FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Security analysis of USB drive
Student:	Bc. David Jagoš
Supervisor:	Ing. Jiří Buček, Ph.D.
Study Programme:	Informatics
Study Branch:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	Until the end of winter semester 2019/20

Instructions

Research existing vulnerabilities of encrypted USB drives. Perform a security analysis of Kingston DataTraveler Vault Privacy encrypted flash drive. Try to find its vulnerabilities in order to extract stored data without knowledge of the password.
- Analyze the internal structure of the flash drive.
- Analyze the client software supplied with the flash drive.
- Analyze and document undocumented APIs.
- Perform experiments in order to analyze the dependency of the encryption key on the password.

Evaluate the results and discuss potential impact of your findings.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 23, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Security analysis of USB drive

Bc. David Jagoš

Department of Digital Design
Supervisor: Ing. Jiří Buček

January 10, 2019

Acknowledgements

I'd like to thank my supervisor, Ing. Jiří Buček, for his counsel and immense patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 10, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Jagoš, David. *Security analysis of USB drive.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce shrnuje bezpečnost flash disků s hardwarovou podporou šifrování a poskytuje bezpečnostní analýzu disku Kingston DataTraveler Vault Privacy.

Klíčová slova šifrování flash disků, Kingston, DataTraveler Vault Privacy, DTVP, Phison, PS2251-63, PS2263

Abstract

This thesis provides an overview of the security of flash drives with hardware encryption support and a security analysis of Kingston DataTraveler Vault Privacy.

Keywords flash drive encryption, Kingston, DataTraveler Vault Privacy, DTVP, Phison, PS2251-63, PS2263

Contents

List of Figures

List of Tables

Introduction

We live in a world of information and protecting that information is becoming more important with each passing year. The never-ending barrage of information leaks forces us to realize just how much damage can be done when sensitive information falls into the wrong hands.

Private citizens risk their personal information being leaked and misused for identity theft or blackmail and corporations could lose their intellectual property or leak business strategies. These scenarios may not seem too frightening, but there are others, much more serious ones as well. Governments have sensitive information about all of us, not to mention lists of undercover police and intelligence operatives, locations of military bases, personnel rosters and details of national defense systems just to name a few. And let us not forget dissidents who could end up being persecuted by oppressive regimes if their data fell into the wrong hands. In these scenarios, security of information is literally a matter of life and death, whichever side you are on.

One of the most prominent ways we store and transport data are USB flash drives. Their small size and portability make them very practical, but also very easy to lose. In a survey conducted by the Ponemon Institute at Kingston's behest [1] 70 percent of respondents reported the organization they were working for suffered a loss of sensitive or confidential data as a result of losing a flash drive. That is a staggering statistic. If those flash drives were recovered by a third party and the data on them were unprotected, this could result in huge loses as well as prosecution by authorities for mishandling personal information.

It is clear that if we are to store sensitive data on flash drives, we need to protect it. But how? There are of course traditional software based encryption tools such as archive managers with encryption support (e.g. 7-zip) and tools for creating encrypted virtual volumes as well as encrypting physical drives (e.g. BitLocker, TrueCrypt and its successor VeraCrypt), but they all share one common weakness—there is no limit to how many passphrases the attacker may try, as well as no rate limiting other than the inherent computa-

tional complexity of the cryptographic algorithms used. If you rely purely on software solutions to secure your data and lose your flash drive, an attacker will be able to try passphrase after passphrase, potentially at the speed of millions per second, for as long as he or she needs to. Are you really confident your passphrase will be able to withstand such a barrage? And what if you accessed your encrypted data on a different computer—the purpose of a flash drive is to be portable after all—which was running a keylogger?

Flash drives with hardware encryption support offer solutions to many of these issues, but do they really work, or are they just empty promises? The relatively long history of failed products in this category doesn't seem very encouraging, but perhaps manufacturers have learned their lesson from past failures. Furthermore, there are no standards or certifications designed specifically for flash drives with hardware encryption support. There are some more general standards for cryptographic modules (like the FIPS-140) which are often applied to these flash drives, but they are hardly comprehensive for this use case.

With identity theft on the rise and the General Data Protection Regulation looming over companies' heads, there has arguably never been a greater demand for secure flash drives.

Privacy is a right and security of information is increasingly often a legal necessity, but do we have the technical means to ensure them?

This thesis attempts to answer those question.

In chapter 1 I explain what are flash drives with hardware encryption support and summarize their defining characteristics. In chapter 2 I go over past attacks against flash drives with hardware encryption support. In chapter 3 I introduce the Kingston DataTraveler Vault Privacy and explain why it was chosen as the subject of this thesis. In chapter 4 I divine the properties of PS2251-63—the controller used in Kingston DataTraveler Vault Privacy. In chapter 5 I analyze the control software supplied with Kingston DataTraveler Vault Privacy and explore its undocumented APIs. In chapter 6 I determine the relation between the user's password and the key used to encrypt the data. Finally, in chapter 7 I attempt to bypass the password try limit by physically modifying the flash drive.

Chapter 1

Flash drives with hardware encryption support

First, I would like to define what I mean when I talk about flash drives with hardware encryption support. This chapter lays out the key characteristics of the encrypted flash drives I encountered in the course of my research and how they differ from their normal counterparts.

1.1 Where is the encryption performed

The main difference between flash drives with hardware encryption support and ordinary flash drives is where the potential encryption is performed. This is their defining characteristic.

1.1.1 Ordinary flash drives

If encryption is used at all with an ordinary flash drive, it is performed by the host computer. This can be problematic, because the cryptographic secrets are—at least for the duration of encryption or decryption—present on the host computer. If the computer is compromised at that time, the secrets can be retrieved by an attacker.

1.1.1.1 Encrypted archives

Among the most commonly used tools for protecting data (not only) on a flash drive are archive managers. Archive managers weren't designed with data encryption in mind. That functionality was only added as an afterthought. For example WinZip introduced encryption support in version 6.2 and only added support for AES in version 9 [2]. Similarly, 7-zip introduced encryption support in version 2.30 [3]. This results in them being quite cumbersome to use for this purpose as well as lacking in security because—since most programs

do not have the ability to access files within archives—for a file to be viewed or edited, it generally needs to be extracted first, meaning it will exist in its unencrypted form somewhere on the computer (probably in some temporary location). Despite this issue (and perhaps because of lack of public awareness), their ubiquity still makes them the first choice of many. In fact, WinZip even explicitly advertises this use case [4].

1.1.1.2 Encrypted virtual disks

Another option is using a product such as VeraCrypt which utilizes encrypted container files whose contents are exposed to the host system as a virtual disk. This greatly improves usability as well as security. As far as the host operating system is concerned, any file being viewed or modified is already present in the clear, so there is no need to copy the file onto an unencrypted disk for viewing or editing; the file's unencrypted version will only exist in RAM [5].

The encryption software can be stored on a flash drive alongside the encrypted container, making this solution portable [6].

1.1.1.3 Full disk encryption

The last commonly used method for protecting data on ordinary flash drives to be discussed in this chapter is full disk encryption. In this scenario the entire content of the drive (including partition headers) is encrypted. The only advantage of full disk encryption over an encrypted virtual disk of which I am aware is plausible deniability (i.e. the owner could claim that his or her flash drive is filled start to finish with random data and it would technically be impossible to prove that it actually contains an encrypted volume [7]). A major disadvantage in usability compared to encrypted virtual disks is that the decryption software can not be stored alongside the data (since the entire drive is encrypted and therefore inaccessible without use of the decryption software), requiring the user to load it onto the target computer separately.

Notable products in this category are VeraCrypt and BitLocker.

1.1.2 Flash drives with hardware encryption support

In flash drives with hardware encryption support, the encryption is performed by the controller of the flash drive. Once the drive is unlocked, the encryption/decryption process is completely transparent to the host computer—as far as it is concerned a completely ordinary flash drive is connected [8][9].

If this scheme is implemented properly, no cryptographic secrets ever leave the flash drive, so even if the host computer is compromised, an attacker should only be able to steal data currently saved on the drive, but should not be able to unlock it him or herself later.

1.2 User input methods

Flash drives with hardware encryption support can be split into categories based on how users authenticate themselves.

1.2.1 Software

Some flash drives authenticate the user via software running on the host computer. These drives usually present themselves to the system as a CD-ROM drive containing the software necessary for unlocking the encrypted partition. After the user is authenticated, a USB mass storage device is presented to the host computer [10].

This approach allows users to comfortably input even very long passphrases, but it suffers from the same weakness as all software-based encryption: the user's passphrase is, even if only for a short time, present in the memory of the host computer.

1.2.2 Hardware

Hardware-based approaches to authenticating users make it possible to keep all cryptographic secrets only on the device itself, but they are often less practical or less secure [10].

1.2.2.1 Keypads

Some drives opt for a physical keypad on the drive itself where a user can enter his or her password. The physical dimensions of a flash drive severely limit the number of distinct characters available (usually decimal digits) and the bad ergonomics limit the practical length of a password. As a result, passwords input via a physical keypad on a flash drive tend to have relatively low entropy. An example of a flash drive with a hardware keypad can be seen in figure 1.1.

Figure 1.1: Kingston DataTraveler 2000 16 GB [11]

1.2.2.2 Fingerprint scanners

Fingerprint scanners offer much better ergonomics as well as much higher entropy, making them, in theory, almost a perfect solution. In practice, however, many finger print scanners have a relatively high rate of false positives and are easily fooled (sometimes even with just a printout of the user's fingerprint) [12]. An example can be seen in figure 1.2.

