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

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]. As for our project, we used JQF to implement a fuzzing generator based on the file format PNG, which is tested by a driver directly in JQF.

PNGJ To test our fuzzer we chose PNGJ, which is a pure, open-source Java library for high-performance reading and writing of PNG images [3]. Especially the reading capability of PNGJ was tested by the fuzzer.

Java Standard Library PNG files use compression algorithms [5] as well as multiple checksum algorithms, which are all provided by the Java standard package `java.util.zip` [6]. The fuzzer is dependent on the correctness of those algorithms and therefore uses this library to ensure reliable results instead of implementing the algorithms itself.

2.2 Process

To implement the PNG fuzzer and test the library, a generator and driver class needed to be written. The generator is a self-contained class, which is called by JQF and returns a specified data type. In our case, the generator returns `PngData`, which wraps a byte array of a functional PNG file. Next, the fuzz driver receives the `PngData`, which in turn is processed by PNGJ. The fuzz driver itself can be designed flexibly to test multiple functionalities in a library. This process is automatically repeated; each time JQF uses the guidance algorithm Zest [2] to guide the randomized seeds from the generator in a more beneficial direction.

3 Generator

3.1 Chunk Structure

The byte stream of PNG images can be broken down into chunks, where each one contains certain information and serves a concrete purpose [8]. 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 [7]. 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

Table 1: Byte strucure of a chunk [7].

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

information about the chunk. The *chunk content* is the main part of each chunk and contains various amounts of data. The *CRC32* is a well-known checksum algorithm, which uses 32 bits. It includes the chunk type and content but not the length field.

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 or streams and receive a complete chunk, which then is concatenated with other chunks to form a PNG file.

3.2 PNG Structure

Chunks follow ordering constraints to form PNG files. There are 4 critical chunks (critical means that the chunk must appear), which determine the ordering of optional chunks.

Every PNG file must begin with a fixed 8-byte signature and the first chunk named `IHDR`. The `IHDR` stores important image information which are commonly accessed by its 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 `IDAT` and `PLTE` are found which hold information about the image data such as pixel color values as well as a color palette for indexed images. Further optional chunks are sorted into the space before `PLTE`, between `PLTE` and `IDAT` or after `IDAT`. For the exact ordering constraints, please refer to the PNG specifications [8].

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 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 given to explain which optional chunks [8] 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 determines 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 an *interlace boolean*. The generator randomizes these parameters, except for the compression method and filter method: At present, only method 0 is defined for both in the `IHDR` chunk [8]. The generator still uses different compression methods as well as filter methods, but they are instead defined in their respective chunks.

Image Size The randomized *image width* and *image 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. A channel refers to a single color value or alpha value. 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* determines the number channels per pixels and the structure how they are stored. 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) [11].

Indexed images (color type 3) are used to decrease memory space and feature a palette in the `PLTE` chunk. Instead of assigning each pixel multiple values for their respective channels (for example 3-bytes for an RGB image with a bit-depth

Table 2: Allowed bit-depths per color type [8].

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)

of 8), indexed pixels only have 1 channel with the size of the bit-depth and hold 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.

As for the exact implementation, 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 determined 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.

3.4 IDAT

The `IDAT` chunk is used to store the pixel data of an image, which is first filtered and then compressed [8] by a *deflate compression* [5], which is a derivative of the *LZ77 compression* used in `zip`, `gzip`, `pkzip` and related programs [9].

The generator uses the standard library `java.util.zip` which already implements the exact compression algorithm used for PNG files. It also includes an additional 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 [11]. Each scanline starts with a 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. 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 [10].

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 [11].

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 and covering branches.

3.6 Other Chunks

All of the critical chunks have been explained to this point but the generator also implements optional chunks. This section will not provide more information about the exact implementation of the rest, because optional are primarily used to store additional information in an image. This is easily implemented by filling those chunks with random bytes, adhering to the size and structural constraints as specifeid [8].

