Force Open

Digital Investigation 20 (2017) S75–S82

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 Europe — Proceedings of the Fourth Annual DFRWS Europe

Force Open: Lightweight black box file repair

Karl Wüst [a, *], Petar Tsankov [a], Saša Radomirović [b], Mohammad Torabi Dashti [a]

[a] Department of Computer Science, ETH Zurich, Switzerland
[b] School of Science and Engineering, University of Dundee, United Kingdom

ARTICLE INFO

Article history:
Received 26 January 2017
Accepted 26 January 2017

Keywords:
File repair
Execution hijacking
Black box testing
Program instrumentation

ABSTRACT

We present a novel approach for automatic repair of corrupted files that applies to any common file format and does not require knowledge of its structure. Our lightweight approach modifies the execution of a file viewer instead of the file data and makes use of instrumentation and execution hijacking, two techniques from software testing. It uses a file viewer as a black box and does not require access to its source code or any knowledge about its inner workings. We present our implementation of this approach and evaluate it on corrupted PNG, JPEG, and PDF files.

Introduction

Corrupted files may occur in many situations, e.g. due to errors in data processing or failures of storage media. Many of these corrupted files still contain most of their information but cannot be opened by a file viewer due to small corruptions in important parts of the file.

It is obvious that reconstructing a corrupted file to its original form is not possible in general. However, this is often also not necessary for the reconstructed file to be usable. Instead, it may be sufficient if the reconstructed file is sufficiently *similar* to the original. Hence an attempted file repair can be considered successful if the resulting file meets the following conditions:

1. A validation program opens the file without crashing or error.
2. The file contains most of the information contained in the original.
3. The file contains very little information that is not present in the original.

Existing approaches to file repair often scan the data of a corrupted file for known patterns. For example, an approach for the repair of corrupted files compressed with the DEFLATE algorithm (e.g. ZIP archives) (Brown, 2011, 2013) scans for patterns such as packet headers and is able to then partially reconstruct a file using information about the content (e.g. English text). However, such approaches are not only limited to repairing specific file formats but also require manual effort in defining how the corrupted patterns are fixed.

In this paper, we present *Force Open*, a novel approach for automatic file repair that offers numerous advantages over existing approaches to file repair. First, Force Open is a black box approach, i.e. it is file format independent and only requires access to a program binary and some valid files of the format of the file that we wish to repair. Second, Force Open does not modify the file but instead modifies the execution of the file viewer, forcing it to open a corrupted file. This is achieved by using binary instrumentation to record executions of the program for multiple valid files to learn how the program behaves if the input file is valid. The recorded information consists of all the branches (i.e. jump instructions and their destinations) where the program behaviour is the same for all valid input files. This learned behaviour is then enforced when opening a corrupted file, where the program execution is hijacked and forced to follow an execution path based on the recorded behaviour for valid input files.

We implemented our Force Open approach and evaluated it for PNG, JPG and PDF files. In our experiments, our approach performed comparably to existing tools—PixRecovery for PNG and JPG files and pdftk for PDF files—in terms of the number of successfully repaired files. More importantly, for all file formats, the majority of the files repaired by our approach were not repaired by these reference tools. Our Force Open approach therefore complements these tools. The total number of repaired files on average increases by 84.82% for PNG files, 35.92% for JPG files, and 31.30% for PDF files.

* Corresponding author.
E-mail addresses: karl.wuest@inf.ethz.ch (K. Wüst), ptsankov@inf.ethz.ch (P. Tsankov), s.radomirovic@dundee.ac.uk (S. Radomirović), mohammad.torabi@inf.ethz.ch (M. Torabi Dashti).

http://dx.doi.org/10.1016/j.diin.2017.01.009

Contributions

This paper makes the following contributions:

- We propose a novel black box approach for repairing corrupted files without requiring any knowledge about the file format. Our approach is based on modifying the execution of a file viewer instead of fixing a file directly.
- We present a set of algorithms to realise this approach, consisting of a training algorithm that learns the behaviour of a file viewer for valid input files and an algorithm that enforces this behaviour for invalid input files.
- We implement and evaluate the approach for PNG, JPG, and PDF files. Our results indicate that the Force Open approach can be used to improve the repair quota of existing file repair programs significantly.

Related work

The most closely related research to that presented here is Docovery (Kuchta et al., 2014), a tool that uses symbolic execution, another technique from software testing, to reconstruct documents without prior knowledge about the underlying file format. The approach presented in Docovery is, however, not a complete black box approach, as it requires access to the source code of the used file viewer for the symbolic execution. In addition, our approach is much more lightweight as it does not require expensive symbolic execution.

Other research regarding file repair is specific to a single file format and requires in depth knowledge thereof, such as Brown (2011, 2013), and Sencar and Memon (2009).

Less closely related research includes the technique for *input rectification* presented by Long et al. (2012), which sanitises inputs to application such that they resemble typical inputs to prevent the exploitation of security vulnerabilities. Similar to the approach that we present here, their approach uses a training phase to collect information about typical input files. However, while our approach modifies the program execution, their approach modifies the input data and is used for input sanitization instead of file repair.