Figure 1.2: Hama "ProtectionKey" FlashPen [13]

1.2.2.3 RFID readers

This approach is mostly used in portable hard drives with hardware encryption support. The drive is unlocked with an RFID tag. The obvious flaw of this scheme is that for practical use, the RFID tag needs to be transported with the hard drive and if an attacker can get one, there is no reason to think he or she will not be able to obtain the other.

1.3 Typical hardware configuration

By combining information from previous research [14] with my own findings I have compiled what I believe to be the typical hardware configuration of a flash drive with hardware encryption support (figure 1.3). The host sys-

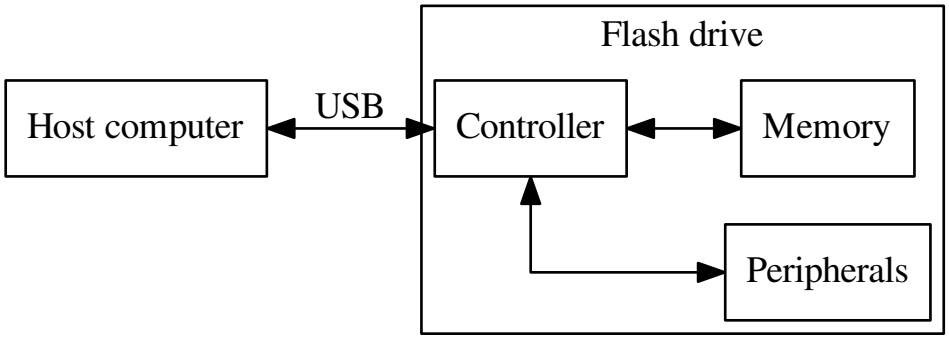

Figure 1.3: Typical hardware configuration of a flash drive with hardware encryption support

tem communicates with the flash drive over USB. The encryption process is completely transparent, so once the drive is unlocked, the communication is exactly the same as with an ordinary USB mass storage device. The controller performs the encryption and decryption, as well as all the standard housekeeping tasks associated with using flash memory. Some controller manufacturers provide the controller and flash memory in a single package for increased security (eavesdropping or MitM would require decapping the chip) as well as simpler PCB layout. Finally the controller may also communicate with peripherals such as keypads or fingerprint readers.

Chapter 2

Past attacks

In this chapter I will go over some past attacks on encrypted flash drives.

2.1 The SySS hack

In late December 2009 SanDisk issued a security bulletin [15] announcing several drives from their Cruzer Enterprise lineup had "a potential vulnerability in the access control mechanism" and that they released a software update addressing the issue. Kingston [16] and Verbatim [17] soon followed suit. Kingston even issued a recall, offering to replace affected drives with newer models.[18]

The vulnerability was discovered by SySS GmbH, a German pen-testing company.[19] While reverse engineering the control application for one of the affected drives, SySS researchers noticed that the application always sent the same binary string to the flash drive to unlock it, irrespective of the password used. They then tried simply replaying this sequence and the drive unlocked. Every one of the affected drives across different manufacturers could be unlocked by simply sending it this magic sequence.

This tells us something very important: Flash drive manufacturers are not the ones implementing the cryptography, that is done by a third party, presumably the controller manufacturer.

This vulnerability affected the following drives:

- SanDisk Cruzer Enterprise USB flash drive, CZ22—1 GB, 2 GB, 4 GB, 8 GB
- SanDisk Cruzer Enterprise FIPS Edition USB flash drive, CZ32—1 GB, 2 GB, 4 GB, 8 GB
- SanDisk Cruzer Enterprise with McAfee USB flash drive, CZ38—1 GB, 2 GB, 4 GB, 8 GB

- SanDisk Cruzer Enterprise FIPS Edition with McAfee USB flash drive, CZ46—1 GB, 2 GB, 4 GB, 8 GB
- Kingston DataTraveler BlackBox (DTBB)
- Kingston DataTraveler Secure—Privacy Edition (DTSP)
- Kingston DataTraveler Elite—Privacy Edition (DTEP)
- Verbatim Corporate Secure USB Flash Drive—1 GB, 2 GB, 4 GB, 8 GB
- Verbatim Corporate Secure FIPS Edition USB Flash Drives—1 GB, 2 GB, 4 GB, 8 GB

Please note that these drives were FIPS 140-2 Level 2 certified by the US National Institute of Standards and Technology.

2.2 The Google hacks

In July 2017 three Google engineers gave a talk at BlackHat USA titled "Attacking Encrypted USB Keys the Hard(ware) Way." Sadly, they did not publish a paper, so all we have is a recording of the talk[10] and a PDF with the slides[14]. They also chose not to directly identify any of the products they had cracked. I did manage to identify most of them, but even in cases where I failed to identify the product, we can still gleam the kinds of mistakes the manufacturers made while designing the drives.

2.2.1 Generic flash drive with a fingerprint scanner

This flash drive is being sold under many obscure brand names as well as without any branding whatsoever (see figure 2.1). As such, the expectations of quality are not too high.

The flash drive has a separate fingerprint manager chip and flash controller and the two communicate over UART (see figure 2.2). The message sent from the fingerprint manager to the flash controller is not some form of a fingerprint digest as one might expect, it is a simple number—an ID of the recorded fingerprint that matched the scanned finger. Furthermore the IDs aren't even randomized, they're assigned sequentially, starting from a known value.

All an attacker needs to do to break into this flash drive is to tap into the UART lines and send the ID of the first recorded fingerprint to the flash controller. This is very similar to the SySS hack. The attack is harder to carry out as the device needs to be physically opened and two wires need be connected (not too much harder though—the UART pins are nicely laid out and labeled on the PCB), but it is worse in that it can not be fixed with a software patch.

Figure 2.1: Generic flash drive with a fingerprint scanner[14]

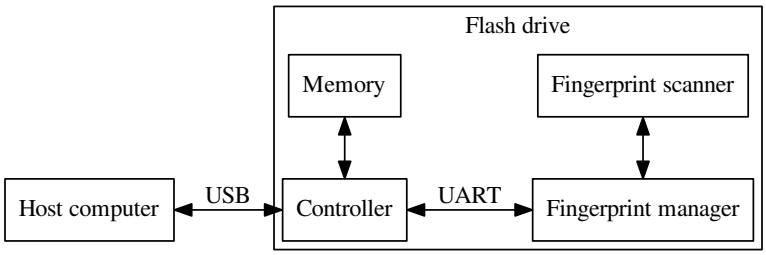

Figure 2.2: Structure of a flash drive with a fingerprint scanner

2.2.2 Digittrade RS64 RFID portable hard drive

While not a flash drive, the weakness demonstrated here could very well apply to one.

The drive is unlocked with a simple RFID tag which can be surreptitiously cloned using equipment anyone can easily and cheaply obtain. If a drive like this is left unattended under the presumption that the data on it can only be accessed by authorized personnel, a malicious actor could quickly read the RFID tag belonging to one of the authorized operators and later create a copy of this tag and use it to access the data.

Another critical flaw in this product is that while it does limit the number of login attempts, the value of this counter seems to be stored in volatile memory and is reset to 0 when the device is power cycled. The process of trying several IDs and power cycling the device could be easily and cheaply automated for a brute-force attack. The feasibility of this attack would de-

Figure 2.3: Digittrade RS64[14]

pended on the length of the ID of the used RFID tag as well as on whether the IDs are assigned randomly or sequentially. This information was sadly not revealed during the presentation.

2.2.3 DM PD061 flash drive with a fingerprint reader

This flash drive uses a fingerprint scanner to authenticate users and a software tool to manage them. Multiple users can be registered simultaneously and to manage users an administrator password is used. It also has an extremely silly undocumented feature: you can request the administrator password and the flash drive will give it to you. In cleartext. With no authentication required.

Figure 2.4: DM PD061[14]

With this password an attacker can then log into the user management program, register him or herself as a new user and unlock the drive using his or her own fingerprint.

2.3 Unverified claims

While working on this thesis I came across a number of individuals claiming they had broken the encryption of various flash drives from obscure ones to Kingston's IronKey D300. The individuals making these claims never published their work, nor have they shown any proof of concept, making their claims unverifiable. While a lot of these claims were obvious lies, some seemed plausible and two of them were made by highly respected gurus on their respective flash drive hacking forums. I am not going to cite any specific claims here, but I will say I strongly believe at least some of them are true and that there are many more compromised flash drives than the public is aware of.

Chapter 3

Kingston DataTraveler Vault Privacy

Originally released in 2009, the Kingston DataTraveler Vault Privacy is quite an old device. It was chosen for analysis despite its age because it is the drive Kingston offered as a replacement for drives compromised in the SySS hack,[18] so there was a reasonable expectation it should be secure and I wanted to ascertain how well that expectation was met.

Figure 3.1: Kingston DataTraveler Vault Privacy 8GB

In this chapter I will go over the drive's specifications, its hardware structure and the control software bundled with it.

3.1 Specifications

The Kingston DataTraveler Vault Privacy boasts the following features[20]:

- All data is encrypted with 256-bit AES in CBC mode.
- Up to 24 MB/s read speed.
- Up to 10 MB/s write speed.
- Can be mounted in a read-only mode.

