PNG-Fuzzing with JQF

Paul Kalz, Marwin Linke, and Sebastian Schatz

Humboldt University of Berlin, Germany

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Background on the File Format PNG

1.1 Overview

TODO: Overview: Gebt einen kurzen Überblick über das ausgewählte Datenformat (Historie, Verwendungszweck,...)

History

Use Case

1.2 Input Specification

TODO: Input Specification: Beschreibt im Detail die Spezifikation des Dateiformats. Wie sind Dateien dieses Formats aufgebaut? Existiert eine formale Spezifikation? Wie ist eine Beispieldatei aufgebaut?

Specifications

Structure

1.3 Security

TODO: Security: Beschreibt mögliche Sicherheitslücken im Zusammenhang mit dem Datenformat. Geht dabei näher auf bereits existierende Fälle ein (case study), ggf. auch im Zusammenhang mit den von euch ausgewählten Tools (Bug-Tracker).

2 Implementation

Noch nicht fertig! *Die Beschreibung der Implementation, des Generators und des Treibers sind im Prinzip fertig, enthalten jedoch noch einige komische Sätze. Man muss es nochmal gründlich durchlesen, anpassen und vor allem alle nötigen Quellen ergänzen.*

2.1 Tools

JQF The coverage-guided testing platform for Java named JQF, developed by R. Padhye, C. Lemieux and K. Sen, is designed for *practitioners*, who want to find bugs in Java programs, as well as for *researchers*, who wish to implement new fuzzing algorithms [1]. For our project, we used JQF to implement a fuzzing generator based on the file format PNG, which can be directly used in JQF to run test drivers for matching libraries.

PNGJ As for the library to test, we chose PNGJ, which is a pure, open-source Java library for high-performance reading and writing of PNG images [2]. For our use case, mainly the reading capability of PNGJ was tested by the fuzzer.

Java Standard Library A large component of writing PNG files are the compression algorithms, CRC32 checksums and Adler32 checksums, which all can be found in `java.util.zip`, a Java standard package. The PNG fuzzer relies heavily on the correctness of those algorithms and therefore uses this library to ensure accurate outputs instead of implementing the algorithms itself.

2.2 Process

To successfully write a PNG fuzzer and test the library, a generator and driver class need to be implemented. The generator is a self-contained class, which is called by JQF and returns a data type. In our case, the generator returns the data type `PngData`, which contains a byte array that resembles a functional PNG file. Next, the fuzz driver receives the `PngData`, which in return is read by PNGJ. The fuzz driver itself can be designed flexibly to test multiple functionalities in a library. This process is automatically repeated many times; each time JQF randomizes the seed to guide the random outputs from the generator in a more beneficial direction.

3 Generator

3.1 Chunk Structure

This section will outline the structure of chunks, an important aspect of PNG files. The byte stream of PNG images can be broken down into chunks, where

Table 1. Byte strucure of a chunk [4].

Chunk length	**Chunk type**	**Chunk data**	**CRC**
4 bytes	4 bytes	*Length* bytes	4 bytes

each one contains certain information and serves a concrete purpose. A chunk is marked by its uniform structure found in the header and trailer surrounding its content.

As shown in table 1, each chunk consists of 4 parts [4]. The *Chunk length* refers to a 4-byte unsigned integer giving the number of bytes in the content field of that chunk, it doesn't include the length of the chunk type nor the CRC checksum at the end. The *Chunk type* is always represented by 4 characters each 1 byte long to uniquely identify each type of chunk. It uses capitalization to imply information about the chunk. The *Chunk content* is the main part of each chunk and contains various amounts of data. The *CRC* is a well-known checksum algorithm, which uses 32 bits and therefore is also called CRC32 here.

We encapsulated this common structure in the class called `ChunkBuilder`. After generating the contents of a chunk, we simply call:

```
ChunkBuilder.constructChunk(chunkType, chunkContent)
```

with the according type and content as byte arrays/streams and receive a complete chunk, which then can be concatenated with other chunks to form a PNG file.

3.2 PNG Structure

Chunks follow ordering constraints to form PNG files, they mainly revolve around 4 critical chunks (critical means that the chunk must appear). Every PNG file begins with a fixed 8-byte signature and the `IHDR` chunk, which always must be first. The `IHDR` stores important image information which are commonly accessed by the following chunks. The end of each PNG file is marked by the `IEND` chunk, which has no content but still a chunk header and trailer as mentioned in 3.1.