Preliminaries

A file may be defined to be corrupted whenever it contains any form of error. However, this definition is too broad for many practical purposes. Take for example a large picture, where the colour of one pixel is changed from blue to green. For almost all purposes of the picture, this change is unnoticeable and thus the corruption is of limited relevance. In many such cases, it is impossible to decide whether a file is corrupted or not.

We will use a narrower definition of file corruption here. File format specifications explicitly or implicitly define a finite set of constraints for files. In the context of this paper, a file is corrupt if it does not satisfy the specification's constraints:

Definition 1. *Let a file format specification be given. A file is corrupt with respect to the file formal specification if it violates at least one of the specification's constraints.*

With this notion of corruption, it is trivial to repair any corrupted file by simply assigning it to a file that meets a given file format specification. We therefore require that a repaired file contains useful information and that it does not introduce false information. Thus a successful file repair must optimize two parameters: It must contain as much information of the original file as possible and introduce as little information that is not contained in the original as possible.

Since a corrupted file violates some constraints, common approaches to file repair try to modify the file so that it meets all constraints. In order to do this, one first needs to know all constraints and second one needs to solve the constraints in a way that preserves as much of the file as possible. Finding and solving all constraints is difficult and requires knowledge about the file format specification.

A key insight that powers our Force Open approach is that the satisfaction of these constraints often manifests in the execution of the file viewer. For example, a file viewer will always take a certain branch if a checksum constraint is satisfied or when a file starts with the correct file signature.

Let us take for example a valid PNG file. One of the constraints given by the PNG specification (World Wide Web Consortium, 2003) is:

> "The first eight bytes of a PNG datastream always contain the following (decimal) values: `137 80 78 71 13 10 26 10`"

This constraint is manifested, for instance, in the function `png_read_sig` in `pngrutil.c` of libpng in the snippet shown in Listing 1.

```
if (png_sig_cmp(info_ptr->signature, num_checked, num_to_check) != 0)
{
   if (num_checked < 4 &&
       png_sig_cmp(info_ptr->signature, num_checked, num_to_check - 4))
      png_error(png_ptr, "Not a PNG file");
   else
      png_error(png_ptr, "PNG file corrupted by ASCII conversion");
}
```

Listing 1: Code snippet that checks for PNG signature.

The function `png_sig_cmp` that is called here, compares the whole signature or parts of the signature of the file to the correct value. In the execution of this code snippet, a file that does not fulfil the file signature constraint, e.g. a file that starts with `137 80 78` **`200`** `13 10 26 10` instead of `137 80 78 71 13 10 26 10` will produce an error here. Valid files satisfy the constraint and thus the execution will always behave the same, i.e. the program will always take the true branch (Fig. 1).

Our approach builds on this and instead of changing the file itself modifies the execution of the file viewer to behave as if its input file is valid.

The intuition why this approach may work for file repair is that the files of the same file format share many similar traits, not only the file signature. For example, other common constraints for files are checksums for some file data. These constraints are manifested by integrity checks that many file viewers use to decide whether the files are valid or corrupted. A file viewer may refuse to open a

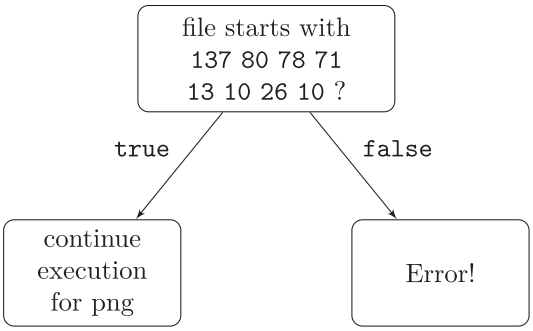

Fig. 1. Checking a PNG file signature in a program execution.

file due to such a check, even though the file might still be recognisable if this check is simply skipped.

These integrity checks are often necessary to prevent unwanted behaviour such as segmentation faults or possible security vulnerabilities. Typically, an integrity check will compute a checksum of the file data and compare this to a checksum that is stored within the file. If the checksums match, the program continues its execution, otherwise it will abort and show some error message (see Fig. 2). However, failing an integrity check does not mean that bad behaviour would necessarily occur, if the execution had been continued. This implies that there exists a subset of corrupted files that do not induce bad behaviour if the checks are ignored even though they will fail an integrity check, i.e. these are files that can be opened by forcing the file viewer to follow the path that continues its execution and thus behaving as if its input is a valid file.

Similarly, traits that are present in all valid files of a file format are assumed to exist in the corrupted file and checks for their existence are bypassed, e.g. we would expect a PNG file that starts with the invalid file signature `137 80 78 200 13 10 26 10` to be nevertheless opened if the program is forced to behave as if the first condition in Listing 1 evaluates to `false`.

In order to accomplish this task, our tool is trained by opening valid input files with a file viewer and collecting information about these executions. In a second step, the program then forces the execution of the file viewer based on the collected information to open the corrupted file. Our tool uses binary instrumentation and execution hijacking and thus does not require access to the source code of the instrumented program.

File repair

In this section we formalize the file repair problem.