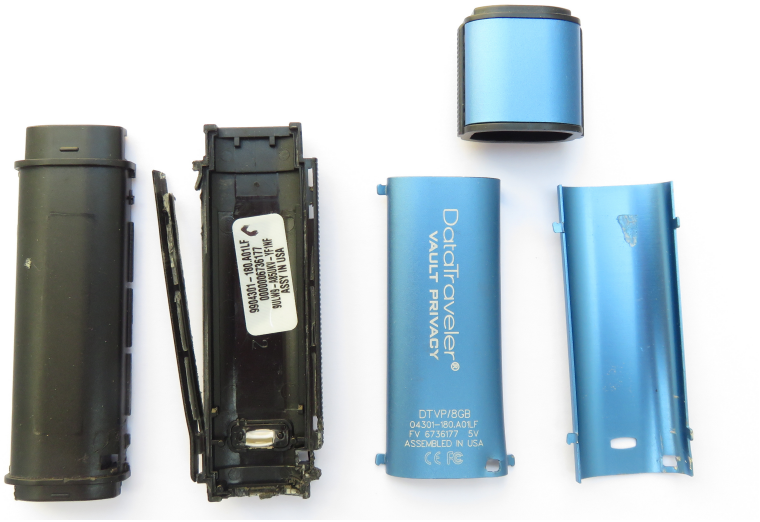

Figure 3.2: Enclosure taken apart

- The drive destroys stored data after 10 unsuccessful login attempts.
- Windows, Linux and Mac OS X support.
- USB 2.0 compliant.
- IPX8 rating (waterproof up to 4 feet).
- Operational temperature from 0 °C to 60 °C.
- Storage temperature from −20 °C to 85 °C.

The drive comes in 4 GB, 8 GB, 16 GB, 32 GB and 64 GB variants.

3.2 Hardware

The circuit board is placed in a black plastic enclosure with an outer aluminum shell (figure 3.2).

The drive is built around the Phison PS2251-63-6 flash controller and one or two 48-pin TSOP NAND flash chips. My testing unit came with a single MT29F64G08CFACA flash chip (figures 3.3 and 3.4).

With the help of the flash chip's datasheet[21] I can decode the part number: It is a 64Gbit MLC NAND flash with an 8-bit shared address and data bus and asynchronous I/O.

I could not find a datasheet for the PS2251-63-6; I will address this issue in chapter 4.

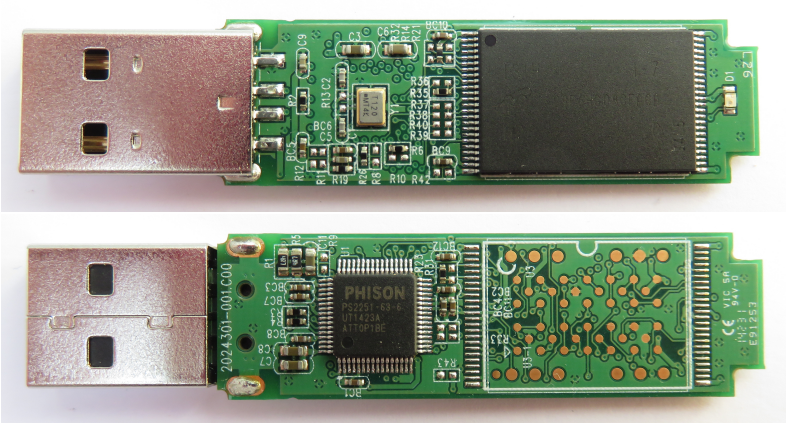

Figure 3.3: Front and back views of the PCB

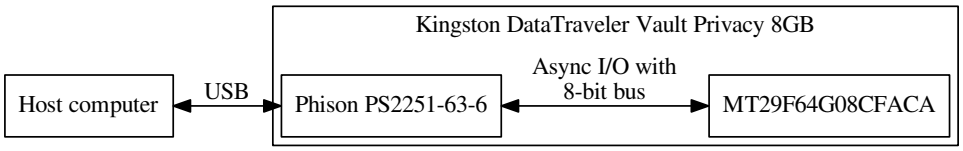

Figure 3.4: Kingston DataTraveler Vault Privacy 8GB hardware diagram

3.3 Software

When the flash drive is connected, it presents itself to the host computer as a CD-ROM drive containing the control software and a manual.

The Windows control software is a simple GUI-based application allowing the unlocking of the drive or resetting it (figure 3.5), changing settings such as the password or contact Information (figure 3.6) and viewing the device information (figure 3.7).

The Mac OS X application is visually and functionally identical to its Windows counterpart. It is written in Objective-C.

Lastly the Linux control software consists of 5 command line tools:

- dtvp_about
- dtvp_forgotpassword
- dtvp_initialize
- dtvp_login
- dtvp_logout

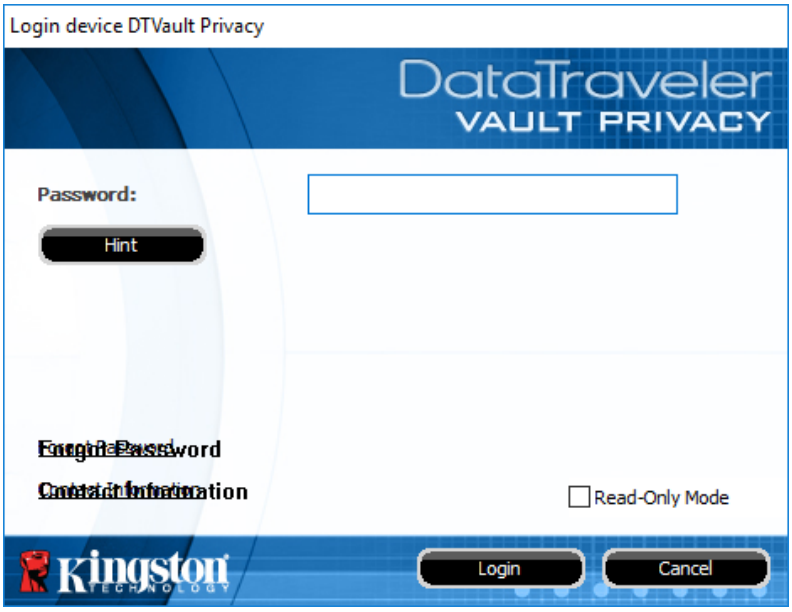

Figure 3.5: Windows unlock dialog

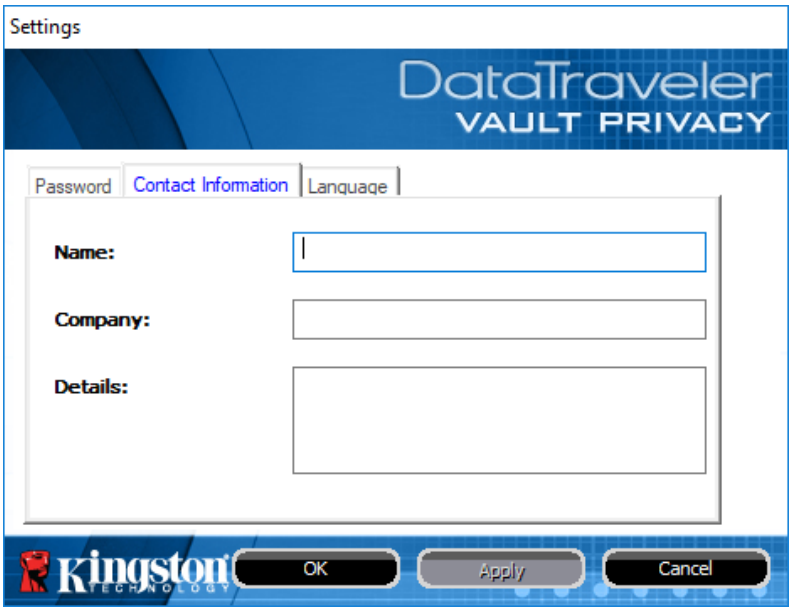

Figure 3.6: Windows settings dialog

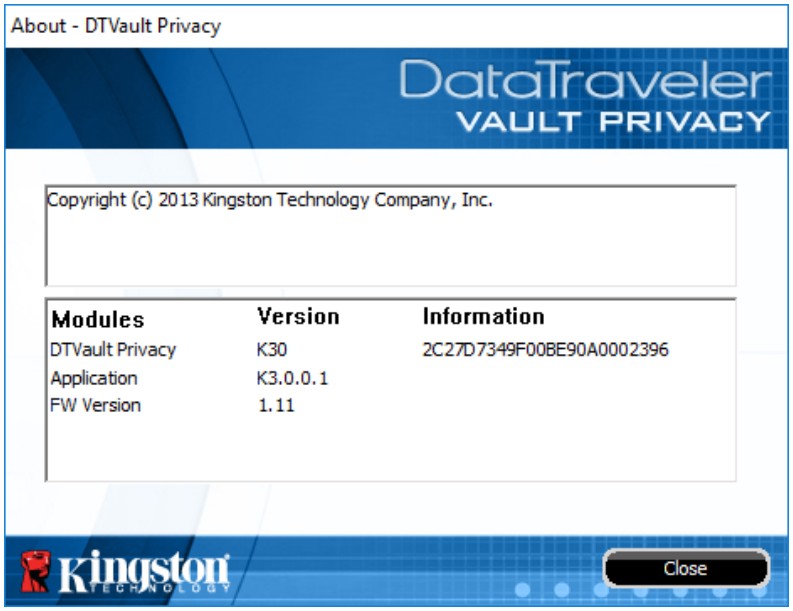

Figure 3.7: Windows device information dialog

The password is limited to a length between 6 and 16 characters and must contain characters from at least three of the following groups: uppercase letters, lowercase letters, digits and special characters (all printable ASCII characters not belonging to the previous three categories).

Chapter 4

Phison flash drive controllers

Since I could not find a datasheet for the PS2251-63-3, alternatively called PS2263 (datasheets are not public, presumably they are only officially available to Phison's customers under an NDA and are therefore often found in very dubious places), I decided to compile information from datasheets of a couple other models of Phison flash controllers (specifically PS2251-31/PS2231 [22] and PS2251-33/PS2233 [23]) as well as bits of information shared on various flash drive repair and hacking forums to get an idea of the general structure and feature set of these controllers.

