Master System and Network Engineering

University of Amsterdam

Cryptanalysis of, and practical attacks against E-Safenet encryption

Jan Laan jan.laan@os3.nl
Cedric Van Bockhaven cedric.vanbockhaven@os3.nl

January, 2014

Abstract

The Chinese company E-Safenet specializes in data leak prevention and provides software to encrypt files using a proprietary encryption algorithm. Manufacturers are using the E-Safenet encryption to protect their source code. Since several licenses require that code modifications are made public, it must be possible to audit the source code for license compliance. For this purpose it is interesting to be able to decrypt this specific file format.

In this report, the E-Safenet encryption algorithm and data format are detailed. Methods are given to find the encryption key and reverse the encryption using a known-plaintext attack, probable-plaintext attack, and a ciphertext-only attack.

Contents

1 Introduction

Many open source software products can be freely used, as long as their license terms are upheld. The GPL [1] license is an example of an often used licensing scheme. Modifications of source code that falls under the GPL license have to be published freely. Some manufacturers however, do not comply with these terms. European vendors sometimes import devices from other countries, such as China, where license terms are frequently violated [2]. In order to make sure the used software license terms are upheld, an audit of the source code has to be performed.

A China-based company called E-Safenet [3] specializes in data leak prevention, and provides tools for source code encryption. Source code encrypted with E-Safenet's proprietary encryption algorithm cannot be read by third parties, which makes it difficult to verify license compliance. Furthermore, the manufacturer of a product can cease to exist, or cease to provide support. With only the encrypted files available, other parties cannot improve the product anymore by having bugs fixed or features added.

This report details the process of reverse engineering the E-Safenet encryption. Tjaldur Software Governance Solutions, a company that specializes in consultancy in the field of software governance and license compliance engineering [4], made the research into this encryption possible by supplying an archive of encrypted source code files. There are possibly other encryption algorithms that were developed by the E-Safenet company. However, the specific encryption used by these source code files is implied when the term "E-Safenet encryption" is used.

2 Research

This research was supervised by Armijn Hemel of Tjaldur Software Governance Solutions. Tjaldur deals with license compliance issues and encountered the E-Safenet encryption during an audit.

The research into E-Safenet encryption aims to answer the following question:

Can files encrypted with E-Safenet encryption be decrypted?

Research questions that are relevant during the reverse engineering process are:

1. Is it possible to decrypt the given archive of files into readable code given known plaintext?

2. How are E-Safenet keys generated? Are there any repetitions or patterns in the keys?

3. Can arbitrary files be decrypted based on file signatures/patterns instead of plaintext?

3 Related work

There seems to be no prior research into the E-Safenet encryption algorithm at the moment of writing. However, encryption software like E-Safenet that can work transparently on the kernel or network level, has previously been looked into [5] [6].

Bruce Schneier's book on applied cryptography discusses many of the encryption mechanisms that are used by the E-Safenet algorithm. Schneier explains how a simple XOR cipher does not provide any security at all, and is easy to break [7]. A paper has been published on how simple XOR encryption can be broken by making use of n-grams in the English language [8].

Previous research has been done in finding software license violations through binary code clone detection, for which the *Binary Analysis Tool* was created [9]. The *Binary Analysis Tool* makes it easier and cheaper to look inside binary code, find compliance issues, and reduce uncertainty when deploying Free and Open Source Software [10]. Another tool named *DebCheck* was released by Cordy, J.R. and Roy, C.K [11], which also allows to check for open source code clones in software systems.

4 Analysis

Tjaldur Software Governance Solutions provided an archive of encrypted source code files that were encrypted using the E-Safenet encryption algorithm. The encrypted files still had the same filenames as the plaintext version of the files. They appeared to be source files from the U-Boot [12] and Android Linux kernel repositories [13]. These repositories contain mostly C files and header files.

The encrypted files all have the same structured header format in which the literals ”E-SafeNet” and ”LOCK” can be read in plain text. This header format is further documented in section 4.2. As illustrated by figure 1, a hex dump of an E-Safenet encrypted file shows a few header bytes, followed by a range of null-bytes, followed by ciphertext.

```
00000000  62 14 23 65 70 00 90 01  93 86 00 01 45 2d 53 61  |b.#ep.......E-Sa|
00000010  66 65 4e 65 74 00 00 00  4c 4f 43 4b 00 00 00 00  |feNet...LOCK....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  75 a6 4c 50 4b be bd fd  86 42 ba 8a 70 cb df 06  |u.LPK....B..p...|
00000080  ce 42 9c ef a2 cd d0 ae  89 ee 21 b4 ce 35 57 d9  |.B........!..5W.|
00000090  96 65 64 df be 2b 68 25  f5 4d 7b db d6 b8 01 99  |.ed..+h%.M{.....|
```

Figure 1: The first 256 bytes of an E-Safenet file

To find out what cipher was being used by E-Safenet, the encrypted files were subjected to statistical techniques for attacking ciphertext, such as autocorrelation.