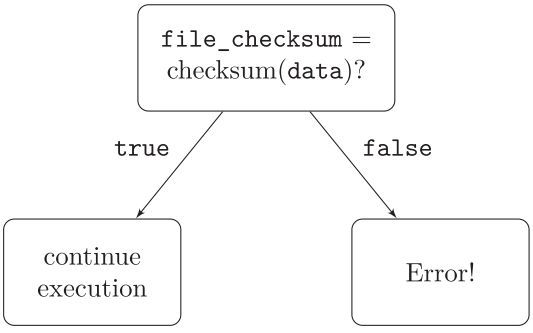

Fig. 2. Checksum check in a program execution.

Terminology

We start with several preliminary definitions. Let I be a set of *inputs* and O a set of *outputs*. We do not further specify I and O. In practice, I and O are infinite sets that contain all finite binary sequences. For example, I may contain binary sequences that represent PNG and JPEG files, while O contains binary sequences send out to output components, such as the RGB values for each pixel of the user's screen. The inputs I are typically stored as files, and we use the terms input, file, and input file interchangeably.

A *specification* $S : I \to O$ is a partial function mapping input to outputs. We call an input file i *valid* for a specification S if $S(i)$ is defined, and otherwise we call i *invalid*. We write I_S for the set of valid inputs for S. For example, the PNG specification (World Wide Web Consortium, 2003) defines the set of valid PNG files. The output of a program that implements a specification S is undefined for any invalid input file. In practice, the program returns an error message for invalid input files. If the program does not detect that a file is invalid it may also throw a runtime error and crash.

The set I_S of valid files for a specification S is typically defined by a set of constraints. For example, the constraint given in Section Preliminaries, which states that the first eight bytes of a PNG file must be `137 80 78 71 13 10 26 10`, defines a constraint that all PNG files must satisfy. We extensionally define a constraint C as a set of files, and the set I_S of valid files is then the intersection of all constraints. An input file i satisfies a constraint C if $i \in C$. Formally, the example PNG constraint contains all files that indeed start with the bytes `137 80 78 71 13 10 26 10`. Note that such constraints are usually infinite sets, and they are intensionally defined as a computable function $\psi_C : I \to \{0, 1\}$ that returns 1 if $i \in C$ and otherwise $\psi_C(i) = 0$.

File repair

File corruptions change the contents of a file and can turn a valid file into an invalid one. We denote the corrupted version of a file i by $\widehat{i}$. As illustrated in Section Preliminaries, even minor modifications to a file can make it invalid. Such modifications typically render the corrupted file unusable, simply because most implementations of the specification refuse to open invalid files. The file repair problem is to mitigate this issue by computing an output o that is similar to $S(i)$ using only the corrupted version of the file (i.e. $\widehat{i}$) and (an implementation of) the specification S. To formalize the notion of similarity, we assume a similarity measure $\sigma : O \times O \to [0, 1]$ that quantifies the similarity between two outputs. For example, a similarity measure between two images displayed on a screen may return the fraction of pixels that differ. Given two image outputs o and $o\prime$, $\sigma(o, o\prime)$ is then 1 if all pixels defined by o and $o\prime$ have identical RGB values. We write $o \approx o\prime$ if $\sigma(o, o\prime) \geq \theta$, where $\theta \in [0, 1]$ is a fixed similarity threshold. Formally, we define the file repair problem as follows.

Definition 2. *Given an invalid file* $\widehat{i}$ *and a specification* S, *find an output* o *such that* $o \approx S(i)$.

Several remarks are due. First, if the invalid file $\widehat{i}$ is too different from the original valid file i, then the file repair problem may be impossible to solve. It is indeed not possible to repair a file i if all bytes of i are replaced by, e.g., zeros. Second, the file repair problem is trivial if we set the similarity threshold θ to 0. This is because if $\theta = 0$ then one can use an arbitrary valid file i to compute an output o from the range of S.

Most existing work on file repair aims to modify the invalid file $\widehat{i}$ and turn it into a valid file i' that is similar to $\widehat{i}$. Typically, this boils down to modifying as few bytes of $\widehat{i}$ as possible so as to transform $\widehat{i}$ into a valid file. Several approaches attempt to first discover which

constraints are violated by $\widehat{i}$ and then use constraint solving to change $\widehat{i}$ into a valid file i'. For example, if the first eight bytes of a corrupted PNG file do not match those prescribed by the PNG specification, then one can replace the first eight bytes by the expected sequence of bytes to derive a file i'; if i' is valid then $S(i')$ is a solution to the file repair problem. Constraint solving can be, however, prohibitively expensive if the specification defines complex constraints. File repair approaches based on constraint solving thus typically focus on a portion of $\widehat{i}$'s bytes, namely those that are most likely responsible for violating constraints.

The Force Open approach

Our program works in two phases. We will call the first phase *training phase* and the second phase *force open phase*. The training phase is used to record information about the execution of the file viewer when opening valid files, and is described in detail in Section Training phase. In the force open phase, the file viewer is then forced to behave as if the input file is valid with the hope that a corrupt input file will be opened successfully. This phase is described in detail in Section Force open phase. A graphical overview of the complete workflow is given in Fig. 3.

Training phase

The training phase is used to gather information about the branches taken by the file viewer during the execution with valid input files. For each branch that is taken during the execution, we record

- the location of the branch in the program and
- the location of the instruction that is executed after branching

We call the algorithm used to collect this information *branchtrace* (Algorithm 1). This algorithm executes the file viewer with an input file and for every branch on the executed path we record a tuple $(source, dest)$ storing the location of the branch ($source$) and the location of the program counter after the branch statement has been executed ($dest$). These pairs are added to a list, which is returned by the algorithm after the file viewer has finished its execution.

