Decompression Quines and Anti-Viruses

Margaux Canet, Amrit Kumar, Cédric Lauradoux, Mary-Andréa Rakotomanga, Reihaneh Safavi-Naini

▶ **To cite this version:**

Margaux Canet, Amrit Kumar, Cédric Lauradoux, Mary-Andréa Rakotomanga, Reihaneh Safavi-Naini. Decompression Quines and Anti-Viruses. CODASPY 2017 - 7th ACM Conference on Data and Application Security and Privacy, Mar 2017, Scottsdale, United States. 10.1145/3029806.3029818 . hal-01589192v2

HAL Id: hal-01589192
https://inria.hal.science/hal-01589192v2

Submitted on 20 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decompression Quines and Anti-Viruses

(This is the author version of the paper to appear in CODASPY 2017.)

Margaux Canet
Univ. Grenoble Alpes

Amrit Kumar
Inria
Univ. Grenoble Alpes

Cédric Lauradoux
Inria

Mary-Andréa Rakotomanga
Univ. Grenoble Alpes

Reihaneh Safavi-Naini
University of Calgary

ABSTRACT

Data compression is ubiquitous to any information and communication system. It often reduces resources required to store and transmit data. However, the efficiency of compression algorithms also makes them an obvious target for hackers to mount denial-of-service attacks. In this work, we consider *decompression quines*, a specific class of compressed files that decompress to themselves. We analyze all the known decompression quines by studying their structures, and their impact on anti-viruses. Our analysis reveals that most of the anti-viruses do not have a suitable architecture in place to detect decompression quines. Even worse, some of them are vulnerable to denial-of-service attacks exploiting quines. Motivated by our findings, we study several *quine detectors* and propose a new one that exploits the fact that quines and non-quine files do not share the same underlying structure. Our evaluation against different datasets shows that the detector incurs no performance overhead at the expense of a low false positive rate.

Keywords

Compression; Quines; Anti-viruses; Denial-of-Service

1. INTRODUCTION

In a world of digital storage and transmission, the density of information within a piece of data is crucial to any information and communication system. To this end, data compression is a useful tool since it effectively reduces resources required to store and transmit data. The popularity of data compression can be easily judged from a W3C 2016 report which states that 68% of websites on the Internet support compression [18] and hence are capable of sending compressed data over the network.

Use of any effective compression algorithm however also comes with the attached security risks. The so-called *decompression bombs* also known as the *zip of death* exploit the efficiency of compression algorithms to mount denial-of-service (DoS) attacks. They often target web servers and anti-viruses. Unfortunately, despite the fact that these threats have been known for years, some security products still remain vulnerable [9, 13].

The common definition of a decompression bomb found in the NIST guide [10] or the textbook [11] considers a small innocuous compressed file that decompresses to a gigantic file. A typical example of a decompression bomb is `42.zip`[1] ($\approx$ 42 Kilobytes) that expands to 4.5 Petabytes. When a software attempts to naively decompress such a file, it consumes all the available memory and eventually crashes.

Decompression bombs are not restricted to the previous definition. In fact, they can be far more dangerous. The previous definition is often associated to bombs that decompress to a finite number of files. It is also possible to design bombs that decompress to an infinite number of files. The idea is to target software packages which decompress files recursively in order to recover all the data. In this case, an adversary can submit the bomb to force the software into running an infinite decompression loop. Such bombs can be characterized as a compressed file that decompresses to itself. We call this particular kind of bomb a *decompression quine*. This designation respects the original definition of quines found in Thompson's seminal paper [17]. It is worth noting that a decompression quine can also be viewed as a fixed point of the decompression function.

In this work, we present a comprehensive analysis of the known decompression quines for the DEFLATE [5] algorithm. DEFLATE is currently implemented in several popular compression routines such as `gzip`, `zlib` and `zip` among others. Based on our study of the existing quines, we generalize the idea presented in [2] to create new ones. The generalization allows us to produce an arbitrary quine inexpensively. Additionally, it renders the obvious signature-based detection impossible. We further conduct experiments with several anti-viruses and incident response frameworks and observe that some of them implement a detector which requires several decompressions to identify a quine. To this end, we also propose a new detection scheme for quines. The underlying objective is to design a faster detector than the ad-hoc ones.

