FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	The use of cryptography in 7-zip
Student:	Josef Hušek
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2019/20

Instructions

Explore the source code of the 7-zip application (https://www.7-zip.org). Describe its SW architecture, its components, and relationships among them.
Analyze the security aspects of 7-zip with respect to the .7z file format:
- the inherent security of the cryptographic algorithms,
- key management,
- and secure design and programming.

Create an encrypted archive and perform an attack on it with the aim to break the encryption.
Discuss your results.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

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

Prague February 11, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

The use of cryptography in 7-zip

Josef Hušek

Department of Computer Systems
Supervisor: Ing. Josef Kokeš

May 16, 2019

Acknowledgements

I would like to first and foremost thank my supervisor Ing. Josef Kokeš for being patient with me, for his incredibly quick email responses and for his willingness to meet and discuss the thesis whenever I asked for it.

Secondly I would like to thank my BI-DPR[1] exercise teacher Ing. Eliška Šestáková who is a great inspiration concerning studies and her enthusiasm to help all the students and pass on her own experiences on thesis writing is matched by few.

Thirdly I would like to thank Mr. Igor Pavlov for creating and maintaining the 7-zip application and releasing its source code under the public domain.

And last but not least I would like to thank my family, my friends and my amazing girlfriend for sticking by me during the writing of this thesis and motivating me when I had trouble to find motivation elsewhere.

[1] university course about bachelor thesis creation

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 May 16, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Hušek, Josef. *The use of cryptography in 7-zip*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce se zaměřuje na použití kryptografie v aplikaci s veřejným zdrojovým kódem jménem 7-zip. 7-zip slouží k ukládání dat do digitálních archivů.

V práci si nejdříve rozebereme jak je 7-zip strukturovaný a jak ho zkompilovat. Poté otestujeme implementaci šifry AES v 7-zip tak, že její výstupy porovnáme s výstupy z knihovny OpenSSL. Dále se zaměříme především na použitou key-derivation-function (funkce-pro-odvození-klíče) která na základě uživatelského hesla tvoří klíče pro AES. Zjistíme, že tato funkce je před kompilací značně přizpůsobitelná, jenže dekódovací část 7-zipu podporuje i dost slabé varianty. Kvůli tomu by bylo možné sestavit 7-zip, který by měl naschvál velmi oslabené šifrování, nicméně ním produkované archivy by stále byly korektní a zpracovatelné běžnou instalací 7-zipu. Belo by ale mnohem snažší jejich šifrování prolomit.

Následně předvedeme jak vlastně takový útok hádající hesla od archivů vypadá, s pomocí další opensource aplikace jménem hashcat. Nakonec ještě sepíšeme pár kuriozit a vlastností, kterých jsme si povšimli během naší analýzy, a které by se za určitých situací mohly projevit jako problematické z hlediska bezpečnosti. Práci zakončíme shrnutím a několika návrhy na další analýzu v rámci 7-zipu.

Klíčová slova 7-zip, AES, funkce-pro-odvození-klíče, heslo, KDF, kryptografie, archiv, 7z

Abstract

This thesis focuses on the use of cryptography in the open source file archiver 7-zip.

We first discuss a bit about how 7-zip is structured and how it is compiled. We then take a look at the included AES implementation and compare its outputs with outputs from the OpenSSL library. After that we mostly focus on the key-dervation-function which transforms user-supplied passwords into AES keys. We find that the key-dervation-function is customizable before compilation, however the decoding part of 7-zip supports even very weak variations. This means a purposefully weak 7-zip build would still produce valid archives – only they would be much easier to crack.

After that we demonstrate how password guessing attacks take place with the help of another open source application called hashcat. Finally we list a few interesting curiosities and properties we noticed along the way, which may or may not prove problematic from a security perspective. We conclude the thesis by a summary and suggestions for future exploration.

Keywords 7-zip, AES, key-derivation-function, password, KDF, cryptography, archive, 7z

Contents

List of Figures

Introduction and goals

This bachelor thesis is about the 7-zip application. 7-zip is a file archiver – a program used for creating digital archives which enables easy data storage. These archives are usually compressed and can also be encrypted for storage of sensitive or secret data.

We picked 7-zip specifically because it is open-source and therefore can be analyzed reasonably well without resorting to reverse-engineering. Moreover, it is very widely used – not only as the actual application, where people install 7-zip on their PC and use it to store, backup or encrypt their data but also as part of other applications and programs. This is achieved either by using pieces of 7-zip code directly in another program or by using the compiled underlying `7z.dll` library.

We aim to analyze the use of cryptography in 7-zip. We will pursue several goals – inspect the software architecture and internal workings of the application, analyze the implementation of the cryptographic and related algorithms and also compare the results of these algorithms to an existing widely accepted solution – the OpenSSL library. Furthermore, we will inspect how the application actually uses the user-supplied password for encrypted archive creation.

Finally, we will use 7-zip to create several encrypted archives and we will use external cracking tools in an attempt to break the encryption. We will conclude the thesis by discussing the results of our analysis.

On the other hand, we do not aim to try and break AES itself or any other theoretical underlying cryptographic algorithms.

Chapter 1

Theoretical background

In this chapter, we will first discuss the need to store digital data either as-is or in encrypted form. After that we will steer our attention towards the 7-zip application – we will talk about its uses and a bit about its history.

1.1 Data storage and encryption

People need to store ever-increasing quantities of digital data, this has been a trend for more than a decade. The hardware used for such storage also increases in its capacity following this trend. Lately, people also started using "cloud" services, where they do not store their digital data locally but on a remote server.

To be able to store or send data which may be really sizable or contain a large number of files we started using file archivers. These are applications which can take many input files and create either one or several digital archives out of them. A usual feature of these is also compression, where we can make use of specialized algorithms and compress the digital data to save space.

We recognize two main types of compression: lossy and lossless. Lossy compression is used almost exclusively with multimedia where we can trade smaller digital size of the data for image quality, sound quality, etc. On the other hand, lossless compression shrinks the supplied data without actually losing any contained information. Of course, such operation has limitations – you can't compress the data indefinitely and there usually is a trade-off between having a smaller compressed file but needing more resources and/or time during the compression/decompression.

Except for compression, another use-case of archivers soon emerged. Sometimes people want to store their data in an encrypted form – it might be company documents, personal files or anything else, the goal is the same – store the data in such a way that only you can read it, even if another person has physical access to the hardware medium that they are stored on. The most

common solution to this problem is using encryption – we can use a symmetric cipher to encrypt the data using a key that only we know. Without the key, the encrypted data is not readable.

Specifically nowadays, the cipher used will often be AES – Advanced Encryption Standard, standardized by the Federal Information Processing Standards [1]. AES is a symmetric block cipher with a key length of either 128, 192 or 256 bits [2]. Usually, a block cipher, such as AES, is not used in its raw form (ECB – or Electronic Codebook – mode of operation) but rather in one of the other modes of operation which often enable them to behave like stream ciphers in some sense and also provide them with better properties [3]. The modes used in 7-zip are CBC (Cipher-Block-Chaining) when handling encrypted `.7z` files and CTR (Counter) when handling AES-encrypted `.zip` files – more on this in Section 2.3.

Depending on the key length we want to use and the specific use-case, we often need a Key Derivation Function (KDF). This is a function that will calculate the secret key to be used in the cipher (AES in our case) usually based on some input. We are interested in password-based key-derivation-functions (PBKDF). These take a user-supplied password and process them in a well-defined way to produce a key of the correct length.

This is the most important part of a KDF however it is not the only one – a good KDF will also ensure that the key has some "good" properties – what properties are "good" might differ between ciphers and use cases. We are mainly interested in "key stretching" – this is a technique that enhances the encryption in a certain sense. The basic premise is that human-made password or passphrases are often easily guessable, so we introduce the PBKDF and we make the transformation from password to key somewhat computationally slow – that way we can limit the number of possible password guesses per second in a brute-force attack.

In a brute-force attack, we simply try to guess the password one by one. Trying all possible combinations of a certain length and a certain character set ensures that we will eventually find the correct password, provided that the number of tries isn't somehow limited and that the password actually falls within this search-space. To further make this attack as inefficient as possible for the attackers, the way the guessed password is verified also shouldn't be as fast as possible but rather quite slow. Finally, of course, the password itself shouldn't be too simple or too short.

There are more elaborate versions of the brute-force attack, such as the dictionary attack. Instead of trying every possible password we only try a certain subset, usually supplied via a simple text file containing the passwords to test – a "dictionary" (hence the name). These dictionaries can further be generated by specialized programs or combined together, etc.

Depending on how we actually receive the information whether our password guess is correct there might be more nuances to this. For example, in the case that somewhere along the way a one-way function – usually a hashing

function – is used, we can calculate large lookup tables containing all possible passwords (of a certain length and character set) as the input into this function. Such tables can either be literal one-to-one (as in one entry per password and its hash) lookup tables or the more elaborate "rainbow tables" invented by Philippe Oechslin[2].

If we know the hash of the correct password we can just use these tables to look the password up, instead of using the brute-force attack. It's essentially a time/space trade-off – the tables still have to be calculated however after that is done, they could be distributed and people can use them to reverse the given function much faster than going through the possibilities again. And indeed this has been done with some widely used hashing functions and there are tools readily available online to be used for this purpose[3].