Algorithm 1 Branchtrace

```
function BRANCHTRACE(program, file)
    branches ← empty list
    for all branch in the execution of PROGRAM(file)
do
        source ← location of branch
        dest ← destination taken by branch
        append (source, dest) to branches
    end for
    return branches
end function
```

The list returned by the branchtrace algorithm is by itself only of limited usefulness. It is simply a record of the execution path taken by a single execution of the file viewer with a specific input file. As the execution paths differ for different valid input files, we need to collect only the branches that have the same behaviour for all valid files. However, we cannot collect information about the executions with *all* valid input files. Therefore, we need a *training algorithm* that collects information from executions with many different valid input files and combines the collected information in a meaningful way.

The *SameBranchBehaviour* algorithm (Algorithm 2) is a simple training algorithm. It takes a list of correct files and a program as input and creates an empty set *branches*. It then loops through all files in the file list and for each file executes the *branchtrace* algorithm (Algorithm 1). The branches returned by *branchtrace* are added to *branches*.

After the loop is completed, the algorithm checks for every tuple of the form $(source, dest)$ in branches whether an entry $(source, dest2)$ with $dest \neq dest2$ exists in *branches*. If that is the case, both tuples are removed from *branches*. The algorithm then returns the set *branches* which contains exactly the branches that always have the same outcome for all valid files. Thus, the output of the

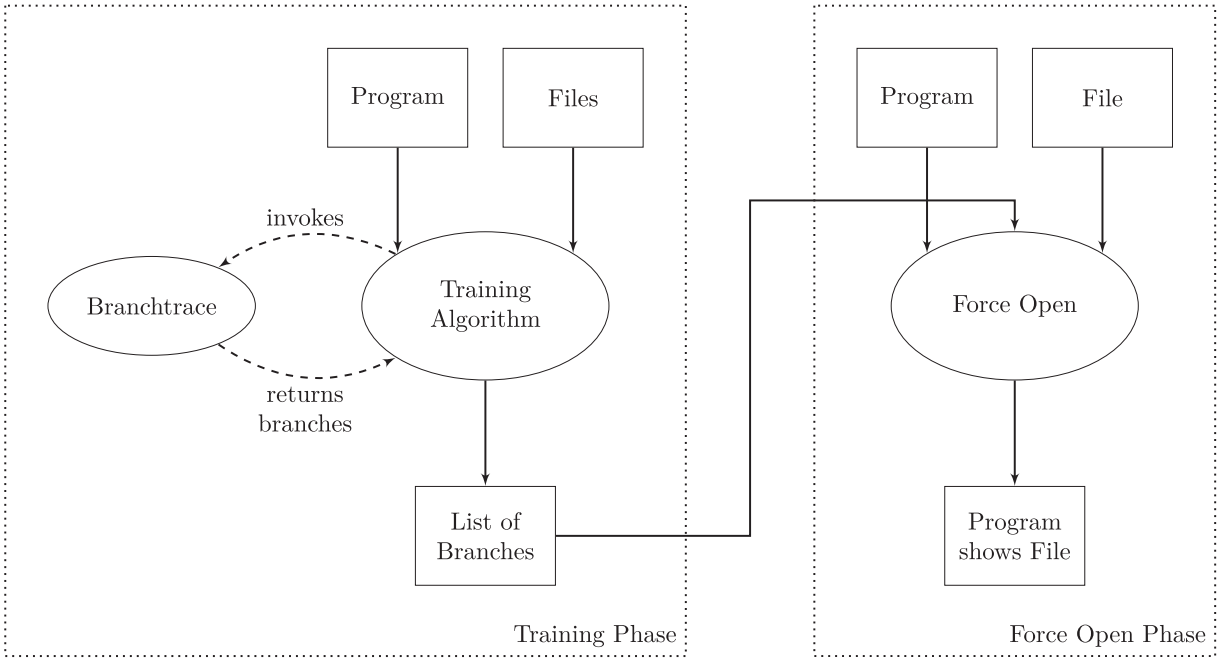

Fig. 3. Force Open workflow.

algorithm contains exactly those branches for which the behaviour is the same for all input files.

If all recorded traces consist of exactly the same set of branches (ignoring the outcome), this algorithm will produce the intersection of the recorded tuples. However, we cannot assume that all traces will contain the same branches, which is why we do not simply compute the intersection. In fact, two traces may consist of almost completely disjoint sets of branches, i.e. they could only share a single branch (with different outcomes, since the execution paths differ from that point onwards) causing the intersection of tuples to be the empty set.

Algorithm 2 SameBranchBehaviour

```
function SameBranchBehaviour(filelist, program)
    branches ← {}
    for all file ∈ filelist do
        temp_branches ← branchtrace(program, file)
        branches ← branches ∪ temp_branches
    end for
    for all (source, dest) ∈ branches do
        for all (source, dest2) ∈ branches where dest ≠ dest2 do
            branches ← branches \ (source, dest)
            branches ← branches \ (source, dest2)
        end for
    end for
    return branches
end function
```

Force open phase

The *force open* algorithm (Algorithm 3) uses execution hijacking in order to force the behaviour of the file viewer. It takes a program, a file and a list of branches as input. The list of branches could in general be any list of branches, but it is intended to be a list compiled by a training algorithm such as Algorithm 2 from Section Training phase.

