James Drennan, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Secure Block Storage

Student's name: James Drennan

This work and its defense approved by:

Committee chair: Philip Wilsey, PhD
Committee member: Karen Davis, PhD
Committee member: Carla Purdy, C, PhD
Secure Block Storage

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering
of the College of Engineering

September 2, 2011

By

James Drennan
BSCompE, University of Cincinnati, 2008

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey
Abstract

Data is becoming much more portable these days with thumb drives and smart phones that can easily have gigabytes of storage. While the portability and ease of transfer of this data is beneficial, it can cause problems when it is lost or stolen. Traditionally securing data on a storage device is achieved using data encryption techniques. This can be accomplished using an encrypted filesystem such as TrueCrypt or some on-device encryption scheme such as that performed by IronKey’s secure flash memory devices. This thesis explores an alternate technique to secure data within a storage device. More precisely, a technique to authenticate each I/O (Input/Output) request issued to the storage device is explored. The device authenticates requests and responds to the request only when the authentication step succeeds. While authentication can occur with each request and is valid only for the individual request, various alternate configurations are possible where authentication is performed for various subsets of the requests. For example, authentication can be enforced only for data reads, data writes, for specific (block) address regions, and so on. For requests that fail authentication, the device may be configured with a number of response mechanisms. While these responses can be virtually anything, some notable response actions would be to: (i) act as a faulty device, (ii) respond with fake data (possibly from some onboard response prepared storage area), or (iii) destroy/erase the stored data rendering the data completely unavailable. Depending on the level of security desired, these failure modes can be temporary or permanent. Lastly, this approach can decouple the act of building an authenticable I/O request from the host and storage devices. More specifically, the I/O requests can be transmitted from the host to a third party for translation to an authenticable form. The third party could be an online server system or a nearby bluetooth device. Thus, a lost or stolen device is decoupled from the platform that builds authenticable requests thereby disabling access to the information stored in a lost device.
In this thesis, the specific authenticating mechanism studied is to add a nonce to all I/O requests and then digitally sign/verify each I/O request. To demonstrate the use of this security scheme, a mechanism for attaching digital signatures to each USB request is constructed. A filter driver is added to a Windows platform to capture block requests to the USB device and route them with a nonce to a bluetooth device for computing the digital signature to attach to the request. The bluetooth device returns the signature and the filter driver packages it with the USB request for transmission to the USB device. A mass storage USB device was modified to receive and authenticate the USB packets. Unfortunately, the process of securing information in this manner is not without cost. Additional time and resources are spent signing, encrypting, and verifying the data. The time spent accomplishing these activities is affected by Bluetooth transfer rates and the microcontrollers on the devices. In particular, the implementation in this thesis impacts performance negatively due to the increase in time needed to apply and verify the security measures. In the worst case, the same actions with security measures enabled take three orders of magnitude more time. The critical performance bottleneck is in the verification step on the USB device. Thus, faster, more advanced, microcontrollers could substantially improve performance and reduce the performance impact to a more manageable level.
Acknowledgments

I would like to thank Dr. Philip Wilsey first and foremost, without his ideas and guidance I would not be where I am today. I would also like to thank the University for helping prepare me to undertake a more in depth project such as this Master’s work. Thanks also goes out to Shamus software for their support and use of the Miracl library, OSR Online for Windows driver support, and Atmel and TI for help with their development kits.
## Contents

1 Introduction ........................................... 1
   1.1 Hypothesis ........................................... 3
   1.2 Thesis Overview ........................................ 4

2 Background .......................................... 5
   2.1 Cryptography .......................................... 5
      2.1.1 Secure Hash Algorithm 256 ................................ 8
      2.1.2 Elliptic Curve Cryptography ................................ 11
   2.2 Device Driver .......................................... 17
   2.3 Universal Serial Bus (USB) ................................ 18
   2.4 Bluetooth ............................................ 19

3 Related Work ......................................... 21
   3.1 Encrypted Filesystems ................................. 21
   3.2 Secure USB Devices .................................... 22
      3.2.1 Generic Secure USB Devices ............................. 22
      3.2.2 IronKey ......................................... 22
   3.3 Two-factor Authentication ............................... 23

4 Overview of Securing transfers .......................... 24

5 Development ........................................... 27
   5.1 Filter Driver ........................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>USB device</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Bluetooth device</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Performance</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions &amp; Future Research</td>
<td>44</td>
</tr>
</tbody>
</table>
# List of Figures

2.1  Client Server interaction during mutual authentication using nonces .......................... 7 
2.2  Functions used in SHA256 ....................................................................................... 9 
2.3  Initial hash values used for SHA256 ........................................................................... 10 
2.4  Pseudo code for the computation portion of SHA256 ................................................. 12 
2.5  Adding two distinct points on an elliptic curve ......................................................... 13 
2.6  Adding a point to its reflection .................................................................................... 14 
2.7  Elliptic curve point doubling ..................................................................................... 15 
2.8  Point doubling when the y component is 0 ............................................................... 16 

4.1  Authenticating Storage: System Overview ................................................................. 25 
4.2  Authenticating Storage with Disconnected Signing Agent .......................................... 25 

5.1  Filter Driver Pseudocode ......................................................................................... 29 
5.2  USB Device Pseudocode ......................................................................................... 33
List of Tables

2.1 Secure Hash Algorithm properties .................................................. 8
2.2 Constants used in SHA256 ............................................................... 10
2.3 Bluetooth transceiver power and range [21] .................................... 20

6.1 Time to verify signature at various operating speeds of USB device .......... 39
6.2 Read times using three setups and file sizes from 4 KB to 4 MB ............... 40
6.3 Read speed using three setups and file sizes from 4 KB to 4 MB ............... 40
6.4 Write times using three setups and file sizes from 4 KB to 8 MB ............... 41
6.5 Write speeds using three setups and file sizes from 4 KB to 8 MB ............... 41
6.6 Write times using three setups, theoretical max transfer speeds for USB, USB 2.0, USB 3.0, on file sizes from 4 KB to 8 MB .................................................. 43
Chapter 1

Introduction

Technology is steadily increasing in many areas including storage, computing, and communication. Advances in storage include increased capacity, power efficiency, gains in solid state drive (SSD) technology, and better interfaces. Western Digital and Intel are two of many that have made improvements. Western Digital offers a 3 TB (Terabyte) hard drive and Intel has a SSD with up to 600 GB (Gigabyte) of storage. Both have made use of SATA (Serial Advanced Technology Attachment) 6 Gb / S technology for faster transfers [1–3]. Continued advancements in flash technology have also allowed for large flash drives such as the 256 GB drive from Kingston [4]. Next generation USB technology, USB 3.0, has also led to faster transfers. This improvement can be seen in flash drives such as Patriot’s Supersonic drive [5]. Gains can also be found in the embedded and mobile market. Atmel has a new microcontroller, the SAM9XE, capable of running at 180 MHz (Mega Hertz) [6]. Qualcomm just recently announced a new quad-core processor for the mobile market [7]. The new Bluetooth 4.0 spec introduces a new type of device known as Low Energy (LE). These devices have reduced power consumption that allows them to run for extended times even on coin cell batteries.

The advances above call for a greater need of security especially with the improvements in storage and the reduction in its form factor. A very large amount of data can now be stored on a medium that is the size of a finger nail and nearly as thin. Three cases show the increased use of these small flash devices to store information that was stolen. The first is a worker that stole trade secret information valued between seven and twenty million from Valspar [8]. Next is a man who used his thumb drive to store fake CIA identification
information [9]. The last case is three people attempting to copy information from a laptop of an aid of an Indonesian delegate [10]. More evidence that security is a growing concern can be seen with the estimated twenty one billion that utility companies will spend on securing the electrical grid [11].

Securing information in flash storage is the issue that requires attention. This attention is warranted because flash storage has high density, small form factor, and is used frequently in devices. These attributes make it possible to store large amounts of information in a small physical space which can easily be misplaced or stolen. Two systems where securing flash storage is valuable are: flash drives connected to secure systems, and mobile devices where the system and storage are one package.

The secure system and the mobile system have similar security needs as well as differences. Both systems must be able grant access to the flash storage to authenticated users. The systems must also be able to prevent information from being altered during transmission such as a man-in-the-middle attack. Another problem shared by the systems is what action to take upon unauthorized access. The main difference between the systems is that the flash storage is part of the mobile system and not generally removable. This alters the problem of securing the system because the attack vectors have moved internal to the system. Mobile devices are more prone to theft due to their form factor and thus authenticating access is more difficult.

There are a few solutions to address the problem of securing flash storage. TrueCrypt is a software solution for on-the-fly encryption and decryption of the filesystems in a storage device [22]. TrueCrypt (and other similar solutions) use cryptographically secure algorithms that utilize a keyfile or password to authenticate access. The keyfile or password can be stored non-locally to increase the security of the system. All operations are done in RAM memory and are never written to the hard drive. A drawback to this solution is that it does not check the integrity of the files being encrypted or decrypted and is thus susceptible to man-in-the-middle attacks. Another solution that migrates the filesystem encryption/decryption into the hardware is IronKey [31]. The main strength of the IronKey solution is its hardware encryption and decryption as well as access systems are located on the flash drive itself. This makes it difficult to compromise the system. Another benefit is remote management through the company’s web tools. However, once authentication is achieved, files transferred to and from the device are not verified and are again susceptible to man-in-the-middle attacks. The encryption and decryption actions will also cause a drop in performance of the flash
drive. Another drawback of this device is once the information has left the drive, it is no longer encrypted.

The solution detailed by this thesis is to secure the flash storage by authenticating I/O requests. A kernel filter driver is used to append authenticating data to I/O requests. The storage device receives and authenticates all I/O requests. The filter driver must be installed once and then runs transparently in the background. The filter driver enables flexibility in selecting which requests need to be authenticated. All of the requests can be authenticated for the most security or a subset, such as only read requests, to increase performance. These requests can also be rerouted to a signing agent, such as a stand-alone Bluetooth device, to carry out the authentication routines. A standard mass storage USB device was modified to accept and authenticate the requests. Failure to authenticate on the USB device could lead to any number of actions including: disconnecting the device, sending bad data, recording further actions, deleting data, and so on.

