Lightweight Runtime Reverse Engineering of Binary File Format Variants

Jeroen van den Bos
Netherlands Forensic Institute (NFI)
The Hague, The Netherlands
Email: jeroen@infuse.org

Abstract—**Binary file formats are regularly extended and modified, often unintentionally in the form of bugs in the implementations of applications and libraries that create files. Applications that need to read data from binary files created by other applications face the complicated task of supporting the resulting many variants.**

Lightweight implementation patterns to perform runtime reverse engineering can be used to handle common extensions, modifications and bugs. This increases application usability by generating fewer errors as well as provides useful automated feedback to maintainers.

This paper describes a set of patterns that are the result of experience in developing and maintaining a collection of automated digital forensics tools. The patterns are illustrated through practical examples and can be directly applied by practitioners.

I. Introduction and Background

Nearly all protocols, platforms and applications serialize their data in some binary file format. Supporting an external file format (i.e. a file format that is part of another application) requires the construction of a parser for binary data structures. Some applications rely heavily on external file formats, such as search engines that attempt to index as many document types as possible, security applications that need to detect threats in any executable or forensic software tools that require assessment of evidence no matter its storage format.

When the binary file format that needs to be supported is fixed and well-documented, developing a parser can still be a challenge. For example, most binary file formats use unsigned numbers, which can cause bugs around edge cases in languages that don't natively support unsigned numbers (e.g., a 16-bit unsigned value n will fit into a Java `short`, unless $n >= 2^{15}$).

In practice however, most binary file formats are not well-documented. Additionally, even though common formats are standardized (e.g., JPEG), many variants tend to exist, using either the format's defined extension points, or implementing non-standard extensions. On top of this, common formats tend to be supported by many different applications and libraries. Differences and bugs in those implementations lead to sometimes subtle differences in the files that are produced.

Applications supporting many external file formats are regularly confronted with parse errors, as they encounter previously unknown versions and variants. If such a variation occurs in a single data structure in a file that does not contain embedded data, the impact of such a parse error may be limited. However, variations in archive file formats or file systems may result in large amounts of data or entire storage devices being inaccessible to an application.

This leads to reduced usability, which in turn leads to considerable amounts of maintenance. To improve usability, binary file format parsers may employ implementation patterns to increase robustness. Additionally, these techniques can be employed to automate some of the reverse engineering associated with the resulting maintenance.

This paper contributes a set of patterns developed over the years during development and maintenance of several automated digital forensics tools. Application of the patterns is illustrated through examples based on cases encountered in practice. Additionally, this paper contains a discussion of the importance and considerations around binary file format reverse engineering and related work.

II. Implementation Patterns

The following subsections describe the implementation patterns by providing some background, along with a description of a common problem and the pattern to handle it.

A. Distinguish between Required and Expected Values

Constants are a simple and effective way to identify serialized data structures. Most file formats use them extensively, often in the form of ASCII strings that are easy to recognize in hex-viewers. An example is the GIF image file format, where each file starts with the ASCII string "GIF".

Problem: Some values were originally intended to be variable, such as flags to signal some type of encoding or a field that could be used to hold extension values. If they end up never being used for different values, they effectively become constants for the parsers that handle those file formats.

This presents a dilemma to parser implementers: if those values are treated as constants, recognition of the file format becomes stronger, but won't accept other values, which are strictly acceptable.

Solution: Common values should be treated as *expected* values instead of required values. An expected value is treated as a constant match when it is encountered, but does not generate a parse error when its value differs.

This is especially useful in combination with other parse errors: if the expected value was encountered, it can be assumed that the preceding data was also correct. If the expected value was not encountered, parsing also continues. Then, if the file ends up unrecognized, the location of the expected value can be used to more precisely determine where the corruption or deviation probably originated.

B. Accept New Constants on Structure Match

Binary file formats often use a common layout for all their data structures. A particularly elegant example is PNG [1]: each structure contains a four-byte length field, a four-byte name field, a variable size data field and a four-byte checksum field (containing a 32-bit CRC over the name and data fields).

Problem: Extensions can be introduced as otherwise well-formed data structures that contain new constant values (to identify them for the producing application). While their content is unknown, generating a parse error solely on an unknown constant value will prevent the parser from reaching subsequent data structures.

Solution: Using custom criteria, unknown constants can be accepted as long as the data structure itself is well-formed. For example, in the case of PNG, if the values of the length and checksum fields can be verified, the data structure appears to be an extension. Verifying length fields in this context refers to checking whether the data structure is followed by another structure that is recognized[1].