The algorithm loops through all branches in the input program and for every branch checks whether the branch is contained in the list of branches. If that is the case, the branch in the program is replaced by an unconditional jump to the destination stored in the branch list for this specific branch. After the loop is finished, the modified program is executed with the input file as argument.

Algorithm 3 Force Open

```
function ForceOpen(program, file, branches)
    for all branch in program do
        source ← location of branch
        if source is a key in branches then
            dest ← branches[source]
            replace branch in program with goto dest
        end if
    end for
    program(file)
end function
```

Implementation

The Pin framework (Luk et al., 2005) can be used to write custom dynamic instrumentation tools, called *pintools*, that work directly on x86 and x86–64 binaries. By using Pin to implement the *branchtrace* and the *force open* algorithms, our approach becomes a complete black box approach that does not require any knowledge about the internal workings of the file viewer. Both the *branchtrace* and the *force open* algorithm are implemented as pintools in C++. As the pintools work directly on the binaries and not on the source code, a *branch* as used in the algorithms is any conditional jump instruction.

The jump instructions and their destinations are identified by their memory addresses in a single execution of the file viewer. However, as many libraries are loaded dynamically and their location in the address space changes from one execution to the other, the memory address of the jump instruction is not sufficient to identify the jump for multiple executions of the program. Instead, we use the name of the module in which the jump instruction is located together with the offset of the jump instruction to the base address of the module.

The *branchtrace* pintool takes a program and corresponding program arguments (e.g. the name of a file that should be opened) as input. The *branchtrace* pintool instruments the specified program by inserting a function call after each conditional jump that records the location of the jump and the location of the instruction that will be executed next and writes them to a file.

The *force open* pintool takes a file containing jump locations and corresponding target locations as input, in addition to a program including arguments. It then instruments the program by inserting an unconditional jump to the respective target before each jump contained in its input file and executes it.

The *SameBranchBehaviour* algorithm (Algorithm 2) was implemented in Python. The script takes multiple files as input and either a setup flag or a list containing branch information. If the setup flag is set, the program creates an empty dictionary to store the branch information and loops through the input files, invoking the instrumented file viewer instrumented to record branch information and adding the recorded entries to the dictionary as described in Algorithm 2. At the end, the dictionary entries are written to a file.

When the setup flag is not set and a branch list is given as input, the Python script acts as a wrapper for the force open pintool, allowing for multiple files to be opened. It loops through the specified files and for each hijacks the execution of the program according to the branch list.

Evaluation

We evaluated our approach for the PNG, JPG and PDF file formats.[1] The file viewer used for PNG and JPG is the *feh* image viewer.

[1] The tools and the data sets are available at https://github.com/fldpi/forceopen.

For PDF files we used the *pdftotext* command-line utility. We compare our results for PNGs and JPGs with the results of PixRecovery, a commercial file repair tool for image files, and for PDFs with the results of pdftk, a command-line utility for manipulating and repairing PDFs.

Test conditions and environment

The machine used for testing uses a 3.4 GHz Intel Core i7-4770 CPU, has 32 GB of DDR3-RAM, and is running a 64-bit Linux distribution. The Python scripts used for the implementations were modified to measure total time, CPU-time, and maximum and average memory usage. In order to create comparable conditions and to increase usability, an option was added to the scripts to automatically close opened pictures. The wrapper that calls the force open pintool for each specified file uses a controller that kills the process of the file viewer if no corresponding window is detected within a reasonable amount of time.

Generating corrupted files for testing

The algorithm that we use to generate corrupted files for our experiments is straightforward. In order to produce files that satisfy our definition of corrupted file, we need to generate files that violate the constraints implied by the file format specification. We approximate this by ensuring that the program that would normally be used to open the file, can no longer open it. We assume that the file viewer is able to open all files that fulfil the constraints. While there may be files that violate constraints and cannot be produced using this method, the corrupted files that are generated will fit our definition. The generated set of corrupted files is thus at least as difficult to repair as a random sample of corrupted files. Our algorithm to generate corrupted files (Algorithm 4) takes as input a program, a file, and a number n specifying the number of corrupted bytes. The algorithm then creates a copy of the file, chooses a byte-aligned position between 0 and $(size(file) - n)$ at random, and overwrites the n bytes starting at that position with random data. It then tries to open the modified copy with the specified program. If the program fails to open the copy, the algorithm returns the copy, otherwise the algorithm is repeated.

Algorithm 4 Corrupt

```
function CORRUPT(program, file, n)
    repeat
        copy ← file
        position ←u.a.r. {0, size(file) − n}
        copy[position : position + n − 1] ← random data
    until PROGRAM(copy) fails
    return copy
end function
```

PNG

The training set for the PNG file format consists of 200 valid PNG files of different sizes. The test set (disjoint from the training set) consists of PNG files that were corrupted with the algorithm described in Section Generating corrupted files for testing with corruption sizes of 2^k bytes for $k \in \{0, 1, 2, 3, 4\}$. We compare the reconstructed images to the original (not corrupted) images using the pHash library (Zauner, 2010), which determines whether two images are visually similar by taking into account different possible transformations on the pictures. Note that in addition, a manual inspection of reconstructed images showed that most of them are visually the same as the original with no or only small artefacts.

