Black-box cryptanalysis of home-made encryption algorithms: a practical case study

Pierre Capillon
pierre.capillon@ssi.gouv.fr

Agence Nationale de la Sécurité des Systèmes d'Information
(French Network and Information Security Agency – ANSSI)

Abstract. Product vendors sometimes develop their own cryptographic algorithms to either protect their intellectual property or ensure information confidentiality (data or communications). Many of these algorithms have been proven to contain critical weaknesses which defeat their purpose, weaken security and might expose customer data or systems.
Most research on home-made algorithms is usually done through reverse-engineering of the hardware or software parts implementing these cryptographic primitives. This article tackles a different approach on an unknown and simple algorithm.
During the study of an embedded system firmware, the author was not allowed to tamper with the targeted product. With hardware-based attacks or research out of the equation, this article proposes a first-hand account on how black-box cryptanalysis was performed on a custom algorithm in order to retrieve a 40 MB firmware from an 18 MB compressed and encrypted image.
The author discloses approaches, methods, ideas and tools developed throughout the part-time 6-weeks process, and discusses explored ideas and attacks which proved to be either successful or dead-ends.

Acknowledgement. This work was conducted in 2014, along with Fabrice Desclaux and Camille Mougey who have provided invaluable help in identifying some key data fields. Cédric Bouhier and Nicolas Vivet have also provided significant contributions to the reverse engineering and implementation of the compression algorithm.

1 Introduction

1.1 Disclaimer

While initially describing the study of obscure systems, *cryptanalysis* [9] (as a discipline) has a strong mathematical connotation. This paper does not tackle much of the mathematical challenges the cryptanalysis of modern cryptography poses. It is rather a collection of considerations and

techniques used to attack an unknown encryption/scrambling algorithm (an index table may be found at section 6.3).

Therefore, in the context of this article, *cryptanalysis* shall be viewed as the historical discipline of retrieving cleartexts, whatever the means and however weak the algorithm is. Do not expect a revolutionary attack on a complex cryptosystem, rather a glimpse of a thought process.

The focus of this article is to provide a first-hand account on the effort such a cryptanalysis requires, as well as dismiss claims that attacking custom-made cryptography is infeasible or requires extraordinary resources.

As such, while identifying details were left out, the algorithm principles have been preserved.

1.2 State of the art in black-box cryptanalysis

Nowadays, black-box cryptanalysis is usually performed on ciphertext *with prior knowledge of the algorithm*. This is mainly due to the various reverse-engineering opportunities of available products, hardware or software. Well-known examples include:

- digital rights management (DVD, etc.);
- access control systems (various NFC technologies);
- Wi-Fi encryption (WEP, WPA+TKIP);
- cellular communications (A5/1 and A5/2 stream ciphers).

Many other examples exist where proprietary cryptography was studied and successfully attacked [7]. This also includes the recovery of firmware images [1].

However, all of these examples involved access to the algorithm details before an attack could be performed on ciphertexts.

Historically, ciphertext-only attacks with no prior access to algorithm details have been devised by rival governments when eavesdropping foreign diplomatic communications. While access to algorithm details would still be seeked through various intelligence means, it was not uncommon to attack foreign cryptography on the basis of intercepted communications only. Unlike the well-known case of the Enigma machine, cryptanalysis of the German Lorenz cipher during World War II was made without access to the physical machine [8]. At approximately the same time, the Japanese Purple code was also broken by the american National Security Agency without needing access to the device itself. A second attempt to break the Purple code with modern techniques was even made by the NSA more recently [5].

Similarly, early paid television services have had their scrambling mechanism attacked. Some of their scrambling algorithms have even been cryptanalyzed without having access to a legitimate decoder or algorithm details [2][3].

This article is an account of a similar attempt on present-day secret proprietary algorithm used for firmware encryption.

1.3 Embedded systems security

Security research is common on current embedded products in order to find and patch vulnerabilities. Such a work may or may not be the result of a collaborative effort with the product vendor. As such, different amounts of information are available to the researcher, who in turn tries to get access to as many information as possible, possibly up to design documents and source codes.

In many cases, source code access is either not possible for multiple reasons or simply not desirable. The researcher often turns to black box analysis to:

- study device communications, *via* traffic captures and protocol fuzzing;
- study its internal mechanisms, *via* reverse-engineering of the device firmware.

To protect those, sometimes vendors still consider the development and use of secret, unproven, home-made scrambling systems or cryptography a worthy investment. Beyond firmware encryption/intellectual property protection, it is not uncommon to find this kind of algorithm in proprietary communication protocols. The rising trend in « smart » devices (the « Internet of Things ») brings its round of similar attempts at bogus security measures. Performance considerations worsen the problem, as vendors try to implement « optimized » algorithms to run decently on limited hardware resources.

1.4 Context

As part of security research on a particular embedded system, a team of several people split the task between various efforts, one being focused on the firmware itself.

To obtain the firmware image, multiple paths may be considered. Here are the main options:

- asking the vendor;

– downloading an image file hoping it is not scrambled;
– retrieving the firmware legitimately from the device;
– exploiting a software vulnerability to extract or study the firmware from the device;
– exploiting an hardware vulnerability to extract or study the firmware from the device.

Unfortunately, none of those are available:

– no proprietary information is to be exchanged;
– the device is lent for a short duration by the manufacturer;
– the device cannot be tampered with;
– the device should be returned in pristine and working condition.

These restrictions rule out any kind of hardware manipulations, such as extracting firmware data directly from the NAND flash chips. Yet this specific manipulation was considered, but deemed too risky after seeing how the SoC, flash chips and contact pads are protected under a heavily glued heatsink (see figure 1).

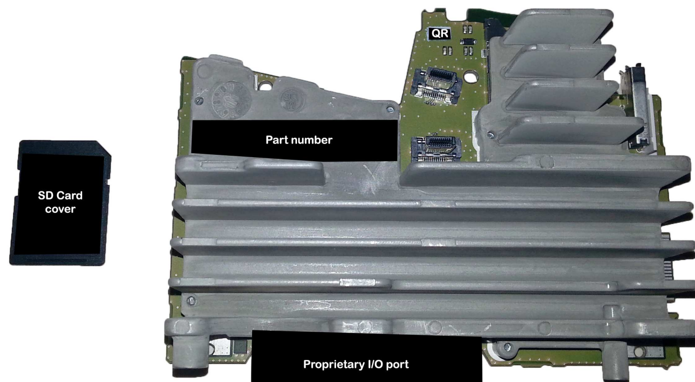

Fig. 1. The target system PCB with a prominent, glued heatsink. Undocumented visible connectors provide no useful signals.

After assembling the unit back together, members of the team go on with fuzzing it, while the author focuses on retrieving the firmware by other means.

For the sake of this article, the different product models can be grouped as follows:

- M1, M2, M3, M5 and M6 form model group A;
- M4, M7 and above form model group B.

Models from the latter group have significant differences in the hardware platform. The available unit is model M6, with later tests done on model M3, whose differences are irrelevant to this paper.

Firmware updates images are available for all models, with five revisions at the time of the study (including a major revision). Downgrading the device to an older revision is officially supported, and the original firmware revision is also available for download. Images for group A are about 18 MB in size, while those for group B are about 50 MB.

It is then decided to have a look at these firmware upgrade files as they stand a good chance of containing the full image.

Unfortunately, these images do not load into reverse-engineering tools as they seem to be encrypted.

1.5 Specific constraints

Modern techniques often involve knowledge of the attacked algorithm and use some of its properties to retrieve information about the plaintext. This information is usually retrieved through reverse-engineering of the software or hardware parts implementing the target algorithm. This approach is not possible due to our constraints.

This paper thus focuses on ideas, approaches, methods and tools developed while attacking the encryption with no prior knowledge:

- the algorithm is entirely unknown;
- the target platform architecture is unknown;
- the hardware configuration is unknown (including possible security modules);
- the software platform seems to include open-source libraries (SSL libraries), as observed by using the product;
- no remote code execution was achieved on the device yet, excluding dynamic analysis or firmware extraction;
- no oracle is available to get feedback from the firmware update process.

The product could very well run an embedded Linux distribution on a general-purpose CPU as well as a bare-bone proprietary operating system on a custom-designed system-on-chip.

1.6 Initial problem and resources considerations

While the practical case on which this article is based ends up being a success, readers shall keep in mind the following considerations:

- the ideas developed in this paper are probably not suitable for strong cryptographic algorithms: a detail feels dangerously off from the beginning, hinting at a weak algorithm;
- even with a weak algorithm, there is no guarantee of succeeding in recovering a plaintext image;
- no knowledgeable help could be obtained;
- *what is observable does not necessarily reflect the algorithm designer's intent*: a very complex proprietary algorithm may result in observed properties which were not initially expected.

A particular emphasis is made on the last consideration.

To achieve decryption, the following resources were used:

- six weeks of part-time research amounting to probably 3 weeks of full-time work on the project;
- a single man effort, with contributions from two colleagues;
- a single desktop workstation for computations;
- lots of sweet and caffeine in various forms.