Between those chunks, the critical chunks `PLTE` and `IDAT` are found which hold information about the image data such as pixel color values as well as a color palette for indexed images. All other (optional) chunks can be sorted into the space before `PLTE`, between `PLTE` and `IDAT` or after `IDAT`. For the exact ordering constraints, please refer to the PNG specifications [5].

Parameters The generator needs to keep track of which chunk is generated and then order them correctly. For that case, every chunk uses a boolean flag to indicate if it is used or not, which is randomly enabled in `initializeParameters()`.

Furthermore, the generator stores parameters about information that is shared between chunks, for example, the bit-depth, the color type and the image size. Before each run, the generator resets its parameters with `resetParameters()`.

Optional Chunks In this section, a short overview will be explained to show which optional chunks [5] our generator can generate.

Color Space Information `cHRM`, `gAMA`, `iCCP`, `sBIT`, `sRGB` are chunks that are used to specify color space information, they define how the image is supposed to be displayed. They must appear before the `PLTE` chunk.

Miscellaneous Information After Palette `bKGD`, `hIST`, `tRNS` are chunks to convey miscellaneous information about the image mainly related to the palette, such as the background color, frequency of colors in a palette or transparency values in a palette. They all appear between the `PLTE` and `IDAT` chunk.

Miscellaneous Information `pHYs`, `sPLT` are also chunks to convey miscellaneous information but can appear anywhere between the `IHDR` and `IDAT` chunk.

Textual Information `tIME`, `iTXt`, `tEXt`, `zTXt` are chunks to hold textual information about the image such as the time the image was last modified or information about the title, author and more. They don't have any ordering constraints.

3.3 IHDR

The `IHDR` chunk, short for image header, stores critical information about the image and defines what type of image the fuzzer generates. It includes the *image width*, *image height*, *bit-depth* (also called *bits per channel*), *color type*, *compression method*, *filter method* and *interlace boolean*. Our generator randomizes most of these parameters.

Image Size The *image width* and *height* can take on values between 1 and 10 pixels each, going larger than 10 pixels didn't seem to affect the number of covered branches. The size of up to 8 pixels contributes to the behaviour of interlacing (See section 3.5), thus greatly increasing the coverage.

Bit-depth The *bit-depth* defines the number of bits per channel, whereas each channel represents one color value of a pixel, for example, the red channel with 1 byte in an RGB image would have a bit-depth of 8. The *bit-depth* is directly dependent on the color type and only certain values are allowed to be used (See table 2).

Color Types The *color type* represents the structure of pixels in a PNG image. Grayscale images (color type 0) consist of 1 channel per pixel, RGB images (color type 2) have 3 channels for red, green and blue. Both of these color types can include an alpha channel, which in turn makes them grayscale with alpha (color type 4) or RGBA (color type 6).

An important aspect of PNGs is indexed images (color type 3), which use a palette to decrease the memory space needed. Instead of assigning each pixel multiple values for their respective channels (for example 3-bytes for an RGB image with a bit-depth of 8), indexed pixels only have 1 channel with the size of the bit-depth and store an index to the palette. The palette stores up to 256 entries (dependent on the bit-depth), each a 3 byte series of values representing red, green and blue. The main advantage comes from an image having the same colored pixel multiple times, which doesn't need to be stored separately in indexed images.

The generator first selects a random color type, then randomly chooses one of the allowed bit-depths and stores both information as parameters. Furthermore, the number of channels is defined based on the color type. The flags of certain chunks are enabled, some optional chunks are enabled by chance, whereas chunks like the palette are mandatory for indexed images. An example of an optional chunk would be the `tRNS` one, which adds corresponding alpha values for entries in the palette.

Table 2. Allowed bit-depths per color type [5].

Color type	**Allowed bit-depths**	**Interpretation**
0	1, 2, 4, 8, 16	Grayscale
2	8, 16	RGB
3	1, 2, 4, 8	Indexed
4	8, 16	Grayscale with alpha
6	8, 16	RGB with alpha (RGBA)

3.4 IDAT

The `IDAT` chunk is used to store the pixel data of an image, which is first filtered and then compressed [5] by a *deflate compression*, which is a derivative of the *LZ77 compression* used in zip, gzip, pkzip and related programs [6].

The generator uses the standard library `java.util.zip` which already implements the exact compression algorithm used for PNG files. It also includes a special checksum, named *Adler32*, which appends to the compressed data.

Image Data The raw image data is a series of bytes which is divided into scanlines (rows) and pixels [8]. Each scanline starts with a so-called filter byte, which represents the filtering method used for that scanline.

The generator calculates the number of scanlines and pixels, based on the image width and height, and iterates over them. After the filter byte, each scanline filters and appends pixels with randomized color channels accordingly. The size and number of channels are based on the color type and bit-depth.