This type of attack is countered by a technique called "adding cryptographic salt". This usually just means that we append the input for the hash function with a randomly generated[4] sequence of a certain length (this sequence is what is referred to as the "salt")) and then we save this sequence in plain text to later be used again during the password verification. Since the salt will always be different it becomes computationally (both time-wise and storage-wise) virtually impossible to pre-calculate the lookup tables.

1.2 7-zip

One example of a file archiver is 7-zip [4]. It is a multiplatform open-source application originally released in 1999 and written and maintained by a Russian developer Igor Pavlov [4]. Since then the application has gone through many changes and the version that we will focus on in this thesis is 18.05.

7-zip supports a multitude of archive formats. Each of these formats supports at least one "filter" to choose from, these filters are mostly compression methods and encryption methods. The way this is implemented makes it possible to later add more custom filters. Among the supported archive types in 7-zip are the widely used `.zip` format, its native `.7z` format, a proprietary `.rar` format (decompression only) and more. We will focus on the `.7z` archive format since its native to the 7-zip application and it is the format of choice when using the default options for archive creation.

During the writing of this thesis, two new versions of 7-zip were released – 18.06 and 19.00 [5]. There weren't any changes concerning security in 18.06, but there was a major security update in 19.00, which addresses one security

[2]Check out `https://link.springer.com/chapter/10.1007/978-3-540-45146-4_36` for more information on rainbow tables

[3]For example, `https://crackstation.net/`, `https://www.onlinehashcrack.com/`, `https://hashkiller.co.uk/`, etc.

[4]preferably using a Cryptographically secure pseudorandom number generator (CSPRNG)

problem that is present in 18.05 (and 18.06 and prior versions) – we will discuss this problem in Section 2.3.

There have been some security problems with 7-zip before. The full list of published vulnerabilities is available in the CVE (Common Vulnerabilities and Exposures) database[5].

A notable one is, for example, the vulnerability concerning `.rar` archives decompression 7-zip 18.03 (and earlier). The state of internal decoding objects can be largely uninitialized and one could then create specially crafted `.rar` archives which would cause the usage of uninitialized memory leading either to a crash or possibly a code execution attack. The vulnerability was introduced on an internet blog concerning computer security by "ladave" (authors real name is David, the surname is unknown) [6].

[5] `https://www.cvedetails.com/vulnerability-list/vendor_id-9220/7-zip.html`

Chapter 2

Analysis of 7-zip

In this chapter, we will focus on our own analysis of 7-zip, with emphasis on the use of cryptography. We mostly analyze version 18.05 of 7-zip – it is as of now out of date, but its source code can still be downloaded from the 7-zip SourceForge page along with most of the released versions [7]. Throughout this chapter we will be making references to the 18.05 source code – it will be available on the attached SD card in the directory `Sources\7z1805-src`, but if the reader doesn't have access to it, we encourage them to download the source code. Even though we will also place certain important outtakes and snippets directly into this thesis. Figure 2.1 shows the contents of the extracted directory. We will consider this directory to be our root in this chapter, when we talk about specific files in the source code (i.e., `CPP\7zip\Crypto\7zAes.cpp`) unless stated otherwise.

Inside the root directory, there is a subdirectory named `myLogs` which is not a part of the 7-zip source code. It contains a tool for simple thread-safe logging to files, which we wrote to help us during the analysis of 7-zip. We included it since it might prove useful if the reader wishes to conduct their own tests or debugging tasks on the 7-zip source code. The tool consists of the header file `myLogger.h` which needs to be included within the files in which we want to use the logging feature and the actual implementation in `myLogger.cpp`.

The directory to which the logs will be saved can be set in `myLogger.h`

Figure 2.1: The source code root directory

```
├─ Asm ..... (The optimized algorithms written in assembler)
├─ C ..... (The source code of the low-level parts of 7-zip, and more)
├─ CPP ..... (The C++ part of the source code)
├─ DOC ..... (7-zip's documentation)
└─ myLogs ..... (Not part of 7-zip. See chapter 2)
```

by redefining the preprocessor macro `PATH_TO_LOGS` on line 15. Finally, there is a file called `StdAfx.h` which is needed because of the precompiled headers when using the logger with 7-zip – it is the same `StdAfx.h` which is included in all 7-zip source files.

To use the logger in 7-zip we need to modify the `CPP\7zip\7zip.mak` file – this is a file that is part of the 7-zip makefile structure and is used in building all the actual binaries (both `.dll` files and `.exe` files). We just need to add

```
\$O\myLogger.obj
```

after the second row of the original file and add

```
../../../../myLogs.cpp$O.obj::
   $(COMPLB)
```

after the 176th row.

This will make sure that when building the binaries our logger gets built and linked to them as well. We will learn how to compile 7-zip in the following section.

The actual usage of the logger is as follows: in the `myLogger.h` file there is preprocessor macro `LOG(name, msg)` – this is what we will use for the logging (no semicolon is needed after the macro invocation).

The first parameter (`name`) specifies the name of the logging file without the path and without the extension (the `.log` extension is appended automatically). The file name is expected to be given as a `std::string` or a value that can be implicitly converted to it (i.e. a C string literal). Each file is treated as a separate thread-safe logger. This means each time the `LOG` macro is called on the same file name we can be sure that the order of the messages written in it is well defined and the messages won't be overlapping. When the log file is first opened it is truncated – when re-running the application and logging to the same file the old log gets deleted.

The second parameter (`msg`) is the actual message that we want to log. The message is automatically prepended by a number indicating the order of the messages (beginning with a 0) and appended by a newline. It is expected to be given as a `std::string` or a `std::wstring` or any values that can be implicitly converted to one of these (for `std::wstring` the literal can be written as `L"the text"` in C++).

Finally, to gracefully close the logging files and delete their structures from memory we should place the preprocessor macro `CLEAR_LOGGERS` (also defined in `myLogger.h`) somewhere in the code where it will surely be called (not called as in `CLEAR_LOGGERS()` – the macro is defined without the parentheses). In the case of the console version of 7-zip this will probably mean placing it in `CPP\7zip\UI\Console\MainAr.cpp` in the `main` function before every possible

`return` statement, to make sure it gets called even when exceptions arise and the `catch` clauses are invoked.

2.1 Software architecture

The compiled 7-zip application normally consists of several components. The `7z.dll` library which provides the actual functionality of compression and decompression, encryption and decryption, etc. This library can of course also be used by other applications and programs than 7-zip.

Then there is `7z.exe` which is the console version of 7-zip – it can be used from the terminal and accepts command line parameters through which the action (add to archive, extract from archive, etc.) and different options can be specified. We will mainly be using this executable in this thesis. An alternative is the `7zG.exe` which also accepts parameters but the output is graphical. It is mostly used either by the shell extension (using 7-zip from mouse right-click in Windows Explorer) or by the third executable which is `7zFM.exe`. This is a built-in file manager which can be used instead of the Windows Explorer (or a different shell when running a different OS).

Finally, there is 7-zip.dll (and 7-zip32.dll when 64-bit 7-zip is installed). This is used by the shell extension (for usage of 7-zip from Windows Explorer – we get several 7-zip commands in the mouse right-click menu).

What we described is the default installation but 7-zip is very modular and can be compiled in different ways. You can, for example, compile a standalone `7z.exe` containing all the functionality of `7z.dll` or compile a `7z.dll` that is reduced in size and supports only specific archive formats etc.

7-zip is written in both pure C and C++ and also some minor parts are even re-written directly in assembler (specifically x86 and ARM assembler) – these might be used during compilation instead of the C/C++ code depending on the platform and compilation options and can provide an increase in performance.

Judging from the `DOC\readme.txt` file and the presence of `.dsp` and `.dsw` files, Mr. Pavlov uses Microsoft Visual C++ 6.0 (MSVC6) to compile 7-zip. We will be using the command line tools included in the Microsoft Visual Sudio 2017 (MVS). To set up the environment we first need to execute the batch file `Common7\Tools\VsDevCmd.bat` located in the MVS installation directory from the `cmd.exe` (Command line). With a default MVS installation, we can also use a prepared shortcut from the start menu called `Developer Command Prompt for VS 2017`. In this environment, we have access to `nmake` which is Microsoft's implementation for the handling of makefiles.

For the build to succeed on this newer environment we either have to define a makefile macro named `NEW_COMPILER` or we have to delete/comment out the `-OPT:NOWIN98` option on line 33 in the `CPP\Build.mak` file.

The individual binaries that we talked about can be compiled separately. When we go through the source code directory tree we will find files called `makefile` – when such a file is present we can run `nmake` in that directory and appropriate parts of 7-zip will get built. All the makefiles in 7-zip actually include the already mentioned `7zip.mak` file which in turn includes the `Build.mak` file. The latter one sets up the correct compiler, assembler, linker, their options and environment in general, and the first one then uses these tools to build what was passed from the actual individual makefiles.

We will mainly be interested in building the console version of 7-zip – the `7z.exe` and the accompanying `7z.dll`. To build the executable we simply have to navigate to the `CPP\7zip\UI\Console` directory from our already set up terminal and execute the `nmake` command. A subdirectory named simply `O` will be created and it will contain object files and other files created during compilation and also the resulting executable file.

Building the library is very similar – this time we need to navigate to the `CPP\7zip\Bundles\Format7zF` directory and there we again execute `nmake`. The `Bundles` directory contains several different versions to compile (independent executables, reduced functionality libraries, SFX modules) and we will use the `Format7zF` – this is the `7z.dll` containing all compression and encryption capabilities of 7-zip without the use of external filters. Again we will get the `O` subdirectory containing the resulting library.