This pattern is only applicable to file formats that have a well-defined and common data structure layout that allows verification of some kind. Additionally, it is difficult to apply if the file format doesn't rely on contiguous data structures and instead relies on some internal addressing mechanism (such as Microsoft Compound Files, which form the basis for document files such as DOC and XLS).

C. Attempt Alternative Encodings

Binary file formats usually require the use of specific encodings for the values they store, such as whether numeric values are signed or unsigned and whether text strings are encoded as ASCII or UTF8.

Problem: Developers implementing file format extensions may not be aware of these restrictions or may be entirely oblivious to the differences in encodings, leading to the use of for example UTF8 encoded characters in strings that were intended to be ASCII-encoded.

Common verification checks are used to determine whether values are acceptable in data structure fields. While these checks increase the confidence in the correctness of the parse result, they will generate parse errors when some alternative and unexpected encoding is used.

Solution: Instead of rejecting values outside the accepted range of some encoding in a file format, alternative encodings can be attempted. Besides alternative string encodings, other uses exist in the area of compression and checksum algorithms (e.g., which value is used for escaping or the initial value of a checksum).

[1]PNG actually anticipates this through the use of upper- and lowercase letters in structure names, but this has not eliminated non-standard extensions.

D. Anticipate Cross-Platform Errors

Apart from encoding of values, binary file formats also commonly require some platform-specific or custom conventions. Examples are the use of padding to specific sizes or multiples, as well as whether text strings are zero-terminated and how special characters are represented.

Problem: Applications and libraries implementing support for binary file formats do not always succeed in fully abstracting away platform-related serialization details. Omitting or incorrectly applying padding and character representation may lead to parse errors.

Solution: When encountering parse errors that generally appear to be close mismatches, alternative conventions can be attempted. An example is the representation of the new-line character in serialized ASCII strings. On unix-based platforms, it is generally represented as a single character (value `0x0a`) while on windows-based platforms it is generally represented as two characters (values `0x0d` followed by `0x0a`). An ASCII string containing n new-line characters may have a calculated size that is n bytes off.

E. Relax Structure Ordering Constraints

Most binary file formats are organized as a sequence of data structures, with constraints on the type of structure that may occur at some position in the file. These constraints are often dependencies between structures, such as a list of parameters to use in an algorithm to decode subsequent structures.

Problem: Many ordering constraints on binary file formats are custom and not described using a simple algorithm. As a result, libraries offering custom serialization of data often do not enforce these constraints. This leads to many files that are well-formed on the individual data structure level, but that violate ordering constraints.

Solution: Instead of implementing the generally complex ordering rules for a file format, a more robust approach is to only track direct dependencies. This means that any order is always accepted, as long as this allows access to values required to parse each directly following structure.

For many modern binary file formats that describe each data structure's size, this may mean that a file can be parsed entirely, but that some parts of the content cannot be decoded. An example is a compressed file that lacks its decompression table. Instead of rejecting it at some location halfway through the file, the error can instead be an indication of the missing structure.

III. Practical Examples

The following subsections discuss examples encountered in practice that have lead to the development of the described patterns, and illustrate their practical applicability.

A. Handling a JPEG Variant with a Malformed Structure

In a case study which investigated 37 file format variants in a corpus of over 1.2M image files [2], several JPEG files contained a data structure with the following contents[2], shown

[2]The example is available at: https://github.com/jvdb/derric-eval/commit/712e5833d7e8ab758d45ae8aadd088e7a74ed491

in a typical hexadecimal view (from left to right: offset, hexadecimal values, ASCII-printable values):

```
0x00b1:  fffe 003c 534e 2031 3635 2d32  ...<SN 165-2
0x00bd:  3031 3036 2d30 322c 2045 6c61  0106-02, Ela
0x00c9:  6e20 476d 6b0d 0a49 6e74 6572  n Gmk..Inter
0x00d5:  6e61 6c20 7573 6520 6f6e 6c79  nal use only
0x00e1:  0d0a 4e6f 7420 666f 7220 7265  ..Not for re
0x00ed:  7361 6c65 00ff c400 1f00 0001  sale........
```

The first byte, at offset `0x00b1` (177) with value `0xff` marks the start of a new JPEG data structure. The next byte, with value `0xfe`, defines it as a Comment structure, which means it most likely has a textual value describing (some aspect of) the image or producing application. Following the markers is a 16-bit unsigned length field, which has the value `0x003c` (60).