This project has strengths and weaknesses similar to other approaches for this problem. A key strength of this solution is the ability to combine it with other security measures. Since the security of this solution is based around the requests, the form of the data being transferred is not significant. This means that encrypting the data before it is sent or after it is authenticated on the device only strengthens the overall security. Another strength is that the filter driver, acting as an authentication gateway, can also prevent unsecure devices from registering on the system. An unsecure device does not contain the functionality to verify signatures and thus will be blocked from communicating with the host. A third benefit is the authentication method prevents man-in-the-middle attacks. The digital signature incorporates the request in its generation and any modification of data during transmission will cause the verification to fail. The main weakness of this approach is the performance costs. The authentication measures employed take time which slows the resulting data transfer rates.

1.1 Hypothesis

The hypothesis of this thesis is that the security of block storage devices can be increased by adding an authenticating signature to the I/O requests sent to those devices. The authentication method must have strong cryptographic properties by utilizing established algorithms. The securing process is orchestrated through a filter driver on a host machine. This driver is designed to have no user interaction, after installation, making the added security mostly transparent to the user.
1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides information on the algorithms and technologies utilized. Encryption specifics as well as kernel drivers are detailed. USB and Bluetooth are also discussed.

Chapter 3 covers solutions that are similar to this work. Secure flash drives such as IronKey are mentioned as well as generalized filesystem encryption solutions such as TrueCrypt.

Chapter 4 presents a high level overview of the authenticating solution explored in this thesis. The decomposition of the solution into the host and storage element components is explained.

Chapter 5 explains how the software/firmware demonstrating the concepts outlined in this thesis was created and refined. Two main configurations are developed for analysis. The first is called secure host and it locates the steps to add the authenticating signature to the I/O requests inside the host computer. The second is called unsecure host (or disconnected signing agent) and it uses a bluetooth device to compute the authenticating signature to the I/O requests.

Chapter 6 presents a performance analysis of the data transfer rates for the two versions of the modified USB storage device. Detailed performance analysis and read/write statistics for the following three configurations are provided: no security, secure host, and unsecure host. Steps to improve performance in the design as well as the affects of microcontroller speed variations are also described.

Finally, Chapter 7 discusses the project as a whole, its impacts, and discoveries. Improvements and future work are also reviewed.
Chapter 2

Background

This chapter explains the lower level protocols and systems of the devices utilized. The first section details cryptography and the specific mechanisms used in this thesis. The next section explains device drivers, specifically a kernel filter driver. USB mass storage devices and their sub-structures are then covered. Finally, Bluetooth and the new Low Energy specification are discussed.

2.1 Cryptography

The definition of cryptography is, “the computerized encoding and decoding of information” [12]. This means altering the data in a predetermined and reversible manner so that it is not transmitted in its original or plain form. Cryptography has two main steps, namely: data encryption and data decryption. Messages are encrypted to a secured form for transmission. After transmission, the messages are decrypted for reading. Assuming a strong encryption method, the message is secured from unintended access during transmission.

Cryptographic algorithms fall into two main categories: symmetric-key and asymmetric-key. The categories are named based on the manner through which keys are managed to encode/decode data. Symmetric-key cryptography uses a common key to encrypt and decrypt the data. Asymmetric-key cryptography (also called public-key cryptography) uses two different keys: one key to encrypt (the public key) and a different key to decrypt (the private key). The advantage of symmetric-key algorithms is a faster encrypt/decrypt algorithm that facilitates larger keys and consequently increased security. However, communicating with symmetric-key cryptography requires that users be able to securely share the secret key. While computa-
tionally more expensive, asymmetric-key cryptography has the advantage that one can openly transmit the (public) encryption key to others and securely retain the (private) key for decryption. Unsecured sharing of the public key does not compromise security of the encrypted data because its decryption is achieved only with the private key (assuming that the public key provides no information to help attackers determine the private key). Both symmetric-key and public-key cryptography enjoy widespread use and are sometimes used together for secure, high throughput communication (e.g., SSL/TLS [13]).

*Digital signatures* are a mechanism that employ cryptographic mechanisms to affix additional information to a message that provides evidence that the message did in fact originate from the specified sender. Digital signatures are similar to their physical counterparts (signatures) and can be used to verify data. One similarity to a physical signature is that they are difficult to forge. In general, the process for creating a digital signature is as follows. A hashing algorithm is used to hash the data intended to be signed. This hash is then encrypted using the signer’s private key. The encrypted hash plus the original data is then the signed data and can be sent to the intended party. Verification is carried out in a few steps. The recipient uses the same hashing algorithm to hash the data received. They then use the sender’s public key to decrypt the encrypted hash. If the hash of the original data and the decrypted hash are the same the data and the signature are verified.

*Hashing* is often used with cryptography to create digest or a more compact way of representing the data. Hashing algorithms take input of data of arbitrary length and generate a digest of fixed length. Even though the length of the input can be arbitrarily long, many hashing algorithms work on fixed block sizes, similar to symmetric key algorithms. Since it is unlikely that the data being hashed will divide perfectly into the block size, the idea of padding is introduced. Padding is a scheme that adds extra data to the original in a predetermined manner that is always the same so that the block size requirement is met. Ideally, a hash algorithm suitable for cryptography has little to no collisions. This means that two independent pieces of data will never hash to the same value (a collision). A strong point of hashing algorithms is that changing a single bit of data should alter the hash completely. In this regard, hashing can be used to ensure the integrity of data. If data that is moved or transmitted is altered intentionally or due to errors, the hash will change.

A *nonce*, or number used once, is another cryptographic tool that can be used to secure transactions. The strength of this object comes straight from the definition, that it is only used once. If the same nonce ever
shows up again then something is wrong. A nonce can help stop replay attacks because of this property. A replay attack is when a valid transaction is captured and resent at a later time in order to impersonate the original. An example use of nonces can be seen with mutual authentication. In this example, a client is trying to log into a secure server. Upon asking to log into the server, the server generates a nonce, $sn$, and sends it to the client. The client then generates its own nonce, $cn$, and hashes its nonce, the server nonce, and their password. The client nonce and hash are then sent to the server. Since the client is known to the server, it has all three pieces as well and can verify the hash. The server then calculates a similar but slightly different hash of the server nonce, client nonce, and the password and sends it back to the client. The client can then verify this hash to ensure that the server is legitimate [14], see Figure 2.1.

![Client Server interaction during mutual authentication using nonces](image)

Figure 2.1: Client Server interaction during mutual authentication using nonces
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Message Size (bits)</th>
<th>Block Size (bits)</th>
<th>Word Size (bits)</th>
<th>Message Digest Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>&lt; (2^{64})</td>
<td>512</td>
<td>32</td>
<td>160</td>
</tr>
<tr>
<td>SHA-224</td>
<td>&lt; (2^{64})</td>
<td>512</td>
<td>32</td>
<td>224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>&lt; (2^{64})</td>
<td>512</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>&lt; (2^{128})</td>
<td>1024</td>
<td>64</td>
<td>384</td>
</tr>
<tr>
<td>SHA-512</td>
<td>&lt; (2^{128})</td>
<td>1024</td>
<td>64</td>
<td>512</td>
</tr>
</tbody>
</table>

Table 2.1: Secure Hash Algorithm properties

example is important because it makes the authentication session hard to replicate. Knowing or storing the nonce does not help an attacker since they are used once with the understanding that the nonce generated is suitably long and random.

Multi-precision arithmetic is also important in cryptography. Today, most computer systems are 32 or 64 bit. This means they can store or operate on unsigned integers of size \(2^{32}\) or \(2^{64}\). While this seems like plenty of numbers or space to work with, cryptography deals with even larger numbers. Instead of storing the numbers as fixed structures, they are stored in arrays. Computations on these numbers may take more time but their size is limited to available memory instead of the computer word size. The Miracl library [15] is one such implementation of multi-precision arithmetic. It also contains implementations of many cryptographic algorithms in c, c++, and assembly. This library is highly customizable and also includes support for constrained environments such as 8-bit microcontrollers. Constrained environments are relevant because both the USB device and the signing agent used in this thesis operate using 8-bit microcontrollers. Their operating environments are limited by size, memory, and speed. While these are capable devices, their processing power does not come near that of a modern computer. The flexibility and ease of use are why this library was chosen for this project.

### 2.1.1 Secure Hash Algorithm 256

This hashing algorithm is one of five that the National Security Agency (NSA) designed and the National Institute of Standards and Technology (NIST) released. The algorithm is described as an, “iterative, one-way hash function(s) that can process a message to produce a condensed representation called a message digest” [16].

SHA-256 can process messages of length less than \(2^{64}\). It works on 512 bit blocks using a word size of
32 bits and produces a 256 bit message digest or hash as seen in Table 2.1. This version was chosen because it has a balance of security, speed, and size. These attributes are important because the devices computing the hashes have limited processing power compared to a modern CPU. The algorithm is completed in two parts: preprocessing and the hashing computation.

The preprocessing section of the algorithm involves padding the message, dividing the message up into blocks, and initializing the variables. The padding scheme starts by adding a 1 to the end of the message, adding a certain number of zero bits, then finally the size of the original message in a 64-bit block expressed in binary. Given the message size, $l$, the number of zero bits, $k$, is determined by solving the equation $l + 1 + k \equiv 448 \mod 512$ with $k$ being the smallest non-negative value possible. The initial hash value is set based on the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

The hashing computation can be broken down into four parts. These four parts are computed in order for each of the blocks that the message was divided. Eight working variables, two temporary variables, and sixty four 32-bit words known as the message schedule are used in these four steps. The first step is to prepare the message schedule. For the first sixteen, this is just the sixteen parts of the current block being used. The next forty eight are calculated using a combination of the functions shown in Figure 2.2 and the first sixteen.
second step is to initialize the working variables. These variables are set to the current hash value broken into eight parts. Initially this starts out with the values that were set in the preprocessing section (Figure 2.3). The third step is sixty four rounds or iterations of updating the temporary and working variables using combinations of constants (Table 2.2), message schedule variables, working variables and functions from Figure 2.2. The fourth and final step is computing the intermediate hash. This is done by taking the current hash and adding the appropriate working variable to it. All additions are done mod $2^{32}$. After all blocks have been processed, the last intermediate hash is presented as a single 256-bit block. Pseudo code for the computation can be seen in Figure 2.4.