These weeks of effort had a noticeable effect on the author's health and sanity, with sleeping and eating disorders occurring over an extended period of time.

2 Exploring file formats

The first step in analyzing firmware updates is figuring out the file format:

- Is it a binary file? Is it executed on the device?
- Is it a filesystem? Is it well-known or custom-made?
- Is it compressed? Is it encrypted?
- Does it contain metadata? A digital signature? Parameters?

Immediate actions are to quickly check whether obtaining a usable binary file looks feasible. Two main steps help achieve that:

- entropy analysis of every firmware update available, for all models in case some differences showed up;
- file format and header reconstruction, which helps postulate various hypotheses which would then be tested.

2.1 Entropy

Entropy analysis [10] is known to help discriminate various types of data (text, binary code, images), as well as their possible status (plaintext, encrypted, compressed). The compression algorithm may sometimes be identified by this method [4].

Figure 2 shows the result on firmware updates from both product model groups.

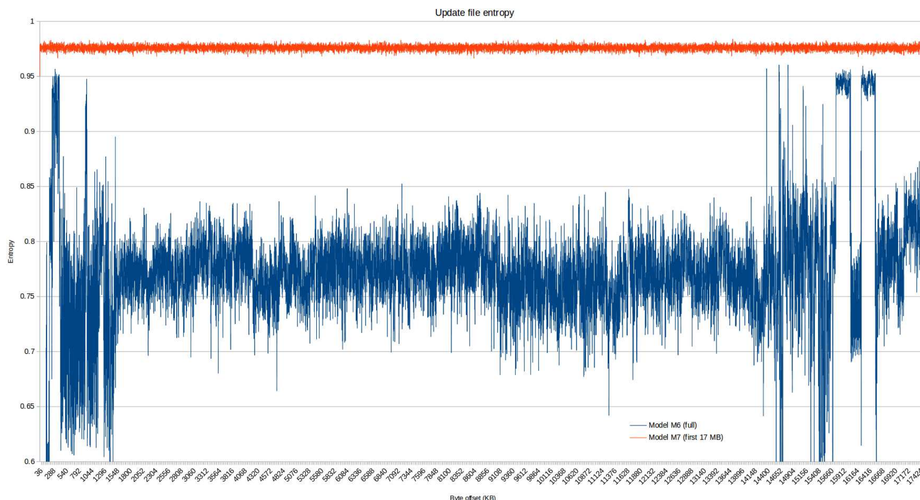

Fig. 2. *Binwalk* entropy analysis on the first 17 MB of the update file for model M6 (group A, bottom) and M7 (group B, top).

Group B models exhibit a very high and uniform entropy average (around 0.98), possibly indicating the use of strong cryptography or compression. However, group A models exhibit a far lower average entropy (0.78) with clear fluctuations. Two areas near the end of the file have a higher entropy, possibly revealing the use of compression or encryption on some parts of the firmware. Also, the beginning and the end of the file show more fluctuations than the largest area between the 10% and 80% filelength marks.

These fluctuations could indicate that more diverse data types are located at these locations, while a single type lies in the largest chunk.

Also, a 0.78 average entropy could be characteristic of executable code, which may suggest that the firmware is not strongly encrypted.

2.2 Header identification

Opening the firmware update in any hexadecimal editor reveals plaintext data, suggesting the presence of a formatted header (see figure 3).

```
./fw-c.01.bin
0000 0000: 04 00 00 00 00 00 00 00  00 00 00 00 56 CC CC 01  ........ ....V...
0000 0010: 4D 4F 44 45 4C 2D 4D 36  2D 56 45 52 53 49 4F 4E  MODEL-M6 -VERSION
0000 0020: 2D 30 31 20 00 00 00 00  00 00 00 00 20 00 00 00  -01 .... .... ...
0000 0030: A3 D2 FD FF 46 49 45 4C  44 31 00 00 00 00 FF FF  ....FIEL D1......
0000 0040: FF FF 46 49 45 4C 44 32  D2 92 13 01 16 B1 9F 8F  ..FIELD2 ........
0000 0050: 46 49 45 4C 44 33 48 00  00 00 1C B4 F0 FF 46 49  FIELD3H. ......FI
0000 0060: 45 4C 44 34 46 49 45 4C  44 31 01 00 EF 00 00 00  ELD4FIEL D1......
0000 0070: 00 00 4D 4F 44 45 4C 2D  4D 36 2D 56 45 52 53 49  ..MODEL- M6-VERSI
0000 0080: 4F 4E 2D 30 31 20 56 00  00 00 46 49 45 4C 44 32  ON-01 V. ..FIELD2
0000 0090: 46 49 45 4C 44 33 10 02  43 6F 70 79 72 69 67 68  FIELD3.. Copyrigh
0000 00A0: 74 20 4D 41 4E 55 46 41  43 54 55 52 45 52 20 49  t MANUFA CTURER I
0000 00B0: 44 20 2D 20 32 30 31 34  20 20 20 20 E0 BF 04 3C  D - 2014     ...<
0000 00C0: C8 02 85 34 32 00 03 24  32 00 02 3C 00 00 A3 AC  ...42..$ 2..<....
0000 00D0: EC 03 86 34 E0 03 42 34  E4 03 84 34 10 00 03 24  ...4..B4 ...4...$
0000 00E0: A0 BF 05 3C 00 00 82 AC  5C 00 A5 34 00 00 C3 AC  ...<.... \..4....
0000 00F0: 00 00 A2 8C 00 FF 03 24  C0 BF 19 3C 24 10 43 00  .......$ ...<$.C.
0000 0100: 55 00 42 34 21 20 00 00  00 00 A2 AC 18 0E 39 37  U.B4! .. ......97
0000 0110: 08 00 20 03 00 00 00 00  3F 3D 3E 41 3A 3D 41 3A  .. ..... ?=>A:=A:
0000 0120: 3F 40 2D 3E 3F 47 3D 3D  47 3D 3D 1A 17 5A 4E 5B  ?@->?G== G==..ZN[
0000 0130: 62 53 4E 50 61 62 5F 52  5F 2D 4F 62 56 59 51 2D  bSNPab_R _-ObVYQ-
0000 0140: 60 66 60 1A 17 53 76 7B  6E 79 2D 5F 72 79 72 6E  `f`..Sv{ ny-_ryrn
0000 0150: 80 72 2D 4F 82 76 79 71  2D 2D 2D 2D 2D 2D 2D 2D  .r-O.vyq --------
0000 0160: 2D 2D 2D 2D 2D 2D 2D 2D  83 F5 EA 21 50 07 02 00  -------- ...!P...
0000 0170: 22 40 AA 11 06 1E 36 95  33 75 83 5A AB C7 DF 50  "@....6. 3u.Z...P
0000 0180: C6 C6 C9 D4 95 D3 88 B3  06 8C 4D 4C 19 0A A8 AE  ........ ..ML....
0000 0190: A3 FB 37 3F 38 8A 11 01  ED B2 CD C7 DC 0C 17 4A  ..7?8... .......J
0000 01A0: EB 26 72 95 77 FC D3 4B  9C 90 60 99 2A E0 FE B2  .&r.w..K ..`.*...
0000 01B0: BC 0D F9 5D 8B 17 8E DB  D7 E6 C9 1F 00 00 00 00  ...].... ........
0000 01C0: E3 B0 C4 42 98 FC 1C 14  9A FB F4 C8 99 6F B9 24  ...B.... .....o.$
0000 01D0: 27 AE 41 E4 64 9B 93 4C  A4 95 99 1B 78 52 B8 55  '.A.d..L ....xR.U
0000 01E0: 30 00 00 00 18 00 00 00  20 00 00 00 02 40 22 92  0.......  ....@".
0000 01F0: 10 00 00 00 C0 01 10 38  32 FA 2B 62 00 08 08 08  .......8 2.+b....
```

Fig. 3. Header fields identified visually. Chunk sizes and hashes can be seen overlayed in green and red (darkest colors).

The relevant model part number is mentionned a few times, as well as a copyright string, both in plaintext form. They might help the update program determine whether the proper update is being flashed to the device.

Also, four 6-character field names have two occurrences each. Their first occurrence is preceded by 8 bytes for each. Four of these bytes turn out to be the size of the data section following their second occurrence.

The first two data sections contain short plaintext data or are simply empty. The last section is identified as a signature, sits at the end of the firmware update and is sufficiently long to hold a MAC value, but not much more. The bulk of the firmware is thus located in the third data storage section, which begins immediately.

2.3 Main data storage section

The main data section starts with 92 unknown bytes, yet they do not vary across firmware revisions nor models. The following section contains weird ASCII values, ending with a long string of 0x2D values. This section is actually a build timestamp/release string obfuscated with a ROT13 pass. The actual purpose of the ROT13 use is still unknown.