4.1 Autocorrelation

To search for patterns in the encrypted files, the autocorrelation of an E-Safenet file was plotted, as shown in figure 2. During autocorrelation, the file is compared with itself at different offsets, and the correlation calculated, in order to identify patterns.

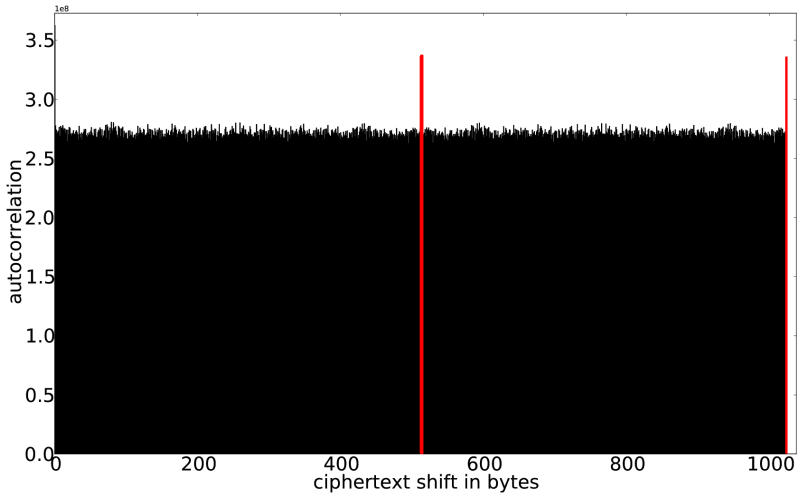

Figure 2: Autocorrelation of an E-Safenet file

The autocorrelation graph clearly shows a spike at values 512 and 1024. Based on these observations, it was assumed that the file was encrypted using a block cipher in ECB mode with a key length of 512 bytes.

When splitting the ciphertext into blocks of 512 bytes, and plotting the byte values, the correlation becomes very visible, as depicted by figure 3. While this figure only shows the results for bytes 480-512, this pattern is visible for all 512 bytes.

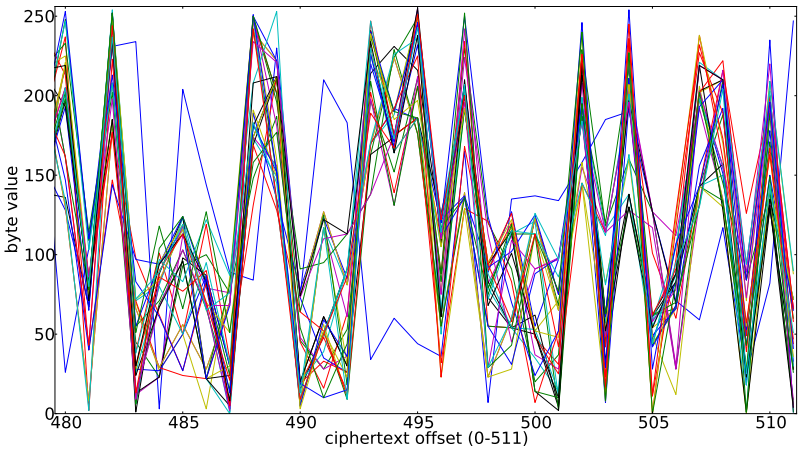

Figure 3: Blocks of 512 bytes of an E-Safenet file. Each color represents one block of 512 bytes.

This pattern emerges because typical C files only contain printable characters, which are in the byte range 32-126, a much smaller range than the 0-255 byte values of random data. In addition, only a subset of these characters is in regular use, namely the lowercase letters, and some special characters such as underscore and parentheses. As a result, characters encrypted with the same key byte result in encrypted bytes that fall within the same range.

Several attacks are proposed in section 5 to extract the encryption key.

4.2 E-Safenet data format

Each encrypted file starts with a custom header. Knowing that a file contains literals in the header that indicate the E-Safenet encryption, and the length of the ciphertext file is still equal to the length of the plaintext file, the only plausible possibility is that compression was used.

After executing a known-plaintext attack on the E-Safenet files, as explained in section 5.1, the plaintext was retrieved. The first block did not contain the actual plaintext, but the expected compressed version. These first 512 bytes are compressed with the LZO algorithm, as is further detailed in section 4.3. After the 21 header bytes and the compressed data, some extra padding is added to total 512 bytes again.

An important remark is that some data cannot be compressed (truly random data), which would result in a failure to preserve the file length. However, it is unclear what happens in this case for E-Safenet files.

The first 6 bytes define where this padding starts and ends, the next 2 bytes contain the size of the compressed data in bytes. Then follow 3 bytes which act as a checksum. These bytes contain the sum of all byte values in plaintext bytes 512-1023. The next byte, byte 11, has an unknown meaning, however its value was always 1 for the files in our possession.

Bytes 12-20 contain the string ”E-SafeNet”, a signature of the company. After this, the padding described above starts. After 3 bytes of padding, the string ”LOCK” is inserted.