We can then copy both the executable and the library into any directory together and they should be fully functional.

More information about compilation of 7-zip, the individual compiled binaries and the source code can be found in the `DOC` directory specifically in the `DOC\readme.txt` file and also in the `readme.txt` file inside the the directory where 7-zip is installed when using one of the installers (these two files despite their name contain different information).

2.2 Coding style

The source code of 7-zip is quite interesting and quite complex. There is actually a lot of functionality coded from scratch that could be used from some widely used solutions – be it from an external library (i.e., OpenSSL) or even the C++ standard library. The reason behind this is probably a concern for extreme in-RAM space effectiveness and multiplatform performance optimization. A few examples of what is created from scratch are containers and types such as a vector or a map, simplified COM interfaces and most importantly for our thesis, the cryptography.

Specifically, in the case of cryptography, this is usually not a recommended approach. With widely used existing solutions (such as the OpenSSL library) we can be reasonably sure that there are no major security problems – because the library has been tested so thoroughly – and if some are discovered they

will swiftly be fixed. A custom implementation may introduce hidden or unexpected security problems.

To further push the optimization Mr. Pavlov defines a lot of makefile macros and C++ preprocessor macros, often nested, which results in some parts of the code actually being different depending on CPU architecture, OS, etc.

Arguably there are a few problematic things about the source-code itself, C and C++ coding conventions are somewhat mixed together here and the encapsulation used while using the objective style of coding is sometimes not respected well. Another might-be-problem is the frequent usage of global variables and sometimes using very long functions – while this might be just a cosmetic problem at first it might prove difficult to deal with later. Global variables can sometimes introduce unexpected problems in the code and long functions make the code much harder to read and understand for other developers.

2.3 Cryptography implementation

As was already mentioned 7-zip uses cryptography for the creation of encrypted archives. This is not the default option when creating an archive through 7-zip and so has to be explicitly asked for when creating the archive. 7-zip can in the default installation create encrypted `.zip` archives where it can either use the "ZipCrypto" or AES-256 encryption (with CTR mode of operation) and `.7z` archives where AES-256 (with CBC mode of operation) is always used. It can also extract encrypted `.rar` archives, but it cannot create them (this is because `.rar` is a proprietary closed source format). We will be focusing on the `.7z` archive format.

2.3.1 AES implementation

This subsection focuses on the implementation of AES in 7-zip. AES is a good choice for data encryption since it is very widely used and therefore thoroughly tested – it is secure and fast, provided that it is implemented correctly.

AES isn't ever used in the raw ECB mode in archives created by 7-zip it is always used either in the CBC mode or the CTR mode. `.7z` archives use CBC mode. With the goal of good cross-platform performance, there are actually three different implementations of AES in 7-zip.

- A pure C implementation[6]
- A pure x86/x64 assembler implementation using the built-in AES new instructions (AES-NI)[7]

[6] in `C\Aes.c`
[7] in `Asm\x86\AesOpt.asm`

- A C implementation using the compiler intrinsics which in turn also use the AES-NI[8]

There is however only one implementation of the key expansion into encryption/decryption round keys – it is pure C and is also present in `Aes.c`[6]. This is not a security problem however it would be possible to further improve the performance by also using the Intel AES-NI instructions for key expansion.

AES-NI is an implementation of an "AES instruction set" consisting of several specialized instructions which are now part of the x86 instruction set. They achieve greater encryption/decryption speeds than regular implementations and should be resistant to side-channel attacks [8].

Compiler intrinsics (sometimes called intrinsic functions) are specialized functions which the compiler knows beforehand – they are built into the compiler itself. They do not need a definition within the scope of the program [9]. They often translate directly one-to-one to an assembler instruction. The Microsoft Visual C++ compiler recognizes several intrinsics among which are also the ones used for AES encryption and decryption – they translate to the aforementioned AES-NI instructions.

The three 7-zip AES implementations and the key-expansion functions in the mentioned files assume that you already somehow generated the base AES key and the initialization vector for the CBC mode – 7-zip does this in the higher-level C++ code about which we will talk later.

We will now ignore the key and IV generation in 7-zip and will test these implementations by themselves by supplying our own key and IV and comparing their results with a widely accepted cryptographic solution – the OpenSSL library. A small C++ program for this purpose is provided on the attached SD card in the directory `Sources\AES_testing`. Inside the directory are the following files:

- `7zTypes.h, Compiler.h, CpuArch.h, CpuArch.c, 7zAsm.asm` – these files are supporting 7-zip files which are necessary for the other ones to work.

- `Aes.h, Aes.c` – these files contain the basic structures for AES encryption in 7-zip, the methods for key expansion, a CBC initialization function and also the pure C AES implementation.

- `AesOpt.c` – this file contains the optimized AES implementation using pure C and compiler intrinsics.

- `AesOpt.asm` – this file contains the optimized AES implementation using x86/x64 assembler and the AES-NI instructions by Intel.

[8] in `C\AesOpt.c`

- `myAesWrappers.h, myAesWrappers.cpp` – these files contain our wrappers for 7-zip AES-CBC encryption and OpenSSL AES-CBC encryption to make their interface as similar as possible – for easy comparison.
- `mainTester.cpp` – this is our main testing file. It calls the encryption routines, compares their results, etc.
- `Building_instructions.txt` – this file contains information about how to build the program with the three different AES 7-zip implementations.

Our program can take several files from the directory in which it was executed as input:

- `iv.txt` The initialization vector (16 bytes)
- `key.txt` The AES-256 key (32 bytes)
- `plaintext.bin` The plaintext to be encrypted (multiple of 16 bytes)

The IV and key are supplied as plain text, where each byte is represented as two hexadecimal digits. No white spaces are expected between the individual bytes. The plaintext to be encrypted is supplied directly in binary form – actual text can, of course, be written inside the file. There just isn't any conversion – the file is read in binary mode exactly as it is. The plaintext size in bytes has to be a multiple of 16 (size of one AES block) because padding isn't implemented in these lower-level 7-zip files. 7-zip, naturally, does use padding however this is implemented in the higher-level C++ code which we aren't testing with this program.

None of these files are mandatory – in the case of some of the files missing there are default hard-coded alternatives (see `default_iv, default_key` and `default_plaintext` in `mainTester.cpp`).

Since there are three different AES implementations in 7-zip (however not all three can work on every platform) we made it so that our program can be compiled against all of these, thus we can check their functionality separately. We have therefore six different variants since we also test the 32-bit and 64-bit version separately.

For compilation, we will again be using the command-line tools made by Microsoft which are included with an MVS installation. The specific tools we will be using are `cl` (this is a program containing both the C and C++ compiler and the linker), `ml` and `ml64` (these are the x86 and x86-64 assemblers respectively).

For compilation of the 32-bit versions, we will open the `cmd.exe` and run the `vcvars32.bat` script included in the MVS installation – this initializes the environment. Within the MVS installation directory we can find this file in the `VC\Auxiliary\Build` subdirectory, however, this might differ in different MVS versions. An easier approach is using the shortcut which is

created in the Start menu by the default installation – it is called `x86 Native Tools Command Prompt proVS 2017` and does what was just described for us. For the 64-bit version, the setup process is the same except we will use the `vcvars64.bat` (or the `x64 Native Tools Command Prompt pro VS 2017` shortcut).

The file `Building_instructions.txt` describes how to compile these different variants of our program. Every source file (`.asm`, `.c` or `.cpp`) has at least 2 variants – 32-bit and 64-bit versions and some have more because of the different AES implementations. These variants are indented. After the colon character, there is a command to copy to the respective building environment and execute as-is to build the desired variety.

The AesOpt.c compiles into four different object files:

- `AesOpt_PURE_C_x86.obj` or `AesOpt_PURE_C_x64.obj` for the pure C AES implementation
- `AesOpt_intrin_C_x86.obj` or `AesOpt_intrin_C_x64.obj` for the C with compiler intrinsics implementation.

The latter differs in the compilation by having the preprocessor macro `intrin` defined through the compilation command line parameters (this is done as `/Dintrin` in `cl`). We have modified the original `AesOpt.c` from 7-zip, so that it chooses the implementation based on the `intrin` macro either being or not being defined – originally the choice is determined based on the target platform (compiler version and CPU architecture).

The `mainTester.cpp` and `myAesWrappers.cpp` are compiled together and linked with the other object files and with the necessary OpenSSL libraries. The names of the resulting executables and commands to build them are also provided in the `Building_instructions.txt` file.

For our program to build and work we also need to install the OpenSSL library first. It is best to download the official sources[9] and compile them ourselves so that they work correctly on our version of Windows, CPU architecture, etc. The downloaded sources come with files describing the installation process somewhat, however, we will in short try to describe it step-by-step here for the convenience of the reader. These steps apply when installing OpenSSL on a Windows machine. We will be using the same compilation environment as when compiling 7-zip – run `Common7\Tools\VsDevCmd.bat` from the MVS installation directory in `cmd`, or the shortcut `Developer Command Prompt for VS 2017`.