We propose a *statistical detector*, where the core idea is to parse compressed files to recover the internal block structure of DEFLATE. We also test the performance of our detector against a corpus of compressed files. The results show that the statistical detector is systematically more efficient than the ad-hoc detectors that include a fixed-point detector and a detector based on bounded recursion. The efficiency of our detector comes at the expense of a low false positive rate.

2. RELATED WORK

The first strike of decompression bombs occurred in the

[1] https://www.unforgettable.dk/

90's with the goal of mounting a DoS attack against FidoNet systems [12]. Since then, several different types of decompression bombs have been designed to attack different services. Decompression bombs can be broadly classified into four types: *single (large) file bombs*, *nested bombs*, *self-reproducing bombs* (decompression quines) and *bogus bombs*.

The basic single file bomb exploits the efficiency of compression algorithms: a large file is compressed into a very small file. The website https://bomb.codes provides several examples of such bombs for the most popular compression formats. These compressed files are designed to crash software applications by consuming all the available memory.

Nested bombs are a composition of single file bombs. They are obtained by using archivers such as `zip` or `tar`. An example of a nested bomb is `42.zip`. The file has a nested level of 6. Each level other than the last one consists of 16 zipped files. The files at the last level have a size of 4.2 GB.

Self-reproducing bombs or decompression quines are the focus of our work. A (decompression) quine is a compressed file that decompresses to itself. The threat posed by a quine during decompression is that it may force the software application to fall into an infinite loop and thus create a DoS exploit. The first decompression quine was provided by Cox [2]. The existence of a decompression quine also means that it is possible to find two different compressed files that decompress to the same file: `r.gz` decompresses to `r.gz` and so does `r.gz.gz`. The work by Cox [2] is the starting point of our study and it is analyzed in depth in the next section.

Finally, bogus bombs target errors and bugs in the implementation of the decompression algorithm. In 2005, Chris Evans handcrafted a special `bzip2` file (http://scary.beasts.org/misc/bomb.bz2) that causes an infinite loop in the decompression algorithm of `bzip2`.

AERAsec Network Services and Security has published the first survey [15] on the threat of decompression bombs. It targets anti-viruses, web browsers and office suites using compressed binary files, compressed `HTML` and image files (such as `PNG`). Forensic tools are also an obvious target of decompression bombs. `bulk_extractor` due to Garfinkel [7] is one of the few tools hardened against this threat. In 2014, Koret [9] repeated the tests of AERAsec on anti-viruses. The author observed that despite the long history of decompression bombs, some anti-viruses were still vulnerable. Pellegrino *et al.* have discovered in [14] new vulnerabilities related to compression/decompression in `HTTP`, `XMPP` and `IMAP` protocols (server side). More recently at BlackHat 2016, Cara [13] has extended the results of [15] to the most popular image formats and compression formats supported by web browsers.

The countermeasures often mentioned in [9, 13, 14, 15] to thwart decompression bombs is to set two limits. The first limit is on the size of the decompressed file. It protects against single file bombs. The second is to limit the number of times a file can be decompressed. It protects against nested bombs and self-reproducing bombs. These two limits have been used in ARBOMB, the first decompression bomb detector [3].

All the related work on the impact of decompression bombs mainly focus on single file bombs and nested bombs. They however do acknowledge the existence of self-reproducing compressed files by citing the work of Cox [2]. Our goal is to study quines on software applications that allow recursive decompression. This leads to the obvious choice of testing against anti-viruses since they are designed to scan files and directories recursively.

3. DEFLATE QUINES

DEFLATE [5] is a lossless compression algorithm implemented in several popular compression routines including `gzip`, `zlib` and `zip`. We note that while, the term INFLATE has been used in the literature to refer to the associated decompression algorithm, in popular use though, the term DEFLATE often subsumes both compression and decompression algorithms. We also abide by the usage of the term DEFLATE to take both denotations.

In this section, we first present an overview of DEFLATE and then in the sequel, we present decompression quines based on DEFLATE.

3.1 Deflate Compression

DEFLATE is based on the LZ77 compression algorithm [19] and the Huffman coding [8]. In fact, LZ77 is the core compression algorithm, the output of which is further compressed using Huffman coding.