SHA was selected as the hashing algorithm because it is secure and flexible. The algorithm allows for a variety of hashing sizes. The output sizes are related to the strength of the hash as well as the computation

*Figure 2.3: Initial hash values used for SHA256*

### Table 2.2: Constants used in SHA256

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>428a2f98</td>
<td>71374491</td>
<td>b5c0fbcf</td>
<td>e9b5dba5</td>
<td>3956c25b</td>
<td>59f111f1</td>
<td>923f82a4</td>
<td>ab1c5ed5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d807aa98</td>
<td>12835b01</td>
<td>243185be</td>
<td>550c7dc3</td>
<td>72be5d74</td>
<td>80deb1fe</td>
<td>9bdc06a7</td>
<td>c19bf174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e49b69c1</td>
<td>efbe4786</td>
<td>0fc19dc6</td>
<td>240ca1cc</td>
<td>2de92c6f</td>
<td>4a7484aa</td>
<td>5cb0a9dc</td>
<td>76f988da</td>
<td></td>
<td></td>
</tr>
<tr>
<td>983ef5152</td>
<td>a831c66d</td>
<td>b00327c8</td>
<td>b597fc7</td>
<td>c6e0bfb3</td>
<td>d5a79147</td>
<td>06ca6351</td>
<td>14292967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2b70a85</td>
<td>2e1b2138</td>
<td>4d2c6dfc</td>
<td>53380d13</td>
<td>650a7354</td>
<td>766a0abb</td>
<td>81c2c92e</td>
<td>92722c85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a2bf8a1</td>
<td>a81a664b</td>
<td>c24b8b70</td>
<td>c76c51a3</td>
<td>d192e819</td>
<td>d6990624</td>
<td>f40e3585</td>
<td>106aa070</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19ac4116</td>
<td>1e376c08</td>
<td>2748774c</td>
<td>34b0bcb5</td>
<td>391c0cb3</td>
<td>4ed8aa4a</td>
<td>5b9cca4f</td>
<td>682e6f3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>748f82ee</td>
<td>78a5636f</td>
<td>84c87814</td>
<td>8cc70208</td>
<td>90befffa</td>
<td>a4506ceb</td>
<td>bef9a3f7</td>
<td>c67178f2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
time. SHA-256 strikes a nice balance of these attributes. Another strong point is that it was developed by the NSA and to date has not been broken like other hash algorithms such as MD5. Finally, the ability to use optimized versions of the algorithm through the Miracl library [15] was a compelling reason to choose SHA-256.

### 2.1.2 Elliptic Curve Cryptography

Elliptic curve cryptography is based on the premise that solving the discrete logarithm is hard. An elliptic curve is a plane curve consisting of the points that satisfy the equation $y^2 = x^3 + ax + b$ [17]. A plain curve is a curve that exists in Euclidean space. A discrete logarithm is the same as a regular logarithm, $\log b$ is the solution of $a^x = b$, confined to a specific group. To make the example above a discrete logarithm, $a$ and $b$ would both be part of a group. Understanding how this applies to cryptography requires a more in depth look at elliptic curve operations.

The operations on elliptic curves required for use in the cryptographic algorithm are based and built using addition. There are two givens when working with elliptic curves: a special point, $O$, the infinity point, and the negative of a point. The infinity point is the additive identity and the negative of a point is its reflection over the x-axis. Given two points, $A$ and $B$, such that $-A$ is not $B$, addition is defined geometrically as taking the reflection of the point, $C$, where a line drawn through $A$ and $B$ intersects the elliptic curve, $-C$ (Figure 2.5). Adding a point to its reflection or $A + (-A)$ is equal to the infinity point, $O$, (Figure 2.6).

Point doubling involves taking the tangent line of a point, $A$, on the curve. If the $y$ component of the point is not zero, then the tangent line intersects the curve at exactly one point, $-C$. The reflection of $-C$ is taken for the solution, $A + A = 2A = C$ (Figure 2.7).

If the $y$ component of a point, $A$, is zero, then point doubling results in, $O$, the infinity point $A + A = 2A = O$ (Figure 2.8). Adding $A$ to itself a third time results in $A$ again, $2A + A = O + A = A$. Given a scalar $k$, even numbers of $k$ for $kA$ result in $O$ and odd result in $A$. Computing the result of $kA$ is known as scalar multiplication of a point.

In order to make calculations precise, the groups used in ECC are the finite fields of $F_p$ and $F_{2^m}$. This means that $F_p$ uses the integers between 0 and $p-1$ and $F_{2^m}$ uses $m$-bit strings. Restricting the calculations to a specific field changes the base equation for curves only slightly, $y^2 = x^3 + ax + b \mod p$. 
For $i = 1$ to $N$:
\{

1. Prepare the message schedule, $\{W_t\}$:
\[
W_t = \begin{cases} 
M_t^{(i)}(t_0) & 0 \leq t \leq 15 \\
\sigma_1^{(256)}(W_{t-2}) + W_{t-7} + \sigma_0^{(256)}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63
\end{cases}
\]

2. Initialize the eight working variables, $a, b, c, d, e, f, g,$ and $h$, with the $(i-1)^{st}$ hash value: $a = H_0^{(i-1)}$
   
   \begin{align*}
   b &= H_1^{(i-1)} \\
   c &= H_2^{(i-1)} \\
   d &= H_3^{(i-1)} \\
   e &= H_4^{(i-1)} \\
   f &= H_5^{(i-1)} \\
   g &= H_6^{(i-1)} \\
   h &= H_7^{(i-1)}
   \end{align*}

3. For $t = 0$ to $63$:
\{
   \begin{align*}
   T_1 &= h + \sum_{1}^{256}(e) + Ch(e,f,g) + K_t^{(256)} + W_t \\
   T_2 &= \sum_{0}^{256}(a) + Maj(a,b,c) \\
   h &= g \\
   g &= f \\
   f &= e \\
   e &= d + T_1 \\
   d &= c \\
   c &= b \\
   b &= a \\
   a &= T_1 + T_2
   \end{align*}
\}

4. Compute the $i^{th}$ intermediate hash value $H^{(i)}$:
\begin{align*}
H_0^{(i)} &= a + H_0^{(i-1)} \\
H_1^{(i)} &= b + H_1^{(i-1)} \\
H_2^{(i)} &= c + H_2^{(i-1)} \\
H_3^{(i)} &= d + H_3^{(i-1)} \\
H_4^{(i)} &= e + H_4^{(i-1)} \\
H_5^{(i)} &= f + H_5^{(i-1)} \\
H_6^{(i)} &= g + H_6^{(i-1)} \\
H_7^{(i)} &= h + H_7^{(i-1)}
\end{align*}
\}

Figure 2.4: Pseudo code for the computation portion of SHA256
Using this information, the elliptic curve discrete problem can be described. Given two points, A and B, on a curve in a group, find a k such that kA = B. When the parameters of the curve and group are set properly, this becomes a hard problem. In a real world example of the difficulty of this problem, a 112-bit key for the prime field case was solved using a cluster of over 200 Playstation 3 consoles. It could have finished in around three and a half months if run continuously [18].

ECC was selected for use in this thesis because it is a strong public key encryption algorithm. It also allows for higher security using fewer bits compared to other public key systems that are based on integer factorization. Retaining high security using less data is important because it reduces the transfer costs incurred for adding security. The inclusion of constrained environment implementations of ECC in the Miracl library [15] also made ECC a solid choice. The curve selected was P-256 from the recommended curves generated by NIST [19]. These curves were generated with security and efficiency in mind.

Figure 2.5: Adding two distinct points on an elliptic curve
$y^2 = x^3 - 5x$

Figure 2.6: Adding a point to its reflection
Figure 2.7: Elliptic curve point doubling

The equation of the elliptic curve is:

\[ y^2 = x^3 - 5x + 6 \]

The points marked are:
- **C(-0.9375, 3.1406)**
- **A(2, 2)**
- **C(-0.9375, -3.1406)**
Figure 2.8: Point doubling when the y component is 0

\[ y^2 = x^3 + 5x - 6 \]
2.2 Device Driver

A driver is a piece of software that is created and run to expose functionality for a piece of hardware. It provides an interface to the data or abilities that the hardware possesses. Drivers can support many devices that fall into a class or a specific device. Multiple drivers can be loaded for a single device. When this happens, a driver chain is formed. The type of driver can affect where in the chain it is loaded. Generally, drivers that have broad capabilities and support, like class drivers, are found lower in the chain while device specific drivers are found higher. Filter drivers are similar to regular drivers in that they provide functionality but are more of a support structure. Filter drivers are inserted above or below a regular driver and generally should not affect the original drivers function. They allow for additional or specific utility without having to alter the base driver.

Drivers communicate with each other using I/O (Input/Output) Request Packets (IRP). Drivers in a chain obtain an IRP and inspect it to see if any work can be done. If the driver is able, it completes whatever routines are applicable to the IRP and either passes it on to the next driver or completes the IRP. There are a few standard types of requests that include: create, read, write, and internal. The internal type is usually used when drivers are communicating between each other. This is more often the case when drivers are creating requests internally and passing them to other drivers rather than receiving the IRPs from another source.

Operating systems usually have two modes of execution, namely: user mode and kernel mode. User mode is where most programs run. User mode programs do not often interact directly with hardware. A user mode program makes an operating system or driver call to interact with the hardware. This call usually crosses the boundary between user and kernel mode. Kernel mode is similar to user mode but has greater privileges.