We will also need to install some implementation of the Perl scripting language – we recommend using ActivePerl[10]. For Perl it's enough to install the 64-bit variant for our purposes even if we plan on compiling both 32-bit

[9] https://github.com/openssl/openssl
[10] https://www.activestate.com/products/activeperl/

and 64-bit OpenSSL (which we should, if we want to perform all six variants of the tests). However for NASM – an assembler – we need both the 32-bit and 64-bit installation[11]. After installing these tools we should add the paths of their installation to the Windows environment variable `PATH`. This variable contains various directories to be searched in many situations – mostly when running utilities and programs without their full paths from the command line. The easiest way is probably appending the 32-bit NASM installation path to the `PATH` variable for the installation of 32-bit OpenSSL; then deleting it and replacing it with the 64-bit NASM installation path for the installation of 64-bit OpenSSL.

Both variants are then built this way:

1. Launch `cmd` with the mentioned MVS environment with administrative privileges and navigate to the directory containing the extracted OpenSSL source code
2. Issue command `Perl Configure VC-WIN32`
3. Issue command `nmake`
4. Launch another session of the same `cmd`, however this time without administrative privileges (and again navigate to the same directory)
5. Issue command `nmake test`
6. Finally return to the administrative command prompt and issue command `nmake install`

For the 64-bit variant just replace the `VC-WIN32` with `VC-WIN64A`.

Now we are ready to build our testing program as described earlier. The finished executable files can then be run from the `cmd.exe` and don't accept any command line parameters – only the input files described earlier. The program first reads the input files or sets the default values if the respective files aren't present. Then it uses 7-zip's AES implementation to encrypt the plaintext. The `key`, `iv` and `ciphertext_7z` buffers (which belong to the supplied AES key, initialization vector and the resulting ciphertext respectively) are then checked for buffer overflows.

This check is achieved by allocating larger buffers than actually necessary to hold these values (specifically the amount of bytes to enlarge the buffers is defined in `myAesWrappers.h` and is called `BUFFER_CHECK` – by default the value is 4, but this can be changed freely). These enlarged buffers are then filled with a randomly generated value (`buffer_fill`). To find out whether an overflow did occur, we check if this value changed in any of the bytes following the normal size of the stored value. If not we assume the buffer didn't overflow.

[11] https://www.nasm.us/

We then perform the same encryption using the OpenSSL library and compare the resulting ciphertext to the one generated by 7-zip. The same checks are then performed for decryption – the buffer overflow check here also checks the `decrypted_plaintext_7z` buffer used to store the decrypted plaintext generated by 7-zip in addition to all the ones that were checked after encryption. The resulting plaintext buffers from 7-zip and OpenSSL are again checked for a match.

The program indicates success if all the checks passed or failure if at least one of the checks failed.

Worth mentioning is the use of the `_aligned_malloc` and `_aligned_free` function. These functions are specific for MSVC (Microsoft Visual C++)[12]. It is necessary to use them for the buffers holding the ciphertext and decrypted plaintext from 7-zip – this is because if we use the special AES-NI assembler instructions they require these buffers to be aligned to 16 bytes. Such an alignment can be guaranteed by neither the C `malloc` nor the C++ `new` operator. You can allocate buffers of static size and align them to 16 bytes (or any other power of 2 bytes) using the C++ `alignas` specifier, however, this isn't usable for dynamically allocated buffers. We actually use the `alignas` in the `myAesWrappers.cpp` file for the allocation of a buffer that 7-zip uses for the storage of both the initialization vector and the expanded key (`ivAes`). The 16 byte alignment technically wouldn't be necessary when compiling against the pure C implementation however it doesn't do any harm if it is present even then.

All tests did pass during our testing – none of the buffers have overflown and the AES outputs are the same from both 7-zip and OpenSSL.

Aside from the AES implementation itself, there are higher level functions in 7-zip taking advantage of this implementation to create the encrypted archives. Specifically, concerning the `.7z` archives, most of the interesting code resides in the `CPP\7zip\Crypto\7zAes.cpp` file and its accompanying header file `CPP\7zip\Crypto\7zAes.h`.

The initialization vector for AES-CBC mode is the size of one AES block – 16 bytes. However, in 7-zip version 18.05, 18.06 and earlier only the first 8 bytes are randomly generated. The rest is always filled with zeroes as seen in figure 2.2.

Furthermore, the random number generator (RNG) for the first 8 bytes is not strong enough for the purposes of cryptography. It uses SHA-256 (Secure Hashing Algorithm version 2, with output hash length of 256 bits) with a constant salt; other inputs are system time and process ID. As seen in figure 2.3 a comment in the file `CPP\7zip\Crypto\RandGen.cpp` even says to use the generator only for salt generation, however instead it is used for the IV

[12] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=vs-2019

Figure 2.2: IV generation (from `7zAes.cpp`)

```
165 for (unsigned i = 0; i < sizeof(_iv); i++)
166   _iv[i] = 0;
167 _ivSize = 8;
168 g_RandomGenerator.Generate(_iv, _ivSize);
```

generation. This problem seems to have been addressed in version 19.00 – the IV is now the full 16 bytes and the RNG was improved – in addition to the existing inputs it now also uses the thread ID and more importantly the RNG provided by the OS.

Figure 2.3: A comment regarding the RNG in 7-zip (from `RandGen.cpp`)

```
24 // This is not very good random number generator.
25 // Please use it only for salt.
26 // First generated data block depends from timer
      and processID.
27 // Other generated data blocks depend from
      previous state
28 // Maybe it's possible to restore original timer
      value from generated value.
```

2.3.2 The key-derivation-function

This subsection focuses on the implementation of the password-based key-derivation-function used in 7-zip. The KDF in 7-zip used with `.7z` archives is implemented in the `CPP\7zip\Crypto\7zAes.cpp` file.

The KDF uses the hashing algorithm SHA-256. The key is derived from the user-supplied password in this manner: the password is appended by 8 bytes that represent an endian-independent counter. Then a constant number of iterations is performed. Depending on the version of 7-zip this number can be different – in 18.05 there are 524 288 (or 2 to the power of 19) iterations. One iteration consists of appending the buffer with the password and counter again (by calling the `Sha256_Update`) and incrementing the counter by one. We can see this in the code snippet in figure 2.4. In the end, we practically get one long buffer where there are the password and counter copied over and over with the counter going up. This long buffer is digested by the aforementioned SHA-256 algorithm and the resulting hash is the key used for AES-256.

This means that the actual full SHA-256 algorithm only takes place once. The `Sha256_Update` is, of course, called as many times as there are iterations however this only simulates preparing the long buffer and then the

Figure 2.4: `.7z` Key-derivation-function (from `7zAes.cpp`)

```
51 size_t bufSize = 8 + SaltSize + Password.Size();
52 CObjArray<Byte> buf(bufSize);
53 memcpy(buf, Salt, SaltSize);
54 memcpy(buf + SaltSize, Password, Password.Size());
55
56 CSha256 sha;
57 Sha256_Init(&sha);
58
59 Byte *ctr = buf + SaltSize + Password.Size();
60
61 for (unsigned i = 0; i < 8; i++)
62   ctr[i] = 0;
63
64 UInt64 numRounds = (UInt64)1 << NumCyclesPower;
65
66 do
67 {
68   Sha256_Update(&sha, buf, bufSize);
69   for (unsigned i = 0; i < 8; i++)
70         if (++(ctr[i]) != 0)
71           break;
72 }
73 while (--numRounds != 0);
74
75 Sha256_Final(&sha, Key);
```

`Sha256_Final` call creates the final digest. Other KDFs (such as PBKDF2) often actually use the output from the first hashing to salt the next hashing round and chain them in a number of iterations [10].

Unfortunately, no cryptographic salt is used in this hashing. At the first glance, the source-code actually seems to support using salt and would appear to use it in the function. It even seems to contain a random-number-generator and a code snippet that would use it for the salt generation – but this code is commented out and never called – thus no salt is actually used.

Even though the function for key-derivation seems to prepend the buffer with salt (see figure 2.4), the salt is actually of zero length and is empty. The consequence of this is that the same user-supplied password will always be transformed into the same AES key. Thus rainbow tables, mentioned in section 1.1, could theoretically be calculated containing the AES keys and the passwords which were used to generate them. Of course, someone (a potential attacker for example) with access to very fast computational hardware and

large digital storage would have to first calculate these tables. These tables could then be used to crack 7-zip archives faster than with a traditional brute-force attack.

However we also have to bear in mind that these tables would only be valid for the same KDF, if it was to change, they wouldn't work anymore since the password-key pairs would change. The KDF in the official releases of 7-zip changed last time with the release 4.58 beta (released in 2008). The change is that before this version the number of iterations was 262 144 – 2 to the power of 18 instead of 19.

The problem is somewhat mitigated by the way the password is actually checked when trying to extract an encrypted archive (more on that in section 2.4) so a brute-force attack would likely still be fairly slow. However, it is still a problem in principle – the code actually seems to salt the password before hashing and so a person concerned about security quickly peaking at the source code might think that salt is used, while it is in fact not.