The idea underpinning LZ77 compression is to find sequences of data that are repeated. This is implemented in practice by maintaining a record of the previously seen data. To this end, a sliding window is employed. Whenever a next sequence of bytes is identical to one that can be found in the sliding window, the sequence is replaced by a pointer where it can be found in the window. The pointer is of the form `<distance,length>`. The `distance` value measures how far back into the window the sequence starts, while `length` counts the number of bytes for which the sequence is identical. Replacing a long sequence of bytes by a pointer effectively compresses the data sequence.

The DEFLATE specification states that the length value is drawn from [3, 258] while the value for distance is drawn from [1, 32768]. With these parameters, the window size is the maximum distance value, *i.e.*, 32,768 bytes.

Once, the data is compressed using LZ77, the output is coded using Huffman encoding [8]. Two different variants of Huffman coding are available in DEFLATE: static and dynamic (briefly discussed below).

Compressed data in DEFLATE are grouped into blocks. There are three types of blocks depending upon the kind of compression applied to the data included into it:

- **Uncompressed blocks** (`STORED`). Such a block is composed of a 2-byte field which contains the number of data bytes in the block. Another adjacent 2-byte field stores the one's complement of the previous field used to validate the length. This is followed by the uncompressed data. We note that no compression whatsoever (not even LZ77) is applied on the data. Uncompressed blocks are limited to 65,535 bytes.

- **Compressed blocks with a static Huffman code** (`STATIC`). The code is defined using several fixed tables. It is to note that the Huffman coded data is pre-compressed using LZ77.

- **Compressed blocks with a dynamic Huffman code** (`DYNAMIC`). The file is parsed during runtime to create a frequency table of symbols. The data is then encoded with this specific table. The block first contains this table that is needed for decompression, which

is followed by the compressed data and finally the literal 256 encoded using the table to mark the end of the block. Again, the data in these blocks are pre-compressed using LZ77.

Each block starts with a 3-bit header. The bits are read from right to left (Big endian encoding). When the first bit of the header is set to one, it indicates that this is the last block of the file. The last two bits of the header are used to specify the block type: uncompressed block (00), static Huffman compressed (01) or dynamic Huffman compressed (10). The compression algorithm terminates a block when it determines that starting a new block would be useful, or when the block size fills up the compression block buffer. The DEFLATE algorithm has been implemented in different compression routines such as `gzip` and `zip`. Each routine implements its own algorithm to determine which block type needs to be used.

3.2 Quines

A *quine* (native to programming languages) is defined as a self-reproducing program, *i.e.*, when executed, the program should generate an output that is identical to its source code. To some extent, a compressed file can also be seen as a program which outputs a result, *i.e.*, the corresponding decompressed file. Hence, a quine for a decompression algorithm is a compressed file which decompresses to itself.

In fact, there is a subtle difference between a regular program and a compressed file. It stems from the fact that the source code of a regular program can be a simple text file, while a compressed file most certainly has a well defined header and a footer (See Appendix A for different file format specifications). The header and footer are required for the file to be correctly interpreted during decompression. For a compressed file, let us use H to denote the header and F for the footer. Using these notations, the actual source code in the context of a decompression quine is the compressed data that lies between H and F.

We note that a quine must generate the same header and the same footer upon decompression. However, when decompression is applied on a quine, the header and the footer part are eventually removed and only the code part gets decompressed (or in other words gets executed). As a consequence, for a decompression quine to be valid, the code part must generate itself and in addition it should also generate the header and the footer.

3.2.1 The LZ77 Language

The reason behind the existence of a quine is that the underlying compression algorithm can be viewed as a pseudo language with a small set of instructions. In fact, the task of constructing a quine essentially reduces to the task of identifying the underlying language. In this section, we present the LZ77 language due to Cox [2]. The language forms the core of any known quines based on DEFLATE. The language has two instructions:

- `literal(n)` followed by n bytes: write these n bytes as the output. For instance, `literal(3)foo` → `foo`.
- `repeat(d,n)` which represents a `<distance,length>` pointer: copy the n bytes found d bytes backward from the current position of the output to the output. For instance, if the output at a given instant is: `incant abracad`, then `repeat(7,4)` → `incant abracadabra`.