At the end of the padding, an encrypted version of the compressed first 512 bytes is stored, followed by an encryption of the rest of the file. Encryption is done by a XOR-operation using a 512-byte key. This key is repeated as often as needed. Note that the compressed data is encrypted with the key starting at byte 0, For plaintext bytes 512 and onwards, the key starts at 0 again and is repeated.

A full overview of each byte and its meaning is shown in table 1.

Byte(s)	Value
0	b (literal string, for Begin)
1	offset, padding starts after this value, always 0x14
2	# (separator?)
3	e (literal string, for End)
4-5	offset, padding ends before this value
6-7	size of compressed data in bytes
8-10	checksum: sum of bytes 512-1023 of the plaintext
11	value 1, meaning unknown
12-20	E-SafeNet (literal string)
21-23	padding, all null-bytes
24-27	LOCK (literal string)
28-xx	null-byte padding, lasts until the value in bytes 4-5
xx-511	encrypted, compressed version of the first 512 plaintext-bytes, xx is the value of bytes 4-5
512-end	encrypted version of the rest of the file

Table 1: E-Safenet data format, all multi-byte numbers are little-endian

4.2.1 Checksum deductions

The checksum that is found in the header of E-Safenet files can sometimes ”leak” information. The checksum is taken over the second 512 bytes block of plaintext.

- The lowest printable ASCII character has an ordinal value of 32.
  $32 * 512 = 0x004000h$
- The highest printable ASCII character has an ordinal value of 126.
  $126 * 512 = 0x00FC00h$

All checksums that fall out of the bounds [0x004000, 0x00FC00] therefore likely indicate binary data, assuming that the file contains at least 1024 bytes. This does not work both ways, as checksums that fall between these bounds do not necessarily indicate printable data.

4.3 Compression

After the decryption of an E-Safenet file, the first 512 bytes are still compressed. When inspecting the hex dump of the compressed data, as shown in figure 4, two important observations can be made. First, when the same data occurs multiple times, the earlier copy is referenced by the following ones, effectively eliminating duplicate strings. Second, duplicate strings do occur after a while, indicating that references can only go back a certain amount of bytes (using a sliding window). This particular way of compressing data is characteristic for Lempel-Ziv based compression [14].

Several Lempel-Ziv based algorithms (LZ77, LZSS, LZJB, LZF, LZO, LZW) were investigated by compressing the same chunk of known plaintext and comparing it with the decrypted, compressed E-Safenet data block. Very similar results were obtained with LZF and LZO, of which LZO will be further discussed in the next section.

```
00000000  00 08 2f 2a 20 43 6f 70  79 72 69 67 68 74 20 53  |../* Copyright S|
00000010  74 61 74 65 6d 65 6e 74  3a 0a 20 2a 48 00 00 01  |tatement:. *H...|
00000020  20 54 68 69 73 20 73 6f  66 74 77 61 72 65 2f 66  | This software/f|
00000030  69 72 6d 60 01 05 20 61  6e 64 20 72 65 6c 50 05  |irm`.. and relP.|
00000040  03 64 20 64 6f 63 75 68  06 0f 61 74 69 6f 6e 20  |.d docuh..ation |
00000050  28 22 4d 65 64 69 61 54  65 6b 20 53 db 06 22 29  |("MediaTek S..")|
00000060  20 50 07 74 09 0c 70 72  6f 74 65 63 74 65 64 20  | P.t..protected |
00000070  75 6e 64 65 72 68 08 07  65 76 61 6e 74 20 63 6f  |underh..evant co|
00000080  70 79 a4 10 02 6c 61 77  73 2e 58 0e 05 65 20 69  |py...laws.X..e i|
00000090  6e 66 6f 72 6d b4 0a 04  63 6f 6e 74 61 69 6e 40  |nform...contain@|
000000a0  07 03 68 65 72 65 69 6e  70 09 40 13 5b 02 66 69  |..hereinp.@.[.fi|
000000b0  64 5b 15 69 61 6c 8c 12  40 0c 09 70 72 69 65 74  |d[.ial..@..priet|
000000c0  61 72 79 20 74 6f 20 27  28 02 01 49 6e 63 2e 60  |ary to '(..Inc.`|
000000d0  04 0f 2f 6f 72 20 69 74  73 20 6c 69 63 65 6e 73  |../or its licens|
000000e0  6f 72 73 2e 64 09 09 57  69 74 68 6f 75 74 20 74  |ors.d..Without t|
000000f0  68 65 20 58 07 4c 04 0c  77 72 69 74 74 65 6e 20  |he X.L..written |
00000100  70 65 72 6d 69 73 73 62  1b 6f 66 28 39 01 69 36  |permissb.of(9.i6|
00000110  39 01 2c 7b 09 61 6e 79  44 1b 4b 11 64 75 63 6c  |9.,{.anyD.K.ducl|
00000120  22 06 2c 20 6d 6f 64 69  66 69 63 8c 19 03 2c 20  |"., modific..., |
00000130  75 73 65 20 4c 0c 06 64  69 73 63 6c 6f 73 75 72  |use L..disclosur|
00000140  54 01 29 6c 01 f4 26 c9  09 64 3b 1d 04 2c 54 03  |T.)l..&..d;..,T.|
00000150  02 20 77 68 6f 6c 4c 08  44 1b 40 16 00 04 61 72  |. wholL.D.@...ar|
00000160  74 2c 20 73 68 61 6c 6c  20 62 65 20 73 74 72 69  |t, shall be stri|
00000170  63 74 6c 79 58 1a 02 6f  68 69 62 69 48 2f 64 1e  |ctlyX..ohibiH/d.|
00000180  09 0a 20 2a 20 4d 65 64  69 61 54 65 6b 11 00 00  |.. * MediaTek...|
```