In our tests, the training phase needed 35 min to complete, while using 182.39% of the CPU (where 100% is one core) with a maximum memory usage of 167 MB. The number of successfully displayed files (i.e. the files that are opened without producing an error) is shown in Table 1, as is the number of files that are recognised by the pHash library as similar to the original. As expected, the success rate decreases as the number of modified bytes increases. The time to open (or failing to open) one image is on average around 15 s of CPU time on the test machine for 1-byte corruptions and 67 s of CPU time for 16-byte corruptions (due to fewer images that can be opened) with an average memory usage of approximately 250 MB. For some files, the process seems to introduce a memory leak, peaking at a memory usage of almost 19 GB. As these peaks occur for very few files that cannot be opened, this could be easily prevented by monitoring the memory usage and aborting the child process if some threshold is exceeded.

For the repaired PNG files with single byte corruptions, the types of corruptions are listed in Table 2. The PNG format is a lossless image compression format. It consists of an 8-byte file header (containing the file signature) and a series of *chunks*. Every chunk consists of a 4-byte *length* field, containing the length of the chunk data, a 4-byte *type* field, the *chunk data* and a 4-byte CRC checksum. Each chunk is either *critical* (IHDR, PLTE, IDAT, IEND) or *ancillary* (i.e. not strictly necessary to decode a file correctly). The IHDR chunk is always the first chunk in the file and specifies attributes such as image dimensions and colour type. It always has the same length. The PLTE chunk specifies the colour palette, this chunk may be optional depending on the specified colour type. There may be multiple IDAT chunks containing the DEFLATE (Deutsch, 1996) compressed image data (World Wide Web Consortium, 2003).

When we take a closer look at the files that are corrupted in one byte and compare the results of our test with the analysis of all corrupted files (Table 2), we see that certain types of corruptions are handled better than others. Most of the repaired corruptions are of a type that we would expect to be repaired successfully. The image header is always the same for each file, as is the location and length of the IHDR chunk. The repaired corruptions (except 1) in the IHDR data are corruptions in the parts specifying either the “filter method” or the “compression method”, for both of which there is only one method specified in the PNG standard (i.e. they contain the same values for all files). Other corruptions in the IHDR data, that are different for most files, are, however, almost never repaired. Corruptions within CRC checksums are also handled well with our approach, which, as mentioned in Section The Force Open approach, fits our expectations. While not all of the corruptions that we would expect to be handled well with our approach (e.g. some CRC checksums) are actually repaired, most of them are (see Table 2).

We used the commercial tool PixRecovery to repair the same test files that were used to test our tool. The results of this test are shown in Table 1. As one can see, the repair rates of our tool and PixRecovery are comparable. However, there is no subset relation between the sets of repaired files.

In fact, the number of files that are only repaired by our tool is larger than the overlap between the sets of reconstructed files (Fig. 4a & Table 1). Thus, our approach can be used to improve the existing heuristics significantly.

Table 1
A summary of our results showing the number of files repaired with Force Open (FO) and a Reference Tool (RT), which is PixRecovery for PNG and JPG, and pdftk for PDF.

	Corrupt bytes	Number of files	Repaired files									
			Force Open (FO)		Ref. Tool (RT)		FO and RT		Only FO		Only RT	
PNG	1	597	306	(51.26%)	207	(34.67%)	144	(24.12%)	162	(27.14%)	63	(10.55%)
	2	624	305	(48.88%)	174	(27.88%)	119	(19.07%)	186	(29.81%)	55	(8.81%)
	4	612	215	(35.13%)	155	(25.33%)	88	(14.38%)	127	(20.75%)	67	(10.95%)
	8	632	129	(20.41%)	98	(15.51%)	46	(7.28%)	83	(13.13%)	52	(8.23%)
	16	630	54	(8.57%)	58	(9.21%)	25	(3.97%)	29	(4.60%)	33	(5.24%)
	1 – 16	3095	1009	(32.60%)	692	(22.36%)	422	(13.63%)	587	(18.97%)	270	(8.72%)
JPG	1	248	37	(14.92%)	58	(23.39%)	9	(3.63%)	28	(11.29%)	49	(19.76%)
	2	256	30	(11.72%)	45	(17.58%)	14	(5.47%)	16	(6.25%)	31	(12.11%)
	4	249	20	(8.03%)	34	(13.65%)	6	(2.41%)	14	(5.62%)	28	(11.24%)
	8	253	14	(5.53%)	25	(9.88%)	3	(1.19%)	11	(4.35%)	22	(8.70%)
	16	255	6	(2.35%)	44	(17.25%)	1	(0.39%)	5	(1.96%)	43	(16.86%)
	1 – 16	1261	107	(8.49%)	206	(16.34%)	33	(2.62%)	74	(5.87%)	173	(13.72%)
PDF	1	312	30	(9.62%)	33	(10.58%)	3	(0.96%)	27	(8.65%)	30	(9.62%)
	2	320	22	(6.88%)	41	(12.81%)	3	(0.94%)	19	(5.94%)	38	(11.88%)
	4	349	16	(4.58%)	49	(14.04%)	0	(0.00%)	16	(4.58%)	49	(14.04%)
	8	358	9	(2.51%)	48	(13.41%)	2	(0.56%)	7	(1.96%)	46	(12.85%)
	16	384	8	(2.08%)	59	(15.36%)	5	(1.30%)	3	(0.78%)	54	(14.06%)
	1 – 16	1723	85	(4.93%)	230	(13.35%)	13	(0.75)	72	(4.18%)	217	(12.59%)

Table 2
Types of corruptions (PNG).