In the rest of this section, we use the notation **Ln** for `literal(n)`, **Rd,n** for `repeat(d,n)` and **Rn** for `repeat(n,n)`. The operator **L0** is also used but gives no output. It should be apparent that LZ77 compression indeed builds upon the afore-described instructions. In fact, an **Ln** instruction represents a non-compressed data, while an **Rn**/**Rd,n** instruction represents a compressed data. The values d, n correspond to the `<distance,length>` pointers of the LZ77 compression.

The language can be used to write the code part of a DEFLATE generated compressed file and for that matter an LZ77 quine. Once, the desired code part is found, a suitable header and footer pair can be manually plugged into the file. We note that the language completely ignores the Huffman encoding that is applied atop LZ77 compression, hence it cannot exactly be considered as a well constructed DEFLATE compressed file. As a matter of fact, constructing a deflate quine essentially boils down to constructing a LZ77 quine. This is because, once an LZ77 quine is found, the data can then be easily grouped into blocks and appropriate Huffman coding can be applied. Hence, in the rest of the discussion we neither consider the block structure of DEFLATE nor the Huffman coding.

EXAMPLE 1. *We now present a simple example to illustrate the basic idea to generate a quine in this language. It is given in Table 1. The example presents a code that generates itself without the header and the footer. After the execution of the first instruction, the output lags behind the code by one instruction (since the execution of **L0** yields nothing). After executing the second instruction, this lag increases to two instructions. The lag remains unchanged after the third instruction. The fourth instruction reduces the lag back to one. The interesting core of this self-producing code is the instruction number five, **L4 R4,3 L4 R4,3 L4**, which is in fact a palindrome. The palindrome property of the instruction allows it to recurse back to a previously appeared instruction of the output. It is evident that any **Ln** instruction increases the lag while any **Rd,n** decreases it.*

In the following section, we present a complete construction for a known quine which includes the necessary header and footer.

3.2.2 Known Quines

Currently, we know six quines for compressed files which use DEFLATE:

1. `droste.zip`[2]: a `zip` quine,
2. `r.gz`: a `gzip` quine proposed by Russ Cox [2]. It uses the same general construction as `r.tar.gz` and `r.zip`,
3. `r.tar.gz`: a tarball quine proposed by Russ Cox. It has the same construction as `r.gz` and `r.zip`,
4. `r.zip`: a `zip` quine proposed by Russ Cox. It has the same construction as `r.gz` and `r.tar.gz`,
5. `rec_fix.gz`: a `gzip` quine proposed by Mahaly Barasz (as a part of comments given on [2]). It contains free bytes which can take any value and yet remain a quine,
6. `rec_tst.gz`: a `gzip` quine very similar to `r.gz`, with `rec_tst.gz` in the filename option to replace `recursive` as in `r.gz`. It also comes from the comments given on [2].

[2]https://alf.nu/ZipQuine

Table 1: A simple example of a quine code. Both the instructions **Ln** and **Rd,n** require 1 byte. In bold, the arguments of the **Ln** instruction. Note that the columns Code and Code Output have the same byte sequence.

Comment	Code	Code Output
1. `print nothing`	L0	
2. `print 4 bytes L0 L4 L0 L4`	L4 **L0 L4 L0 L4**	L0 L4 L0 L4
3. `print nothing`	L0	
4. `go 4 bytes back in the output and repeat the next 3 bytes`	R4,3	L0 L4 L0
5. `print 4 bytes R4,3 L4 R4,3 L4`	L4 **R4,3 L4 R4,3 L4**	R4,3 L4 R4,3 L4
6. `go 4 bytes back in the output and repeat the next 3 bytes`	R4,3	R4,3 L4 R4,3

Table 2: Quine construction for `rec_fix.gz`. H is a header of length 18 bytes, F a footer of length 8 bytes and **FP** are free bytes of length 4. **Ln** is encoded using 5 bytes while **Rd,n** using 3 bytes. In bold, the arguments of the **Ln** instruction.