Figure 4: Decrypted, compressed E-Safenet bytes

4.3.1 Lempel-Ziv-Oberhumer

The LZO (Lempel-Ziv-Oberhumer) algorithm was particularly interesting because it produced compressed data that only differed from the E-Safenet

version in how it referenced previous string occurrences. The choice of which bytes were compressed was the same as in the decrypted compressed E-Safenet data.

The LZO library provides different algorithms (LZO1, LZO1A, LZO1B, LZO1C, LZO1F, LZO1X, LZO1Y, LZO1Z, and LZO2A). Each of these can also be used with different compression levels. None of these algorithms exactly matched the decrypted compressed data. However, LZO1X-1 produced very similar results.

On FTP archives, a total of 13 different LZO versions were found. After making small modifications to 1.x LZO versions, a shared library was compiled for each version. The library was then linked against a C file that outputted the compressed version of some plain text, after which the compression was compared with the compressed header of an E-Safenet file.

LZO1X-1: LZO1X algorithm compression level 1	
Version	Hash of compressed data
lzo-1.00	`d2dc6668f8bb5d2dc6668f8bb5279`
lzo-1.02	`d2dc6668f8bb5d2dc6668f8bb5279`
lzo-1.04	`d323f82fbd201147edb0c7aaa4237`
lzo-1.06	`d323f82fbd201147edb0c7aaa4237`
lzo-1.07	`d323f82fbd201147edb0c7aaa4237`
lzo-1.08	`d323f82fbd201147edb0c7aaa4237`
lzo-2.00	`d323f82fbd201147edb0c7aaa4237`
lzo-2.01	`d323f82fbd201147edb0c7aaa4237`
lzo-2.02	`d323f82fbd201147edb0c7aaa4237`
lzo-2.03	`d323f82fbd201147edb0c7aaa4237`
lzo-2.04	`d323f82fbd201147edb0c7aaa4237`
lzo-2.05	`b3fdf08c375a63637f52e73ccaf7c`
lzo-2.06	`b3fdf08c375a63637f52e73ccaf7c`
E-Safenet	`d2dc6668f8bb5d2dc6668f8bb5279`

Table 2: LZO compression differs among different LZO versions.

Out of the versions that were tested, lzo-1.00 and lzo-1.02 compressed the test data in the exact same way as the E-Safenet file, when using the LZO1X-1 algorithm. Table 2 shows how LZO1X-1 compression produces different output for different LZO versions. Now that the exact compression algorithm was known, the full plaintext of E-Safenet files could be recovered.

Choosing LZO as compression library may indicate that the authors of E-Safenet had performance in mind. LZO outperforms most compression algorithms in speed, although at the cost of compression ratio [15].

4.4 The E-Safenet company

To further understand the E-Safenet file format, it is desirable to know the origin and reasoning behind its existence. The E-Safenet files are of Chinese origin. This claim is validated by the fact that many of the encrypted files contain Chinese text, were found on Chinese websites, and by the existence of a Chinese company called E-Safenet [3].

This company provides multiple solutions for data leak prevention. An interesting product is UltraSec, which promises to be an easy to use file encryption platform. Important is the mention of real time and transparent encryption, which explains the algorithms focus on performance. In normal use, the UltraSec files are plaintext, but when stored/saved they are encrypted transparently. UltraSec is marketed to high-tech, embedded and mobile software development, which fits the source code files which were found.

Another product is SmartSec, which does something similar as UltraSec. It provides transparent encryption and decryption of documents in the same way. This would explain the found Microsoft Excel and Word files encrypted using the E-Safenet format.

We believe that UltraSec and SmartSec use the E-Safenet file format which is the target of our research. This claim, however, could not be validated. It is possible that these products use another method, or that the E-Safenet encryption method is used by other software of this company.

5 Attacks

Three different types of attacks were constructed for E-Safenet encrypted files. Implementations of these attacks can be found in the source code repository belonging to this paper. See Appendix C.

5.1 Known-plaintext attack

Thanks to the archive of E-Safenet encrypted source code supplied by Tjaldur, a working known-plaintext attack could be constructed. For this attack, 512 bytes of the plaintext are `xor`ed with the corresponding fragment of ciphertext. This immediately yields the encryption key. This key can then be used to decrypt the entire file, or files using the same encryption key. The first data block still has to be decompressed to complete the decryption process, as described in section 4.3. A python implementation for the known-plaintext attack is given in Appendix B.