According to the JPEG specification [3], length fields include the size of the length field itself, but not of the markers. This means that the payload of the Comment structure in this case should be $60 - 2 = 58$ bytes. The following data structure should begin at offset $181 + 58 = 239$ (`0xef`), but this is not the case. Instead, the first following valid JPEG marker is at offset 242 (`0xf2`), which means the length field's value should be 63 instead of 60.

A possible explanation for the error is that the code that serialized the structure calculated the size of the string it contains incorrectly. The string contains two instances of the Windows-version of the new-line sequence `0x0d`, `0x0a` and a single terminating zero. In a high-level string representation, the new-line was most likely represented as a single character (`\n`), and the terminating zero could be omitted when performing a typical length calculation (e.g., `strlen()` in C).

Anticipate Cross-Platform Errors described in section II-D is an example of handling this problem by attempting different alternative values for the length field based on platform-specific considerations. This can include attempting a size of both 1 and 2 for new-line sequences as well as including and omitting the size of the terminating zero.

There is no definite answer on the cause of this specific problem: the deviation of three bytes in the value of the length field could have been caused by many other issues. Still, in this case it would have been sufficient to apply this pattern and it would have automatically provided a plausible explanation for the encountered data.

B. Distinguishing Between Corrupted and Fragmented Files

When metadata documenting the location of files on a storage device (e.g., the file system) is unavailable, a content-based approach to recover data is commonly employed. This *file carving* [4] amounts to attempting to recognize files based on their internal structure. A particularly difficult problem is that of file fragmentation, where the contents of files is stored in non-contiguous blocks on a storage device, as a result of optimizing the available space.

If a parse error is encountered during an attempt to recognize a piece of data as being in some known file format, it is crucial to determine the error location with high precision. If the parse error occurs in a block of data where no previous matches (such as constants or verified length fields) have been located, the file is most likely fragmented. However, if the parse error occurs in a block where some positive matches have already occurred, it is more likely that the file is corrupt (or is a variant that warrants additional investigation).

As an example, consider a sequence of contiguous clusters C_1 to C_4, as shown in Fig. 1.

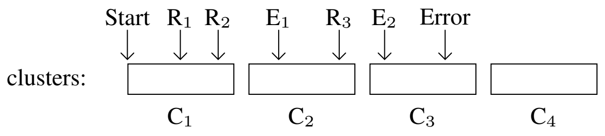

Fig. 1. Required and expected values across clusters.

Parsing started at the beginning of C_1 and continued through C_2 and C_3 until a parse error was encountered in C_3. Some required constants prescribed by the file format were encountered along the way, denoted as R_1, R_2, and R_3. Additionally, some expected constants as discussed in section II-A were encountered as well, denoted as E_1 and E_2.

In this situation, only distinguishing between required and variable values would suggest that the file is most likely fragmented: in C_1 and C_2 required constants were matched and none in C_3, where the parse error occurred. However, in C_3 there was a positive match on an expected value: a formally variable value that nearly always has a specific value in all encountered instances.

This changes the conclusion: this match suggests that the cluster does belong to the file that is being parsed, but instead the file is possibly corrupted or of some unknown variant of the file format. Instead of attempting some combinatorial approach to find the correct next cluster, the application can instead suggest to the user that the file is corrupt or unknown.

C. Accept Missing Footers

Most file formats specify a kind of terminator to formally end a file of its type. However, instances exist of practically all file formats that omit these footers. While this may present problems for applications in specific domains such as data recovery that often have difficulty determining when to stop looking for pieces, for most parsers this should not present a significant problem.

As discussed in section II-E, handling the ordering constraints for file format data structures can be made more robust by replacing the formal rules by a dependency handler. This handler collects all encountered instances of all data structures and only generates an error when parsing cannot continue due to a value that should have been encountered.

An example of this is the GIF file format, that specifies a double footer. First, the sequence of data blocks should be terminated with a block that has zero size, represented by a single byte with value `0x00`. Next, the actual terminator is a single byte with the specific value `0x3b`. In practice, any combination is quite common: both values present, both values missing, only the zero present, and only the `0x3b` present.

This illustrates that a seemingly simple two-byte footer can be interpreted by developers in many different ways.

IV. Discussion

A. The Rise of Binary File Formats

The early personal computing era was dominated by Microsoft, which meant most applications only required support for MS-DOS- and Windows-related file formats. After Internet-based applications and service-oriented architectures became more commonplace, local storage even appeared to become a thing of the past. In a way, this is still the case as documents and media are increasingly no longer stored primarily locally, but are streamed or copied on-demand the cloud.