I focused on USB 2.0 NAND flash controllers.

4.1 General characteristics

They are an all-in-one solution specifically designed for USB flash drives. The following characteristics seem to be common to all models:

- USB 2.0 support
- four endpoints
  - Endpoint 0: 64 Bytes CONTROL transfer
  - Endpoint 1: 512 Bytes BULK transfer for IN transaction
  - Endpoint 2: 512 Bytes BULK transfer for OUT transaction
  - Endpoint 3: 64 Bytes INTERRUPT transfer for IN transaction
- supports SLC and MLC NAND with 2 kiB and 4 kiB pages
- 3.3V and 1.8V NAND
- in-system programming via USB
- 8051 compatible processor core

Following are characteristics supported only by some models which I believe to be relevant to the PS2251-63-6:

- MLC NAND with 8 kiB pages
- Secure Hidden Area
- 256-bit AES hardware module
- hardware RNG

It seems reasonable to assume that the encryption key or a part of it could be stored in the Secure Hidden Area.

4.2 Firmware

As expected, I could not find a PS2251-63 firmware image. I did manage to find leaked firmware images for 6 different Phison flash controllers, but none of them were for controllers with hardware AES support (or Secure Hidden Area support), so there would be little to gain from analyzing them.

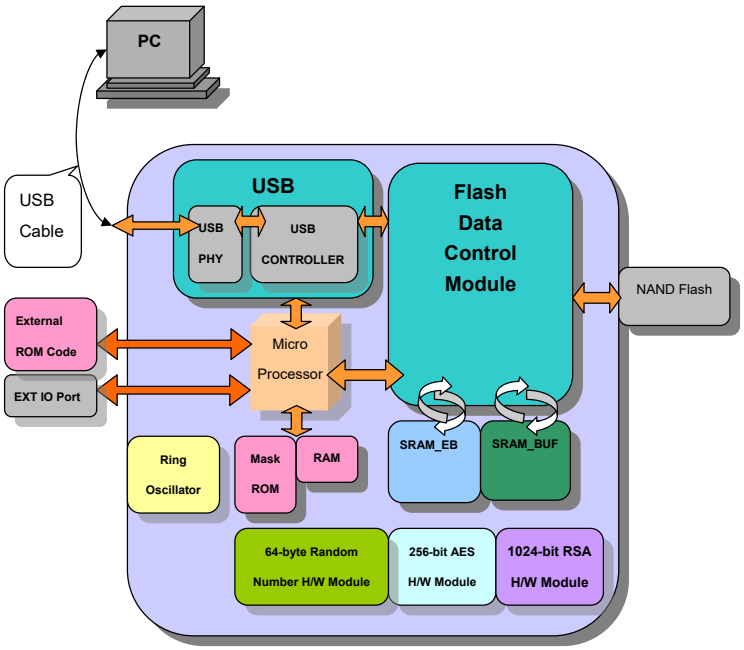

Figure 4.1: PS2251-31 block diagram [22]

According to a block diagram from the PS2251-33 datasheet (figure 4.1), the controller stores one part of the firmware in mask ROM (programmed

during chip fabrication, unchangeable; probably a bootloader of sorts) and the other (presumably main) part is stored off-chip. Since there are no chips on the board other than the controller and the NAND flash, the firmware must be stored in the NAND flash memory.

There is a leaked toolset called MPALL for flashing firmware on flash drives based on Phison controllers via USB. According to USBDEV.ru, there also is a tool for dumping the firmware via USB, both however require a so-called "burner file" which is unique for each controller and is usually leaked alongside the firmware image. Unfortunately the burner file for PS2251-63 seems to not have been leaked, so if the firmware were to be obtained, it would need to be retrieved from the NAND flash directly.

4.3 Features

While the firmware for PS2251-63 hasn't been leaked, the firmware configuration tool bundled with MPALL does support it, so, combined with a leaked manual for an older Phison firmware configuration tool, we can gleam what features this controller has to offer.

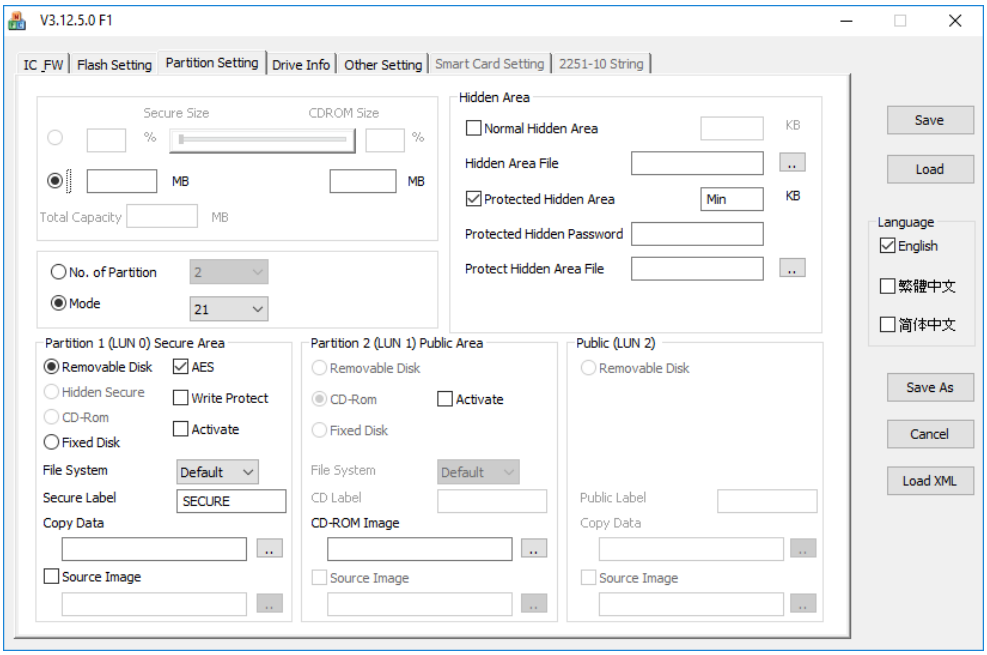

Figure 4.2: Phison firmware configuration tool

The most interesting part of the configuration tool is partition settings tab seen in figure 4.2 (the rest is about flash NAND used, device name and IDs and how the LED should behave). Here we can see that the controller supports up to 3 partitions which can be presented to the system as a removable drive, a

CD-ROM or a hard drive (the grayed-out options are disabled because of the selected mode, not because the controller does not support them). The first partition can be encrypted with AES. We can also choose data to be preloaded onto the partitions. Finally, we can configure the Normal Hidden Area and the Protected Hidden Area which seems to be protected with a password.

According to a patent awarded to Phison[24], the Hidden Area is a part of the flash the host computer will not be allowed to access by the controller. Combined with the fact that the controller seems to have no writable non-volatile on-board memory, the Hidden Area (and the Secure Hidden Area especially) seems like a good place for cryptographic secrets to live.

Chapter 5

Control software analysis

In this chapter I will analyze the control software supplied with Kingston DataTraveler Vault Privacy. The goal of this analysis is to identify ways to (in order of desirability):

- retrieve the password or encryption key
- dump the contents of the NAND flash for purposes of offline cracking and obtaining the firmware
- reset or bypass the wrong password counter

5.1 Challenges

A huge setback to all my efforts was the fact that we were only able to source one testing unit. Being released in 2009, the drive had not been for sale in shops for some years. I searched high and low, combing through Amazon, eBay and even Craigslist. There were some offers, but the new drives were prohibitively expensive (by then more or less only the 64 GB and 32 GB models were left for sale new and those always were hideously expensive). Even though not ideal for research as they theoretically could have been tampered with, I would have settled for second-hand drives, but none of the sellers were willing to ship internationally. I even tried asking on reddit, offering to pay cash or exchange the drive for the updated USB 3.0 model. I did have a couple takers initially, but when it came down to business, they backed out citing privacy concerns.

This meant that I had to very carefully consider the risks involved with performing all actions and avoid ones deemed too dangerous as damaging the drive would prevent further testing.

5.2 Choosing a target

As stated earlier, the flash drive comes with three control applications—one for Windows, one for Mac OS X and one for Linux.

The Windows application is a single 1 MiB GUI executable built on top of the Microsoft Foundation Class Library (MFC). No form of obfuscation seems to be used. There are over 3000 functions, but most of them are MFC library code. Analysis of this application would be possible, only made a bit tedious by the presence of the extraneous GUI handling code.

The Max OS X application is a fat binary containing Mach-O executables compiled for the x86 and PowerPC architectures. The application is written in Objective-C. Class and method names are present, which makes it an interesting target, however the fact that I do not own a Mac OS X computer means I would be limited to static analysis of the code. That combined with my limited experience reverse engineering Mac applications led to me ruling out the Mac application as a candidate for analysis.

Finally there is the Linux application. Or rather applications—there are five separate command line utilities for performing different tasks. When I loaded one of the binaries into IDA, I got a very nice surprise—Kingston did not strip the binaries. That means I have the name and prototype of every function in these binaries. On top of that, Linux distributions are free and readily available, making dynamic analysis possible. This, combined with the fact that there is virtually no superfluous code for handling the user interface, makes the Linux utilities the perfect target for analysis. They were written in C++ and compiled for the x86 and x86-64 architectures. In my analysis I am going to use the 32 bit x86 variants.