Right after this ROT13 section, 120 bytes of data look much different from the next ones, which have far more null values that the rest of the header. Among those 120 bytes, a sharp eye can identify the 32-byte SHA256 hash value of an empty string (`e3b0c44298fc1c14 9afbf4c8996fb924 27ae41e4649b934c a495991b7852b855`). The four bytes immediately preceding the hash are NULL, which may reveal a new size field.

As three SHA256 hashes fit in 120 bytes, with 24 bytes left, there is room for a 32-bit size field and an additional 4-byte value. Indeed, the other 4-byte values fit the description of a size field: adding all three fields gives precisely the size in bytes of the remainder of the main data section.

Therefore, these three data chunks seem to be stored in the main data section of the firmware update file, with the last chunk being empty. Notably, this chunk of data is not empty among models from group B.

Unfortunately, those SHA256 hashes do not match any subset of the stored data.

From this manual analysis, many parts of the firmware update header can be reconstructed. Figure 4 shows the resulting documentation. A few fields are identified by reverse-engineering the firmware update application, but the software does not parse them further than the first 144 bytes (0x90), while some can be identified further along. Parsing of those fields is probably done on the device itself.

Identified fields are then compared across product models and firmware revisions, to help mark fields whose values are either static or changing. Some fields have their value depend solely on the firmware revision, sharing it across models. Some other fields have their value change completely between models, even from the same group.

2.4 Hypotheses and initial thoughts

A summary of firmware update c.01 for model M6 is shown in figure 5.

Two interesting data chunks are identified: a very small one, and a large one occupying the bulk of the firmware update file. The latter is likely to contain the firmware image.

Format

Fields **in bold** exhibit varying values depending only on chunk #1 or chunk #2 content.

v01.xx.yy																
Offset	***Byte***															
	0 **0**	**1** **1**	**2** **2**	**3** **3**	**4** **4**	**5** **5**	**6** **6**	**7** **7**	**8** **8**	**9** **9**	**10** **A**	**11** **B**	**12** **C**	**13** **D**	**14** **E**	**15** **F**
0x00000000	(0x04)	(0x00)	(0x00)		(0x00)		(0x00)		(0x00)				0x56 ('V')	Version (BCD)		
0x00000010	Article Number (string)															
0x00000020	Article Number (string)				(0x00)				(0x00)				size (LE)			
0x00000030	Unknown				KEYWORD ("FIELD1")						Size (LE)				Unknown	
0x00000040	Unknown		KEYWORD ("FIELD2")						**size (LE)**				**Unknown**			
0x00000050	KEYWORD ("FIELD3")						size (LE)				**Unknown**				KEYWORD ("FI")	
0x00000060	KEYWORD ("ELD4")				"FIELD1"						0x01	0x00	0xEF	0x00	0x00	
0x00000070	0x00		Article Number (string)													
0x00000080	Article Number (string)						Version (BCD)				"FIELD2"					
0x00000090	"FIELD3"						0x10	0x02	Copyright (string)							
0x000000A0	Copyright (string)															
0x000000B0	Copyright (string)												Unknown			
0x000000C0	Unknown															
0x000000D0	Unknown															
0x000000E0	Unknown															
0x000000F0	Unknown															
0x00000100	Unknown															
0x00000110	Unknown								Build string (ROT13 string)							
0x00000120	Build string (ROT13 string)															
0x00000130	Build string (ROT13 string)															
0x00000140	Build string (ROT13 string)															
0x00000150	Build string (ROT13 string)															
0x00000160	Build string (ROT13 string)								Unknown				**chunk #1 size (little endian)**			
0x00000170	**SHA256 chunk #1 cleartext?**															
0x00000180	**SHA256 chunk #1 cleartext?**															
0x00000190	Unknown				**chunk #2 size (little endian)**				**SHA256 chunk #2 cleartext?**							
0x000001A0	**SHA256 chunk #2 cleartext?**															
0x000001B0	**SHA256 chunk #2 cleartext?**								Unknown				chunk #3 (little endian)			
0x000001C0	SHA256 chunk #3 cleartext? (chunk is not present)															
0x000001D0	SHA256 chunk #3 cleartext? (chunk is not present)															
0x000001E0	... **chunk #1 data** ...															
0x1E0 + chunk #1 size	... **chunk #2 data** ...															
0x1E0 + chunk #1 size + chunk #2 size EOF - 0x4d (78 bytes)	"FIELD4"						0x00	0x00	0x00	0x01	0x2F	0xEC	0xE9	0x02	**Unknown**	
EOF - 0x3d (62 bytes)	**Unknown**															
EOF - 0x2d (46 bytes)	**Unknown**															
EOF - 0x1d (30 bytes)	**Unknown**															
EOF - 0x0d (14 bytes)	**Unknown**															

Fig. 4. Header fields as retrieved from the manual file analysis. Bold text highlights variable contents between revisions and models.

Analysis with automated tools, such as *binwalk*, does not identify a known compression algorithm or data type.

Consequently, the recovery efforts then focus on firmware updates for models from group A, as they do not seem strongly encrypted.

At this point, a few hypotheses are formulated on the identified data chunks:

- one of the chunks must contain executable code, as well as other plaintext data we should obtain by using the device (GUI strings, web pages);

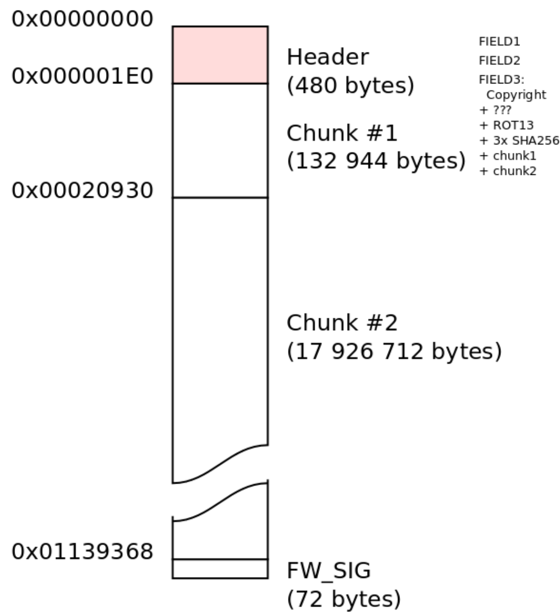

Fig. 5. Firmware update file format summary. Firmware image is expected to be found within chunk #2.

- the chunks could be flashed « as-is », as there seems to be strong integrity mechanisms in place;
- identified SHA256 hashes must correspond to the output after properly processing data chunks, indicative of either (or both) compression or encryption;
- downgrading ability, format similarities and varying header values across firmware revisions must mean the update file is self-sufficient, and that the updater software is backward compatible.

We were then left with two data chunks, of which we assumed to have the SHA256 values of their corresponding plaintext. The recovery efforts then focus on each chunk, as recovering the smallest one could eventually help recovering the largest chunk.

Up to this point, the recovery efforts lasted for a single day.

3 Initial intuition, trial and errors

3.1 Finding code patterns

Not knowing whether the binary data is in plaintext, compressed, or somehow encrypted form, a colleague suggested to try and match code patterns.

Finding instruction opcodes A statistical analysis can be performed on byte groups of varying length, to match the results to those of typical CPU architecture instruction sets.

This task is quite difficult, especially with variable-length instruction sets.

Although it was briefly explored, initial results proved too uncertain to properly match any given architecture.

Function calls Byte sequences related to function calls/returns or function prologue/epilogues can be searched for, and matched against those of known architectures. A match would also help identify the underlying CPU architecture. One would find roughly the same proportion of call instructions than that of return instructions. The same goes for prologue/epilogue patterns (matching combinations of push/pop instructions).

While data entropy is somewhat similar to that of binary code from various architectures, this approach proved unsuccessful.

3.2 One more hypothesis

After failed attempts at directly identifying executable code, a new hypothesis is tested.

The plaintext data is likely to be either some form of binary executable code, some data files, or parameters. Such data is likely to be structured: common executable formats have a well-defined header, whose structure and values do not vary that much between compilations. The same is true for other data types.

Therefore, we are likely to find a format header or container structure which probably does not vary much between firmware updates.

3.3 Initial discovery

After the firmware update header, data from chunk #1 immediately shows a lot of null values, as can be seen in figure 6: chunk #1 data starts at `0x1E0`.