Also, it doesn't really make much sense not to use the salt when the code-base is ready to use it. There is a SourceForge discussion thread originally from 2005 asking about other encryption options beside AES in 7-zip, which evolved into a discussion about the usage of cryptographic salt. The interesting part of the discussion took place in 2008 mostly between Mr. Pavlov (author of 7-zip) and two 7-zip users with nicknames Hitcher and Marco Certelli. This discussion shows that Mr. Pavlov is aware of what the missing salt means, however, he doesn't think it is a major security problem. Mr Pavlov writes:

> "7-Zip's AES decoder supports salt. But it's disabled in encoding code. I still think that there is no big gain from salt using."

and

> "Salt can help from dictionary attack. But if password is random (small latin characters + digits), you need too big dictionary:
> 5 characters in password: 1 GB dictionary
> 6: 30 GB
> 7: 1 TB
> 8: 30 TB
> 9: 1000 TB"

and

> "7-Zip uses random AES-CBC init vectors. So probably it's pretty strong even without salt."

and

> "Note, that you still need SHA calculation Hardware to create that dictionary. Salt can prevent some complex attacks when some

> orginization with big budget creates really big dictionary (lets suppose 10000 TB). And then it uses that big dictionary to check big number of .7z archives. So the cost of crack per archive will be reduced. But if there is such big organization, they can invest same money to fast SHA-calculation hardware (that can be probably 100 times faster). In that case they don't need big dictionary. So 32 bytes (key size) of HDD can be more expensive than 256000[13] iterations of SHA-256 in some cases."[11]

It is not quite clear from the discussion why Mr. Pavlov decided to actually code the parts necessary for salt usage but then they aren't used in archive creation in a normal release 7-zip installation.

He also mentions random initialization vectors as an answer to a users question regarding cryptoanalysis – it is true that different IVs mean that the resulting ciphertext will always be different even for the same encrypted data and the same AES key used. This point is a bit undermined by the fact that this discussion took place in 2008, but as we already pointed out in 2.3 up until version 19.00 (which was released in 2019) the IV used in `.7z` is also quite problematic.

We can successfully create archives which do use salt by modifying the code on line 197 in `CPP\7zip\Crypto\7zAes.cpp`. This line is normally commented out as seen in figure 2.5. If we uncomment this line and build 7-zip the KDF is the same except the buffer that is copied over and over and then hashed now consists of the salt + the password + the coutner. This means now the same password won't generate the same AES key and thus rainbow tables cannot be used to speed up a potential password guessing attack. We can even change the size of the salt used by changing the number 4 to the desired number of bytes, however, the maximum size is 16 bytes (which is more than enough).

Figure 2.5: CEncoder constructor (from `7zAes.cpp`)

```
195  CEncoder::CEncoder()
196  {
197    // _key.SaltSize = 4; g_RandomGenerator.Generate
         (_key.Salt, _key.SaltSize);
198    // _key.NumCyclesPower = 0x3F;
199    _key.NumCyclesPower = 19;
200    _aesFilter = new CAesCbcEncoder(kKeySize);
201  }
```

In figure 2.5 we should also notice an initialization of a data member called `NumCyclesPower`. On line 198 the member would be initialized with

[13] At the time this discussion took place only 262 144 iterations were performed in the KDF

the hexadecimal value 3F but the initialization is commented out. On line 199 the actual initialization takes place with the decimal value 19. This is very important – this value represents the power to which we raise the number 2 and the result is the number of iterations used during the KDF (the number of times we copy the prepared buffer before hashing it). This value can also be changed and after the compilation of such modified 7-zip we would have a different number of iterations. The maximum power that we can use is 24 as defined on line 25 of the same source file.

Both the salt and the `NumCyclesPower` are actually saved in the resulting archives and if we modify 7-zip in this way, it still creates valid `.7z` archives. Other installations and versions of 7-zip will be able to open, extract and modify the archives since they read the values from the archives first and set up their decoding to match these values.

This can, on one hand, be really good – we can actually use salt if we want and by increasing the `NumCyclesPower` value we can slow the KDF function down even more – this way we can create very safe encrypted archives, which will still be working perfectly even with unmodified installations of 7-zip but will be virtually uncrackable (assuming the underlying algorithms AES and SHA-256 will not be broken in the future).

It is also, however, a double-edged sword. We could use this to lower the security of 7-zip and then distribute this maliciously modified installation or perhaps the `7z.dll` library between would-be victims. This could be at first glance achieved by lowering the number of iterations performed in the KDF.

However, there is a much bigger problem. The commented line assigning the 0x3F to `NumCyclesPower` isn't there by accident. If we use this value instead, the KDF changes entirely and the security is dramatically reduced. As seen in figure 2.6 when `NumCyclesPower` is equal to 0x3F the KDF actually doesn't use the hashing algorithm at all. Instead the AES key is created directly in this way: first, the salt (which is usually empty) gets copied over to the key buffer, then the user-supplied password and the rest of the AES key is filled with zeroes.

7-zip uses two bytes for storage of every character, thus allowing Unicode characters in passwords. This means that a 16 character password can fit in this 32 byte AES key in this weakened KDF. If a user inputs a longer password than 16 characters only the first 16 are used, the rest is disregarded and 7-zip doesn't give any indication of that happening.

We can further weaken the KDF even more by intentionally using salt of the maximum size – 16 bytes. As can be seen in figure 2.6 the salt is then going to populate half of the AES key and we only have 16 more bytes for the password, resulting in a password of maximum 8 characters. Most people still use passwords consisting only of ASCII characters, this makes it likely at least half of the remaining 16 bytes are filled with zeroes. In conclusion, if we want to attack an archive which used this KDF, we already know 24 bytes of the 32 byte AES key – the salt is stored in plaintext in the archive and half

Figure 2.6: Beginning of the CalcKey function (from `7zAes.cpp`)

```
39  if (NumCyclesPower == 0x3F)
40    {
41      unsigned pos;
42      for (pos = 0; pos < SaltSize; pos++)
43        Key[pos] = Salt[pos];
44      for (unsigned i = 0; i < Password.Size() &&
           pos < kKeySize; i++)
45        Key[pos++] = Password[i];
46      for (; pos < kKeySize; pos++)
47        Key[pos] = 0;
48    }
49    else
50    {
51          ...
```

of the password bytes are zeroes (assuming the user chose ASCII characters for their password).

This is a major problem. The vulnerability doesn't lie in just the ability to weaken the source code and then compile a compromised 7-zip library/executable – that could arguably be done with most open source applications. The problem here is, that the decoding part of 7-zip is prepared for these changes and accepts them normally, even in the release version, without any indication to the user. So if we create these weakened archives where the KDF is for all intents and purposes instantaneous, they will behave like any other archive in any 7-zip installation – the only difference is that only the first n[14] characters from the password the user chooses actually get used. When the user then tries to extract the archive the password will be evaluated as correct every time these first n characters are correct. Very observant users might perhaps notice this. However, data can be extracted as expected and we can even add more files to the archive (even though if we add them with an unmodified 7-zip installation the added files will again have the stronger encryption thanks to the normal KDF). Password guessing on archives created with this version of the KDF will be much faster.

In the same manner, we could also weaken the initialization vector – we could even hard code it to always be the same. This would weaken the encryption as well and unmodified versions of 7-zip again wouldn't have any problems handling such archives. However this problem is difficult to avoid, the IV has to be saved in the archive for proper decryption anyway. Unlike the problem with the weakened KDF which just shouldn't be present in the

[14] $n = (32 - SaltSize)/2$

application in the first place.

We will see some examples of archives with the modified and original values and attempts at cracking them in the last subsection of section 2.4.

2.4 Archive cracking

In this section we will first briefly describe how a `.7z` archive is structured and how the password verification process works. Then we will setup our working environment to perform password guessing attacks against `.7z` archives. Finally we will attempt these attacks and talk about the results.

2.4.1 Archive structure

The `.7z` archives contain 'folders' or 'blocks'. These are continuous streams of data which are processed by the same methods. This means that an archive can contain several of these blocks and each might be using different methods. These can be set up manually by using command line parameters when running 7-zip. This is described quite well in an external document written by one of the 7-zip users called "Understanding 7z Compression File Format" [12]. Unfortunately the document was never finished, but the information in it is still useful.

If we do not specify any blocks in this way 7-zip does it automatically, choosing usually to compress everything with the LZMA2 compression method (also developed by Mr. Pavlov [13]) and when handling either `.dll` files or `.exe` files also first applying the BCJ/BCJ2 filter which can then allow for better compression. Also, a new block is always created when new files are added to the archive after it has been created.

The encrypted `.7z` archives by default don't have the headers encrypted, that means the directory and filenames and some metadata within the archive can be viewed even without knowing the password.

In this mode, the archive contains the decompressed size and the decompressed CRC32 (cyclic redundancy check – a checksum) of the first file in each block. This means that in order to verify a password used to encrypt one of the blocks, at least as much data as the uncompressed size of the first file in that block must be extracted. We can then calculate the CRC32 of the decompressed file and compare it with the one stored in the archive, thus finding if the password was correct or not.

This is the reason why brute-forcing the password might be pretty slow. Extracting the file actually consists of decryption and decompression and depending on the specific file used this could potentially be a very costly operation. It also means that the speed of the attack actually depends on what files are stored in the archive and in which order 7-zip places them in the blocks since some files will take longer to decompress then others.

How the information is structured inside the `.7z` archives is described in `DOC\7zFormat.txt` and also in the already mentioned external document [12].

2.4.2 Cracking setup

We will demonstrate possible attacks on the encrypted archives using an external open source program called hashcat (the authors themselves write it with a lower-case 'h' so we will too). There are other utilities out there capable of password-guessing attacks however hashcat is nowadays the most widely used and it is very well optimized. It uses the GPU (graphical processing unit) of a computer to achieve very high parallelization [14]. We can download or clone hashcat source code from the official GitHub page[15] and it is also included on the attached SD card in the `Sources\hashcat` directory. When we mention files and paths in this section, we consider the root directory to be the hashcat source code directory, unless stated otherwise.