4 Fuzz Testing

4.1 Fuzz Driver

As for the fuzz driver, we decided to use code samples, provided directly by *PNGJ* itself [4], 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, albeit 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 dependent on the color type is that most functionalities are only applicable to certain color types, generating more valid inputs.

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. However, we decided to not use it and stick to the default guidance settings.

PNG is 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. In contrast, the precise specifications of PNG files make this a difficult task. Due to many checksums, compression algorithms and concrete identifiers, PNGs are rare to come across purely randomly. Practically every byte must be at its exact location with its exact, intended value. The explicit nature of PNGs may offer approaches like grammar fuzzing to be successful, whereas mutational fuzzing would pose a great challenge.

5 Evaluation

5.1 Experiments

To evaluate the efficiency of our implementation, we fuzzed the *PNGJ*-libary with the following four fuzzers using our fuzz driver.
Firstly, our **Complete Fuzzer**, which uses the full capability of our generator to correctly generate all critical and optional chunks as specified [8].
Secondly, our **Random Fuzzer** as a baseline, which generates random chunks in which only the signatures and the CRC checksums of the chunks are calculated correctly.
Thirdly, our **Probable Fuzzer** a version of our random baseline that also generates a correct IHDR and a correct IDAT chunk. The structure of the PNG files is therefore correct at least in the cirtical chunks.
Fourthly, our **Simple Fuzzer** a simplified version of our Complete Fuzzer, which contains only IHDR, IDAT, PLTE and IEND chunks, all colour types, only bit-depth of 8, default compression, no interlacing and no filtering. This fuzzer therefore generates only correct PNG files.
We tested PNGJ with each fuzzer 10 times, with each repetition taking one hour. All experiments were executed on a Windows 10 system using a GTX 1060 6 GB GPU, a Intel i5 7600k CPU with 16 GB DDR4-3200 RAM. The experiments were run sequentially.

5.2 Results

We did not find any bugs or crashes during our tests, so we were only able to analyse the coverage of the fuzzers.
In the following diagrams, the blue graph always belongs to the Complete Fuzzer, the red graph to the Probable Fuzzer, the green graph to the Simple Fuzzer, and the black graph to the Random Fuzzer.