Additionally, most bytes have a very low number of bits set. This seems to confirm the previous hypothesis that data might be structured and have bitfield values or flags stored first.

```
./fw-c.00.bin
0000 01C0: E3 B0 C4 42 98 FC 1C 14  9A FB F4 C8 99 6F B9 24  ...B.... .....o.$
0000 01D0: 27 AE 41 E4 64 9B 93 4C  A4 95 99 1B 78 52 B8 55  '.A.d..L ....xR.U
0000 01E0: 00 00 80 01 00 00 C0 00  00 00 00 01 12 91 14 00  ........ ........
0000 01F0: 00 00 80 00 80 C0 01 0E  5F 11 93 D1 40 40 00 40  ........ _...@@.@
0000 0200: 00 00 00 00 00 00 00 00  00 00 02 80 00 00 00 40  ........ .......@
0000 0210: BE 40 E8 22 0D 00 00 00  6A 81 00 00 00 04 00 5C  .@.".... j......\
0000 0220: 0D 00 08 00 00 00 0A C0  02 40 02 80 00 00 00 40  ........ .@.....@
0000 0230: 00 40 00 00 00 00 00 20  00 00 00 20 C8 00 00 00  .@.....  ... ....
0000 0240: C8 00 00 00 01 00 00 40  00 40 00 00 00 00 00 40  .......@ .@.....@
0000 0250: 00 7C 00 00 00 3C E8 22  00 3C E8 22 D4 04 00 00  .|...<." .<."....
0000 0260: D4 04 00 00 01 00 00 C0  00 40 00 00 00 00 00 40  ........ .@.....@
./fw-c.01.bin
0000 01C0: E3 B0 C4 42 98 FC 1C 14  9A FB F4 C8 99 6F B9 24  ...B.... .....o.$
0000 01D0: 27 AE 41 E4 64 9B 93 4C  A4 95 99 1B 78 52 B8 55  '.A.d..L ....xR.U
0000 01E0: 30 00 00 00 18 00 00 00  20 00 00 00 02 40 22 92  0.......  ....@".
0000 01F0: 10 00 00 00 C0 01 10 38  32 FA 2B 62 00 08 08 08  .......8 2.+b....
0000 0200: 00 00 00 00 00 00 00 00  00 10 00 40 00 08 00 00  ........ ...@....
0000 0210: 5D C4 17 08 00 A0 01 00  00 40 2D 10 80 0B 80 00  ]....... .@-.....
0000 0220: 01 A0 01 00 01 18 00 40  00 50 00 48 00 08 00 00  .......@ .P.H....
0000 0230: 00 00 00 08 00 04 00 00  00 04 00 00 00 00 19 00  ........ ........
0000 0240: 00 00 19 00 00 28 00 00  00 00 00 08 00 08 00 00  .....(.. ........
0000 0250: 00 00 80 0F 5D 04 80 07  5D 04 80 07 00 80 9A 00  ....]... ].......
0000 0260: 00 80 9A 00 00 38 00 00  00 00 00 08 00 08 00 00  .....8.. ........
```

Fig. 6. Chunk #1 binary data differences between consecutive minor revisions of model M6. Note how the same number of bits are set in each 4-byte word.

Binary differences between two consecutive minor revisions reveal another interesting detail: there is the same number of bits set for each corresponding 4-byte group.

This observation leads to a major discovery: if the stored data is actually structured, it is unlikely that the format markers differ by much. Bits should then be stored at the same position. Since it is not the case, there are yet precisely the same number of bits set for each 4-byte group, **the content has probably been scrambled at least by a bitwise rotation of some sort**.

This is first verified manually by calculating bitwise rotations differences on little-endian 4-byte words, as shown on figure 7.

So the scrambling system is likely to use bitwise rotations to obfuscate data from chunk #1, as we are able to get perfect matches of the first data bytes by applying a specific rotation to an update revision compared to another.

Up to this point, two days of effort had been spent in the analysis.

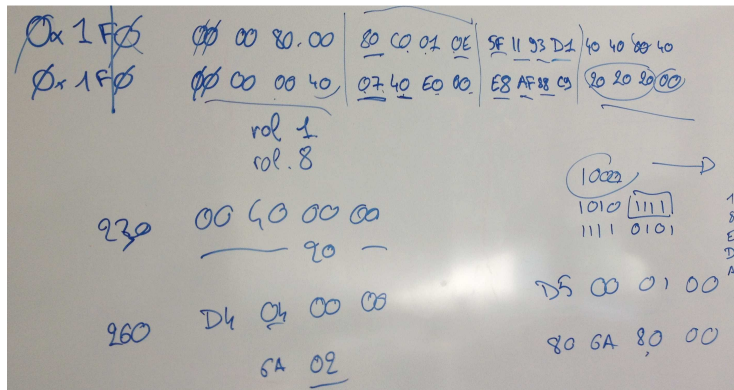

Fig. 7. Whiteboard with initial discovery of bit-shifting similarities across firmware revisions.

3.4 Comparing update revisions

The previous finding means that one might be able to match some if not all data from an update revision to another, by simply applying the same bitwise rotation to each 4-byte word.

Comparing SHA256 hashes from the update headers between various models and revisions reveals that chunk #1 data could be the same within the same major firmware revision. Figure 8 shows that all minor revisions of the b.xx and c.xx branches respectively have identical hashes.

Model	Version	chunk #1 SHA256	chunk #2 SHA256
M3	a.01	302e7a968837c8e947a0838072e090cd31decdf43d214ce49159eb8c4e696749	7fa8e2f14f03dafb1b490e5e9433ecb403ae6c9d85afb1d04e126c3b2cc19b3d
M3	b.00	f8e7175e9468a32c93dd6fafd349ea2080b7a9eb5a355ac6688451139ccfb9a3	75860f5f64ac5c00bf821096515536ac6d2ee8a60fd883969fd277d3198abfb7
M3	b.01	f8e7175e9468a32c93dd6fafd349ea2080b7a9eb5a355ac6688451139ccfb9a3	0ece3c7b50c8970f96026e0109de951fba570700a42a48e1e4e45916de74f263
M3	b.02	f8e7175e9468a32c93dd6fafd349ea2080b7a9eb5a355ac6688451139ccfb9a3	7e812b3b993321b44c749e54d2c7ba25623b58c0921d6b095738cbe59ed13950
M3	c.00	2240aa11061e36953375835aabc7df50c6c6c9d495d388b3068c4d4c190aa8ae	4d8afeba9d3fc325212743874d78ff41bf9b0e5ce61ae3cbd4cb94da9354c726
M3	c.01	2240aa11061e36953375835aabc7df50c6c6c9d495d388b3068c4d4c190aa8ae	edb2cdc7dc0c174aeb26729577fcd34b9c9060992ae0feb2bc0df95d8b178edb
–			
M6	a.01	3ca0a9187f17d3d548ba59a6fe742eff2360b5329285056d6aec59d99e2e341f	f60c382ae3f88b7ddc8f39bd6240a2d17a47fef9ca42cac8002d4eef7f2f32dd
M6	b.00	2c084c2ca46b9df64a3ed9b8346f47020ce98e6b33895d5c0ea8865d1b92075d	3782f7fb4561b9522e590788574029766858 83fb2dbe89f84b2eab7e15258581
M6	b.01	2c084c2ca46b9df64a3ed9b8346f47020ce98e6b33895d5c0ea8865d1b92075d	5dc6b540499e800d8ee2905c61486c82ae41cc3ceb7bccbbdec1e5fab24cfc10
M6	b.02	2c084c2ca46b9df64a3ed9b8346f47020ce98e6b33895d5c0ea8865d1b92075d	915b44a142fb73d7096274603f058cbf2087f48f257a2d9f6e752f009cf9009e
M6	c.00	31d0569f058654ef501820b978e55b3b7ef33fece1964d5ff6d992d48c627cf6	f32c8e97273f7b555e3a8f71c149ef90e4304408942cfc03004df6d0b20f3306
M6	c.01	31d0569f058654ef501820b978e55b3b7ef33fece1964d5ff6d992d48c627cf6	a317ae03851805d9e8cbe723d154b804bcb8bd956ab6ca88a8848574733807fa

Fig. 8. SHA256 hashes found in headers from update files for different versions and models.

Therefore, if our earlier hypothesis—that SHA256 hashes are those of cleartext data—is true, it should be possible to perfectly match chunk #1

data found among those minor revisions, both in cleartext and encrypted form.

Ciphertext match does not necessarily mean the encryption has been broken yet, as the bitwise rotation may be applied as a final pass. However, if this is the only pass, it is then possible to fully decrypt the data. In any case, being able to match ciphertext from an update to another can still prove useful.

Figure 9 shows what the decryption approach would be in such an event.

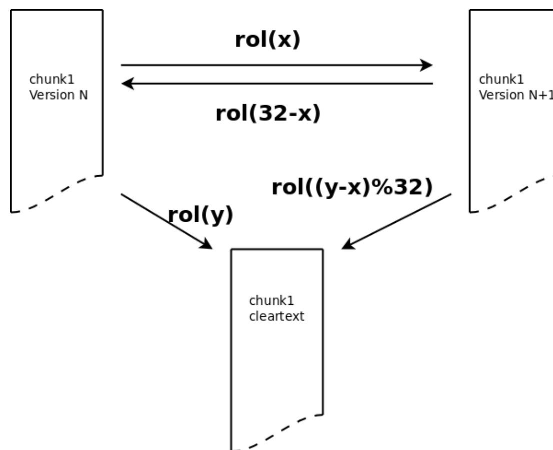

Fig. 9. Decryption approach for chunk #1 in the event bitwise rotation is the only scrambling pass.