5.2 Probable-plaintext attack

Known-plaintext decryption of E-Safenet files was shown to be easy. However, known plaintext is not always available. A probable-plaintext attack can be constructed based on expected patterns in the encrypted text, in order to find the encryption key. This method requires the definition of specific patterns for each file type however.

A pattern is a small piece of probable plaintext, which is expected to occur in the targeted file(s). For example, in a C file, the keyword `return` is expected to occur often.

After defining a set of patterns, a part of the encrypted file is decrypted using these patterns one by one. Start with the first pattern at the start of the encrypted text, and XOR the pattern with the encrypted text. The result is assumed to be part of the key. These key bytes are then applied at future offsets in the 512 bytes blocks of the ciphertext. If printable characters[1] are found after applying these key bytes, this part of the key will be assumed to be correct. Then, the pattern is `xor`ed starting with the second byte of the encrypted text, and so on. This process is repeated until the end of the encrypted text, and for all defined patterns.

[1] A printable character is any character with a decimal ASCII value in the range 32-126. Here, 9 (tab), 10 (LF) and 13 (CR) were included as well.

Each pattern will result in a set of key bytes. Where the key bytes overlap, the most occurring byte value is chosen as correct key byte. Combining all the key bytes should result in the complete key, given enough data.

This method is especially useful in programming languages, as they are highly structured, and have frequently occurring (longer) patterns. A natural language, such as English, also has frequently occurring patterns, but these are usually smaller words (n-grams), resulting in less key bytes found per match, and a larger chance of false positives.

5.2.1 A probable-plaintext attack for source code files

Partial decryption, without known plaintext, was first attempted with two different file types: C programming language files, and PHP files. Both are programming languages and are highly structured. A list of common patterns was defined for these languages [16], as shown in table 3. Finding the key for another plaintext format can be done by changing the patterns to frequently occurring patterns in that specific file format.

(a) Probable PHP patterns

this
array(
function_
return_
return;
public
<?php
class_
false
true
null
_____
_____
Copyright

(b) Probable C patterns

return_
return;
#include_
#DEFINE_
#define_
#IFDEF
static void
const char
const
char *
struct
extern
static
_____
_____
malloc
printf
sprintf
stdio.h
Copyright
GNU General Public License

Table 3: Probable plaintext patterns

5.2.2 A probable-plaintext attack for binary files

For binary data there is no easy way to check if a pattern results in a correct decryption, as the method of finding printable characters at future offsets cannot be applied in this case. Some binary files, however, have other interesting characteristics. Files using Microsoft's MS-XML format for instance (the old Microsoft Excel, Powerpoint and Word format) contain many null-byte sequences. Null-bytes are often used as padding in binary file formats [17]. Possibly, many software implementations create data structures of a fixed size, which initialize the data to null-bytes due to underlying calls to the *calloc* function in C.

By searching for common substrings over different blocks in the encrypted text, and then assuming them as null-bytes, a probable cipherkey can be extracted.

The longest of the found common substrings are then combined, until the entire cipherkey can be covered. This is illustrated by figure 5. The result of this can immediately be assumed as the correct key, since `xor`ing with 0 has no effect.

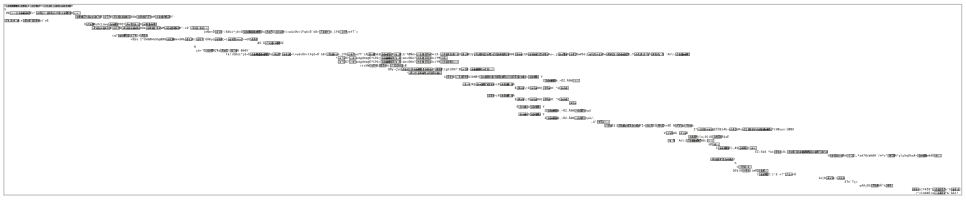

Figure 5: Common substrings spread over the entire key size

It is interesting to note that if the binary file has a checksum of 0x000000, the second block of ciphertext can directly be assumed as the cipherkey. A checksum of 0x000000 can only be obtained if this second block of plaintext bytes contains null-bytes only, as explained in section 4.2.1.

5.2.3 Results

Several decryption attempts were performed using the probable-plaintext attack for source code described in section 5.2.1. The results of the attack on various amounts of encrypted data are shown in table 4. Finding the entire correct key was successful with as few as 17 kB of encrypted data for source code files. However, results varied greatly, even within files of the

same programming language. More data will in general result in a greater chance of obtaining the entire correct key, however, chance plays a large role in the amount of data needed.

For binary files, the test data was limited to files that could publicly be found, being Excel and Word documents.