Chunk	Field	# of files	# of repaired files
File header		74	59
IHDR	Chunk type	32	9
IHDR	Chunk length	30	26
IHDR	Chunk data (compression/filter)	25	18
IHDR	Chunk data (other)	106	1
IHDR	CRC	44	33
IDAT	Chunk type	55	20
IDAT	Chunk length	1	0
PLTE	Chunk type	3	0
PLTE	Chunk length	3	0
PLTE	Chunk data	70	33
PLTE	CRC	5	2
Ancillary chunks	Chunk type	72	50
Ancillary chunks	Chunk length	77	49

JPG

The training set for the JPG file format consists of 79 valid JPG files of different sizes. The test set consists of JPG files that were corrupted with the algorithm described in Section Generating corrupted files for testing with corruption sizes of 2^k bytes for $k \in \{0, 1, 2, 3, 4\}$ (the same as for PNG). We again compare the reconstructed images to the original images using the pHash library (Zauner, 2010).

The training phase needed 6.2 min to complete, while using 133.90% of the CPU (where 100% is one core) with a maximum memory usage of 26 MB. The results are summarised in Table 1. The success rate again decreases as the number of modified bytes increases. The time to open (or failing to open) one image is on average around 19 s of CPU time with a maximum memory usage of approximately 50 MB.

We again compare our results to the results of PixRecovery and while the repair rates are higher for PixRecovery (Table 1), especially for larger corruptions, the overlap is again small, i.e. our approach still improves the overall repair rate significantly if used in addition to a traditional approach (Fig. 4b & Table 1).

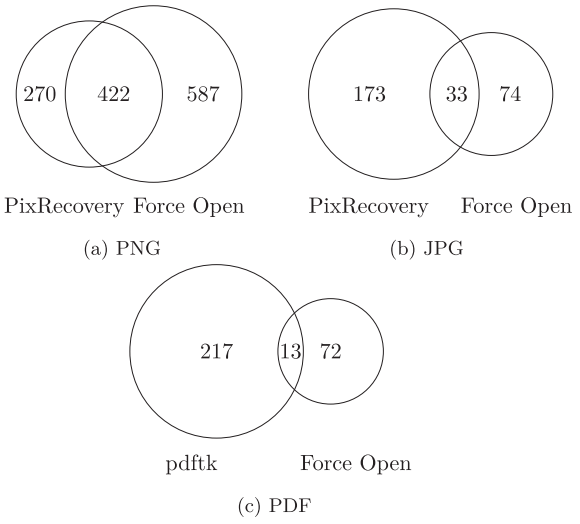

Fig. 4. Overlaps of repaired files.

PDF

For PDF files, we used to *pdftotext* command line utility as file viewer. The training set for PDF consists of 158 text based PDF files. The test set consists of text based PDF files that were corrupted with the algorithm described in Section Generating corrupted files for testing, again with corruption sizes of 2^k for $k \in \{0, 1, 2, 3, 4\}$. The output of the execution using our tool is then compared to the output of *pdftotext* when given the original as input using the Levenshtein distance as a metric.

The training phase needed 4 h and 44 min to complete, while using 216% of the CPU (where 100% is one core) with a maximum memory usage of 115 MB. The results are summarised in Table 1. For all files that were successfully opened, the Levenshtein distance of the produced output to the output of *pdftotext* with the original file as input is zero, i.e. the files are either completely repaired or not at all. The success rate again decreases as the number of modified bytes increases. The time to open (or failing to open) one file is on average around 7 s of CPU time with a memory usage of less than 90 MB (and peaks of up to 2.3 GB in rare cases).

For PDF, we compare our results to the results of pdftk (Table 1), which is much more successful for larger corruptions and comparable for single byte corruptions. However, the overlap of repaired documents is again small (Fig. 4c & Table 1), i.e. our approach repairs additional files and could thus be used as additional heuristic to improve the overall results.

Discussion

Limitations

The design of our approach inherently introduces some limitations. For Example, Force Open is not capable of repairing corruptions that are highly data dependant (e.g. image dimensions) and are not in the form of some if-then-else statement. Large corruptions, e.g. corruptions of the size of a file system block, also pose a problem, since they will usually introduce data dependency.

In addition, our method may introduce unwanted behaviour in the hijacked program. Our approach is designed to circumvent integrity checks and as such also introduces a possibility for malicious exploitation. This should be taken into account when applying our method by sandboxing its execution.

Different training algorithms

In addition to the training algorithm (Algorithm 2) described in Section Training phase, we also experimented with two other simple training algorithms. However, they did not work as well, which is why we only briefly describe them here. Both other training algorithms require not only valid files but additionally a corrupted version of each valid file.