At the same time, the diversity in platforms running locally on personal computers is increasing: Microsoft is still a large player, along with Apple on the desktop. Mobile devices are divided mostly between Android and iOS. All these platforms each run several different web browsers and many native applications, which have regained popularity as a way for platforms to gain competitive advantages.

The large amount of different combinations of platforms, browsers, applications and (cloud-based) storage has caused many different binary file formats to appear. All these files are interesting to many applications, including search engines, security applications, and data recovery tools.

Effective ways of supporting the necessary reverse engineering work is needed. The patterns described in this paper can help ease the maintenance burden caused by the requirements of supporting many binary file formats.

B. Related Work

Related reverse engineering research tends to focus on either binary reverse engineering (where binary refers to binary executables) or data reverse engineering (where data refers to data descriptions such as database layouts or structure descriptions in source code). Binary data reverse engineering has received relatively little attention, most likely due to the points discussed in the previous subsection.

Notable related work is in the area of extracting data structures from binary executable files, for example to reconstruct symbol tables. In this area there are dynamic approaches such as Howard [5] that tracks how an application uses memory. Static approaches such as mapping well-known library calls are in broad use since they are implemented in popular reverse engineering tools such as IDA PRO [6].

Additionally, there is considerable work in the security community in the area of automated reverse engineering of network protocols. An approach that has proven useful in this area is that of Polyglot [7], performing data flow analysis of messages between systems (assuming that a flat sequence exists). More advanced techniques are implemented in Tupni [8], which was also capable of automatically deriving sequence information for several binary file formats.

A key difference between most of the existing approaches and the patterns described in this paper is that the latter are intended to improve the robustness of existing parsers, instead of inferring or approximating a parser directly. As such, the patterns can potentially be applied to improve the results of existing approaches.

Related work also exists in the area of developing parsers for binary file formats, including the DSLs PADS [9] and Derric [10], as well as the C parser combinator library Hammer [11]. Such tools make it relatively easy to apply the patterns described in this paper: they can be implemented in the accompanying runtime library or code generator. An example of this is the `expected` keyword in Derric, that implements the pattern described in section II-A.

V. Conclusion

Reliably supporting a binary file format in a software application is not trivial. Common causes for this are lacking or unclear specifications, multiple existing standards, conflicting developer interpretations and buggy existing implementations. Rejecting everything that does not exactly conform to some interpretation, version or standard will result in many generated errors at the user level, degrading usability significantly.

Instead of considering every version and variant separately, a binary file format parser can be made more robust through the use of lightweight runtime reverse engineering patterns. These patterns correct for common deviations and alternatives. Additionally, when they do not succeed in parsing a file, they assist in providing more precise and useful error messages to alleviate the resulting maintenance.

Finally, changes in the personal computing landscape have caused the development of many new and evolving binary file formats. This creates a need for more research into the automated reverse engineering of this type of software artifact.

References

[1] W3C, “Portable Network Graphics (PNG) Specification,” 2003, http://www.w3.org/TR/PNG/.

[2] J. v. d. Bos and T. v. d. Storm, “A Case Study in Evidence-Based DSL Evolution,” in *9th European Conference on Modelling Foundations and Applications (ECMFA’13)*, ser. LNCS, vol. 7949. Springer, 2013, pp. 207–219.

[3] ITU/CCITT, “Rec. T.81 (JPEG Compression Specification),” 1992.

[4] A. Pal and N. Memon, “The Evolution of File Carving,” *Signal Processing Magazine, IEEE*, vol. 26, no. 2, pp. 59–71, 2009.

[5] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A Dynamic Excavator for Reverse Engineering Data Structures,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS’11)*. The Internet Society, 2011.

[6] Hex Rays, “IDA,” https://www.hex-rays.com/products/ida/index.shtml.

[7] J. Caballero, H. Yin, Z. Liang, and D. Song, “Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS’07)*. ACM, 2007, pp. 317–329.

[8] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, “Tupni: Automatic Reverse Engineering of Input Formats,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS’08)*. ACM, 2008, pp. 391–402.

[9] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer, “PADS/ML: A Functional Data Description Language,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’07)*. ACM, 2007, pp. 77–83.

[10] J. v. d. Bos and T. v. d. Storm, “Bringing Domain-Specific Languages to Digital Forensics,” in *33rd International Conference on Software Engineering (ICSE’11)*. ACM, 2011, pp. 671–680.

[11] UpstandingHackers, “Hammer,” https://github.com/UpstandingHackers/hammer.