One element that is different about the kernel is the interrupt request level (IRQL). This is basically a value that determines what can interrupt or be interrupted in the kernel. When processes are running in the kernel, they can run at a specific IRQL. Some processes raise or lower the IRQL as well. The higher the IRQL the less that can interrupt the current process running. User mode programs all run at passive level, the lowest IRQL. This means that any other process with a higher IRQL can interrupt a user mode program. Having this level is important because the computer needs to be able to service certain requests
more frequently or before others. An example would be a thermal sensor inside the computer. If the computer got too hot, a process running at a higher IRQL would alert the operating system and hopefully shut the system down gracefully. This would include alerting the user and hopefully saving anything they were working on before shutting down. This is opposed to the computer continuing to run until it malfunctioned because the thermal sensor could not interrupt other processes.

Windows has a tool called the Kernel Mode Driver Framework (KMDF). This makes writing drivers faster, easier, and with better support. The new framework encapsulates much of the startup code for drivers. This makes it easier to get a functioning driver created and running. The KMDF also reduces the number of function calls or lines of code needed to perform tasks. This helps to make the code more readable and easier to manage. Another benefit for filter drivers is that the KMDF automatically processes IRPs that aren’t being filtered. This removes the need to inspect the IRP and write code to pass it on or complete it. The KMDF has support for memory management as well. This is very important in kernel code since the memory space is dedicated to the system. Any problems with memory will almost certainly cause the computer to crash. Since the framework is newer, it also provides functionality for systems such as USB and Bluetooth.

2.3 Universal Serial Bus (USB)

USB is a well established interface for connecting devices to computers. Understanding in more detail how some of the lower layers of USB work will aid in the explanation of this thesis. When a USB device is connected to a computer, one of its configurations is selected. Most USB devices have only one configuration. USB composite devices are one example of having more than one configuration. Composite devices are like having two or more USB devices in one. Each sub-device of the main USB device has its own configuration and is set up independently during initialization. After the configuration is selected, the rest of the device is configured.

During this configuration process, the pipes associated with the selected configuration are enumerated. There are four types of pipes or endpoints that a USB device can use, namely: control, bulk, interrupt, and isochronous. Bulk pipes are relevant to this project. They provide the most bandwidth of the pipes but have no time guarantee. When data is sent through these pipes, it is broken down into sessions.

The sessions are further broken down into three phases, namely: token, data, and acknowledge (ack).
The first phase or the token phase, informs the USB device what action or inaction to take. The second phase or data phase, is optional and may contain the data that the token phase specified. The final phase, the ack phase, signals the end of the current session. In each phase, data sent to or received from the device through the pipes is sent as packets. The size of the packet sent can vary based on the type of pipe used as well as the type of USB connection. Changes in the packet size affect how many packets are sent or received during the data phase as well as the speed of transfers.

Mass storage is one class of USB devices. Three endpoints, one control and two bulk are used in this class. The control endpoint is used to configure the device and set up the other two endpoints. The two bulk endpoints, in and out, are used to transfer data to and from the device.

Mass storage commands issued on the USB are really Small Computer System Interface (SCSI) commands [20]. These commands are found in a command descriptor block (CDB). Inspection of a USB packet along with knowledge of the CDB gives information about the purpose of the USB transfer. This is useful when filtering packets in a driver so that only the appropriate packets are modified.

A URB is the basic structure that allows for all possible commands to be sent to a USB device. The URB is broken down into the header and the function. The header is always included in a URB and contains basic information about the request being sent. The function portion of the URB varies based on the type of pipe and the command. Since the device being used is in the mass storage class, the bulk or interrupt transfer function is of interest.

Among the fields of this structure are the transfer buffer, transfer MDL (Memory Descriptor List), and the transfer buffer length. The transfer buffer or the transfer MDL is used, never both. The transfer buffer is used for smaller packets such as tokens. Transfer MDL’s are used for larger transfers such as the data phase. The transfer buffer length specifies how large either of these structures is. The transfer buffer of a token packet can be further resolved into a CDB. The part of this structure that is useful is the SCSI command. This is the command that ultimately tells the USB device what to do.

### 2.4 Bluetooth

Bluetooth is a short-range wireless technology. It operates in the unlicensed ISM (Industrial, Scientific, and Medical) band at 2.4 GHz with a few power levels and data rates. The different power levels are also known
as classes. Class 1 is the strongest, using the most power and having the greatest range, see Table 2.3. Class 3 and Low Energy use less energy with Low Energy having the lowest minimum specified. “The system employs a frequency hop transceiver to combat interference and fading” [21]. Connections are set up using a master/slave role. The master is the source of synchronization data such as the clock and frequency hop pattern. Pairing is the process that the master and slave go through to establish a connection. Security is one aspect that is set up during pairing.

Bluetooth is a strong candidate for communication to a signing agent because it is well established, power efficient, and localized. Most new electronic devices come with Bluetooth capabilities such as cell phones and laptops. Adding Bluetooth to older or unequipped devices is not difficult either since Bluetooth dongles (usually with USB connectivity) are small and generally inexpensive. Using class 3 or Low Energy Bluetooth devices allows for low energy consumption that can allow a stand-alone device to work for extended periods of time even using batteries. Lower energy also lowers the broadcast range of the devices. This is a desirable security feature because it limits the distance that eavesdroppers could successfully operate.

<table>
<thead>
<tr>
<th>Power Class</th>
<th>Max Power</th>
<th>Min Power</th>
<th>Range (approximate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>100 mW (20 dBm)</td>
<td>1 mW (0 dBm)</td>
<td>100 m</td>
</tr>
<tr>
<td>Class 2</td>
<td>2.5 mW (4 dBm)</td>
<td>0.25 mW (-6 dBm)</td>
<td>10 m</td>
</tr>
<tr>
<td>Class 3</td>
<td>1 mW (0 dBm)</td>
<td>N/A</td>
<td>1 m</td>
</tr>
<tr>
<td>LE (Low Energy)</td>
<td>10 mW (10 dBm)</td>
<td>0.01 mW (-20 dBm)</td>
<td>50 m</td>
</tr>
</tbody>
</table>

Table 2.3: Bluetooth transceiver power and range [21]
Chapter 3

Related Work

This chapter contains information on solutions similar in nature to this thesis. These works include encrypted filesystems such as TrueCrypt [22], secure USB devices such as IronKey [31], and two factor authenticators such as RSA’s SecurID [32]. The encryption solutions and secure devices are similar to the authentication methods in that they are securing the data. The two-factor authenticator is similar to the stand-alone signing agent. Both devices provide unique information that can be used in authentication.

3.1 Encrypted Filesystems

The solutions in this section all aim to secure data from unauthorized access. This is accomplished by encrypting files, folders, partitions, or whole drives. Authentication methods vary slightly but the most common are pins, passwords, or keyfiles. A strength of these solutions, when used on the whole drive, is that even the free space of the drive is encrypted. This makes identifying areas on the drive that contain useful data very difficult. Although a one time cost, this also increases the time to encrypt the drive. Most of the solutions have easy recovery systems in the event that the main authentication method is lost. Some also contain centralized servers for storing keys and managing multiple encrypted disks or systems. Possible downfalls to these solutions are performance hits and possible loss of data due to lost keys and backups.

While all of the solutions secure data through encryption, some of the contain extra features. TrueCrypt for example allows the creation of hidden volumes [22]. The hidden areas are not revealed with the initial password to decrypt the main volume. The hidden volumes are stored where “free” space would normally
be in the outer volume and are therefore undetectable because they look like random data. SecurStar allows for the creation of fake passwords that reveal pre-configured fake data [23]. This allows the user to provide a password instead of refusing altogether. BitLocker is able to integrate the TPM (Trusted Platform Module) [24] chip into its security scheme [25]. TPM chips are found in many newer systems. These chips contain keys that are “burned in” during production which can be used to authenticate and secure systems. The TPM chips can also be used to verify that the system has not been tampered with. This is accomplished by generating a hash of the hardware and software components during initialization and comparing it whenever queried.

The most common approach to the silent encryption and decryption of data is through the use of a driver. The driver is invoked whenever any type of access is requested. The use of a driver can alleviate issues with permissions since they are usually given higher access. Running in the kernel, along with the elevated permissions, can make it more difficult to circumvent the security the driver is providing as well.

### 3.2 Secure USB Devices

#### 3.2.1 Generic Secure USB Devices

There are many secure USB flash drives that offer varying degrees of protection. Almost all of these solutions make use of encryption and authentication to accomplish this security. The encryption scheme used most often is AES 256-bit. Corsair makes a drive called the Flash Padlock 2 that makes use of a physical keypad entry system to unlock the drive [26]. Corsair also makes a drive called The Survivor which utilizes TrueCrypt to secure the drive [27]. Another software encrypted drive, JumpDrive Secure II Plus, is made by Lexar [28]. Imation has a drive, the Pivot Plus Flash drive, that utilizes hardware based encryption and a password based authentication system to secure the drive [29]. A slightly different approach by USB-Secured.com is to use a built-in fingerprint scanner on the drive or a password to unlock the drive [30].

#### 3.2.2 IronKey

IronKey sells secure flash drives with three levels of features. The Enterprise version contains the most features with the personal and basic versions having a smaller and smaller subset. Data stored on the
drives is secured using strong AES 256-bit hardware encryption. Anti-tamper measurements include an epoxy compound that fully encases the chips used inside. An internet based management system allows for administrators to set policies, such as password specifications, on the drives. Some of the drives also contain active scanning for malware and viruses. Another service allows for lost devices to be rendered unreadable until verified, locked out completely, or even to run its self-destruct sequence. These devices use public/private keys so that multiple administrators can monitor the devices. IronKeys can be used as two-factor tokens with their ability to support One-Time Passwords [31]. These devices are similar to the USB mass storage device used in this project. IronKey devices have management software for part of their security where as the USB device for this project utilizes a Bluetooth signing agent and a driver.