Filtering There are five different filtering methods, whereas an empty byte (0) represents no filtering. Methods 1 to 4 indicate the use of *Sub*, *Up*, *Average* or *Paeth filtering*. The reason to use filtering is to represent the color values relative to their neighbouring pixels, this allows the deflate algorithm to compress patterns which are found relatively between pixels. The generator implements the filtering algorithms following the pseudo-code written in the specification [7].

3.5 Interlacing

Interlacing is a procedure to render an image over multiple passes. It starts with a low resolution and increases with each pass until the complete image is shown. This allows large PNG files to be rendered smoothly instead of waiting for the complete image to be rendered at once. Interlacing is done by including multiple `IDAT` chunks in the PNG file, each one representing one pass of the rendering procedure. The first `IDAT` chunk only contains 1/64 of the pixels, the next one contains 1/32, then 1/16 and so on. The final, seventh pass contains the complete image. Which pixels are included is based on an 8-by-8-map of pixels which can be found in the specification [8].

The generator implements interlacing by generating multiple `IDAT` chunks with a lower image width and height, this doesn't represent true interlacing since each pass uses randomized pixels but it doesn't hinder most PNG readers, rendering the image regardless.

3.6 Other Chunks

All of the critical chunks have been explained to this point but the generator also implements a lot of optional chunks. We are not going into much detail further because almost all optional chunks are just a way to store additional information about PNG images. That's why the actual implementation mostly fills optional chunks with random bytes adjusted to the size and constraints found for that specific chunk [5].

4 Fuzz Testing

4.1 Fuzz Driver

As for the fuzz driver, we decided to use code samples, provided directly by *PNGJ* itself [3], and combine them to test multiple functionalities. The largest contribution to the covered branches came by simply reading the PNG data, this covers all chunks superficially. To test for detailed aspects of the library we also included a pipeline that changes the PNG image depending on its color type and rereads the data in every step. The reason why we decided to make the driver partially dependent on the color type is that most functionalities are only applicable to certain color types.

In the first step, the fuzz driver checks for indexed images that use a palette and converts them to true color images that use RGB or RGBA. In the second step, all converted images and images that were generated with true color by default are desaturated and converted to grayscale. Lastly, all converted images and images that were generated with grayscale by default are tested by mirroring the image.

Although this approach tests multiple functionalities, it is still far from covering all functionalities. It would need a very detailed fuzz driver to cover everything, which would pose its own challenge and lie outside the scope of this project.

4.2 Guidance

JQF also offers the possibility to alter its guidance properties, this is a useful aspect of fuzzing, we decided to not use it though and stick to the default guidance settings.

PNGs are a very interesting data format when it comes to fuzzing. The general idea of fuzzing is to generate random data that tries to cover most of the functionality of a program. However, the precise specifications of PNG files make this quite difficult. Due to many checksums, compression algorithms and concrete identifiers, PNGs are difficult to come across purely randomly. Practically every byte must be at its exact location with its exact value. The precise nature of PNGs may offer approaches like grammar fuzzing to be successful, whereas mutational fuzzing would pose a great challenge.

5 Evaluation

TODO: Beschreibt die durchgeführten Experimente und deren Ergebnisse. Wie hoch war die erreichte Coverage? Konnten Bugs/Crashes gefunden werden? Wenn ja, welche?

5.1 Experiments

5.2 Results

6 Result Discussion

TODO: Versucht die Ergebnisse der Experimente zu interpretieren und zu erklären (discussion). Zieht Sie Folgerungen aus den Ergebnissen (conclusion). Beschreibt die nächsten Schritte, die durchgeführt werden müssten/könnten/sollten (future work).

6.1 Discussion

6.2 Conclusion

6.3 Future Work

References

1. Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3339002
2. PNGJ GitHub-Page, https://github.com/leonbloy/pngj?tab=readme-ov-file
3. PNGJ Samples, https://github.com/leonbloy/pngj/tree/master/src/test/java/ar/com/hjg/pngj/samples
4. LibPng: File structure, http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html
5. LibPng: Chunk specification, http://www.libpng.org/pub/png/spec/1.2/PNG-Chunks.html
6. LibPng: Deflate algorithm, http://www.libpng.org/pub/png/spec/1.2/PNG-Compression.html
7. LibPng: Filtering, http://www.libpng.org/pub/png/spec/1.2/PNG-Filters.html
8. LibPng: Data representation, http://www.libpng.org/pub/png/spec/1.2/PNG-DataRep.html