(a) C source code files

Amount of data (kB)	22	34	36	49	53	60	60	66	78
% of key found	51	100	39	98	100	100	99	100	100

(b) PHP source code files

Amount of data (kB)	4	10	17	27	30	33	47	90
% of key found	43	97	100	97	100	99	100	100

(c) Binary files

File type	xls	xls	xls	docx	docx	docx
Amount of data (kB)	19	27	183	49	51	1605
% of key found	100	100	100	96	100	100

Table 4: Data needed for probable-plaintext decryption

5.3 Ciphertext-only attack

There are cases in which the file is not structured, does not contain patterns, or the file type is unknown. Hence, no probable-plaintext attack can be constructed. For instance, this may happen for encrypted text documents that contain a natural language.

A strategy for gaining more insight into the contents of the files is to choose the cipherkey in a way that the resulting plaintext bytes fall as much as possible in an expected, predefined range. E.g. for text documents, the cipherkey can be chosen in a way that the number of printable characters[1] in the plaintext will be maximized.

This attack may reveal recognizable patterns or keywords in the plaintext. In the best case, the key can be recovered entirely. The result can also be used to create a more general, faster probable-plaintext attack.

The model for this optimization problem is constructed as follows. An OPL implementation for CPLEX[2] can be found in the source code repository (see Appendix C).

[2]IBM ILOG CPLEX Optimization Studio: MIP solver

Data:

C_i : encrypted ciphertext bytes
$C_{i,0..7}$: encrypted ciphertext bits (boolean)

Decision variables:

K_i : key bytes [0, 512) P_i : plaintext bytes
$K_{i,0..7}$: key bits (boolean) $P_{i,0..7}$: plaintext bits (boolean)

Objective:

$$max \sum_{i}^{P} 32 \leq P_i \leq 126 \tag{1}$$

Subject to:

$$P_{i,b} = C_{i,b} + K_{(i \ mod \ 512),b} \ (mod \ 2) \qquad \forall i \in C, \forall b \in [0,7] \tag{2}$$

$$C_{i,b} = \left\lfloor \frac{C_i}{2^b} \right\rfloor \ (mod \ 2) \qquad \forall i \in C, \forall b \in [0,7] \tag{3}$$

In this model, (2) means that each ciphertext bit is `xor`ed with the corresponding key bit, resulting in a plaintext bit. In modulo 2, this can be represented as an addition.

The amount of printable characters is then maximized, in accordance with (1). The range $[32, 126]$ is assumed for printable characters, although 9d (horizontal tab) 10d (new line) and 13d (carriage return) are also often occuring and may be included in this range.

Furthermore, (3) is provided as translation constraint between the binary and the bit representation.

5.3.1 Practical implementation

A practical implementation for this model can be developed as follows.

First, the data is chopped into sections of 512 bytes. These bytes are then grouped together (all first bytes of all sections, all second bytes, etc.). Each group is `xor`ed with all 256 possible key values (K_i) and a record is kept of how many of the resulting plaintext bytes P_i are in the printable range. The most likely key byte is the one with the most printable characters found during this process, K_i.

This process is repeated 512 times to get all 512 bytes of the key. In only $256 \times 512 = 1 \times 10^6$ attempts, the most likely key is found.

Because not all possible characters occur in every printable text, the chance that all K_i are actually the real key value, i.e. the amount of data needed for this to succeed, varies per file type. However, when the file type is known, a more specific attack can be performed, as shown in section 5.2.

5.3.2 Results

Results vary using this method. Applying it on a number of C files, totalling 220 kB of data, returned the entire correct encryption key. When this method was applied on a number of PHP files, totalling 440 kB, the entire key was not found. Most bytes of the 512-byte key were found correctly, however, some bytes were wrong, but still near the correct value. For example, the incorrectly decrypted text showed strings such as `Lepser Geneqal Publjc License`, `$this.>macros` and `Ex`el_Style+true`. For these cases, most characters are correct, and some are only off by a few characters. It is easy to correct this, given some knowledge about the language used. The strings in the example can be fixed to show `Lesser General Public License`, `$this->macros` and `Excel_Style=true`. By fixing these strings manually, all 512 bytes of the key can be found. Of course, each wrong offset only needs to be fixed once, after which the correct character will show in every instance of that offset.

6 Keys

It is interesting to know how E-Safenet encryption keys are generated, and how they correlate. If multiple keys are similar in some way, this information can be used for decryption of other E-safenet files. If it is known how the keys are being generated, creating new, valid keys becomes a possibility.

In order to find out more about the keys, they will be placed next to each other, and analyzed to see if there are similarities.

For this purpose, there are seven keys available, retrieved from various E-Safenet files and archives. The keys were not available immediately, they were calculated from the decryption of the files. This is a relatively small number of keys for this kind of analysis, therefore expectations were low.

6.1 Key correlation

As expected, no real correlation between the keys could be found. There were no overlapping sequences of bytes, or bytes which were the same for all keys. The frequency distribution for bytes throughout the keys was very random.