Total Coverage

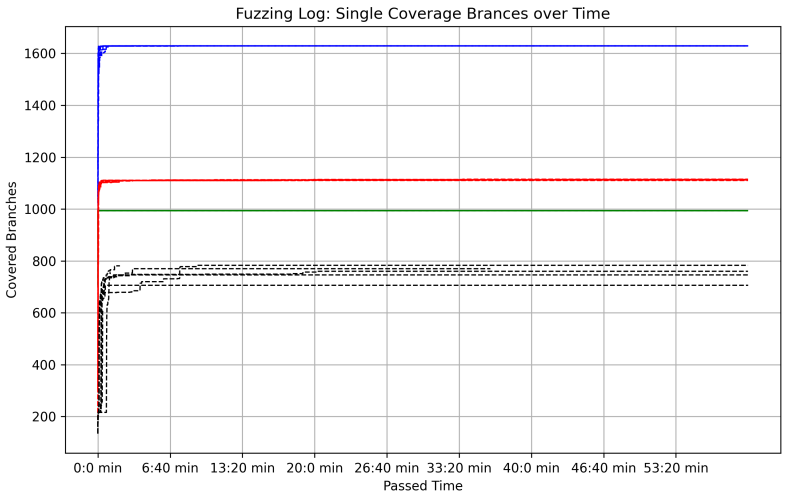

Fig. 1: Coverage for each repetition over time

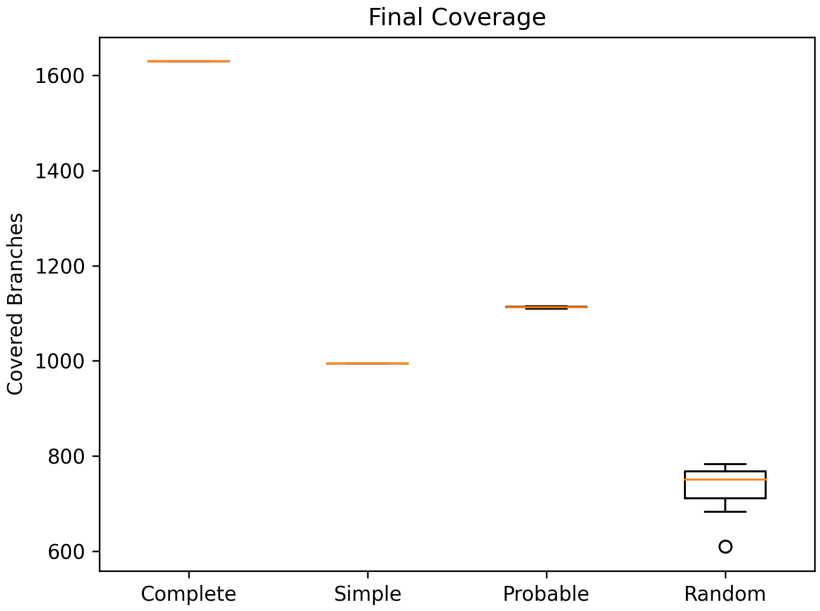

Fig. 2: Final Coverage

With our Complete Fuzzer, we managed to cover 1629 branches in every repetition. The Probable Fuzzer covered an average of 1112.9 branches. The Simple Fuzzer covered 994 Branches in every repitition. And the Random Fuzzer covered an average of 732,2 branches.
The number of covered branches for the Complete Fuzzer and the Simple Fuzzer did not change during the runs. This is why Figure 2 does not show any variance in the final coverage. The coverage of the Probable Fuzzer fluctuated slightly between 1110 and 1115 branches during the test runs. The coverage of the Random Fuzzer varied significantly more from 610 to 783 branches.
It can also be observed that the fuzzers find a lot of new branches in the first minute and that the coverage almost stops to increase after that.

Valid Coverage

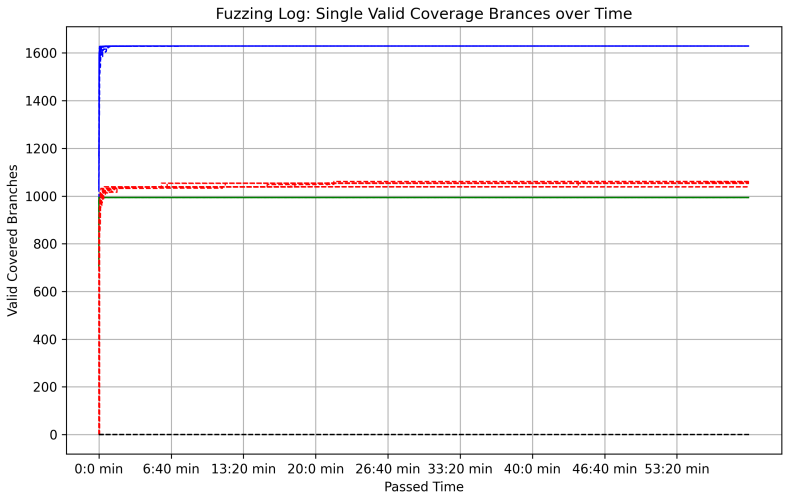

Fig. 3: Valid Coverage over time

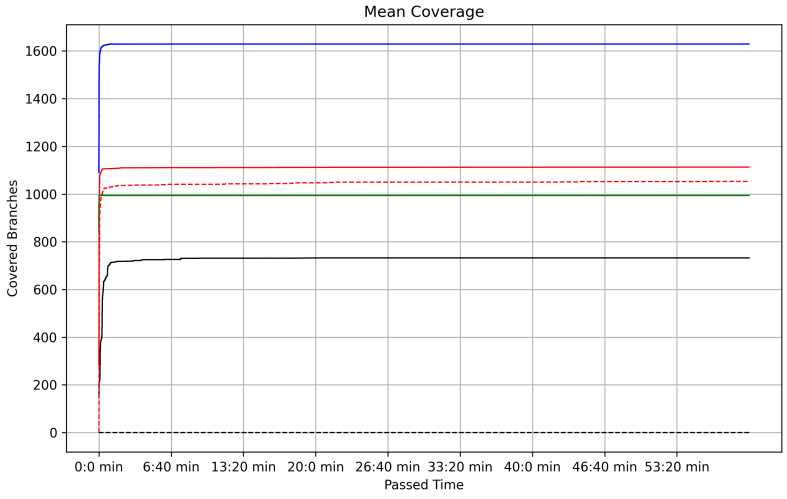

Fig. 4: Mean Coverage over time (dotted graphs are valid coverage)