Hashcat can take input in more ways, however, we will use input files which contain something that in the hashcat community is referred to as a "hash". It doesn't have to be an actual hash as in output from a hashing function though (and it isn't in the case of 7-zip) – hence the quotation marks. It is simply an input containing just enough information so that the program can guess passwords and verify whether they are correct or not. In this section when we use the word hash we mean this input for hashcat unless stated otherwise.

There are many ways to set up the actual guessing and we will not talk about them in great detail. All the necessary information is available on the hashcat wiki and forum [16], but at least in short: with hashcat we usually do not use a pure brute-force attack but rather a so-called "mask attack". We supply hashcat with a mask which is a string describing what the password should look like – the mask and parameters can be set up in such a way that we do actually get back to the classic brute-force and try every combination possible, but we can also set it up to be more specific. For example, if we know how long the password is, what character types (lower-case characters, upper-case characters, numbers, special symbols, ...) are used (even in specific positions in the password that is being guessed) we can specify that with the mask.

Also, a classic brute-force attack for example in the case of a five letter password made up of lower-case letters would proceed lexicographically guessing first `aaaaa`, then `aaaab`, `aaaac` and so on until reaching `zzzzz`. Hashcat also goes through all possibilities specified by the mask, however, the order is different. There are some very advanced thoughts behind the algorithms that choose which combinations to try sooner than others and we will not be describing them in this thesis, however for the interested reader a good

[15] `https://github.com/hashcat/hashcat`

[16] `https://hashcat.net/wiki/`, `https://hashcat.net/forum/`

starting point for studying would be Markov chains and learning how hashcat uses them.

In addition to the mask attack, there is, of course, the possibility of using a dictionary attack. There are also tools, included with hashcat, that can help generate large dictionaries by combining words from other dictionaries, changing letter case, appending or prepending numbers or symbols, replacing letters for numbers which visually resemble said letters and so on.

All-in-all hashcat is a very advanced password recovery tool. The team developing it regularly attends password cracking competitions and often wins[17].

Hashcat uses so-called "modules" – these describe how to attack different types of hashes. We will be using module 11600 which is the 7-zip module designed to help crack the `.7z` archives. The module expects a certain format for the input hash. To get this hash from an existing `.7z` archive we will use a Perl script `7z2hashcat.pl` developed by one of the hashcat developers Philipp "philsmd" (the surname is not publicly known). We can download this script from the authors public GitHub page[18] and it is also included on the attached SD card in the `Sources` directory.

The script itself is quite clever as well – if there are more encrypted blocks in the archive it finds the one which has the smallest first file – thus ensuring faster cracking. Because of the way 7-zip verifies passwords the hash produced actually contains the encrypted (and possibly compressed) data from that file.

We can either use the release version compiled into a binary executable if we use Windows, or the Perl script itself either on a Linux system (or any Unix-like system) or on Windows as well. For the latter variant we need Perl installed and a Perl module `Compress::Raw::Lzma`. While making this thesis we used the Perl script as-is from the Windows subsystem for Linux (WSL – an emulator layer translating Linux API calls to WinAPI calls – allows for usage of Linux tools from Windows). We used WSL instead of a Windows Perl implementation because it is easier to install the `Compress::Raw::Lzma` module for it – on a Debian based distribution we just use the `apt` package manager to install the `libcompress-raw-lzma-perl` package and we are ready.

As for hashcat, itself we also can use the released compiled binaries or compile them ourselves from source. We did the latter since we want to be able to modify the code. There are more ways to build hashcat – we decided to use WSL and the cross-compiler `x86_64-w64-mingw32-gcc` to compile for Windows (hashcat doesn't seem to officially support compilation by the MSVC tools).

The compilation instructions are included with the hashcat source code

[17] `docs\team.txt`

[18] https://github.com/philsmd/7z2hashcat

in the file `BUILD.md` and also in the comments of the makefile[19] itself (line 549–552). According to these comments we also need to build and install `win_iconv` which is a hashcat dependency. We can download or clone the source of `win_iconv` from its GitHub page[20] and it is also included on the attached SD card in the `Sources\win-iconv` directory.

To cross-compile hashcat for Windows from a Linux system:

1. Use `apt` to install the package `g++-mingw-w64`
2. Download/git clone the source code from `https://github.com/win-iconv/win-iconv`
3. Use the Linux tool `patch` to patch the `Makefile` included with the downloaded source code with the `tools\win-iconv-64.diff` file included with hashcat
4. Issue the command `make install` from the `win_iconv` source code directory.
5. Restart bash and issue the command `make win` from the hashcat source code root directory.

When modifying the source code of hashcat we only need to re-do the last step to build the binaries with our changes.

To extract the "hash" from an existing `.7z` archive we simply need to run the `7z2hashcat.pl` script and pass the name of the archive in question as the first parameter. The hash is then written to the standard output, so we usually redirect it to a file that we can later use with hashcat[21].

Hashcat can handle different values of the `NumCyclesPower` variable described in section 2.3 well, as long as the values lie within the range 0–24. However if we use value 0x3F therefore using the weak KDF (also described in section 2.3) hashcat won't handle it correctly – it will actually treat it just like any other value thus setting the number of iterations to compute in the KDF to 2 to the power 0x3F (0x3F is 63 in decimal). This would of course be a very large number of iterations which would render the attack computationally impossible, but even if we somehow could do such calculations in a reasonable time, they would be wrong anyway. As was already discussed no iterations are actually used in the KDF when `NumCyclesPower` is equal to 0x3F instead the password is just copied over to the AES key (prepended by a salt, if present) and the rest of the key is filled with zeroes.

Hashcat isn't prepared for this, however, we managed to modify the source code so that it could crack such archives as well. The modifications are what

[19] `src\Makefile`, not the `Makefile` in the root dir

[20] https://github.com/win-iconv/win-iconv

[21] e.g., `7z2hashcat.pl test_archive.7z > test_archive_hash.hash`

could be described as "crude hacks" they in fact very much break the normal functionality of hashcat and only make it usable to crack these type of archives. This could be fixed with more work and probably optimized better than we did, however these modifications are just for demonstration purposes – to show the difference in cracking speed. We need to modify 2 files for this purpose: `src\modules\module_11600.c` and `OpenCL\m11600-pure.cl`. The latter is included in its modified form on the attached SD card in the `Sources` directory. This file normally handles the whole KDF. In our modified file we commented out most of the code because it is not necessary for the weaker KDF (no hashing is performed). We left the data structures and the `m11600_comp` function at the bottom as is, commented out all functionality of the other functions and added new code to the `m11600_hook23` function. This code can be seen in figure 2.7 and basically just copies the password characters to the AES-key.

Figure 2.7: Modified `m11600_hook23` function (from `m11600-pure.cl`)

```
303  const u64 gid = get_global_id (0);
304
305  if (gid >= gid_max) return;
306  hooks[gid].ukey[0] = 0;
307  hooks[gid].ukey[1] = 0;
308  hooks[gid].ukey[2] = 0;
309  hooks[gid].ukey[3] = 0;
310  hooks[gid].ukey[4] = 0;
311  hooks[gid].ukey[5] = 0;
312  hooks[gid].ukey[6] = 0;
313  hooks[gid].ukey[7] = 0;
314
315  const u32 pw_len = pws[gid].pw_len;
316  for (u32 i = 0, idx = 0, j = 0; i < pw_len && j < 4; i +=
         4, idx += 2, j += 1)
317  {
318    u32 pw_tmp = pws[gid].i[j];
319    hooks[gid].ukey[idx+1]   |= ((pw_tmp & 0xff000000) >> 8)
           | ((pw_tmp & 0x00ff0000) >> 16);
320    hooks[gid].ukey[idx]     |= ((pw_tmp & 0x0000ff00) << 8)
           | ((pw_tmp & 0x000000ff) );
321  }
```

The `module_11600.c` only needs one modification and that is changing line 615 from `salt->salt_iter = 1u << iter;` to `salt->salt_iter = 1u;`. This will guarantee that hashcat will not unnecessarily try to perform many iterations of no hashing, we only need to go through the copying code once. When we modify the `.c` source files we need to re-run the `make win` command, however not with the `.cl` files. These are compiled into something that is referred to as "kernels" by hashcat and they are built on every run,

unless already present. In the hashcat source code directory, there will be a subdirectory `kernels` (unless we didn't yet run hashcat at all, or issued `make clean` recently). When we change any of the `.cl` files we need to delete the contents of this subdirectory and then re-run hashcat. The kernels will be built again, with our changes incorporated.

With these modifications we will need to run hashcat with an additional command-line parameter – `--self-test-disable` – this is because the internal self-test isn't prepared to work with those modifications and would therefore always fail.

Hashcat also doesn't support the usage of salt in `.7z` archives. The code actually seems to have some basics implemented for the usage of salt, however since 7-zip doesn't normally use it, this functionality isn't finished and salted archives cannot correctly be cracked with hashcat. On the other hand, the hash format does support it and the `7z2hashcat.pl` correctly extracts both the length and contents of the salt. Fixing hashcat to support the salted archives would take a bit more work and is out of the scope of this thesis, however, it definitely would be possible. This means when we test the weak version of the KDF we will use NumCyclesPower = 0x3F but we will not be using salt. The results would be the same anyway if we test with a password of 8 characters or less.

Furthermore, hashcat has a built-in limitation for `.7z` archives – it can only process hashes up to a certain size (specifically the encrypted data in the hash cannot exceed 655 056 bytes. The thought behind it probably is that since the size of the data has a significant impact on the speed of one password guess, larger files would probably slow it down too much and render the attack impractical.