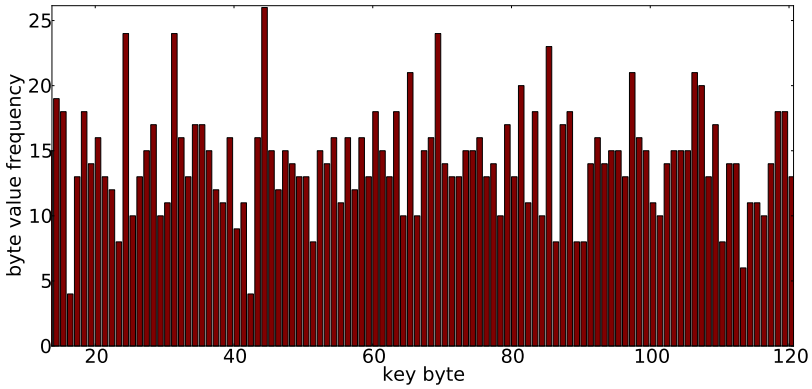

Figure 6: Typical excerpt from key values. This shows that the value of key bytes appear to be random.

As shown in figure 6, every byte occurred in the keys. On average each byte should occur 14 times. (There are 256 possible values in seven 512-byte keys. $\frac{7\times512}{256} = 14$) Due to the small number of keys, this could not be seen, however the graph does show results near that number for every byte, with

all bytes occurring between 5 and 25 times. Figure 7 shows that subsequent keys do not have any bytes in common.

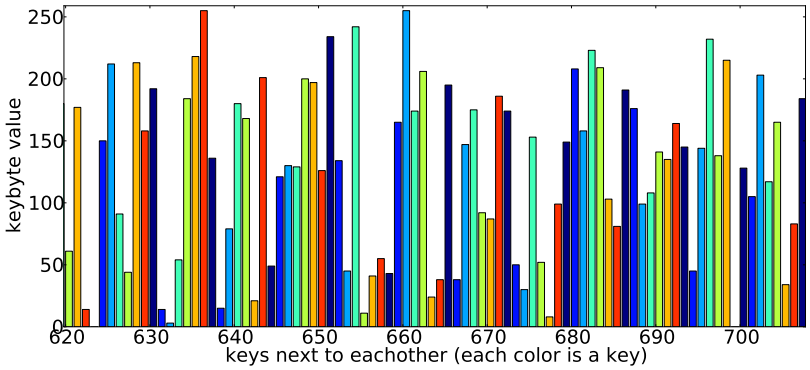

Figure 7: Typical excerpt from key values. This shows that different keys share no same key bytes between them.

No relation could be found between the obtained E-Safenet keys. They could be random, created by some key derivation method, or obtained by some other method.

7 Suggestions for improvement

The E-Safenet algorithm is a very fast, but insecure encryption method. Nothing is known about the authors' true intentions and design goals, however it seems that performance was a major factor. The XOR encryption is extremely fast, and the chosen LZO compression algorithm favors performance over compression ratio. Even with these design goals in mind, improvements can be made. In this section, some possible improvements are discussed.

The XOR encryption is extremely weak. No suitable encryption method, however will outperform it. As a replacement, another cipher is needed. Stream ciphers have generally better performance than block ciphers. A secure modern stream cipher that performs well is Salsa20/12 [18]. While not as fast as the XOR encryption, Salsa20/12 was designed to have good performance. Its performance is good enough for almost any application, and provides proper security [19].

The first 512 bytes of the data are compressed in order to keep the encrypted file length, including E-Safenet header, the same as the file length of the plaintext file. This header of the encrypted file contains readable information about the plaintext file, such as the checksum of bytes 512-1023 and the length of the compressed data. If keeping the same file length is not a requirement, this metadata should be omitted, as it leaks information about the plaintext file.

If both keeping the same file length after encryption, and adding text such as `E-Safenet` in front of the encrypted file are required, for whatever reason, compression is a good option, with the observation that truly random data cannot be compressed, resulting in a failure to preserve the file length.

At least the 3-byte checksum could be omitted. The checksum is of little value, as it only validates the second 512 bytes of the data. As documented in section 4.2.1, it could leak information about the plaintext.

8 Conclusion

The E-Safenet encryption as described in this research uses a very weak XOR cipher. It can be easily broken. In this research, three methods to obtain the decryption/encryption key have been shown, each with its own merits.

When known plaintext is available, finding the key is trivial (5.1). Without known plaintext, but with knowledge of the file format, the key can be obtained with relatively little data using a probable-plaintext attack (5.2). Finally, when nothing is known about the encrypted data, maximizing the printable characters in the plaintext can reveal useful information, as demonstrated by the ciphertext-only attack (5.3). Either a probable-plaintext attack can be constructed from the result, or the encryption key can be derived immediately, besides some manual work. It is unknown how the encryption keys are generated.

The range of presented attack methods show the weakness of the encryption. The encryption is an excellent example of the statement "Don't roll your own crypto". Therefore it is unsuitable to securely store data. A better algorithm is needed for this purpose.