The first alternative algorithm compares the executions of the file viewer for a valid file with the execution for its invalid version and records only the first differing branch, i.e. the branch where the execution paths first start to diverge. This training algorithm was not successful. By forcing the recorded branches with this algorithm, Force Open was unable to open a single file.

The second alternative algorithm also compares executions of the file viewer for a valid file with the execution for its invalid version. However, it records all branches where the execution paths differ, i.e. the set of collected branches is a subset of the branches collected with the *SameBranchBehaviour* algorithm (Algorithm 2). While this training algorithm was faster, the success rate for this training algorithm was significantly lower than for the *SameBranchBehaviour* algorithm.

Another advantage of the main training algorithm presents itself if one considers that when training with one of the alternative algorithms, we produce corrupted files for training with the same algorithm that we used to generate our test files. Thus, the types of corruptions in the test files are artificially generated with the same distribution as in the training files. This means that the results apply only to corrupted files for which the distribution of the types of corruptions can be simulated. The *SameBranchBehaviour* algorithm does not have this disadvantage since it does not use corrupted files for training.

Instrumenting only parts of the file viewer

It is possible to only instrument some modules of the file viewer. For example, we can exclude some library that is assumed to have no influence on the decision of whether a file is valid or not. This suggests two possible advantages:

- We remove the possibility of erroneously inserted jumps that may cause the program to fail.
- The instrumentation functions are called fewer times, resulting in a faster runtime.

However, selecting the optimal configuration of instrumented modules is a difficult problem. For example, the image viewer *feh* uses approximately 70 modules when opening a PNG file. Testing all possible combinations is therefore infeasible. Manually selecting modules that seem to play a part in deciding the validity of a file is possible, but this requires some deeper knowledge about the file viewer or the file format, i.e. this would no longer constitute a complete black box approach.

Nevertheless, we experimented with manual module selection to investigate whether the success rate increases and whether the speed is improved. In our experiments, the tested module configurations performed comparably to instrumenting all modules with regards to the number of successfully repaired files while being slightly faster due to less instrumentation.

This means that in principle, one can achieve similar success rates while being slightly quicker by instrumenting only parts of the file viewer. However, since this requires manual intervention, the process is no longer completely automated and requires additional knowledge for each file format. It also adds a time overhead since some experimentation is needed to find a good configuration of modules. Thus, the benefits of instrumenting all modules outweigh the benefits of instrumenting fewer modules.

Conclusion

In this paper, we presented Force Open, a black box file repair solution based on binary instrumentation and execution hijacking that is adaptable to different file formats without requiring knowledge about said file format or the inner workings of the instrumented file viewer. It is a lightweight approach without the necessity of resource expensive techniques such as symbolic execution. We implemented and tested our approach with three different file formats and compared it to state of the art file repair solutions. In our experiments, we were able to show that our approach can be used to significantly improve the repair quota of existing programs.

References

Brown, R.D., 2011. Reconstructing corrupt DEFLATEd files. Digit. Investig. 8, S125–S131. http://dx.doi.org/10.1016/j.diin.2011.05.015. http://dx.doi.org/10.1016/j.diin.2011.05.015.

Brown, R.D., 2013. Improved recovery and reconstruction of DEFLATEd files. Digit. Investig. 10 (Suppl. (0)), S21–S29. http://dx.doi.org/10.1016/j.diin.2013.06.003. The Proceedings of the Thirteenth Annual DFRWS Conference 13th Annual Digital Forensics Research Conference. http://www.sciencedirect.com/science/article/pii/S1742287613000492.

Deutsch, P., 1996. Deflate Compressed Data Format Specification Version 1.3.

feh, http://feh.finalrewind.org/, (Accessed 10 December 2015).

Kuchta, T., Cadar, C., Castro, M., Costa, M., 2014. Docovery: toward generic automatic document recovery. In: International Conference on Automated Software Engineering (ASE 2014).

libpng, http://www.libpng.org/pub/png/libpng.html, (Accessed 10 December 2015).

Long, F., Ganesh, V., Carbin, M., Sidiroglou, S., Rinard, M., 2012. Automatic input rectification. In: Proceedings of the 34th International Conference on Software Engineering, ICSE'12. IEEE Press, Piscataway, NJ, USA, pp. 80–90. http://dl.acm.org/citation.cfm?id=2337223.2337233.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40 (6), 190–200. http://dx.doi.org/10.1145/1064978.1065034. http://doi.acm.org/10.1145/1064978.1065034.

pdftk, https://www.pdflabs.com/tools/pdftk-server/, (Accessed 10 December 2015).

PixRecovery, http://www.officerecovery.com/pixrecovery/, (Accessed 10 December 2015).

Sencar, H.T., Memon, N., 2009. Identification and recovery of JPEG files with missing fragments. Digit. Investig. 6 (Suppl.), S88–S98. http://dx.doi.org/10.1016/j.diin.2009.06.007. The Proceedings of the Ninth Annual DFRWS Conference. http://www.sciencedirect.com/science/article/pii/S1742287609000437.

World Wide Web Consortium (W3C), November 2003. Portable Network Graphics (PNG) Specification, second ed. http://www.w3.org/TR/2003/REC-PNG-20031110.

Zauner, Christoph, 2010. Implementation and Benchmarking of Perceptual Image Hash Functions (Master's thesis). Upper Austria University of Applied Sciences.