Unfortunately, applying the same bitwise rotation to the full chunk #1 data only yields a partial match: data ceases to match after an arbitrary length (see figure 10).

The bitwise rotation distance changes after a given length, hence revealing that the scrambling process works with 4-byte aligned blocks of arbitrary sizes. Repeating the bitwise rotation allows identification of 4 blocks of different sizes with different rotation parameters for the full chunk.

Note that the first two blocks include large amounts of consecutive NULL bytes. As these NULL bytes are insensitive to bitwise rotation, it is impossible to determine how many blocks span across those.

```
c1-vc.00
0001 EFC0: 2E 0E F0 84 00 90 9E 40  00 94 9E 50 C0 8F 91 00  .......@ ...P....
0001 EFD0: E0 83 01 24 00 88 BE 42  00 02 01 84 2E 0A F0 84  ...$...B ........
0001 EFE0: D5 0B 91 F0 E0 03 01 24  FF 4F 90 E8 FE 03 58 28  .......$ .O....X(
0001 EFF0: 2E 0A F0 C4 C0 13 31 12  00 88 3E 52 E0 03 01 24  ......1. ..>R...$
0001 F000: FF 4F 90 C8 F3 A0 2B 8C  FF 16 F0 80 2E 0A F0 C4  .O....+. ........
0001 F010: C0 17 31 02 FB 0A F0 BC  BD 27 A8 FF 88 8C 00 00  ..1..... .'......
0001 F020: B0 AF 48 00 B0 27 10 00  B2 AF 50 00 B1 AF 4C 00  ..H..'.. ..P...L.
0001 F030: BF AF 54 00 80 00 21 88  A0 00 21 90 00 02 21 20  ..T...!. ..!...!
0001 F040: 00 00 21 28 06 24 34 00  00 01 09 F8 07 24 34 00  ..!(.$4. .....$4.
0001 F050: A3 93 10 00 02 24 7F 00  62 10 09 00 A3 93 11 00  .....$.. b.......
c1-vc.01
0001 EFC0: 2E 0E F0 84 00 90 9E 40  00 94 9E 50 C0 8F 91 00  .......@ ...P....
0001 EFD0: E0 83 01 24 00 88 BE 42  00 02 01 84 2E 0A F0 84  ...$...B ........
0001 EFE0: D5 0B 91 F0 E0 03 01 24  FF 4F 90 E8 FE 03 58 28  .......$ .O....X(
0001 EFF0: 2E 0A F0 C4 C0 13 31 12  00 88 3E 52 E0 03 01 24  ......1. ..>R...$
0001 F000: FF 4F 90 C8 F3 A0 2B 8C  FF 16 F0 80 2E 0A F0 C4  .O....+. ........
0001 F010: C0 17 31 02 FB 0A F0 BC  FE F7 9E A0 00 20 32 02  ..1..... ..... 2.
0001 F020: 01 C0 BE 22 00 C0 9E 40  01 C8 BE 42 01 C4 BE 32  ..."...@ ...B...2
0001 F030: 01 FC BE 52 20 02 02 84  40 82 02 84 80 00 08 84  ...R ... @.......
0001 F040: A0 00 00 84 00 18 90 D0  E0 03 04 24 00 1C 90 D0  ........ ...$....
0001 F050: 00 8C 4E 42 01 08 90 FC  00 88 41 24 00 8C 4E 46  ..NB.... ..A$..NF
```

Fig. 10. Block boundary detection in chunk #1 at `0x1F018` due to a sudden content mismatch.

3.5 Bruteforce

Testing all 31 possible bitwise rotations on chunk #1 finally reveals some plaintext, indicating that no other scrambling has been applied to data. The first block reveals the ELF magic value, while the last block reveals ELF section names and some compilation string artefacts.

As shown in figure 11, two blocks are left with no identifiable plaintext in any of their rotated counterpart. This leaves $31^2 = 961$ combinations (compared to the initial $31^4 = 923521$ combinations), allowing a bruteforce attack on chunk #1 to recover the plaintext. Candidate plaintexts would be tested against the SHA256 hash found in the update header, granted our initial hypothesis is correct.

All 961 candidates are generated with their SHA256 hash, and a match is found for chunk #1, validating all previously stated hypotheses.

A valid ELF binary follows a small, custom header. This also reveals that the underlying architecture is a MIPS CPU.

3.6 Chunk #1 analysis

Reverse-engineering the recovered chunk #1 ELF reveals interesting details. The binary has three main modes of operation:

- parsing headers with a format matching the one found before the ELF magic;
- uncompressing a blob to an arbitrary memory location, with a variant of an LZ compression algorithm;

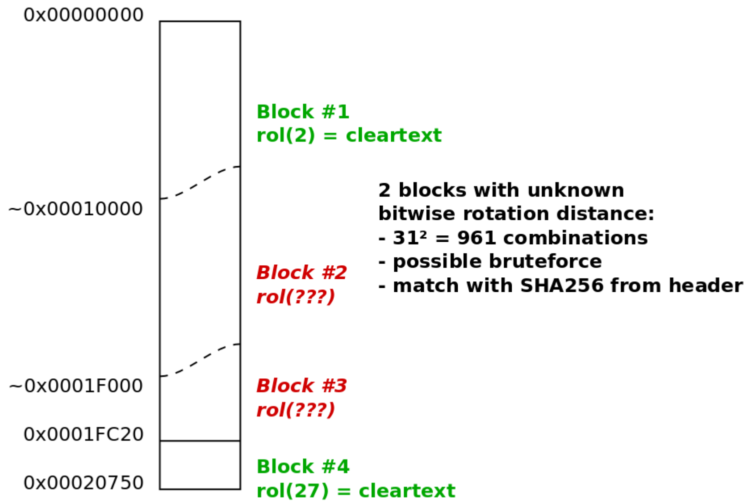

Fig. 11. Block decryption bruteforce for chunk #1.

– acting as an ELF image loader and jumping to its entry point.

Oddly enough, it looks like this binary is executed on the device and gets the decrypted chunk #2 as an input, decompresses and runs it. The binary could act as a bootloader and be used to load the firmware image into memory.

The compression algorithm is then reimplemented to mimic the binary implementation, including corner cases and error conditions. However, the chunk #2 data needs to be fully decrypted before being able to fully uncompress it.

3.7 Trial and error

While bruteforce was an option for chunk #1 (even with the initial 923521 combinations), this seems a very unlikely possibility on chunk #2 due to the potentially large number of blocks. A rough estimate gives more than 2000 blocks with the observed mean block size of 8 KB.

Chunk #2 also displays a valid ELF header at the beginning, after a similar custom header. However, identifiable data quickly surfaces in the few kilobytes immediately after the ELF header:

– large data areas padded with `0xFF` values;
– an embedded filesystem image, with structures similar to directory entries;

- corrupted web content (HTML pages and GIF images).

Binwalk successfully identifies a very limited number of images, though they are split across multiple storage blocks of the filesystem and are not yet fully recoverable.

The chunk #2 data is still compressed at this point. Due to how LZ compression works, early bytes will probably be left unaffected before the compression dictionnary fills with enough context. However, control bytes are inserted every 8 bytes, corrupting contents.

Quickly enough, the compression mechanism is sufficiently active to deter naive recovery attempts. Also, block boundaries for the encryption algorithm become less and less clear as binary data is much more difficult to identify. It is then required to recover all block boundaries as well as the rotation distance for each of these blocks before being able to uncompress the firmware.

As could be expected, contrary to the chunk #1 data, no two firmware updates share identical SHA256 hashes. Identifying block boundaries is not feasible using the same comparison technique.

About two weeks of part-time work had been spent on recovering the chunk #1. The next sections will cover how the actual recovery was performed on the 18 MB chunk #2.

4 Hypotheses, heuristics and validation

Two major parameters are still unknown:

- how the bitwise rotation distance is chosen;
- how the block size is determined.

Given the size of the chunk #2 data, these parameters cannot be guessed by bruteforce alone.

4.1 Refining hypotheses and using them to find strong heuristics

Given the previous findings regarding the chunk #1 executable, one can safely assume compressed, cleartext chunk #2 data is processed by a similar (if not identical) binary running on the device itself. The following hypotheses may then be formulated:

- decryption must occur before any further processing;
- downgrade support means the decryption binary must either apply a common algorithm for every revision or identify the target revision;

- firmware update files must be self-sufficient to yield correct decryption parameters.

Careful comparison between update revisions and across different models reveals the following facts:

- rotation distance varies based on version number only, as the same values are used for chunk #1 data across different models;
- block sizes vary by both version number and model; however as no identical SHA256 is found across models, the latter may be unimportant;
- block sizes do not vary between minor firmware revisions for the same chunk #1 data.

Therefore, block sizes and rotation distances could be determined by:

- two hard-coded, data- and version-dependent lists of values;
- a value generation algorithm seeded by information within the update file header;
- the cleartext data itself, or a value derived from the cleartext data (possibly its hash value).