5.3 Analysis setup

To analyze the binaries I decided to use IDA Pro 7 with the Hex-Rays Decompiler plugin. However I only have access to a license for the Windows version, so to facilitate dynamic analysis of the Linux utilities I set up a virtual machine running Ubuntu 16.04 and used IDA's remote debugging feature. To allow the utilities to interact with the flash drive, I redirected one of the USB ports on my PC to the virtual machine.

5.4 Static analysis

There are five command line utilities in total:

- dtvp_about
- dtvp_forgotpassword

- dtvp_initialize
- dtvp_login
- dtvp_logout

Looking at symbols present in the binaries with objdump suggests they all share the exact same codebase. The file and function names are identical across all five binaries, suggesting only the main function was ever changed. Comparing the binaries with zynamics' BinDiff reveals all the functions with the exception of main are indeed identical, confirming this hypothesis. It would seem Kingston did not even bother changing the main source file name—it is "DTVP_Login.cpp" in all five tools. Following is a list of all source file names:

- crtstuff.c
- DTVP_Login.cpp
- KT_SDK.cpp
- Rijndael.cpp
- FunctionalityLayer.cpp
- HashSDK_PWS.cpp
- integer.cpp
- nbtheory.cpp
- RSA.cpp
- SCipher.cpp
- sha2.cpp
- rand.cpp
- CommandLayer.cpp
- hashFFour_SDK_PWS.cpp
- hashFOne_SDK_PWS.cpp
- hashFThree_SDK_PWS.cpp
- hashFTwo_SDK_PWS.cpp
- TransportLayer.cpp

Many of the files seem to be dealing with standard cryptographic functions and can be safely ignored. DTVP_Login.cpp seems to only contain the main function.

5.5 Dynamic analysis

While a big portion of the analysis could be performed statically, I prefer dynamic analysis, so I performed the majority of the analysis with the help of a debugger. In this section I am going to go over tracing through the normal execution flow of select binaries (specifically dtvp_login and dtvp_forgotpassword), trying to call some functions with my own arguments and finally examining some functions not used by any of the five binaries and calling some of them.

5.5.1 Tracing dtvp_login

First I wanted to know how the user's password was used for authentication—whether it was verified by the application itself (least secure), whether an encryption key was derived from it by the application and sent to the flash drive, or whether the password itself was sent over (most secure)—so I began my analysis with dtvp_login.

The program is rather straight-forward: It starts by calling the KTInitAPI function, collects the password from the user and calls the KTLogin function, checking for and reporting errors along the way.

KTInitAPI looks for the flash drive by iterating over all the /dev/sg* devices (backed by the Linux SCSI General driver), sending SCSI inquiry command to each using the ioctl function (in fact, that is how all the SCSI communication is handled). If the returned device type is 5 and the area designated for vendor specific data (offset 0x24) contains either the string "PMAP1234" or "FFAP1234" the device is selected. Then the serial number and version are retrieved and finally the ReadInfo command (a vendor specific SCSI command 0xc6) is issued to determine whether the device is locked or unlocked.

KTLogin begins by setting up a Secure Session. The device and control application first authenticate each other by exchanging public RSA-512 keys and issuing a challenge to each other. Curiously, the application does not compare the public key provided by the device to a known value (or verify it in any other way) which lead me to wonder whether the device checks the application's public key. To test this, I patched the program with my own RSA-512 key and ran it again. This time the Secure Session failed to initialize with the device returning error code 0x53, indicating it does verify the application's RSA key. All communication within the Secure Session is encrypted with AES-128 (which explains the difficulties I experienced during my initial attempts to monitor the communication with Wireshark). Finally the password is sent unaltered (only encrypted by the Secure Session) to the device and the Secure Session is terminated.

The program ends by displaying a status message indicating the result of KTLogin—success (0), wrong password (0x38), wrong password and retry limit reached (0x39) or other.

5.5.2 Tracing dtvp_forgotpassword

Next I wanted to know how a password is set and whether it would be possible to change it without losing the data stored on the drive. To that end I needed to analyze a utility used to set the password, that means either dtvp_initialize or dtvp_forgotpassword. I had previously initialized my drive and since dtvp_initialize does not allow users to re-initialize an already initialized drive, the choice was rather simple. Also, since—as an attacker—I would not know the password, dtvp_forgotpassword seemed like the more fitting choice anyway.

Once again, the program is fairly straight-forward: It starts by calling the KTInitAPI function, warns the user that proceeding will result in loss of all data on the drive, collects the new password and calls KTSetCredentials.

KTSetCredentials starts off by calling CS03::PRA__Configure_Private_Area which creates a Secure Session and sends the new password to the drive. At this point the drive is unlocked. Next a FAT32 partition is created (there is also support for FAT16 partitions for drives up to 2 GiB in size, however the smallest Kingston DTVP sold has capacity of 4 GB, so this function is never used). Finally KTLogout and KTLogin are called, presumably to force the OS to mount the newly created partition.

It is possible to skip the new partition creation process using a debugger, however that only results in garbled data where the partition header should be, presumably because the encryption key has changed.

I should note that there is a bug in this utility which results in the new partition header being invalid, however the rest of it works as intended.

5.5.3 Dtvp_about, dtvp_initialize and dtvp_logout

The remaining three utilities—dtvp_about, dtvp_initialize and dtvp_logout—turned out to be not very interesting.

Dtvp_about simply calls KTInitiAPI, followed by KTGetDeviceInfo (something all five utilities do) and displays the returned information (serial number of the drive and firmware version). The only information unique to this utility is the version number of the Linux utilities—2.0.0.2.

Dtvp_initialize is almost identical to dtvp_forgotpassword in terms of functionality. The only functional difference is that dtvp_initialize only works on uninitialized drives, whereas dtvp_forgotpassword only works on initialized drives.

Dtvp_logout calls KTInitAPI and then KTLogout. The only thing notable about this utility is how badly it is written. KTLogout still takes a credentials structure as an argument, although the password is completely unused. Not only does main explicitly check for error codes that should not be possible (e.g. 0x38—wrong password), but the password is copied into the credentials structure from an uninitialized buffer.

5.5.4 Dumping the Protected Hidden Area

As mentioned in chapter 4, the drive should contain a 128 kiB Protected Hidden Area. All the utilities (with the exception of dtvp_about) read from the Protected Hidden Area, but they only ever access three pages: 9, 0xa and 0xc. Page 9 contains device configuration, page 0xa contains copyright notice ("Copyright (c) 2013 Kingston Technology Company, Inc.") and a support page address ("http://www.kingston.com/support/") and page 0xc contains the password hint put in by the user. Naturally, I was wondering what is stored in all the other pages.

The Protected Hidden Area is split into 512 byte pages. Individual pages are read by the KTReadHiddenPage function. KTReadHiddenPage first reads the page and then decrypts it with AES, using the following key:

```
50 44 4D 47 FE 87 31 30 B9 61 88 45 02 C6 78 20
```

This key seems to be quite ubiquitous in Kingston products. For example it was also used to decrypt an ISO image in an updater utility for a different encrypted flash drive from Kingston.

The prototype of KTReadHiddenPage is as follows (return type and parameter names filled in by me):

```
int KTReadHiddenPage(char device, ushort page_number,
                     uchar *buffer, ulong buffer_len)
```

I used IDA's Appcall functionality to call KTReadHiddenPage with my own parameters. I wrote a simple debugger script which read all 256 pages (128 kiB) of the Protected Hidden Area and stored their contents into individual files. Sadly, after examining the resultant files it became apparent that only pages 9, 0xa and 0xc contain any data. All the other files only contained a repeating sequence of 16 bytes:

```
57 E7 A8 A1 5B 1F DF 91 35 04 2A 16 19 29 4F 62
```

It turns out that these bytes are the result of decrypting zeros with the aforementioned key.

I also tried fuzzing the parameters to see if the function could be used for reading arbitrary data from the flash. Page numbers higher than 255 (beyond the 128 kiB of the Protected Hidden Area) resulted in an error and providing a larger buffer resulted in the same data being returned, only padded with zeros. I traced execution flow all the way to the SCSI command being sent and it seems the bounds check is being performed by the drive itself.

5.5.5 Unused functions

There are quite a few functions in the Linux binaries which are not used by any of the tools. I picked out a few that I felt were worth looking into:

- CS03::HA__Get_Current_Number_of_Attempts
- CS03::HA__Read_Page
- CS03::HA__Write_Page
- CS03::HA__Read_Secure_Page
- CS03::HA__Write_Secure_Page
- CSCSICommand::ClearPassword
- CSCSICommand::LoginPassword
- CSCSICommand::SetPassword
- CSCSICommand::ReadKeyFromFlash
- CSCSICommand::WriteKeyToFlash
- CSCSICommand::ReadKeyFromSRAM
- CSCSICommand::WriteKeyToSRAM
- KTChangeCredentials
- KTUnblockCredentials
- KTReadBlockRaw
- KTWriteBlockRaw

CS03::HA__Get_Current_Number_of_Attempts seemed interesting to me because its analysis could reveal the location of the password try counter and perhaps even a way of overwriting it. Unfortunately, to get this information it uses a vendor specific SCSI command, which does not reveal the counter's location. An undocumented write variant of the command could potentially exist, but if it does, its structure is not obvious.

CS03::HA__Read_Page and CS03::HA__Write_Page seemed interesting because they share the same prefix ("HA__", which I am guessing could stand for Hidden Area) with the attempt count retrieval function and could therefore potentially provide a different way of reaching the counter. CS03::HA__Write_Page is completely unused while CS03::HA__Read_Page is only used by KTGetHiddenSN which itself is unused and doesn't provide much insight into how the function is supposed to be used (most of the parameters—including the ones I presumed to be buffer size and page number—are zero). I tried calling the function and fuzzing the parameters, but to no avail.