Comment	Code	Code Output
1. `print 23 bytes H (18 bytes) L23 (5 bytes)`	L23 **H L23**	H L23
2. `go 23 bytes back in the output and repeat the next 15 bytes`	R23,15	H[1..15]
3. `print 16 bytes H[16..18] (3 bytes) L23 (5 bytes) R23,15 (3 bytes) L16 (5 bytes)`	L16 **H[16..18] L23 R23,15 L16**	H[16..18] L23 R23,15 L16
4. `repeat the 16 previous bytes of the output`	R16	H[16..18] L23 R23,15 L16
5. `print 16 bytes R16 (3 bytes) L16 (5 bytes) R16 (3 bytes) L16 (5 bytes)`	L16 **R16 L16 R16 L16**	R16 L16 R16 L16
6. `repeat the 16 previous bytes`	R16	R16 L16 R16 L16
7. `print 16 bytes FP (4 bytes) R12 (3 bytes) 00 (1 byte) F (8 bytes)`	L16 **FP R12 00 F**	FP R12 00 F
8. `repeat the 12 previous bytes`	R12	R12 00 F
9. `add extra padding`	00 (Padding)	

We note that there exist some quine variants such as the file `rec_dup.gz` (see the comment section of [2]). The file is an "*ever expanding quine*" in the sense that the file upon decompression generates a `.gz` file twice the size of the original file. Hence, if a recursive decompression is applied on the initial file, at each recursion step, one obtains a file twice as large as the file at the previous recursion step.

In this section, we focus on the construction used in the quine `rec_fix.gz`. The construction of other quines essentially follows the same pattern. It assumes that the **Ln** instruction is coded using 5 bytes and the **Rd,n** instruction using 3 bytes. It is also assumed that we have a header H of 18 bytes and a footer F of 8 bytes. The header and footer size respect the `.gz` file format (details on the header and footer specifications can be found in Appendix A).

In order to present the code in a readable form, we further use the shorthand **H[i..j]** to collectively represent all the $(j - i + 1)$ bytes between the i^{th} and the j^{th} byte of the header H. The notation **FP** is used to denote the four free bytes available in `rec_fix.gz`. We recall that free bytes can take any value while maintaining the quine property. The quine construction is shown in Table 2.

One can easily verify that the output sequence is exactly the same as that of the content in the code except that the output additionally has the header H in the beginning and the footer F at the end. To understand how the construction works, one may notice that at the first instruction, the output lags behind the code by **H L23**. In the second instruction, the output is behind by the sequence **H[16..18] L23 R23,15**. The next two instructions allow the output to catch up on this delay. This is followed by a succession of **R16 L16** which reverses the situation with the code behind the output. At the end, we have the code and the output at the same point with the footer in addition (in the output). The padding is used to have enough bytes for the seventh instruction. Even, in this complete construction, the palindrome **L16 R16 L16 R16 L16** in the fifth instruction plays a crucial role.

3.2.3 A Generalization

The problem with the previous quine construction is the constraint that the header must be 18 bytes long. We propose in this section a natural generalization to this construction for a header of any arbitrary length.

First, we define p the length of the header H and k an integer such that $k = p - 3$. In fact, the header must be of a length greater than 3 bytes (due to the file format specification, see Appendix A for further detail), hence we have $p \geq 3$. The generalized quine construction is given in Table 3. As in the previous construction, the **Ln** instruction must be coded using 5 bytes, while, the **Rd,n** instruction using 3 bytes.

4. QUINES VERSUS ANTI-VIRUSES

All the quines presented in the previous section were tested on three online anti-virus aggregators (Virus Total, Jotti and VirScan), on two incident response frameworks (Mastiff v0.7.1 and Viper v0.12.9) and on several anti-viruses (see Table 4 and Table 5). The tests were conducted in August 2016.

Online anti-virus aggregators allow a user to submit files to be checked by several anti-viruses. Our goal was to determine if quines were detected by anti-viruses and if yes, identify the method employed.

VirusTotal (www.virustotal.com) is operated by Google and it aggregates 54 anti-viruses. It is by far the most popular online anti-virus aggregator. Jotti (virusscan.jotti.org) and VirScan (www.virscan.org) aggregate 19 and 39 anti-viruses respectively. The information on the different anti-viruses including their version are available for VirScan. It

Table 3: Construction for the generalization of `rec_fix.gz`. H is a header of length p bytes, F a footer of length 8 bytes and **FP** are free bytes of length 4. **Ln** is encoded using 5 bytes while **Rd,n** using 3 bytes. In bold, the arguments of the **Ln** instruction.