We tried lifting this limitation by modifying the source code again but it seems the limitation might lie somewhere deeper than just the files related to module 11600. We did manage to turn off the checks that stop hashcat from loading hashes that are over the size limit and enlarge some buffers in such a way that hashcat does actually go through to the main cracking loop and attempts to crack the larger file. However, it becomes unstable (probably because of segmentation faults). The larger the file supplied the more likely for hashcat to crash, though we did manage to crack files which are slightly larger than the original size limit, so it is possible in theory.

These are the necessary modifications to lift the size limit (but with no guarantee on the application remaining stable):

- In the file `src\modules\module_11600.c` on line 83 there is a declaration of a statically allocated buffer – `u32 data_buf[81882];`. The `u32` stands for an unsigned integer 32 bits in size. This buffer needs to be enlarged if we want to try cracking larger archives than the limit.

- Similarly in the same file on line 135 – `u32 out_full[81882];`.
- In the same file in the section where the hash is parsed into tokens, there are limits as well – on lines 404 and 410 we assign 6 into two variables which limit the size of the ninth and tenth field of the input hash – these represent the length of encrypted and decrypted data. To allow for larger files we might have to increase this limit.
- Similarly with the buffer representing the actual data that need to be decrypted to perform the password check. This is the 12th field in the hash and its limit is initialized on line 416 (normally the limit is 655 056 hex characters – or 327 528 Bytes of data).
- In the same file still, there are some data verification checks from line 504 to line 530. We could just comment them all out, or we can just comment out the ones that have something to do with the data length.
- In the file `include\common.h` on line 108 there is a macro define – `#define HCBUFSIZ_LARGE 0xb0000`. We also have to increase this value.
- Finally in the script `7z2hashcat.pl` on line 115:

  `my $PASSWORD_RECOVERY_TOOL_DATA_LIMIT = 655 056;`
  (The script doesn't have any problems evaluating larger `.7z` files, this limit is here only because of hashcat itself having these limits. So we can just increase this value as much as we want.)

We do not provide any specific enlargement values since it depends largely on the size of the data that we want to use. These "hacks" aren't very clean anyway and they make the application largely unstable, we are just including them for demonstration purposes. These modifications should be enough for hashcat to get past the hash decoding part and to the cracking loop even with files that are over the limit, however after that the behavior is undefined. Sometimes the application crashes, sometimes it can actually crack the archive. To make hashcat truly support larger files would likely be more complex since the issue probably lies somewhere in the hashcat engine itself.

2.4.3 Cracking attempts

In this subsection, we will showcase actual attempts at cracking encrypted `.7z` archives. We will use 7-zip to create several archives with different files inside them, different password lengths and different encryption settings and then attempt to crack them with hashcat. We will take notes about how fast each attack is going – this is usually represented by a number of password guesses per second, often called hashes per second (H/s).

We would like to emphasize that the actual numbers presented here, resulting from our testing do not have any significance on their own. Much more

important is how they relate to each other and their orders of magnitude – this is because the precise value will, of course, vary depending on the hardware used, the OS, other programs running on the system, etc.

On the attached SD card is a directory named `Cracking`, it includes example files and archives, that we used for this testing. The three subdirectories called `small`, `medium` and `large` each contain the original file to be encrypted – `small.bin` which is 100 Bytes in size, `medium.bin` which is 15 000 Bytes in size and finally `large` which is 320 000 Bytes in size (just below the official supported size limit of hashcat). These files are made up of random bytes (and therefore are also difficult to compress) and we generated them from a bash shell like this: (for different sizes replace the number after `-c`)

```
head -c15000 </dev/urandom >/archives/medium.bin
```

Each file has been archived 9 times. We used three variants for the `NumCyclesPower` variable in 7-zip (19 – the default value, 24 – highest possible and 0x3F (63) – the weak KDF). For every variant we encrypted each file thrice – once with a 4 character password ("7zip"), once with an 8 character password ("7zipPass") and once with a 10 character password ("7zipPassWD"). The respective archives reflect these values in their filename (for example `small_0x3f_10.7z` is the encrypted `small.bin` using a 10 character long password and the 0x3F variant of the KDF). The hashes generated by `7z2hashcat.pl` are also included in these directories, their name only differs in extension from their respective archives (`.hash` instead of `.7z`).

Finally, there are files with the same name and the `.txt` extension, these contain the output of hashcat after a few seconds of attempting to crack the respective files. The output contains information about how many hashes per second we are guessing and also an estimation of when the cracking will finish (more precisely when all the password possibilities for the given settings are exhausted – of course, in reality, it isn't very likely that we would guess the correct password on the last try). When using the 0x3F value for `NumCyclesPower` we need to use our modified version of hashcat described earlier, for the other two we use the original files.

The command that we execute to start the cracking attempt on the example file `small_19_8.hash` is:

```
hashcat.exe -a3 -w3 -m11600 --potfile-disable --self-test-disable -1?l?u?d archives\small\small_19_8.hash ?1?1?1?1?1?1?1?1
```

- `-a3` – This tells hashcat that we want to use the mask attack.
- `-w3` – This is the optimization level, it is not necessary.
- `-m11600` – This chooses the correct module for the input hash (7-zip).
- `--potfile-disable` – This means that the results do not get saved to a "potfile". If we leave this out then cracked hashes get saved. If we

attempt to crack the same hash again we get a message about it already being cracked and hashcat exits.

- `--self-test-disable` – As we already explained this is needed in cases we modified the hashcat source code to the point where the self-test would always fail.
- `-1?l?u?d` – Here we specify a custom charset number 1 which includes the built-in charsets l – lowercase letters, u – uppercase letters and d – numbers.
- `archives\small\small_19_8.hash` – We specify what hashfile we want to attempt to crack.
- `?1?1?1?1?1?1?1?1` – This is the mask. The number of question marks is the number of characters in the guessed password and the character following a question mark specifies the charset for that position – we use our charset 1 defined earlier[22]. For guessing a shorter password we have to make this mask shorter[23].

For other files, we have to modify the archive name and the mask accordingly.

We encourage the reader to view our results in the text files and perhaps even perform tests with other password lengths, charsets, or with dictionaries. With our knowledge of the underlying algorithms and now the available test results we can summarize what impact do different values have. The time to perform one guess (and therefore the number of guesses per second) depends mostly on the size of data we need to extract to perform the check and on the value of the `NumCyclesPower` variable. On the other hand, the total count of the guesses we need to perform depends on the password length and the charset(s) used in the password.

When using our charset of lowercase letters, uppercase letters and digits it still isn't reasonably doable to crack most archives if they use a password of at least 8 characters, even when we use our significantly weaker KDF. This, however, changes if we were to change the charsets, or if we were to use dictionary attacks instead of these brute-force-like attacks. The important number is the hashes per second in that case.

With a normal release version of 7-zip, this should be perfectly safe. Now, let's return to our idea of the purposefully weaker 7-zip. We already have

[22] If we knew some characters exactly we could replace the question mark and charset specifier for a certain position with a specific letter. There are much more complex ways to execute the attack, for example, using "rules" but we will not focus on them in this thesis – the hashcat website has all the necessary information to get started.

[23] Hashcat also allows for incremental masks – we can specify at least and at most how many characters a password has and hashcat tries all the possibilities in that range

control over the `NumCyclesPower` variable. Could we do something about the data size as well, while keeping the archives valid?

We actually could. This would be a change that would be a bit more noticeable, than just changing the encryption values though. We could modify 7-zip in such a way that when creating an encrypted archive, it purposefully adds a new block, which doesn't use compression (only encryption) and populates it with a very small file. This file could then always be used for password verification and would make the guessing speed the same for archives of all sizes.

This change would, of course, mean that suddenly every archive would have one more file in it and upon extraction, it would be extracted with all of the other files. If we wanted to develop an actual attack there could be some smart ways around this fact to make everything look as normal as possible. For example, we could create a plain text file called `desktop.ini` which would only contain the semicolon character (;) followed by 15 other characters including the linebreak. We would create the file with the `hidden` and `system` attributes. Files with these attributes will be invisible in Windows Explorer by default. Furthermore, the `desktop.ini` is actually generated by Windows in directories when it needs to store additional information about that directory (for example changed directory icons, etc.), so it wouldn't be that suspicious. The file cannot be empty if we want to use it in the password verification process. This way the file is only one AES block long and still a valid `.ini` file, since the semicolon is used for line comments in them.

There are probably many other ways to hide additional files into the archives which could attempt to be inconspicuous while still upholding the fact that the archive is a valid `.7z` archive, usable with any 7-zip installation. For example, if the user uses 7-zip to compress a larger directory structure we could hide the file deep inside the directory structure. Or we could forcefully add the `-sns` command line parameter to any archive creation – this enables NTFS alternate data streams functionality in 7-zip. We might be then able to hide the small file inside another file's alternate stream.

This way we would actually be able to speed up the cracking process even more since we do not actually need to decompress any data and we do not even need to calculate the CRC32 checksum. This is because we know the plaintext of the file. Since it's just one AES block, the verification for every password-guess is just decryption and then comparison to our known plaintext.