### 3.3 Two-factor Authentication

RSA is one manufacture of two-factor authentication systems. Their system makes use of a RSA Authentication Manager, RSA Authentication Agents, and RSA SecurID authenticators. The SecurID authenticators come in a couple form factors including hardware and software versions. The hardware keyfobs are the most relevant to this project. These are small devices that can fit on a key ring and contain a unique symmetric key. The key plus an algorithm is used to generate one-time passwords (OTP) every sixty seconds. This password plus a secret personal identification number (PIN) is part of the something you have and something you know scheme to generate combinations that are near impossible to guess. Since these devices are synchronized with a server, they can be used to authenticate or verify users [32]. The authenticators are similar to the signing agent that this project uses. The Bluetooth signing device is passive though and does not take or give any data to the user directly.
Chapter 4

Overview of Securing transfers

The security system developed for this thesis centers around the filter driver. This filter driver can be set up in two different modes, namely: secure host and unsecure host. The first configuration assumes a secure host and therefore does not require an external signing agent. In this mode, the authentication of I/O requests is completed inside the filter driver itself. In the second configuration, no assumptions can be made about the host and thus an external signing agent is utilized. In the solution explored in this thesis, a separate signing agent communicating to the host with Bluetooth is used. The authentication of I/O requests are forwarded to the signing agent. This situation can be viewed in Figure 4.2. In both cases, the digital signature that was generated from the authentication step is attached to the original I/O request and sent to the USB device. This can be viewed on the left half of Figure 4.1.

The USB device is the flash storage device the security system is designed to protect. The USB device is modified to verify the digital signature and, based on the outcome of the verification, act accordingly. If the signature is verified, access to read and write the flash storage is enabled. If the signature is not verified, the anti-tamper system takes over and can be used to carry out various security measures. The right half of Figure 4.1 illustrates this.

Both configurations, described above, can also be used with a USB device that does not verify the digital signature. The result of this setup is a way of denying unsecured devices. In this specific case, the digital signature added to the I/O request causes a standard USB storage device to malfunction and not properly mount on the host machine.
The system developed has benefits and costs. The main benefit is a semi-transparent security solution that seamlessly combines with other security means. The solution is also modular and allows for different authentication methods to be implemented. With current authentication methods, the solution is resistant to man-in-the-middle attacks. With the filter driver running in the kernel, it is harder to monitor what the driver is doing and possibly modify its behavior. With modifications, this solution can work for any block storage device. The main cost is the performance hit incurred for securing the I/O requests. Authenticating the digital signatures on the USB device causes a severe drop in performance. This drop is even more significant when the Bluetooth keyfob is used to authenticate the I/O requests. Another drawback is key distribution and pairing is not automated. If using the Bluetooth keyfob, it is another device to keep track of
and keep powered.
Chapter 5

Development

This chapter details the development of the three pieces of the project. The first section describes the creation of the kernel filter driver. The filter driver is the central piece of software in the project. It facilitates the communication of the devices as well as the authentication. The next section details the firmware modified for the USB device. This device is an example of a flash storage device that could use the security described in this thesis. The final section explains the firmware developed for the Bluetooth device. This device is an example of a signing agent that could be utilized for unsecure hosts.

5.1 Filter Driver

The filter driver took the most development time. Developing this type of driver involved windows kernel programming. Kernel programming is similar to user mode programming but comes with more responsibility and more involved debugging. Since the kernel is a privileged space, the code written for it must be very exact and have no bugs or defects. In user mode these types of problems crash the program and rarely affect the operating system (OS) itself. This is not true in the kernel. A problem in the kernel almost certainly results in a blue screen (a crash) of the computer and could possibly corrupt the OS. An interactive debugger usually makes finding and correcting errors in code relatively easy. Kernel programs can be debugged interactively but require another computer to debug. This is necessary because the act of debugging in the kernel stops execution on that machine. The secondary computer then controls the first and debugging can be completed normally. This process along with having to wait for the development machine to reboot...
causes the development time to increase. The filter driver is the central piece of the project controlling and initiating communication between the USB device and the Bluetooth signing agent. This is another reason why more time was spent on the driver.

The main goal of the USB filter driver is to access specific USB packets enroute to the USB device and modify them to add security but not break existing functionality, see Figure 5.1 for pseudocode. The code to accomplish this is based of the OSR online filter driver code [33]. This example filter driver was intended to show how a filter driver could be created using the KMDF. In its original form the filter driver reported information back to the user via a corresponding user mode program and did not modify any of the data. The code to report information to the user mode program was removed and debug messages were used instead. It was removed because having the driver be as silent/passive as possible was another goal. The installer portion of the driver was not modified. This program is run from the command line. Upon installation, the program registers the driver but does not insert it as an upper or lower filter. Inserting the entry into the registry has to be done manually or programatically. The correct location to insert the driver is discussed below.

Finding the correct location to insert the filter driver was an important task. The best location is where the other drivers in the stack do as much of the work as possible. This reduces the amount of work and code needed in the filter driver. Simplifying the filter driver also reduces errors, memory requirements, and speed penalties. The first location chosen for the filter driver did not work. The filter driver was able to obtain the data from the other driver but it was not in a form that was conducive for modifying. A better understanding of the packet structures and requirements along with a limited number of locations to insert the filter resulted in success with the next tried position. This location was a lower filter in the sub-field of the device found in Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB. The data packets were in a form that could be decoded or fit into existing structures. This allowed for a more detailed look at the individual pieces of each packet.

USB device setup information can be obtained once the filter driver is inserted in the correct position. Filtering the configuration and interface packets found by looking for the select configuration function and the select interface function in the URBs allow setting up KMDF structures related to USB. These structures, including the USB pipes, allow for easy access to the USB device within the filter driver. This access is used
Figure 5.1: Filter Driver Pseudocode

to initialize the nonce that is shared between the driver and the USB device. A read packet is sent to the USB device after it has been configured and set up by the OS but before normal stead-state operation has begun. The read packet is modified to have a custom SCSI command that only the custom USB device recognizes. In addition to the custom SCSI command, the token packet of the read command is slightly longer than normal to further specify that the driver is asking for the nonce value. Since both the driver and USB device modify the nonce value with the same frequency and function, they only need to sync at start up. Filtering of specific packets is possible now with the known information. Since SCSI commands are based on a standard it is possible to look up the different control codes. Reads and writes are the most important actions that need to be secured and correspond to the hex codes: 0x28 and 0x2a respectively. When a token packet with either of these codes is detected in the filter driver, the driver invokes functions to add security to the packet before it is sent on to the next driver and ultimately the USB device. Protection could be added to additional packets if the corresponding SCSI commands were later found to be in need of such protection. Limiting security code to specific packets allows the system and the USB device to perform normally and faster since no additional actions are taken on non-filtered packets.

The filtered packets are secured by adding signature data to them. The first attempts to accomplish this
involved modifying system structures to add a signature field. The new structure was then allocated and all of the original data was copied to it in addition to a small static signature. The signature was made small and static to make testing and verifying it on the USB device simpler and easier. The attempts with modifying the system structures failed. The system was unable to recognize the modified structures. The next attempt was to use the same structures found in the filtered packets but create, initialize, and send them from within the filtered driver. This approach was met with some success. The first problem was attempting to send the newly created packets synchronously. Synchronous transfers were selected so that order could be maintained in the flow of packets to and from the USB device. Sending data synchronously did not work due to invalid handle issues and improper IRQL. Moving to asynchronous calls fixed these issues. The first packets that were created and sent successfully were token packets. Though successful in sending to the USB device, they eventually caused it to stop working properly. The problem this time was not conforming to the USB spec. The USB device was also expecting an ack packet. Once this packet was built and sent after the token packet, the device would continue to work. The final solution to send modified packets was developed to try and reuse as much of the original packets as possible and reduce or remove the need to send extra packets to the USB device. This solution was accomplished by determining which type of buffer the bulk or interrupt packet was using, a local buffer or an MDL, and create a larger version with the signature in it. After the correct buffer was determined, the original contents were copied into the new buffer along with the signature size and the signature itself. The buffer address field of the current packet being modified was then changed to the new buffer. The transfer buffer length field was also updated.

The IRQL issue resurfaced with the realization that the driver would have to wait while the signature was being processed on the Bluetooth device. The first solution to run at passive IRQL, where waiting is allowed, was to create a worker thread. Worker threads always run at passive level. While this solution solved the IRQL issue, it increased the complexity of the filter driver. Worker threads are asynchronous by definition. A separate thread running at a lower IRQL is spawned and queued items are executed when that thread is serviced. The use of a worker thread was abandoned when a simpler solution involving the filter thread’s queue was found. When the filter driver is loaded, it creates a queue to deal with the incoming data. One of the options for configuring this queue includes the IRQL that it runs at along with the functions that support that queue. Even though running at this lower IRQL introduces the possibility of other services
interrupting the filter driver, this does not happen very often and the ability to wait for periods of time is necessary.

The signature that provides the security that is being added to the packets is based on the ECC algorithm which uses SHA2 as an input. For development purposes, the ECC algorithm was implemented in the driver first. This was done to reduce communication issues and verify that the algorithm was working properly. The ECC and SHA2 algorithms are based on the code found in the Miracl library [15]. C code versions of these algorithms were modified to remove the need for external file access and ensure that they could compile in the driver. The external calls were removed so that the filter driver was stand alone and to increase the security of the process. Generating a signature of the current packet involves sending the transfer buffer data and the current nonce to the ECC function. Since token packets can be very similar or even identical, the nonce is the first data input into the SHA2 algorithm followed by the transfer buffer data. Adding the nonce to this process produces hashes and then signatures that are very different even when the transfer buffer contents are the same. After each full hash is computed, the nonce value is updated. This function is the same as on the USB device and Bluetooth keyfob so they stay in sync. The signature and the size of the signature are returned to the driver. Similar to the process of obtaining the nonce, the driver saves the current SCSI code and replaces it with a custom code that the USB device recognizes. The actual SCSI code, the size of the signature, and the signature are then appended to the buffer for token packets or pre-pended for data packets. The USB device did not respond well to signature data following the data packet and thus it had to be added to the start of the packet. Adding the nonce to the hash did cause verification issues at first. This problem was resolved when it was determined that the signature algorithm was being called for all packets instead of just the filtered read and write packets.