CS03::HA__Read_Secure_Page and CS03::HA__Write_Secure_Page piqued my interested because some sort of a secure area sounds like a nice place

to store passwords and keys. Once again, these functions are not used at all in the five utilities. Despite my best efforts I could not get this function to work. No matter what I tried it always returned an error. Perhaps I didn't identify the parameters correctly or maybe it requires some form of authentication (the Phison flash controller datasheets do mention a host with "Trusted host ID" being able to access the secure area).

CSCSICommand::ReadKeyFromFlash sounds like exactly the API I am looking for. Unfortunately, it only retrieved 5 bytes of data.

CSCSICommand::ReadKeyFromSRAM did retrieve 32 bytes of data, which is the correct length for an AES-256 key, the data doesn't look random at all and doesn't change between drive resets. For what it's worth, here it is:

```
12 01 00 02 00 00 00 40 51 09 0D 16 10 01 01 02
03 01 04 03 09 04 FF FF FF FF FF FF FF FF FF FF
```

KTChangeCredentials takes two passwords—current and new. It seems to require the correct current password in order for it to work, and both passwords are directly passed to the device, so it wasn't much use to me. I believe this is the function used by the Windows application to change the password without loss of data (a feature the Linux utilities lack).

I hoped KTUnblockCredentials might be used to reset the retry counter. Frustratingly, it requires the correct password to work. Internally it calls a function called CS03::PRA__Make_SecureDisk_Public (not used by anything else) which encrypts the password and passes it to the device. I got it to work (return 0), but I have no idea what it actually does.

KTReadBlockRaw and TWriteBlockRaw start by calling CS03::GetHiddenAreaSize. Since there is no Normal Hidden Area on this device (it is configured to have size 0), the function immediately returns. If the device had a Normal Hidden Area, next the CS03::ReadHiddenArea or CS03::WriteHiddenArea function would be called. I tried calling CS03::ReadHiddenArea anyway with various parameters, but unsurprisingly it always failed.

Out of fear of bricking the drive, I did not attempt to call functions that could write potentially invalid values to the flash, namely CSCSICommand::ClearPassword, CSCSICommand::LoginPassword and CSCSICommand::SetPassword.

5.6 Summary

Despite the oversight of leaving in the debug symbols, the software is quite secure. There seems to be no way to directly retrieve either the password or the encryption key. All the tested APIs verify the supplied credentials correctly. There are no APIs for dumping the NAND flash or resetting the password try counter. I tried to get to this data by out of bounds access through other

APIs, but in all the tested APIs the bounds checks are performed by the flash drive and seem to be implemented correctly.

I am a bit perplexed by the implementation of the Secure Session. The short RSA key aside (I do not think RSA-512 was a good idea even back in 2009 and it certainly did not age well [25]), the fact that both the application's public and private keys are hard-coded in the binary, combined with the fact that the application does not verify the device's public key, makes the authentication process unreliable since an attacker can easily extract the private key needed to communicate with the flash drive from the binary. Encrypting the communication between the host system and the drive is certainly a good idea, but the implementation of the authentication process is inadequate.

The stark difference in code quality between the main functions and the low level functions (cryptography, SCSI, etc.), together with many of the source file names containing "SDK" suggests multiple developers. Presumably, a large portion of the code comes directly from Phison and the individual flash drive vendors for the most part only develop the user interface (this would explain why several vendors were affected in the 2009 SySS hack).

Chapter 6

Relation between the password and the encryption key

In this chapter I explore the relation between the user's password and the key actually used to encrypt and decrypt data stored on the drive. Primarily I am interested in whether the key is derived from the password, randomly generated or fixed. I attempt hardware-based approaches as well as software-based black box testing.

6.1 The NAND flash interface

The NAND chip used in my testing unit is MT29F64G08CFACA in a TSOP-48 package. According to its datasheet [21] it is an Open NAND Flash Interface (ONFI) 2.2 compatible chip with two dies (referred to as LUNs—logical units—by the ONFI standard) using asynchronous I/O. The ONFI is a unified interface for NAND flash chips designed by a working group composed of dozens of companies manufacturing and using NAND flash memory for the purposes of compatibility and ease of integration [26]. The ONFI 2.2 specification [27] is freely available from the ONFI website. The pinout of the NAND flash is described in table 6.1.

6.2 Logic analyzer

The most reliable way to determine whether the encryption key used changes with the user's password would be to write and read the same data before and after a password change or a drive reset and monitor the data actually being written to the flash. My first naive attempt was using a logic analyzer. I used a generic Chinese 16 channel logic analyzer. To record and analyze the data I used the Sigrok software suite with the PulseView GUI. I could not find a plugin for ONFI decoding, so I would have to decode the data manually using

Table 6.1: Pinout of MT29F64G08CFACA

TSOP-48 pin number	Pin name	Description
17	ALE	Address Latch Enable. Loads data on the bus into the address register.
9	CE	Chip enable for the 1st die.
10	CE2	Chip enable for the 2nd die.
16	CLE	Command Latch Enable. Loads data on the bus into the command register.
29	DQ0	Data input/output bus. Bit 0.
30	DQ1	Data input/output bus. Bit 1.
31	DQ2	Data input/output bus. Bit 2.
32	DQ3	Data input/output bus. Bit 3.
41	DQ4	Data input/output bus. Bit 4.
42	DQ5	Data input/output bus. Bit 5.
43	DQ6	Data input/output bus. Bit 6.
44	DQ7	Data input/output bus. Bit 7.
8	RE	Read Enable. Transfers serial data from flash to host.
18	WE	Write Enable. Transfers commands, addresses and serial data from host to flash.
7	R/B	Ready/busy for the 1st die (LUN). Pulled low when the 1st die is busy.
6	R/B2	Ready/busy for the 2nd die (LUN). Pulled low when the 2nd die is busy.
19	WP	Write Protect. Disables program and erase operations. Active low.
12, 37	V_{CC}	Core power supply (3.3 V).
13, 36	V_{SS}	Core ground (0 V).

the ONFI 2.2 specification. I considered writing my own decoder plugin for PulseView, but I quickly realized just how extensive the ONFI protocol is and decided against it.

There are 18 signals in total (see table 6.1), but my logic analyzer only had 16 channels, so I had to omit at least two signals. I decided to omit the R/B and R/B2 signals as they only serve as an indication to the host that the device is ready to receive new commands and therefore should not be essential to decoding the commands and data. I discovered that the pads on the unpopulated flash chip footprint on the other side of the board (see figure **??**) were electrically connected with the pads on the front side. All the signals were in the same positions except the two pairs of chip enable and read/busy signals which were swapped. This allowed me to solder on wires for my logic

analyzer with minimal risk of damaging the flash chip on the other side of the board.

The first experiment I planned was to write 8 kiB (one page) of zeros to a fixed 8 kiB aligned address using the dd utility, reset the drive, write zeros to the same address again and compare the recorded data. After recording the first trace I was overwhelmed by the sheer amount of data. I was anticipating a simple program command, but I had neglected to consider the overhead of the file system. I tried to get around this problem by only looking at the program commands, but there still were many more than I had anticipated. Even more frustrating was the fact that even without changing the password or resetting the device I could not get two traces that would look remotely similar. This was probably in part caused by Ubuntu's implementation of the FAT32 driver. I believe the data was not actually written to the drive as a single 8 kiB block, but rather in smaller pieces. Since, according to the Kingston DTVP datasheet **??**, the flash drive is using AES in cipher block chaining mode, a change in the middle of a page would cause all the data following it to change as well (this would also explain why the drive's rated write speed is so much lower than the read speed).

Faced with these issues, I decided not to pursue this approach any further unless no satisfactory alternative could be found.

6.3 Black-box testing

Black-box testing is a method of examining the behavior of a system without peering into its internal structures and workings. I noticed that the password change function only took a few seconds to complete. If the encryption key were changed by this operation, the whole drive would have to be re-encrypted, which would be a very lengthy process. This alone meant that the encryption key could not be derived solely from the user's password. Similarly, the drive reset operation—which supposedly destroys all data on the drive—also only took a few seconds. That meant it could not be actually overwriting the data. Presumably, it was only changing the encryption key.

I tried opening the mounted protected partition in a hex editor and discovered it was full of large blocks of a repeating 16 byte sequences (see figure 6.1). I assumed this was the default value stored in the flash decrypted with the current key. If this assumption were correct, it would mean I have a way to observe a key change.

To test this hypothesis and to examine the relation between the password and the encryption key, I picked one of these blocks of repeating data and wrote down the address. Then I performed a series of experiments on the drive and checked the data at this address after each one.

1. I tried simply disconnecting and reconnecting the drive. After this operation the data was unchanged.

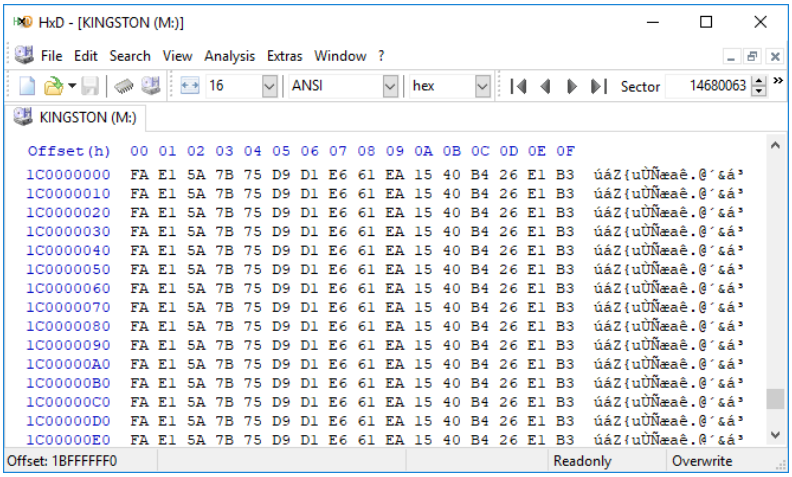

Figure 6.1: Free space decrypted with the current key

2. I reset the drive, choosing a different password. The data changed after this operation.

3. I changed the password. After this operation the data was unchanged.

4. I reset the drive, using the current password (i.e. the password was the same before and after the reset). After this operation the data did change.