It is highly unlikely that hardcoded lists of values are used for each version, as they should be either determined from the start of the product lifecycle, or pushed in advance for upcoming upgrades by the previous ones. This process seems very unlikely for a device with both downgrade support and the ability to skip major revisions.

The next efforts are focused on recovering how both block sizes and rotation distances are determined. At that point, only a handful of these values have been determined: 4 blocks for chunk #1 data, and only 2 blocks for chunk #2 data.

4.2 Expanding the sample size

To get a better idea of the various block sizes and rotation parameters, it is useful to gather as much actual values as possible, sorted by revision number and model.

Fortunately, chunk #1 data for model M1 is 560 KB long. As multiple versions share the same data, the same comparison technique can be used between minor revisions to identify block boundaries.

Within two minor revisions 22 and 24 blocks can be identified, with confirmed lengths varying between 128 and 15712 bytes (blocks starting or ending in byte arrays neutral to bitwise rotation are ignored as their

boundary cannot be precisely determined). No obvious correlation pattern can be identified with such a small sample size.

Rotation parameters to fully decrypt chunk #1 cannot not be brute-forced. Indeed, only five blocks are recovered based on string values, leaving 17 to 19 unknown parameters, far too much to guess accurately. Again, the sample size for both block size and rotation parameter is insufficient to detect any noticeable correlation.

As the comparison method cannot be reliably used on chunk #2 data because of varying contents between revisions, other ways of confirming block boundaries are developed.

4.3 Instrumenting compression

The decompression algorithm will fail if it encounters certain particular invalid control bytes or empty dictionary entries.

This allows using the decompression algorithm to validate its own input: incoherent data will not be decompressed correctly. This implies:

- bitwise rotation on 4-byte words will corrupt or shift control bytes at incorrect positions;
- decompression dictionary will slowly fill with incorrect contextual data, leading to decompression errors later on, sometimes very far from the current position, depending on data duplicity.

If wrong block sizes or rotation parameters are guessed, decompression is likely to fail. This fact can be exploited to validate guessed values, by trying to uncompress the data and having the decompression routine display offsets where errors have been encountered.

4.4 Finding multiple reliable checks

Block boundaries will end in the middle of stored data, which may be used as a validity check:

- if an executable binary is located around suspected block boundaries, disassembling instructions should yield a coherent output, rather than corrupted or uncommon instructions;
- if a filesystem is discovered, its structure could be reverse-engineered to help validate coherence of guessed block boundaries and rotation parameters;
- if files of known format are discovered, they can be used to validate the decompression output.

Various data are found that could be used to validate block boundaries after guessing block size and bitwise rotation distance:

- a data structure first attributed to a NAND flash image format (however, this turned out to be misleading);
- a proprietary filesystem derived from FAT-like structures;
- a few web pages;
- a few PNG and GIF images;
- multiple X.509 certificates for HTTPS support.

This « almost known » cleartext data helps provide reliable checks to confirm candidate block boundaries and rotation parameters: either check for consistency or simply compare with a copy obtained from the embedded web server. GIF files, web pages and certificates are used in that regard.

4.5 Unreliable data

A block boundary ends up within an SSL certificate for a different model, which is still present in the image[1]. Due to the random nature of such data, and unavailability of the relevant model, it is impossible to fully validate this block boundary.

Furthermore, the embedded filesystem image seems to contain forensic artefacts of deleted data within the web root, which is confusing as these files could not be retrieved by accessing the web server (as if the filesystem image had been generated and updated on an actual device).

These uncertainties lead to the decompression progressing or failing in similar ways given multiple different candidate block boundaries. This quickly increased combinatorial complexity.

This type of error becomes more and more common as the decompression progresses through binary input data which is not easily identifiable/verifiable. Indeed, the decompression dictionary quickly fills, leaving less and less empty entries, which results in less and less decompression failures (even from incorrect data).

4.6 Combining steps to discard branches

As described in figure 12, the recovery process outline then becomes as follows:

[1] The fact that generic SSL certificates are shared and hardcoded is a worrying fact on its own.

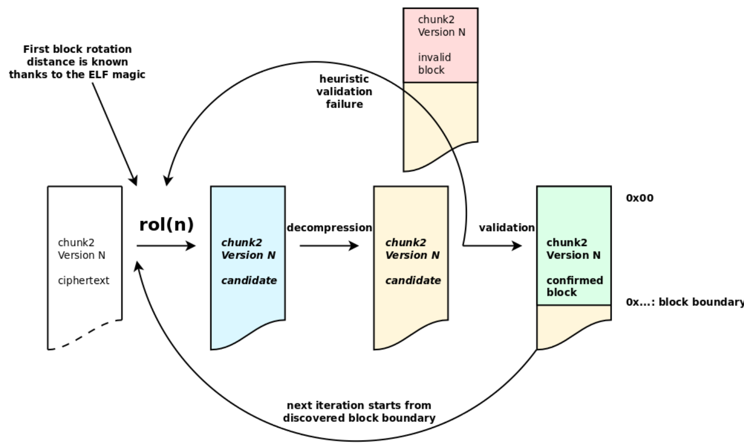

Fig. 12. Chunk #2 decryption approach.

1. guess rotation parameter for the first block, using the ELF magic value, or bruteforce the rotation distance ;
2. run instrumented decompression on candidate cleartext chunk. Decompression failure occurs at or after the actual block boundary: decompression should then have advanced significantly if the parameters were guessed correctly;
3. check validation heuristics on candidate uncompressed chunk;
4. try the next candidate rotation parameter or block boundary if validation fails. Go on to the next block if validation passed.

Advancing decompression along with guessing attempts provides the following benefits:

- bitwise rotation distance is bruteforced for each subsequent block sequentially instead of trying every combination for multiple consecutive blocks. Doing so contained combinatorial explosion within acceptable limits;
- compression control codes may be checked for coherence before attempting decompression, effectively reducing the number of combinations;
- decompression can be attempted on multiple candidate block boundaries, with candidates advancing the output further likely being the correct ones;
- multiple, different heuristic checks may be tested against the last few uncompressed candidates.

This approach limits computational requirements progressively by running the most CPU-intensive tasks on less candidates as they are being discarded along the way.

4.7 Initial results

This approach yields very interesting results. As shown in figure 13, multiple block boundaries and rotation parameters can be retrieved.

It becomes evident that the rotation distance comes from a cyclic list of 24 values. This list is shared across all revisions and models. Only the starting index differs and it is determined by the version number within the update file header.

This discovery enables full decryption of model M1 chunk #1, which ends up having no helpful difference with respect to chunk #2 data recovery.

At this point however, block boundaries do not seem to be shared between models or revisions, nor do they look cyclic.

Uncertainties in block boundaries, as well as decompressed data ambiguity lead to finding new ways of validating our guesses, as discussed in subsection 4.9.

Version n+1 Block offset	Version n+1 Bit rotation (rol)	Version n Bit rotation (rol)	Version n Block offset
0x00000000	19	21	0x00000000
0x00000080	6	19	0x00001BA0
0x000030A0	9	6	0x00005120
0x00004BCC	15	9	0x00006468
0x00007974	29	15	0x0000918C
0x0000B858	10	29	0x0000C850
0x0000D04C	26	10	0x0000F7A0
0x0000D9A4	5	26	0x00010E60
0x00010228	11	5	0x00013FAC
0x000114E0	14	11	0x00014AE8
0x00014A10	23	14	0x00018784
0x00016D64	3	23	0x00018B50
0x00018F94	31	3	0x00018F80
0x000191AC	22	31	0x000193F4
0x00019DC4	1	22	0x0001B864
0x0001C294	18	1	0x0001BAC4
0x0001F100	27	18	0x0001EAEC
0x00020FFC	7	27	0x0001FA08
0x000231C4	25	7	0x00022EE0
0x00023400	30	25	0x00026A58
0x00024420	17	30	0x00027610
0x00027528	13	17	0x00028E94
0x00029C34	2	13	0x0002B040
0x0002DA9C	21	2	0x0002CA78
0x0002FE8C	***19***	***21***	0x00030800
0x00030C98	***6***	***19***	0x00032FD4
0x00031964	***9***	***6***	0x00033650
0x00035638	***15***	***9***	0x00034AC0
0x00039214	***29***	***15***	0x00036C30
0x0003B9B0	***10***	***29***	0x00036E54

Fig. 13. Chunk #2 identified block boundaries and repeating bitwise rotation pattern.