In order to use the Bluetooth keyfob, additional code had to be added to the filter driver. Since the USB dongle that actually does the Bluetooth communications shows up as a serial device, serial device initialization code had to be added. A serial monitoring program was used to find the correct initialization steps that were needed to set up the USB dongle. After this was complete, the serial commands that were necessary to establish a connection, read, and write to the keyfob were recorded from TI’s BTool program and implemented in the filter driver. Since the filter driver can execute commands faster than the BTool program, delays had to be inserted so that the keyfob would not get congested with commands and stop.
responding. Fine tuning these delays allowed for better performance of the transfers.

Adding the filter driver to a device that does not recognize the modified packets can be beneficial. When the filter driver was added to filter a standard flash drive, the drive never fully registered with the OS. Windows device manager recognized a device was plugged in but it was unable to fully configure it. This was caused by the additional data being added to the USB packets. The extra data would cause the device to get out of sync. This can be helpful because it allows only recognized devices with the security enabled firmware to connect to the host. Installing the filter driver for all block storage devices or as a class filter would enable this functionality.

5.2 USB device

The USB device is based on the AT90USBKEY evaluation kit from ATMEL. This small evaluation board comes with a micro USB connection, an 8 MHz crystal, two data banks totaling 16 MB of flash storage, two LEDs, and break out pins for the JTAG (Joint Test Action Group) interface. The device is capable of USB 2.0 connections at Full speed. The LEDs on the board have two colors that can be lit independently which allow for four different visible outputs. The device is programmable through AVR Studio and a program called FLIP or using a JTAG programmer. FLIP makes use of the boot loader included on the board and allows for quick and easy programming of the board with new code. The JTAG interface exposes more programming options including fuses to change how the board operates as well as interactive, real-time debugging.

The firmware or code that is programmed on the device is based on the mass storage code from ATMEL. In its unaltered form, this firmware causes the device to behave like a flash drive with three endpoints, control, bulk in, and bulk out. During inspection of this base firmware, two areas of interest were located. The first area is where the token packets are processed. This is important because knowing exactly how the device processes the tokens allows for creating well formed tokens as well as modified ones. Code found in this area also revealed how the different parts of the system interacted, specifically the mass storage code with the USB controller. The USB controller is the system that interacts with the host computer and sends or receives data. The second area of interest is where the SCSI command is decoded and the appropriate action is taken. Exploring this code path revealed how the device actually reads or writes data on a low level. This
is useful if data needs to be accessed and possibly modified or verified. Pseudocode for the altered firmware can be viewed in Figure 5.2.

Modifications to add security to the USB firmware started simple like the filter driver. The first changes involved adding code to recognize the custom SCSI command created to inform this device that the driver was sending modified token packets. The built in LEDs were used programmatically to verify when certain code paths were run. Using the LEDs was helpful in determining what was happening during runtime on

```
{
    DeviceStartup()
    DeviceConfiguration()
    while(Device is connected and active)
    {
        ProcessTokenPackets()
        if(SpecialPacket)
        {
            if(FirstPacket)
            {
                SendNonce()
            }
            else
            {
                SaveSignature()
                VerifySignature()
            }
        }
        if(DataPhase)
        {
            ProcessDataPacket()
            if(SpecialPacket)
            {
                SaveSignature()
                VerifySignature()
            }
            ProcessAckPacket()
        }
    }
    DeviceShutdown()
}
```

NOTE: Surprise removal can occur at any time.

Figure 5.2: USB Device Pseudocode
the USB device but was also limited. These outputs are useful for simple debugging but pale in comparison to a full runtime debugger. After verification that the device was running the new code based on the LED output, additional code was added to check a small static signature.

The first attempt to increase debugging capabilities on the USB device was to introduce a secondary device making it a USB composite device. The secondary device was a serial communication interface. The code for this type of device was also obtained from ATMEL. In addition to adding this code to the firmware being compiled for the USB device, some of the configuration files had to be modified to ensure that both devices were set up properly and recognized by the host machine. Now when the device was plugged into a host machine it showed up as a mass storage device and a communication device. Global variables were used to allow the two devices to communicate between each other on the USB device. A global string could be updated with variable and state information that was then sent to the host machine every time it was updated. The contents of this string were then printed in a serial console session that was established after the device was plugged in. When it worked, this method of debugging was fairly effective. Unresolved issues establishing a connection with the communication part of the USB device or the operating system recognizing the USB composite device hindered this method. Moving from Windows XP to Windows Vista made the problem worse. This move was made because Windows XP lacks full driver support of Bluetooth devices. Ultimately the composite solution was dropped due to lack of functionality and usability.

Adding the ECC algorithm for signatures was the next major addition to the USB firmware. This development was pursued even with the lack of decent debugging abilities. This was possible because the algorithms being added could still be tested on the device internally even if the signatures could not be compared to ones being sent from the driver. The ECC algorithm was based on the code found in the Miracl library [15]. A stripped down version containing assembly code was modified to run on the USB device. Using assembly code for internal functions of the algorithm allows it to run faster and in less space on constrained environments like the 8-bit MCU of the USB device. Other modifications included updating the constants that the algorithm used to be in line with the signing agent and to use the nonce as the first input to the hash algorithm before verifying signatures. Using the output of a test string, “abc”, to the hash algorithm was used to verify that the SHA2 algorithm and the ECC algorithm were running correctly and producing the same results as the signing agent.
The runtime of the signing algorithms was taking a considerable amount of time on the USB device. Changing the signing to make use of MD5, a smaller, simpler, and less secure algorithm, did not make significant improvements in speed. Reducing the number of rounds that SHA2 was running, and reducing its security, also did not prove to be an effective solution. Further investigation into the speed issue revealed that the USB device was shipped with an 8 MHz operating speed but was being reduced to 1 MHz with an internal divider. This divider is active to ensure stability on the shipped device and is modified by a fuse setting. The fuse settings cannot be modified by any means shipped with the stock kit such as through the bootloader. To resolve this issue, an AVR Dragon was obtained from ATMEL. This is a development board that allows for programming and debugging of many of ATMEL’s chips. One such connection on this device is a JTAG interface. The JTAG interface allows for programming and debugging on devices. The programming capabilities also include modifying fuses. Removing the divider on the clock improved the device speed but it is still considerably slower than a desktop machine.

The next step was to generate a full signature from a filtered packet and verify it on the USB device. This task was difficult before the use of the AVR Dragon because a lot more data was being sent. Finding errors or mismatches in the increased amount of data could not be completed quickly or easily using only LEDs. Interactive debugging with the AVR Dragon revealed that all but the last few bytes of the signature were being transmitted to the USB device correctly. It was determined that the last few bytes were being read incorrectly because the USB controller was running out of data but still servicing the request resulting in garbage data. After sixty four bytes of data, the USB controller needed to receive an ack to be able to fetch the next chunk of data. When the code to correct for this error was added, the signatures came across fine and were verified on the USB device.

Adding code related to the nonce was the final piece of firmware added to the USB device. The first addition was the initialization code for the nonce. This code is executed once when the special SCSI command, 0xC1, is sent by the filter driver. The nonce is generated using four calls to the rand function resulting in an unsigned thirty-two bit integer. The random number generator (RNG) is seeded using the USB frame number. The frame number is an eleven-bit incrementing number used to synchronize the USB bus. The RNG was seeded with this because it is rarely the same value when the device is plugged in and the initialization code is called. The nonce is stored in global memory on the USB device so that the other functions,
including the cryptography functions, can utilize it. The final step to the initialization is to send the nonce back to the host so that the signing agent can use it. The code to start the hashing algorithm with the nonce was also added. This is the same as the signing agent. The update routine was the last code added related to the nonce. This function simply increments the nonce and rolls it back over to 0 if it reaches the maximum. The update function could be anything as long as it matches the signing agent. An update function with as large of period as possible is desirable to reduce the chance for roll over. Roll over and specific data could lead to a reply attack succeeding but the conditions for this to occur are very small.

5.3 Bluetooth device

The Bluetooth solution is based on the Bluetooth low energy (BLE) Mini Development Kit from Texas Instruments. The kit came with three devices, two of them with CC2540 chips and a hardware debugger. The two with the CC2540 were a USB dongle and the other was a battery powered keyfob like device. Each device came with firmware preloaded onto it. The USB dongle contained firmware to accept Host Controller Interface (HCI) commands which allow it to communicate with the keyfob. The keyfob contained firmware to accept connections, read, write, and inquiry commands from the USB device. The keyfob firmware is based on the SimpleBLEPeripheral project provided from TI.

In order to interact with the SimpleBLEPeripheral firmware, TI also provided a program called BTool. This simple program allows the user to connect to the USB dongle through a USB/serial connection and issue commands to the keyfob through a user interface. The program provides a log of all commands and the response generated during program use. This was convenient because it allowed for replication of the serial commands sent to the USB dongle in the filter driver. After initialization, connection and read/write sequences were replicated in the driver, additions and modifications were made to the keyfob firmware.

The first addition to the firmware was the signing code. This involved adding the correct files from the miracl library and compiling them in with the original firmware. A problem that was immediately apparent was the code size limitations of the keyfob. The removal of unnecessary code was the first step to combat this issue. Next, the signing algorithm was reevaluated to see if any of the steps could be simplified. It turned out that many of the signing algorithm steps could be precalculated. This reduced the computation time and the amount of extra code that needed to be compiled and flashed onto the device. The final step
to get the signing code working was to increase the stack size on the device to compensate for some of the larger structures that were used by the algorithm. Much like the USB device, the signing code was verified using a set nonce and a test string. Using the debugger, the code was broken into after the signing code was run and the signature was checked against the known outcome.

The next step was to modify the firmware to accept larger amounts of data to be read from the keyfob and written to it. The original profile for the keyfob contained four characteristic values that were each only one byte. These values had varying read/write permissions. All four characteristic values were extended to be able to hold nineteen bytes each and have both read and write permissions. The values were extended to nineteen bytes because this was the max amount of data that could be read or written in one command to the keyfob. The read and write functions were also updated to handle the larger characteristic values as well as process the data to generate a signature. On a write to the keyfob, the processing involved adding each byte written to the hash for the signature generation. When all of the data was written, the signature was generated and the size of the signature along with the signature itself were saved into the four characteristic values. The four values could then be read sequentially to obtain the size and the signature.