From the fact that the pattern remained constant across power cycles, but did change after drive reset, I conclude this method is a valid way of detecting an encryption key change.

From the fact that the encryption key did change after resetting the drive with the same password, I conclude that the encryption key is not solely dependent on the password and is at leas partially randomly generated. The presence of a hardware random number generator (see figure 4.1) supports this conclusion.

Finally, from the fact that the encryption key did not change after a password change (without drive reset), I conclude it is either completely independent of the password and stored alongside the encrypted data, or only partially dependent on the password. An example of a system where the encryption key is dependent on the password, but allows the password to be changed without the need to re-encrypt the data, would be one where a random value is generated and stored alongside the data and the encryption key itself is obtained by adding this value to the password (or a hash thereof) by some reversible operation (e.g. XOR). To change the password in such a system, one simply needs to add the existing password to the stored value (obtaining the encryption key) and then subtract (in case of XOR by simply performing

XOR) the new password and storing the resultant value in place of the original. When the new stored value is added to the new password, the same key is produced. Or in a more comprehensible way:

$$key_1 = rnd_1 + pass_1$$

$$rnd_2 = key_1 - pass2 = rnd_1 + pass_1 - pass_2$$

$$key_2 = rnd_2 + pass_2 = key_1 - pass2 + pass_2 = key_1$$

6.4 Summary

I conclude with high confidence that the encryption key is randomly generated and is either completely independent of the user's password or only partially derived from it. I believe the key or its part to be stored on the NAND flash alongside the encrypted data, but without dumping the flash memory (and possibly analyzing the controller firmware, if it could be obtained), I can not make that claim for certain.

Chapter 7

Attempting to bypass the password try limit with a hardware modification

Since I found no way to bypass or reset the password try counter while analyzing the control software, I wondered if it could be done in hardware.

7.1 Plan

When I was reading through the ONFI standard, one signal caught my attention—write protect. Supposedly, when held low, this signal should prevent all program and erase operations, but still allow read operations. Since I believe the password try counter to be stored in the flash memory, I hoped that forcing this signal low could prevent updating the counter after an unsuccessful login attempt, while still allowing the password verification to take place.

To determine how the WP signal was currently driven and how it could be safely tied to ground, I followed the trace on the PCB (see figure 7.1, WP trace highlighted in red). Since this was my only testing unit, I had to be absolutely certain that any hardware modification I perform would not damage the device.

The trace goes directly from pin 38 of the controller to pin 19 of the NAND chip as well as pin 19 of the unpopulated NAND footprint. This meant that if I were to tie the signal to ground, I would be shorting the output of the controller directly to ground, which would probably lead to irreparable damage the moment the controller tries to bring the signal high. To avoid this I would first have to break the connection between the controller and the flash chip. This could be achieved either by desoldering and lifting the corresponding leg on either the controller or the flash chip or by cutting the trace connecting the two. I decided cutting the trace was the safest option as it avoided the risk of

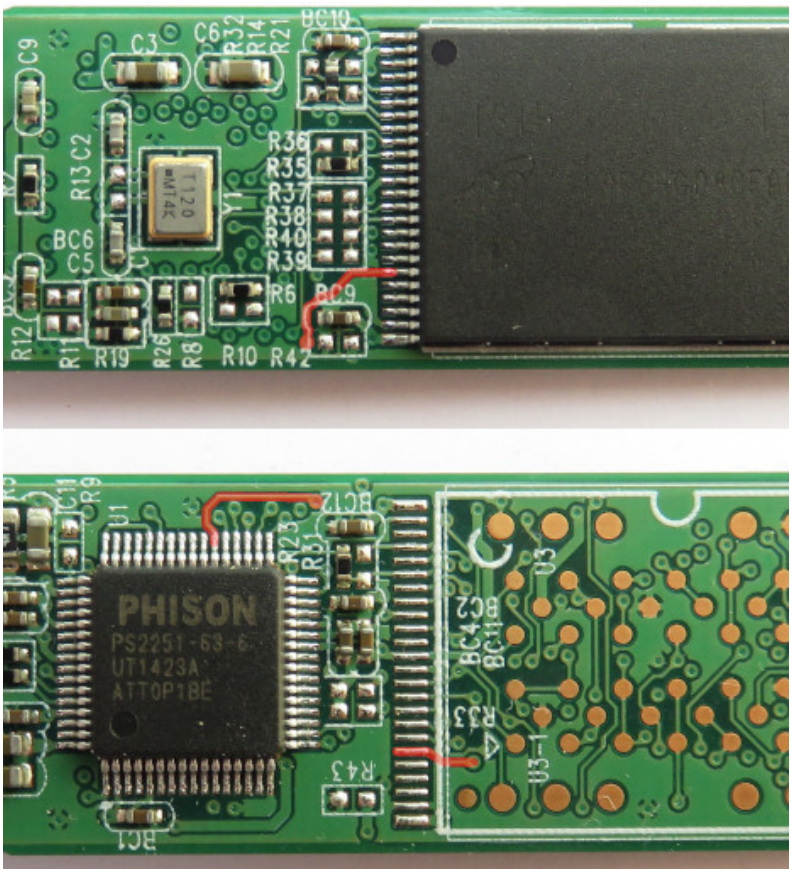

Figure 7.1: Write protect signal trace on the PCB (highlighted in red)

damaging either of the chips with heat. As in the logic analyzer experiments, I would use the unpopulated footprint on the back side of the PCB to access the signal. The closest pad with a solid ground connection was pin 23 of the same footprint. The proposed modification is visualized in figure 7.2.

In case I needed to be able to toggle the write protect signal at will, I could reconnect the cut trace through a high value resistor and connect the write protect signal to ground through a switch. The switch would function as an override. With the switch open, the controller could drive the signal through the resistor normally. Since logic inputs tend to have very high input impedance, the relatively low impedance of the series resistor should have no significant effect on the signal. With the switch closed, pulling the signal to ground, the series resistor would limit the current drawn from the controller (e.g. a 1 kΩ resistor would limit the current to 3.3 mA). This version of the modification is visualized in figure 7.3.

I decided to implement the second version of the modification.

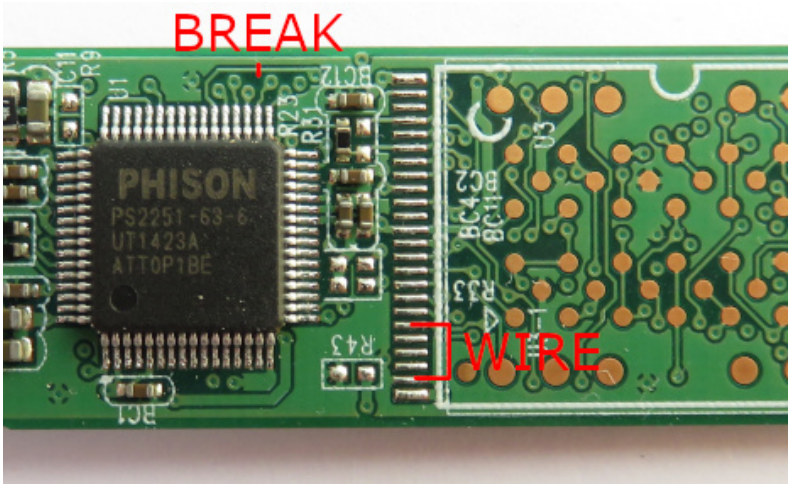

Figure 7.2: Safely tying the write protect pin to ground

Figure 7.3: Safely tying the write protect pin to ground using a switch

7.2 Execution

First I started by carefully cutting the trace and connecting pads 19 and 23 of the unpopulated footprint with a piece of wire as indicated in figure 7.2. At this stage I tried plugging the flash drive into my computer. Unfortunately, in this state the flash drive was not functional. It was picked up by the system and the virtual CD-ROM appeared, but it was inaccessible.

I proceeded with the modification as planned. First I chose an appropriate resistor. Due to space constraints I decided to go for an 0603 package. Out of the 0603 resistors I had on hand I picked a 1.2 kΩ one. I measured where

its ends would land on the PCB and scraped off the solder resist from the cut trace in those places. Because solder connection to such a thin trace would have very low mechanical strength, I secured the resistor to the board with superglue and then soldered both ends. Finally I soldered two wires to pins 19 and 23 of the unpopulated footprint, connected their other ends to a slide switch and secured it to the PCB with hot-melt adhesive. I covered the wires with Kapton tape to prevent snagging. The result can be seen in figure 7.4.