4.8 Implementation

A small tool is developed specifically to implement our approach (figure 14):

- given an initial rotation distance, it will guess the next block boundaries recursively by running decompression attempts on multiple candidate boundaries;

```
[-] Declared block #001: shift=21 start: 0x00000000
[+] FW UPD size: XXXXXXX (XXXXXXXX bytes)
[+] Chunk 1 size: 0x00020750 (132944 bytes)
[+] Chunk 2 size: 0x01118630 (17925680 bytes)
[+] SHA256 Hash #1: 2240aa11061e36953375835aabc7df50c6c6c9d495d388b3068c4d4c190aa8ae
[+] SHA256 Hash #2: 4d8afeba9d3fc325212743874d78ff41bf9b0e5ce61ae3cbd4cb94da9354c726
[+] SHA256 Hash #3: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
[+] Performed ROT13 on Build string
[-] Allocating and rotating 32 chunk2 buffers [....5....10....15....20....25....30..] done
[-] RAW Data:
  20930  00 00 00 06 00 00 00 03 00 00 00 04 48 44 52 00  ............HDR.
  20940  00 00 00 06 00 00 00 08 80 80 00 00 02 7e 6f f8  .............~o.
  20950  00 00 00 08 01 06 00 fc 01 06 00 fa 02 7e 6f f4  .............~o.
  20960  01 00 7f 45 4c 46 00 01 01 01 00 00 00 00 00 80  ...ELF..........
[-]     explore(): Step 0: guard
[-]     explore(): Step 1: find crash offset...
[-]     explore(): Step 2-5: find next shift
[-]     explore(): Step 6: generate candidates
[-]     explore(): Step 7: run stage2 on candidate boundaries
[-]     explore(): Step 8: discard candidates below threshold
[-]     explore(): found candidates:
[+]     |- 0x00001BA0    shift: 19    pos: 0x00005126    [depth: 1]
[+]     |- 0x00001B9C    shift: 19    pos: 0x00005126    [depth: 1]
[+]     |- 0x00001B94    shift: 19    pos: 0x00005126    [depth: 1]
[+]     |- 0x00001B90    shift: 19    pos: 0x00005126    [depth: 1]
[-]     explore(): Step 9: recurse into valid candidates
[-]        explore(): Step 0: guard
[-]        explore(): Step 1: find crash offset... 0x00005126
```

Fig. 14. Developed tool for block boundary and bitwise rotation parameter discovery.

- decompression is reimplemented in C (from our initial ruby implementation), to accommodate for some performance issues;
- rotation parameter guessing is reimplemented using the discovered hardcoded values to speed up boundary discovery and eliminate unreliable guesses.

The tool is obviously very specific to the attacked algorithm and firmware update format.

4.9 Analyzing revision-specific differences

While different firmware revisions contain different data, one could assume minor revisions would not differ by much.

Even if parts of the executable code change to reflect patches, most static resources and code will remain the same. This means one should be able to find identical data within two consecutive minor revisions, granted the build process is streamlined and produces a similar content layout for each revision.

Therefore, an attempt is made to recover two consecutive firmware revisions in parallel. The rationale behind this effort is as follows: as different versions have different block sizes, block boundaries are likely to end up in different parts of their contents. This means that for any

given data block, one will probably be longer than the block containing the same data from the other revision. It is then possible to leverage this length difference to cross-validate block boundaries by comparing contents from candidate blocks of one revision to the longer block from the other revision.

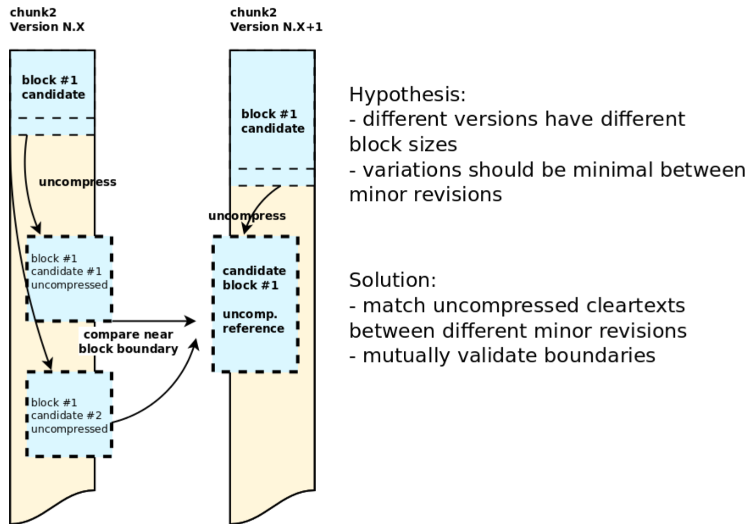

Fig. 15. Chunk #2 mutual validation approach.

Proceeding this way enables cleartext recovery by alternating the reference block used for validation between two consecutive firmware revisions: whichever block ends at the farthest output offset will be used as reference. This « mutual validation » approach is described in figure 15.

4.10 Final results

The mutual validation between consecutive revisions proves successful enough to enable the recovery of several hundreds of data blocks.

During the manual recovery of these block boundaries, some statistics are kept. It becomes obvious that their sizes are also following a cyclic pattern of 31 values (see figure 16), although they are different for every firmware revision. Like block boundaries of the chunk #1, it is thought that the values do not depend on revision numbers, but only data contents.

Model M5

V c.00	V c.01	V c.00	V c.01	V c.00	V c.01
block boundaries		decimal		block sizes (dec)	
0x00000000	0x00000000	0	0		
0x00001BA0	0x00000080	7072	128	7072	128
0x00005120	0x000030A0	20768	12448	13696	12320
0x00006468	0x00004BCC	25704	19404	4936	6956
0x0000918C	0x00007974	37260	31092	11556	11688
0x0000C850	0x0000B858	51280	47192	14020	16100
0x0000F7A0	0x0000D04C	63392	53324	12112	6132
0x00010E60	0x0000D9A4	69216	55716	5824	2392
0x00013FAC	0x00010228	81836	66088	12620	10372
0x00014AE8	0x000114E0	84712	70880	2876	4792
0x00018784	0x00014A10	100228	84496	15516	13616
0x00018B50	0x00016D64	101200	93540	972	9044
0x000193F4	0x00018F94	103412	102292	2212	8752
0x0001B864	0x000191AC	112740	102828	9328	536
0x0001BAC4	0x00019DC4	113348	105924	608	3096
0x0001EAEC	0x0001C294	125676	115348	12328	9424
0x0001FA08	0x0001F100	129544	127232	3868	11884
0x00022EE0	0x00020FFC	143072	135164	13528	7932
0x00026A58	0x000231C4	158296	143812	15224	8648
0x00027610	0x00023400	161296	144384	3000	572
0x00028E94	0x00024420	167572	148512	6276	4128
0x0002B040	0x00027528	176192	161064	8620	12552
0x0002CA78	0x00029C34	182904	171060	6712	9996
0x00030800	0x0002DA9C	198656	187036	15752	15976
0x00032FD4	0x0002FE8C	208852	196236	10196	9200
0x00033650	0x00030C98	210512	199832	1660	3596
0x00034AC0	0x00031964	215744	203108	5232	3276
0x00036C30	0x00035638	224304	218680	8560	15572
0x00036E54	0x00039214	224852	234004	548	15324
0x00039A6C	0x0003B9B0	236140	244144	11288	10140
0x0003B968	0x0003BBFC	244072	244732	7932	588
0x0003D508	0x0003E820	***251144***	256032	***7072***	11300
0x00040A88	0x0003E8A0	***264840***	***256160***	***13696***	***128***
0x00041DD0	0x000418C0	***269776***	***268480***	***4936***	***12320***
0x00044AF4	0x000433EC	***281332***	***275436***	***11556***	***6956***
0x000481B8	0x00046194	***295352***	***287124***	***14020***	***11688***
0x0004B108	0x0004A078	***307464***	***303224***	***12112***	***16100***
0x0004C7C8	0x0004B86C	***313288***	***309356***	***5824***	***6132***
0x0004F914	0x0004C1C4	***325908***	***311748***	***12620***	***2392***
0x00050450	0x0004EA48	***328784***	***322120***	***2876***	***10372***
0x000540EC	0x0004FD00	***344300***	***326912***	***15516***	***4792***
0x000544B8	0x00053230	***345272***	***340528***	***972***	***13616***
0x000548E8	0x00055584	***346344***	***349572***	**1072**	***9044***
0x00054D5C	0x000577B4	***347484***	***358324***	**1140**	***8752***
0x000571CC	0x000579CC	***356812***	***358860***	**9328**	***536***
0x0005742C	0x000585E4	***357420***	***361956***	**608**	***3096***
0x0005A454	0x0005AAB4	***369748***	***371380***	**12328**	***9424***
0x0005B370	0x0005D920	***373616***	***383264***	**3868**	***11884***
0x0005E848	0x0005F81C	***387144***	***391196***	**13528**	***7932***
0x000623C0	0x000619E4	***402368***	***399844***	**15224**	***8648***

Fig. 16. Chunk #2 identified block boundaries across firmware update revisions.

Using both the block size and rotation distance lists to decrypt the firmware results in a clear text image matching the SHA256 hash found in the update file header.

This concludes the six weeks of research, with most advances always occurring only a few days apart. The lack of long period without significant breakthroughs supported arguments in favor of continuing the cryptanalysis effort.