The last modification was to allow for a special write to the keyfob to initialize the nonce. This modification was an additional change to the write function on the keyfob. If a write to the first characteristic value of size four was encountered, the four bytes were saved as the nonce and also processed as the first four bytes of the first hash to be used in the first signature. The remaining hashes were initialized with the updated nonce after the generation of a signature.
Chapter 6

Performance

This chapter reviews the performance implications when using the solution detailed by this thesis. The areas investigated are read and write speed impacts. These results are compared based on the unmodified USB device, the secure system setup, and the unsecure system (utilizing the signing agent). Additional results include microcontroller speed variations and ideal case calculations.

The main goal of the performance analysis was to find the impact of the developed solution and improve it, if possible. The second goal was to benchmark the performance based on read/write speeds using the three configurations: unmodified, secure system, and unsecure system. The two areas that caused the most slow down were found to be in verifying the signature on the USB device and getting a signature from the signing agent. These preliminary results were obtained by adding timing code to the filter driver and monitoring how long each section took.

With the resources at hand, the verification could not be improved upon. This was due to the nature of the code used to verify the signature as well as hardware restrictions. The library that the verification code is included in is already optimized. Furthermore, unlike the signature code, no offline or code that always evaluates to the same result could be removed to speed up the verification process. A function for reducing the operating speed of the USB device was discovered and used to investigate how the operating speed affected the performance of the verification routine. Without being able to modify the software or the hardware, the operating speed had the most direct impact on verification runtime.

The results from changing the speed on the verification time can be seen in Table 6.1. The last entries
Table 6.1: Time to verify signature at various operating speeds of USB device

<table>
<thead>
<tr>
<th>Speed(MHz)</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>52.4886</td>
</tr>
<tr>
<td>2</td>
<td>24.3433</td>
</tr>
<tr>
<td>4</td>
<td>11.7574</td>
</tr>
<tr>
<td>8</td>
<td>5.7653</td>
</tr>
<tr>
<td>16</td>
<td>2.7283</td>
</tr>
<tr>
<td>180</td>
<td>0.2092</td>
</tr>
</tbody>
</table>

of 16 MHz and 180 MHz are based on a fit of the curve generated from the first four points resulting in a power curve with the equation: $y = 51.697 \times x^{-1.061}$ with an $R^2 = 0.9997$. $y$ is the result in seconds using a speed, $x$, in megahertz. 16 MHz is the maximum operating speed of the AT90USB1287 chip used on the USB device. 180 MHz is based on the SAM9XE microcontroller by Atmel which is capable of running at that speed [6]. Slower speeds were capable but below 1 MHz the stability of the security solution was not ensured due to timeouts from the system. The times used with each speed are an average of ten verifications. An average of the verification times was used because of the random nature of the signatures being verified.

The second problem, getting the signature from the signing agent, was improved significantly. Reevaluation of the code that sent and received data through the Bluetooth devices revealed that the timeouts being used were excessive. Refining this code reduced the time spent getting the signature from around thirty seconds down to around three. An exhaustive search to find the lowest timeout and delay values could be performed to reduce this time even more. Given the current setup, the time to get the signature from the signing agent could be reduced to zero if the next packet that needed to be signed was pre-fetched. The system has to wait for the USB device to verify the signature, which takes about six seconds. If the next packet was pre-fetched the three seconds it takes to get the signature would be included in the time already spent waiting for the verification.

The performance impact of the security solution can best be viewed by the read/write speeds on various file sizes. Each of the tests performed were measured on the three system setups: no security, secure system, and unsecure system. The read speed test performed was created by running a script that would report the time it took to run the “TYPE” command on the specified file. The TYPE command reads the specified file and prints the contents to a console window. The results of this test can be seen in Table 6.2 and Table 6.3.
Table 6.2: Read times using three setups and file sizes from 4 KB to 4 MB

<table>
<thead>
<tr>
<th>File size(bytes)</th>
<th>No Security</th>
<th>Secure system</th>
<th>Unsecure System</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.13</td>
<td>6.06</td>
<td>18.11</td>
</tr>
<tr>
<td>8192</td>
<td>0.37</td>
<td>6.20</td>
<td>18.72</td>
</tr>
<tr>
<td>16384</td>
<td>1.31</td>
<td>7.30</td>
<td>19.42</td>
</tr>
<tr>
<td>32768</td>
<td>3.16</td>
<td>9.03</td>
<td>18.42</td>
</tr>
<tr>
<td>65536</td>
<td>6.75</td>
<td>15.73</td>
<td>31.00</td>
</tr>
<tr>
<td>131072</td>
<td>14.59</td>
<td>29.53</td>
<td>47.92</td>
</tr>
<tr>
<td>262144</td>
<td>29.61</td>
<td>44.83</td>
<td>66.61</td>
</tr>
<tr>
<td>524288</td>
<td>58.93</td>
<td>75.06</td>
<td>104.43</td>
</tr>
<tr>
<td>1048576</td>
<td>121.57</td>
<td>132.59</td>
<td>173.56</td>
</tr>
<tr>
<td>2097152</td>
<td>244.23</td>
<td>248.39</td>
<td>328.00</td>
</tr>
<tr>
<td>4194304</td>
<td>481.19</td>
<td>484.56</td>
<td>628.22</td>
</tr>
</tbody>
</table>

The write test utilized a script to copy a file of a specified size to the USB device from the host computer and record the time it took to complete. The results from this test can be seen in Table 6.4 and Table 6.5.

The read and write tests both show a significant performance penalty when adding the security system. The performance penalty in both tests is due mostly to the time spent verifying the signature. This penalty is increased in the unsecure host system due to time spent getting the signature from the signing agent. A trend that both tests show is the larger the file being tested the better the performance. This performance is still below the unmodified system but in the secure host approaches the same speed for the read test. One reason

Table 6.3: Read speed using three setups and file sizes from 4 KB to 4 MB

<table>
<thead>
<tr>
<th>File size(bytes)</th>
<th>No Security</th>
<th>Secure system</th>
<th>Unsecure System</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>31508</td>
<td>676</td>
<td>226</td>
</tr>
<tr>
<td>8192</td>
<td>22141</td>
<td>1321</td>
<td>438</td>
</tr>
<tr>
<td>16384</td>
<td>12507</td>
<td>2244</td>
<td>844</td>
</tr>
<tr>
<td>32768</td>
<td>10370</td>
<td>3629</td>
<td>1779</td>
</tr>
<tr>
<td>65536</td>
<td>9709</td>
<td>4166</td>
<td>2114</td>
</tr>
<tr>
<td>131072</td>
<td>8984</td>
<td>4439</td>
<td>2735</td>
</tr>
<tr>
<td>262144</td>
<td>8853</td>
<td>5848</td>
<td>3936</td>
</tr>
<tr>
<td>524288</td>
<td>8897</td>
<td>6985</td>
<td>5020</td>
</tr>
<tr>
<td>1048576</td>
<td>8625</td>
<td>7908</td>
<td>5872</td>
</tr>
<tr>
<td>2097152</td>
<td>8587</td>
<td>8443</td>
<td>6394</td>
</tr>
<tr>
<td>4194304</td>
<td>8717</td>
<td>8656</td>
<td>6676</td>
</tr>
<tr>
<td>File size(bytes)</td>
<td>Time(sec)</td>
<td>No Security</td>
<td>Secure system</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>4096</td>
<td>0.45</td>
<td>215.75</td>
<td>345.33</td>
</tr>
<tr>
<td>8192</td>
<td>0.45</td>
<td>192.62</td>
<td>335.99</td>
</tr>
<tr>
<td>16384</td>
<td>0.50</td>
<td>216.31</td>
<td>335.14</td>
</tr>
<tr>
<td>32768</td>
<td>0.69</td>
<td>181.31</td>
<td>309.12</td>
</tr>
<tr>
<td>65536</td>
<td>0.86</td>
<td>215.84</td>
<td>336.44</td>
</tr>
<tr>
<td>131072</td>
<td>1.34</td>
<td>205.46</td>
<td>309.14</td>
</tr>
<tr>
<td>262144</td>
<td>2.37</td>
<td>234.55</td>
<td>373.13</td>
</tr>
<tr>
<td>524288</td>
<td>4.39</td>
<td>272.30</td>
<td>465.18</td>
</tr>
<tr>
<td>1048576</td>
<td>8.25</td>
<td>346.95</td>
<td>543.06</td>
</tr>
<tr>
<td>2097152</td>
<td>16.10</td>
<td>469.97</td>
<td>738.99</td>
</tr>
<tr>
<td>4194304</td>
<td>31.74</td>
<td>701.02</td>
<td>1099.25</td>
</tr>
<tr>
<td>8388608</td>
<td>63.06</td>
<td>1323.11</td>
<td>2043.92</td>
</tr>
</tbody>
</table>

Table 6.4: Write times using three setups and file sizes from 4 KB to 8 MB

<table>
<thead>
<tr>
<th>File size(bytes)</th>
<th>Speed(byte/sec)</th>
<th>No Security</th>
<th>Secure system</th>
<th>Unsecure System</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>9102</td>
<td>19</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td>18204</td>
<td>43</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td>32768</td>
<td>76</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td>47490</td>
<td>181</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>76205</td>
<td>304</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td>97815</td>
<td>638</td>
<td>424</td>
<td></td>
</tr>
<tr>
<td>262144</td>
<td>110609</td>
<td>1118</td>
<td>703</td>
<td></td>
</tr>
<tr>
<td>524288</td>
<td>119428</td>
<td>1925</td>
<td>1127</td>
<td></td>
</tr>
<tr>
<td>1048576</td>
<td>127100</td>
<td>3022</td>
<td>1931</td>
<td></td>
</tr>
<tr>
<td>2097152</td>
<td>130258</td>
<td>4462</td>
<td>2838</td>
<td></td>
</tr>
<tr>
<td>4194304</td>
<td>132146</td>
<td>5983</td>
<td>3816</td>
<td></td>
</tr>
<tr>
<td>8388608</td>
<td>133026</td>
<td>6340</td>
<td>4104</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Write speeds using three setups and file sizes from 4 KB to 8 MB
for the increase in performance as the file size increases is buffering. This means the transfer, including the security routines, completes faster than the given operation, reading or writing. When this case is true, the read or write operation does not have to wait and can execute more efficiently. This is especially true for the read test since the TYPE command takes time to execute. Another reason for the performance penalty is the retransmission of IRPs. Even though the same number of packets were being transferred, the secure host and unsecure host were processing more packets than the unmodified host. This is most likely due to the filesystem, which is higher up the driver chain, reissuing the IRP. In the worst case, each packet was reissued resulting in double the number of packets processed.