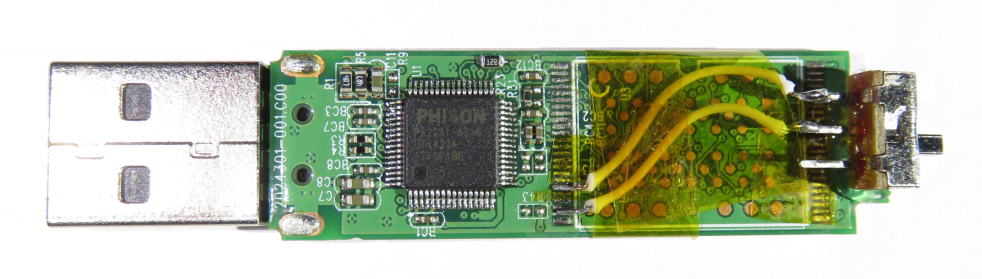

Figure 7.4: Kingston DTVP with the write protect override switch installed

7.3 Result

As mentioned in the previous section, connecting the flash drive with write protect already enabled resulted in the drive not working at all, but now I was able to toggle write protect with ease, so I could try enabling it during various phases of authentication.

First I tried enabling write protect after connecting the drive and running the login utility, but before entering the correct password. When the password was entered, the login utility returned error 0x34 and the LED started flashing rapidly (my best guess is the controller kept trying to write to the flash over and over) and the device became unresponsive. To bring it out of this state, the device had to be power cycled. Next I tried entering an incorrect password and the login utility returned error 0x15 (and the device entered the same unresponsive state as before). I was very excited. If the counter did not decrement, this would mean I had a way to test an unlimited amount of passwords, albeit very slowly. I immediately started trying other passwords. Unfortunately, this turned out to be a fluke. All subsequent attempts with both correct and incorrect passwords returned error 0x34.

As a last resort I tried running the login utility in a debugger, pausing right before the actual SCSI command containing the password is sent and enabling write protect at that point. Sadly, this resulted in the same 0x34 error code. I even tried examining the data returned by the SCSI command, but it was always identical; the validity of the password had no effect on it.

In the end I was not able to get an infinite number of password guesses using this method (although the counter was indeed not decremented).

Conclusion

Even though I ultimately failed to break the encryption of Kingston DataTraveler Vault Privacy, I believe I still achieved the objective of this thesis. My analysis of the control software was very thorough; there were only a handful of APIs I did not dare test out of fear of breaking my only testing unit (and given how robust the rest of the APIs seemed to be, I would be very surprised if testing these remaining APIs lead to any significant discoveries).

I determined with high confidence the relation between the user's password and the encryption key and I also demonstrated the drive is resilient to simple hardware attacks.

Overall Kingston DataTraveler Vault Privacy is a significant improvement in terms of security over the drives it replaced after the 2009 hack. My only recommendations to Kingston for future improvements would be stripping debug symbols from the binaries where possible and implementing the authentication of the Secure Session more securely, since the current implementation relying on an RSA-512 key hard-coded in the binaries is, in my opinion, inadequate.

The only worthwhile avenue I did not explore is dumping the contents of the NAND flash chip. This task is non-trivial to say the least, but it could lead to very interesting discoveries from determining how and where the encryption key and password try counter are stored to extracting and reverse engineering the firmware of the flash controller. I would recommend this subject for future research.

Bibliography

[1] Ponemon Institute, LLC. The State of USB Drive Security in Europe. 2011, [cit. 2018-05-09]. Available from: https://media.kingston.com/pdfs/Ponemon/Ponemon_research_EMEA_summary_UK_1111.pdf

[2] Corel Corporation. WinZip Version History. [cit. 2019-01-09]. Available from: https://www.winzip.com/win/nl/version.html

[3] Pavlov, I. History of the 7-Zip. December 2018, [cit. 2019-01-09]. Available from: https://www.7-zip.org/history.txt

[4] Corel Corporation. Data Protection. [cit. 2019-01-09]. Available from: https://www.winzip.com/en/features/data-protection.html

[5] IDRIX SARL. Introduction. In *VeraCrypt Documentation*, [cit. 2019-01-09]. Available from: https://www.veracrypt.fr/en/Introduction.html

[6] IDRIX SARL. Portable Mode. In *VeraCrypt Documentation*, [cit. 2019-01-09]. Available from: https://www.veracrypt.fr/en/Portable%20Mode.html

[7] IDRIX SARL. Plausible Deniability. In *VeraCrypt Documentation*, [cit. 2019-01-09]. Available from: https://www.veracrypt.fr/en/Plausible%20Deniability.html

[8] Kingston Technology Corporation. *DataTraveler Vault Privace User Manual*. [cit. 2019-01-09]. Available from: http://media.kingston.com/support/downloads/DTVP_userManual_002.pdf

[9] Apricorn, Inc. *Aegis Secure Key 3.0 User's Manual*. August 2017, revision 1.7, [cit. 2019-01-09]. Available from: https://www.apricorn.com/media/document/file//a/s/ask3_manual_online.pdf

[10] Picod, J. and Audebert, R. and Bursztein, E. Attacking Encrypted USB Keys the Hard(ware) Way. December 2017, [cit. 2018-05-09]. Available from: https://www.youtube.com/watch?v=jVKl3GuazEs

[11] Kingston DataTraveler 2000 16 GB. [cit. 2018-05-09]. Available from: https://cdn-reichelt.de/bilder/web/xxl_ws/E910/KINGSTON_DT2000_16GB_02.png

[12] van der Putte, T.; Keuning, J. Biometrical Fingerprint Recognition: Don't get your Fingers Burned. In *Smart Card Research and Advanced Applications*, edited by J. Domingo-Ferrer; D. Chan; A. Watson, Boston, MA: Springer, 2000.

[13] Hama "ProtectionKey" FlashPen. [cit. 2018-05-09]. Available from: https://www.hama.com/bilder/00124/abx/00124197abx.jpg

[14] Picod, J. and Audebert, R. and Blumenstein, S. and Bursztein, E. Attacking encrypted USB keys the hard(ware) way. 2017, [cit. 2018-05-09]. Available from: https://www.blackhat.com/docs/us-17/thursday/us-17-Picod-Attacking-Encrypted-USB-Keys-The-Hard%28ware%29-Way.pdf

[15] SanDisk Corporation. Security Bulletin December 2009. December 2009, [cit. 2018-05-09]. Available from: https://web.archive.org/web/20091220042009/http://www.sandisk.com/business-solutions/enterprise/technical-support/security-bulletin-december-2009

[16] Kingston Technology Corporation. Kingston's Secure USB Drive Information Page. December 2009, [cit. 2018-05-09]. Available from: https://web.archive.org/web/20091224102747/http://www.kingston.com/driveupdate/

[17] Verbatim Americas, LLC. Important Security Update December 2009. December 2009, [cit. 2018-05-09]. Available from: https://web.archive.org/web/20100108171617/http://www.verbatim.com/security/security-update.cfm

[18] Kingston Technology Corporation. Kingston Digital to Replace Affected Secure USB Flash Drives with Upgraded Security Architecture, New Drives. 2010, [cit. 2018-05-09]. Available from: https://www.kingston.com/us/company/press/article/40506

[19] Schmidt, J. NIST-certified USB Flash drives with hardware encryption cracked. *The H*, 2010, [cit. 2018-05-09]. Available from: http://www.h-online.com/security/news/item/NIST-certified-USB-Flash-drives-with-hardware-encryption-cracked-895308.html

[20] Kingston Technology Corporation. *Kingston DataTraveler Vault - Privacy datasheet*. 2013, [cit. 2018-05-09]. Available from: `https://www.kingston.com/datasheets/DTVP_en.pdf`

[21] Micron Technology, Inc. *Micron Technology 64Gb, 128Gb, 256Gb, 512Gb Asynchronous/Synchronous NAND Features*. November 2009, rev. A.

[22] Phison Electronics Corporation. *USB 2.0 Flash Controller Specification PS2231*. October 2007, revision 1.6.

[23] Phison Electronics Corporation. *USB 2.0 Flash Controller Specification PS2251-33*. January 2009, revision 1.2.

[24] File protecting method and system, and memory controller and memory storage apparatus thereof. 2012, United States, [cit. 2018-05-09]. Available from: `http://www.freepatentsonline.com/8954692.html`

[25] Valenta, L.; Cohney, S.; et al. Factoring as a Service. Cryptology ePrint Archive, Report 2015/1000, October 2015. Available from: `https://eprint.iacr.org/2015/1000`

[26] ONFI Workgroup. Why ONFi. [cit. 2019-01-07]. Available from: `https://www.onfi.org/about`

[27] ONFI Workgroup. *Open NAND Flash Interface Specification*. October 2009, revision 2.2.

Appendix A

Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CBC Cipher Block Chaining

FAT File Allocation Table

I/O Input/Output

LED Light-emitting Diode

LUN Logical Unit

MFC Microsoft Foundation Class

MitM Man in the Middle

MLC Multi-level Cell

NDA Non-disclosure Agreement

ONFI Open NAND Flash Interface

PCB Printed Circuit Board

RAM Random Access Memory

RFID Radio-frequency Identification

RNG Random Number Generator

ROM Read-only Memory

RSA Rivest Shamir Adleman

SCSI Small Computer System Interface

SDK Software Development Kit

SLC Single-level Cell

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

Appendix B

Contents of enclosed CD

- `dtvp_cd` ..... DTVP read-only CD partition contents
- `re` ..... directory containing reverse engineering outputs
  - `ha_dump` . directory containing the dump of the Protected Hidden Area
  - `idb` ..... directory containing IDA databases
  - `dtvp_login_rsa_patch` ..... dtvp_login utility patched with a custom RSA-512 key
  - `function.txt` .. complete list of function names and prototypes of the Linux utilities
  - `hw_notes.txt` .. notes pertaining to the hardware analysis of Kingston DTVP
  - `sw_notes.txt` . notes pertaining to the analysis of the Kingston DTVP control software
- `src` ..... source codes directory
  - `thesis` ..... source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format