Since we have control of the initialization vector as well we could just make it always the same. That would allow us to calculate lookup tables which would contain password:ciphertext pairs, where ciphertext is generated with the given password from our planted one-AES-block-sized file (this means the ciphertext is also just one AES block in size). The lookup tables would be calculated in advance for a certain set of passwords and then sorted lexicographically by the resulting ciphertexts. Then if we wanted to check if an encrypted archive, created by such a 7-zip build, used a password that was on

our list, we could just search for the ciphertext of our planted file in this table and get the password without any guesswork at all. These modifications are however much more noticeable and tech-savvy users might notice at least one of them, thus making them ineffective.

2.5 Other problems

In this section, we will talk about other findings which might prove problematic from a security standpoint, not always necessarily connected to the use of cryptography.

There is a slight problem with how the AES key and the password are handled in memory. Both are copied around quite a lot and mostly stored in the custom-coded string types defined in `CPP\Common\MyString.h`. When we look at the destructors and methods of these string types we find that the pointers pointing to the stored characters are eventually just deleted by using the C++ `delete` operator. This, however, doesn't guarantee that the memory where they were stored will actually be overwritten, it merely "forgets" the pointer to that part of memory and notifies the OS to make it available for future allocation. A better way would be to first overwrite the memory, preferably with the WinAPI function `SecureZeroMemory` [15].

Furthermore, when handling sensitive information (again – key and password) we should make sure that it doesn't get swapped to the hard disk in plaintext, where it could potentially be read by someone unauthorized. We can achieve this by either: using the WinAPI function `CryptProtectMemory`, which encrypts part of the memory so that even if it gets swapped, it is unreadable; or by using another WinAPI function `VirtualLock`. This ensures that the data actually stays in RAM and doesn't get swapped to the hard disk at all. Neither of these functions is ever used in 7-zip.

7-zip does, however, support the use of large memory pages when using the command-line parameter `-slp`. Since large pages are always locked in memory, 7-zip first enables the `SeLockMemoryPrivilege` (which is needed for large pages) for the user account that it is running from. This isn't a security feature however, instead it is supposed to speed up the compression of large files. To enable this privilege 7-zip needs to be running from an administrator account. From a security standpoint, this can actually be harmful instead of helpful. The `SeLockMemoryPrivilege` is left enabled on the user account that ran 7-zip even after it terminates. Since the user isn't notified about this in any way, they will probably leave it enabled and it could potentially result in a "denial–of–service condition" later as described in the Microsoft docs [16].

7-zip actually supports the modification of the `.7z` archives after they've been created and it also supports having some files encrypted, some unen-

crypted or even different files being encrypted with a different password within the same archive.

This creates another potential problem – a user of 7-zip might assume that creating an encrypted archive would prevent it from being tampered with – unfortunately this is true only when the headers are encrypted as well, but that option is not enabled by default[24]. When just the data is encrypted but the headers are not, it is actually possible to add files to the archive in an unencrypted form. This is probably an intended feature – when opening such an archive in the `7zFM.exe` you can see well which files in the archive are encrypted and which are not. However often the user would just use the "extract all" command and in such a case he would be asked for a password and all files would be extracted, including the unencrypted ones, which were added later. This allows for planting of malicious files into encrypted archives – even though the unencrypted files get extracted before the password is entered, so more observant users might notice it.

When we create encrypted archives through the GUI version, we either check the option to make the password visible normally during the input, or we have to type it twice – this is the classic check against typos in passwords. However, in the command line version where we use the `-p` parameter to set up a password 7-zip behaves differently. We can either specify the password directly during the 7-zip invocation like this: `-pPASSWORD` or we can specify just `-p` by itself. We will then be asked to type the password, but the password isn't echoed and is only typed in once. If we use this together with the `-sdel` parameter which makes 7-zip delete the original data after they are compressed, we might lose the data entirely, if we make a typo in the password.

[24] to encrypt the headers when running console 7-zip, use the switch `-mhe=on`

Conclusion

In this chapter, we will conclude this thesis, summarize what we found out about the 7-zip application and provide suggestions for related exploration in the future.

Summary

In its current form (version 19.00) 7-zip should be sufficiently secure for storage of sensitive data inside its native `.7z` archives, under a few conditions:

- The user has to be sure that his 7-zip installation hasn't been modified by someone else.
- Preferably the user should build 7-zip from source and as described in section 2.3 turn on the usage of salt and possibly even increase the number of iterations performed in the KDF.
- When creating encrypted archives the user should also use the option to encrypt the headers, if possible.

As we can see from the attached results of our cracking attempts, the unmodified version of 7-zip is quite safe by itself. The modified version with a stronger KDF (and possibly also enabled salt) creates encrypted archives which are pretty much uncrackable with current technology and knowledge.

Future exploration

There is still much more that could be analyzed and explored concerning the security of 7-zip. To name a few examples:

- Run fuzzing tools against input fields in the GUI version
- Look for buffer overflows in other parts of the application

- Analyze the crypto implementation for other archive formats, such as `.rar` and `.zip`
- Help improve hashcat by implementing and optimizing the necessary code to support the usage of salt and the weaker KDF in `.7z` archives and possibly archives of larger sizes.
- Try to calculate rainbow tables of a certain size for the default `.7z` KDF
- etc.

Bibliography

1. *Advanced encryption standard (AES)* [online]. 2001. Available from DOI: 10.6028/nist.fips.197.
2. DAEMEN, Joan; RIJMEN, Vincent. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
3. DWORKIN, M J. *Recommendation for block cipher modes of operation: Methods and Techniques* [online]. 2001. Available from DOI: 10.6028/nist.sp.800-38a.
4. PAVLOV, Igor. *7-Zip* [online]. 2019. Available also from: http://www.7-zip.org/. Accessed on 04/24/2019.
5. PAVLOV, Igor. *HISTORY of the 7-Zip* [online]. 2019. Available also from: https://www.7-zip.org/history.txt. Accessed on 04/24/2019.
6. L., David. *7-Zip: From Uninitialized Memory to Remote Code Execution* [online]. philsmd, 2018. Available also from: https://landave.io/2018/05/7-zip-from-uninitialized-memory-to-remote-code-execution/. Accessed on 05/15/2019.
7. PAVLOV, Igor. *7-zip Source code* [software]. SourceForge. Available also from: https://sourceforge.net/projects/sevenzip/files/7-Zip/. Accessed on 05/02/2019.
8. GUERON, Shay. *Intel® Advanced Encryption Standard (AES) New Instructions Set* [online]. 2010. Available also from: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. Accessed on 05/15/2019.
9. ROBERTSON, Colin; B., Michael; JONES, Michael; HOGENSON, Gordon; CAI, Saisang. *Compiler Intrinsics* [online]. 2016. Available also from: https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=vs-2019. Accessed on 05/15/2019.

10. KALISKI, Burt. *PKCS# 5: Password-based cryptography specification version 2.0*. 2000. Available also from: `https://www.rfc-editor.org/rfc/pdfrfc/rfc2898.txt.pdf`. Technical report. RSA Laboratories. Accessed on 05/15/2019.

11. PAVLOV, Igor. *Encryption options* [online]. 2007–2008. Available also from: `https://sourceforge.net/p/sevenzip/discussion/45797/thread/d6924a97/`. Accessed on 05/15/2019.

12. J., Gordon. *Understanding 7z Compression File Format* [online]. 2015. Available also from: `https://docs.google.com/document/d/1-VvYP950eKIC4Yua45CksNgBiZ_eQYKn5u1xj_UdENQ/edit`. Accessed on 05/15/2019.

13. PAVLOV, Igor. *LZMA spec?* [online]. 2004. Available also from: `https://sourceforge.net/p/scoremanager/discussion/457976/thread/c262da00/`. Accessed on 05/15/2019.

14. *hashcat – advanced password recovery* [online]. Hashcat Team. Available also from: `https://hashcat.net/hashcat/`. Accessed on 04/24/2019.

15. RANDOM65537. *Would it be good secure programming practice to overwrite a "sensitive" variable before deleting it?* [online]. 2014. Available also from: `https://security.stackexchange.com/a/74281`. Accessed on 05/15/2019, the author's real name is not known however they are a very active StackExchange (security community) user.

16. HALL, Justin; BICHSEL, Andrea. *Lock pages in memory* [online]. 2017. Available also from: `https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/lock-pages-in-memory`. Accessed on 05/15/2019.

Appendix A

Acronyms

AES Advanced Encryption Standard

AES-NI (Intel) AES New Instructions

CBC Cipher Block Chaining (AES mode of operation)

CRC32 Cyclic redundancy check (32 bytes output length)

CSPRNG Cryptographically secure pseudorandom number generator

CTR Counter (AES mode of operation)

CVE Common Vulnerabilities and Exposures

ECB Electronic Codebook

GPU Graphical processing unit

KDF Key-derivation function

MSVC Microsoft Visual C++

MVS Microsoft Visual Studio

PBKDF Password-based-key-derivation function

RNG Random number generator

SHA Secure Hashing Algorithm

Appendix B

Contents of attached SD card

- `Cracking` ... directory containing example archives for cracking attempts
- `Sources` ... directory containing relevant source codes
  - `7z1805-src` ... 7-zip 18.05 source code + my logger
  - `AES_testing` ... our AES testing program
  - `hashcat` ... hashcat source code
  - `Thesis` ... LaTeX source codes of this thesis
  - `win-iconv` ... win-iconv source code
  - `7z2hashcat.pl` ... the script to help with `.7z` archive cracking
  - `m11600-pure.cl` ... the modified `m11600-pure.cl` – see section 2.4
- `Assignment.pdf` ... the assignment for this thesis
- `BP_Hušek_Josef_2019.pdf` ... this thesis in PDF format