Another performance related aspect that was investigated was the ideal case. In this case, all of the individual parts would work at their peak performance. Areas where the ideal case was applied were getting the signature, sending data to the USB device, and verification of the signature.

Getting the signature involves sending and receiving data over the air via Bluetooth. Currently this process takes about 3.25 seconds. 3 of the 3.25 seconds is spent waiting to ensure that the signing agent can process the commands and send and receive the data. A total of 217 bytes are sent and received for each Token packet that needs to be signed. If this amount of data is sent, minus the wait time, it results in a transfer rate of 868 bytes/sec. $\frac{217}{0.25} = 868$. If the theoretical max transfer rate of 1 Mb/s was achieved in the ideal case, the transfer would take 0.00166 secs. $\frac{217}{131072} = 0.00166$. In the ideal case, the transfer costs could be reduced by three orders of magnitude. The generation of the signature is not in the same order of magnitude as the transfer time but could be brought close with a faster microcontroller. With a more direct interface to the USB dongle that sends and receives the data to the signing agent the delays introduced could be reduced as well.

The next area that the ideal case can be applied to is the USB transfer rates. Even though the USB device used is a USB 2.0 device it does not support high speed. It instead runs at full speed. The difference in the theoretical max of full speed and high speed is 12 Mb/s and 480 Mb/s. Additionally, USB 3.0 adds super speed at a rate of 5 Gb/s. A comparison of the time to transfer the tested files at the theoretical speeds can be seen in Table 6.6. If the solution provided ran at the theoretical max of full speed it would reduce the time taken by an order of magnitude and possibly two (assuming the ideal case for all other areas).

The last piece of the ideal setup is the USB device’s microcontroller. Increasing the operating speed up
Table 6.6: Write times using three setups, theoretical max transfer speeds for USB, USB 2.0, USB 3.0, on file sizes from 4 KB to 8 MB

<table>
<thead>
<tr>
<th>File size (bytes)</th>
<th>Unsecure System</th>
<th>Secure system</th>
<th>No Security</th>
<th>Full speed (12 Mb/s)</th>
<th>High speed (480 Mb/s)</th>
<th>Super speed (5 Gb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>345.53</td>
<td>215.75</td>
<td>0.45</td>
<td>0.0026</td>
<td>6.5104E-05</td>
<td>5.1035E-06</td>
</tr>
<tr>
<td>8192</td>
<td>335.99</td>
<td>192.62</td>
<td>0.45</td>
<td>0.0052</td>
<td>0.00013</td>
<td>1.2207E-05</td>
</tr>
<tr>
<td>16384</td>
<td>335.14</td>
<td>216.31</td>
<td>0.50</td>
<td>0.0104</td>
<td>0.00026</td>
<td>2.4414E-05</td>
</tr>
<tr>
<td>32768</td>
<td>309.12</td>
<td>181.31</td>
<td>0.69</td>
<td>0.0208</td>
<td>0.00052</td>
<td>4.8828E-05</td>
</tr>
<tr>
<td>65536</td>
<td>336.44</td>
<td>215.84</td>
<td>0.86</td>
<td>0.0417</td>
<td>0.00104</td>
<td>9.7656E-05</td>
</tr>
<tr>
<td>131072</td>
<td>309.14</td>
<td>205.46</td>
<td>1.34</td>
<td>0.0833</td>
<td>0.00208</td>
<td>0.00020</td>
</tr>
<tr>
<td>262144</td>
<td>373.13</td>
<td>234.55</td>
<td>2.37</td>
<td>0.1667</td>
<td>0.00417</td>
<td>0.00039</td>
</tr>
<tr>
<td>524288</td>
<td>465.18</td>
<td>272.30</td>
<td>4.39</td>
<td>0.3333</td>
<td>0.00833</td>
<td>0.00078</td>
</tr>
<tr>
<td>1048576</td>
<td>543.06</td>
<td>346.95</td>
<td>8.25</td>
<td>0.6667</td>
<td>0.01667</td>
<td>0.00156</td>
</tr>
<tr>
<td>2097152</td>
<td>738.99</td>
<td>469.97</td>
<td>16.10</td>
<td>1.3333</td>
<td>0.03333</td>
<td>0.00316</td>
</tr>
<tr>
<td>4194304</td>
<td>1099.25</td>
<td>701.02</td>
<td>31.74</td>
<td>2.6667</td>
<td>0.66667</td>
<td>0.00625</td>
</tr>
<tr>
<td>8388608</td>
<td>2043.92</td>
<td>1323.11</td>
<td>63.06</td>
<td>5.3333</td>
<td>0.13333</td>
<td>0.01250</td>
</tr>
</tbody>
</table>

to 180 MHz as shown in Table 6.1 would reduce verification times to the tenth of a second range. While this is an order of magnitude better than the current results, it is feasible that this time could be reduced even further. Including a dedicated security co-processor and moving to a more advanced microcontroller are two improvements that could be made. The use of a security co-processor could reduce the time spent on computationally expensive code segments since the co-processor is highly optimized in this area. A more advanced microcontroller would have greater word lengths (16, 32, or 64 bit) and faster interfaces to sub-systems such as the memory bus. Larger word lengths help reduce the number of instructions executed since a greater part of the data can be operated on in a single call.
Chapter 7

Conclusions and Suggestions for Future Research

Overall, the project was a success and the hypothesis was proven possible. While possible, the performance results show that the current version of the solution does not make it feasible to use on a daily basis (unless the user is willing to sacrifice a significant amount of time). It is certainly possible to improve the performance of the system with faster more efficient parts. An actual construction with these parts would have to be completed to know for certain whether the solution could be used in a consumer market.

The solution developed provided many learning experiences. The filter driver was a major portion of the project that presented many challenges that had to be overcome. Developing and working in the Windows kernel is one such challenge. Pre-existing experience with setting up the kernel debugger environment and actually debugging in the kernel was invaluable. This knowledge made it possible to write the filter driver since errors in the filter driver caused system crashes. Debugging allowed the errors to be viewed unlike in normal operation where error codes, that are usually unhelpful, are presented on a blue screen for a short time. Using the KMDF was a great choice since it simplified the code and handled a lot of the memory management. Memory management and using the proper variables in function calls are two areas that can easily get a driver writer into trouble. Improper allocation or freeing of memory is a runtime error that causes system crashes and is easy to overlook. Even though documentation and forum support is growing, it is still difficult to find helpful resources in driver programming.
As difficult as it was, the choice to use a filter driver was the right one. The filter driver, in this case, offers the best of both worlds with respect to accessibility and responsibility. A kernel filter driver allows access to low level functions and the hardware that is not easily available in user mode. This access also increases the risk and severity of errors. This is another reason that the filter part of a filter driver is beneficial. With the KMDF, most of the work is done with respect to cases that are uninteresting or unnecessary to the driver writer. This is opposed to writing a full blown driver where the start-up and various cases that the driver encounters all have to be developed. Along these same lines, using a filter driver allows the original driver to do most of the work.

There are two ideas, in this area of drivers, that could improve the solution. The first would be to interface more directly with the Bluetooth dongle instead of through the serial interface. Issuing commands directly to the USB endpoints of this device might reduce the inter-driver communication needed on unsecure hosts. It also might allow for more efficient transfers from the USB dongle to the Bluetooth fob though this might involve writing more firmware to handle the longer reads and writes necessary for this project. The second idea is a filter driver higher up in the chain near the file system. Access to the original requests might make it possible to reduce retransmissions due to the long time needed for signing. This idea might not be necessary if the verification time and the time needed to get a signature from the signing agent was reduced.

Both embedded devices were good choices to prove the hypothesis of this thesis. The interactive development environments (IDE) provided by Atmel and IAR (for the TI chips) worked very well and made development quicker and easier. If accurate simulation software for the embedded device used is not available, an interactive debugger is invaluable. Without the aid of the AVR Dragon to debug the Atmel chip, it is unlikely that the problems encountered would have been solved without a serious investment of time. The disadvantage of the test kits used was the reduced performance. The trade off for ease of development versus performance was well worth it in the end though.

The cryptographic algorithms used is another area of discussion. The use of SHA256 and ECC worked well. The balance of speed and security by choosing this combination is valid but more investigation in this area can be done. The lack of proper debugging early in the project moved the focus to finding a solution that functioned rather than the very best solution. Finding and testing more cryptographic algorithms on the hardware used could possibly lead to a combination that ran with better performance.
The mostly transparent solution to this problem is a benefit. A minimal amount of user interaction to set up and use the system helps increase the security. Less interaction decreases the areas of vulnerability for an attacker. On the flip side, adding a password element to the solution could also increase the overall security. The transparent ability to combine this with other security solutions is also a benefit. The fact that the solution provided by this thesis combats security in a different manner allows it to be combined with other solutions. The use of requests to establish integrity and signatures to maintain that integrity as opposed to obscuring the data is the main difference.

Even though USB autorun functionality is limited and has caused problems with security, it could allow this solution to be deployable from the USB device itself. The use of secure and unsecure areas of storage on secure USB drives is not new. Using the unsecure storage area to store the driver and an installation program could easily be developed. Having multiple secure devices and more widespread deployment would bring about other issues that would need to be resolved. One such problem is making the filter driver robust enough to interface with multiple USB devices and various signing agents. Allowing one signing agent to generate signatures for multiple devices is also possible but a more complex problem to solve.
Bibliography