The valid coverage of the Complete Fuzzer and the Simple Fuzzer is identical to their total coverage. The Probable Fuzzer covers between 1038 and 1061 valid branches. That is slightly less than its total coverage. The Random Fuzzer had no valid coverage in our tests.
We checked the statistical significance of the differences in coverage using Mann-Whitney U tests. However, it is already clear from the figures that there is no

overlap between the measurements of the fuzzers and that the differences are therefore significant.

6 Result Discussion

6.1 Discussion

As expected, the Complete Fuzzer provides the greatest coverage. As it contains many optional chunks and features, it can utilise many of PNGJ's functions. This becomes particularly clear when it is compared with the Simple Fuzzer, which covers a much smaller number of branches due to its limited options. The Probable Fuzzer has decent coverage for its relatively low implementation effort. However, due to the complexity of a PNG file, it is very unlikely to randomly generate one of the options that would enable even higher coverage. The same applies to the Random Fuzzer.
The fact that the fuzzers reach their maximum coverage after about a minute may be explained by the limited complexity of the PNGJ library as well as our test driver. The final coverage of the Probable and Random Fuzzers varies between runs, as sometimes options are found randomly that lead to more coverage and sometimes not. With the Complete and Simple Fuzzer, on the other hand, the structure is more strictly predefined, so it is more likely to generate all possible options, which seems to have happened during our runs.
When testing the Random Fuzzer, a Heap Error sometimes occurred which caused our fuzzing run to crash. We have not yet been able to find the cause of this error.
The valid coverage of the Complete Fuzzer and the Simple Fuzzer is identical to their total coverage, because these two Fuzzers only generate valid PNG-files. The valid coverage of the Probable Fuzzer is on average slightly below its total coverage, which should be due to the fact that this fuzzer can generate all critical chunks correctly. The random fuzzer has no valid coverage, as it is very unlikely to generate a correct PNG file at random.

6.2 Conclusion

Our generator creates valid PNG files that can contain all critical and optional chunks. This gives us significantly higher total and valid coverage on our fuzzing targets than simpler fuzzers. PNG is a relatively complex file format in which there are many dependencies between individual components. Therefore, a PNG generator needs a fairly fixed structure to generate valid files. Generating PNGs is thus not efficient with simple fuzzers that are largely based on chance.

6.3 Future Work

Our generator could be tested on other more complex fuzzing targets. In addition, the JQF guidance could be adapted to possibly deliver better results with our generator.

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. Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. 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, 12 pages. https://doi.org/10.1145/3293882.3330576
3. PNGJ GitHub-Page, https://github.com/leonbloy/pngj?tab=readme-ov-file
4. PNGJ Samples, https://github.com/leonbloy/pngj/tree/master/src/test/java/ar/com/hjg/pngj/samples
5. RFC 1951, Deflate Compressed Data Format Specification, https://datatracker.ietf.org/doc/html/rfc1951
6. Package `java.util.zip`, https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/util/zip/package-summary.html
7. LibPng: File structure, http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html
8. LibPng: Chunk specification, http://www.libpng.org/pub/png/spec/1.2/PNG-Chunks.html
9. LibPng: Deflate algorithm, http://www.libpng.org/pub/png/spec/1.2/PNG-Compression.html
10. LibPng: Filtering, http://www.libpng.org/pub/png/spec/1.2/PNG-Filters.html
11. LibPng: Data representation, http://www.libpng.org/pub/png/spec/1.2/PNG-DataRep.html