5 Instrumentation considerations

5.1 Planned evolutions, areas of research

While further development was not required, a few evolutions and areas of research were discussed internally, in case the manual and mutual validation approach proved insufficient:

- automating the mutual validation process;
- automating binary disassembly as a dedicated validation check;
- spending more effort reverse-engineering the proprietary filesystem to develop efficient validation heuristics;
- using a third data source for validation, across either three consecutive minor revisions or between identical revisions for different product models;
- instrumenting decompression to provide dictionary hit and miss statistics, and devise heuristics based on expected behaviour.

5.2 On multithreading

The developed tools used for bruteforce or block boundary discovery are not multithreaded.

The author believes spending time on properly multithreading the process would have shifted focus on irrelevant matters, rather than finding clever ways of attacking the problem. To an extent, the author recommends performance issues not be resolved by multithreading before any other option has been fully explored.

However, some critical steps may hit performance issues, as was the case with our initial ruby implementation of the decompression algorithm. One should take extra time to consider whether multithreading would lift a significant bottleneck or help advance to an ulterior step.

A basic attempt was still made to make use of the Open MP multithreading library, although performance was not ultimately an issue for our needs.

5.3 Statistical methods for data analysis

While simple analysis of block sizes and rotation parameters helped identify cyclic patterns, a true statistical analysis was not relevant for our data set.

However, this method should always be considered, and the author purposedly kept track of various observed properties in case a statistical study would be required later on.

6 Final thoughts and industry efforts

6.1 Recovered algorithm and data

The efforts succeeded in recovering a plaintext firmware image from the update files only. The result is a 42.2 MB ELF file recovered from the initial 18.1 MB firmware update. The file loads properly in IDA and finally reveals a custom-designed, proprietary real-time operating system running on a MIPS architecture.

The encryption algorithm resulted in bitwise rotation performed on 4-byte words, with parameters shared within data blocks of varying sizes ranging from 128 bytes to 16 KB. Bitwise rotation distance is taken from a cyclic list of 24 values, shared accross all firmware versions, with a starting value depending solely on the firmware update version. Block sizes vary with 31 possible values, also taken from a cyclic list, whose values vary depending on the firmware update chunk content.

The reader should keep in mind the observed resulting algorithm may not necessarily reflect the designer's intent, but might as well be the result of unintended consequences of an improper design.

Reverse engineering of the discovered firmware revealed enormous, unrolled routines implementing cryptographic primitives believed to participate in the update process. Multiple debug symbol names reveal the algorithm was actually thought as a cryptosystem, with decryption and encryption routines being explicitly named as such.

6.2 Required effort and resources

As stated in subsection 1.6, the effort spanned over six weeks of part-time work for a single researcher.

Coworker contributions included reverse-engineering of the ELF binary contained in the first chunk as well as a reimplementation of the decompression routine in both Ruby and C, in addition to various tool optimizations.

The developed tools helped us figuring out block boundaries, which allowed recovery of plaintext images in less than half an hour of manual checking. Computations were single-threaded and performed on a Dell T5500 workstation with 12 GB of RAM and dual Xeon X5650 (6 cores/12 threads each, 2.67 GHz).

In preparation for a similar attempt, the author would like to give readers the following advices:

- take detailed notes of everything: wikis are great for that;

- take regular breaks, sometimes for extended periods of time;
- step back and go over your approach on a regular basis to avoid getting lost in your own misconceptions;
- try to find ways to refine and/or validate hypothesis;
- have your ideas cross-examined by coworkers who are not involved in your project, keep notes of every suggestion;
- do not take a path without having considered at least one or two other possible options in case of failure.

6.3 Summary of discussed techniques and ideas

The following tables provide a summary of analysis techniques and ideas discussed in this article, along with their relevance to the recovery of our firmware data.

File format identification

Technique/Idea	Ref.	Comments/Relevance
Entropy analysis	2.1	Helped identify compressed and weakly encrypted data
Field delimiter identification	2.2	Helped recover data-container structure
Well-known value identification	2.2	Helped identify hashes, timestamps, sizes, etc.
Size-field identification	2.2	Helped delimit raw data chunks
Static/variable data identification	2.3	Helped delimit fields, identify value significance, devise hypotheses

Encryption scheme identification

Technique/Idea	**Ref.**	**Comments/Relevance**
Finding instruction opcodes	3.1	Unsuccessful attempt at finding executable code
Matching calling conventions	3.1	Unsuccessful attempt at finding code-like patterns in case the encryption kept symbols alike
Bitwise binary differences	3.3	Helped locate bitfields, headers, parameters, etc.
Comparing multiple samples	3.4	Helped identify static data BLOBs, enable statistical analysis, exploit potential similarities
Bruteforce	3.5	Helped recover data without known/identifiable plaintext
Reverse-engineering	3.6	Helped identify and implement the compression algorithm, helped find useful error/corner cases
Binary data fingerprinting	3.7	Unsuccessful attempt at recovering complete files from firmware image

Data recovery

Technique/Idea	Ref.	Comments/Relevance
Devising strong heuristics	4.1	Helped develop data validation routines used for automating exploration
Keeping statistics and numbers	4.2	Helped identify patterns, cyclic values, helped prepare further analysis
Chaining decryption with other processing	4.3	Helped validate decryption, block boundaries, decompression input
Instrumenting compression	4.3	Helped discard incoherent compression control bytes
Validating data formats	4.4	Helped validate successful decompression, helped discard corrupted bruteforced/guessed candidate blocks
Chaining data validation	4.6	Helped limit combinatorial complexity
Cross-comparison between multiple source data versions	4.9	Helped exploit similarities across data revisions, helped derive validation heuristics
Multithreaded tools	5.2	Unused
Statistical methods for data analysis	5.3	Unused, though it was strongly considered and prepared for

Thought process

Technique/Idea	Ref.	Comments/Relevance
Taking long breaks	6.2	Proved essential to keeping clear thoughts, ideas and goals
Peer review, suggestions and rubber duck debugging	6.2	Proved essential to validate and find ideas, helped ensure the implemented methods were appropriate and correct

6.4 On the use of home-made cryptography

Most researchers would not bother trying to attack the algorithm directly when they could afford attacking the hardware directly.

While it is widely recognized that security by obscurity is not a viable option in the long term, the successful efforts presented herein are a strong reminder that custom-made cryptography is not a strong deterrent to reverse-engineering. Even worse, the development and use of such algorithms tend to provide vendors with a false sense of security, which ultimately is fatal in environments where security is key.

Hiding firmwares to deter reverse-engineering ultimately results in the security community failing to analyze products and get vulnerabilities fixed through responsible disclosure, while leaving resourceful, malicious actors able to spend the required effort to succeed. These actors usually have no incentive to publish their findings or get bugs fixed, leaving legitimate customers vulnerable.

All of these considerations hold even without considering the fact that designing a secure cryptographic system can prove very difficult [6].

References

1. Dominic Chen. Firmware deobfuscation utilities. `https://github.com/ddcc/drive_firmware/`, 2015. [Online; accessed 22-January-2016].
2. Technical Zone Experiment. Codage et décodage des chaines analogiques de 1984 à 2010 (partie 1). `http://wintzx.fr/blog/2014/01/codage-et-decodage-des-chaines-analogiques-en-1984-partie-1/`, 2014. [Online; accessed 22-January-2016] (French).
3. Technical Zone Experiment. Codage et décodage des chaines analogiques de 1984 à 2010 (partie 2). `http://wintzx.fr/blog/2014/02/codage-et-decodage-des-chaines-analogiques-de-1984-a-2010-partie-2/`, 2014. [Online; accessed 22-January-2016] (French).
4. Craig Heffner. Differentiate encryption from compression using math. `http://www.devttys0.com/2013/06/differentiate-encryption-from-compression-using-math/`, 2013. [Online; accessed 18-January-2016].
5. NSA. Red and purple: A story retold. *Cryptologic Quarterly*, Vol. 3(n°3-4):63–80, 1984-1985. NSA analysts' modern-day attempt to duplicate solving the Red and Purple ciphers.
6. Bruce Schneier. Security pitfalls in cryptography. 1998. `https://www.schneier.com/essays/archives/1998/01/security_pitfalls_in.html`.
7. Roel Verdult. *The (in)security of proprietary cryptography.* PhD thesis, Radboud University Nijmegen and KU Leuven, Netherlands, 2015.
8. Wikipedia. Cryptanalysis of the lorenz cipher — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Cryptanalysis_of_the_Lorenz_cipher&oldid=695707661`, 2015. [Online; accessed 29-December-2015].
9. Wikipedia. Cryptanalysis — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Cryptanalysis&oldid=692497875`, 2016. [Online; accessed 18-January-2016].
10. Dennis Yurichev. Analyzing unknown binary files using information entropy. `http://yurichev.com/blog/entropy/`, 2015. [Online; accessed 18-January-2016].