9 Further work

The E-Safenet format has been analyzed, and the meaning of all but one byte in the file header is known, however this single byte does not seem to be of any significance. It was possible to decrypt most available files, however some other file formats may prove more difficult. This can be solved by either obtaining more data encrypted with the same key, or by improving on the used decryption/key finding methods.

The compressed first 512 bytes are not used at all in finding the key. However, they do contain a great amount of readable data, even after compression. This data can be exploited to aid in finding the encryption key.

The presented probable-plaintext attack works only for specific file types, and a list of probable patterns must be defined for every plaintext file type. For binary files, the probable-plaintext attack is heavily dependent on large amounts of the same byte value occurring after each other in an encrypted file.

It is still unknown how the 512 byte E-Safenet keys are being generated. Analysis revealed no similarities between the obtained keys. Perhaps they are generated by some key derivation function, or simply randomly generated and stored. More keys are needed to thoroughly analyze this.

It is unknown what happens with the E-Safenet file if the first 512 bytes cannot be compressed. If this occurs, there is no room for the E-Safenet header without changing the file size or compressing another part of the data.

10 References

[1] GNU General Public License. http://www.gnu.org/licenses/gpl.html.

[2] Nilay Patel. Open Source and China: Inverting Copyright. *Wis. Int'l LJ*, 23:781, 2005.

[3] E-Safenet. http://www.esafenet.com.

[4] Tjaldur Software Governance Solutions. http://www.tjaldur.nl/.

[5] Yang Li, Li Yeli, and Zheng Liangbin. Transparent encryption based on network file system filtering driver. In *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, pages 6339–6342, April 2011.

[6] Jinxin Ma, Zhoujun Li, and Jia Li. A novel secure virtual storage device scheme. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 271–275, Oct 2010.

[7] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[8] Alexander Griffing. Solving XOR Plaintext Strings with the Viterbi Algorithm. *Cryptologia*, 30(3):258–265, 2006.

[9] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011.

[10] Binary Analysis Tool. http://www.binaryanalysis.org/en/home.

[11] J.R. Cordy and C.K. Roy. Debcheck: Efficient checking for open source code clones in software systems. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 217–218, June 2011.

[12] Das U-Boot – the Universal Boot Loader. http://www.denx.de/wiki/U-Boot.

[13] Android Samsung 2.6.35 Gingerbread Git repository. https://android.googlesource.com/kernel/samsung/+/android-samsung-2.6.35-gingerbread.

[14] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337, 1977.

[15] Quick Benchmark: Gzip vs Bzip2 vs LZMA vs XZ vs LZ4 vs LZO. http://pokecraft.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO.

[16] Kenta Motomura. The Most Frequent Word in Source Code on GitHub.

[17] Martin Karresand and Nahid Shahmehri. Oscar — file type identification of binary data in disk clusters and ram pages. In *Security and Privacy in Dynamic Environments*, volume 201 of *IFIP International Federation for Information Processing*, pages 413–424. Springer US, 2006.

[18] Christophe Cannière. eSTREAM Software Performance. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 119–139. Springer Berlin Heidelberg, 2008.

[19] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: analysis of salsa, chacha, and rumba. In *Fast Software Encryption*, pages 470–488. Springer, 2008.

Appendix A: Definitions

ECB Electronic codebook cipher mode: the message is divided into blocks, and each block is encrypted separately.

LZO Lempel-Ziv-Oberhumer compression: lossless compression algorithm.

ASCII American Standard Code for Information Interchange: encoding scheme based on the English alphabet. In this report, the 8-bit variant is assumed.

GPL GNU General Public License: Often used licensing scheme for open-source software. See [1].

Known-plaintext attack: an attack where both the plaintext and corresponding ciphertext are known.

Probable-plaintext attack: an attack based on plaintext that likely, but not necessarily occurs in an encrypted file.

Ciphertext-only attack: an attack that relies solely on a set of known ciphertexts.

Appendix B: Decryption code

```
1  import simplelzo1x
2
3  """
4  Decrypt a given text, with the key.
5  Returns the file's text
6  """
7  def decrypt_file(text, key):
8          plain = ""
9          # offset is stored in these 2 bytes in little-endian order.
10         offset = ord(text[4]) | ord(text[5]) << 8
11         decr_header = __xor_with_key(text[offset:512], key)
12         plain_header = simplelzo1x.decompress(decr_header)
13         plain_file = __xor_with_key(text[512:], key)
14
15         return plain_header + plain_file
16
17 """
18 Xor any text with a given key.
19 The key should be an array of byte values.
20 An Esafenet key has a length of 512 bytes.
21 """
22 def __xor_with_key(text, key):
23         xored = ""
24         for idx, c in enumerate(text):
25                 xored += chr(ord(c) ^ key[idx % len(key)])
26         return xored
```

Appendix C: Full source code

The full source code of the encryption/decryption programs created, as well as some scripts to perform analysis on E-Safenet files, can be found on GitHub at https://github.com/c3c/E-Safenet. This includes practical implementations of all attack methods.