Comment	Code	Code Output
1. `print p+5 bytes H (p bytes) L23 (5 bytes)`	Lp+5 **H Lp+5**	H Lp+5
2. `go p+5 bytes back in the output and repeat the next k bytes`	Rp+5,k	H[1..k]
3. `print 16 bytes H[k+1..p] (3 bytes) Lp+5 (p+5 bytes) Rp+5,k (3 bytes) L16 (5 bytes)`	L16 **H[k+1..p] Lp+5 Rp+5,k L16**	H[k+1..p] Lp+5 Rp+5,k L16
4. `repeat the 16 previous bytes of the output`	R16	H[k+1..p] Lp+5 Rp+5,k L16
5. `print 16 bytes R16 (3 bytes) L16 (5 bytes) R16 (3 bytes) L16 (5 bytes)`	L16 **R16 L16 R16 L16**	R16 L16 R16 L16
6. `repeat the 16 previous bytes`	R16	R16 L16 R16 L16
7. `print 16 bytes FP (4 bytes) R12 (3 bytes) 00 (1 byte) F (8 bytes)`	L16 **FP R12 00 F**	FP R12 00 F
8. `repeat the 12 previous bytes`	R12	R12 00 F
9. `add extra padding`	00 (Padding)	

is important to note that aggregators may use a common anti-virus but not necessarily the same version. This may eventually produce inconsistent behavior across the aggregators for the same anti-virus.

The result of the submission of the quine to the aggregators and incident response frameworks is given in Table 4. We first observe that quines are detected only by a few anti-viruses. Let us first start with the `zip` quine `droste.zip`. This quine is detected as a decompression bomb by `Sophos`, `ESET` in VirusTotal and Jotti. `Qihoo 360` also detects it in Jotti and VirScan. The second quine tested was `r.zip`. It is only detected by `Sophos` in Jotti only and not by `ESET` and `Qihoo 360`. `Sophos` being used by all the aggregators, we observe an inconsistent behavior which might be caused by the different setups or versions of the anti-virus.

At this step of our analysis, we may conclude that `Qihoo 360` and `ESET` employ a signature-based detection. These anti-viruses maintain a list of signatures of known bombs and check whether a given file has a signature that belongs to the list. `Zoner` (for VirusTotal) and `F-Secure` (for Jotti) are not able to provide an analysis for both files (53 answers out of 54 for VirusTotal and 18 out of 19 for Jotti). It means that they can be a potential victim of a decompression quine.

Table 4: Quine detection by anti-virus aggregators and incident response frameworks. x/y means that x out of y anti-viruses detected the file as a quine. ✓ for detecting the file as a quine, while ✗ for not doing so. ∞ implies that the tool ran an infinite recursion loop.

Quine	Agregators and Frameworks				
	VirusTotal	Jotti	VirScan	Mastiff	Viper
`droste.zip`	3/53	2/18	1/39	∞	✗
`r.zip`	0/53	1/18	0/39	∞	✗
`rec_tst.gz`	0/54	1/19	0/39	✗	✗
`rec_fix.gz`	0/54	1/19	0/39	✗	✗
`r.tar.gz`	0/54	0/19	0/39	✗	✗
`r.gz`	0/54	1/19	0/39	✗	✓

In the second step of our analysis, we look at `gzip` quines. Most of the time, all the gzip quines are declared as safe. `Sophos` in Jotti is the only anti-virus that identifies all `gzip` quines except `r.tar.gz`. A possible explanation behind why `Sophos` misses `r.tar.gz` could be that the `tar` archive format is not well supported by most anti-viruses. Otherwise, `Sophos` detects decompression quines in a consistent way. To understand how it works, we took a safe file and compressed it several times. We submitted the resulting compressed file to Jotti. It appears that after compressing the file 6 times, `Sophos` declared it to be a decompression bomb. We can safely assume that the decompression bomb detector in `Sophos` is similar to the Python code provided in Code 1.

Code 1: Quine detector with a recursion bound.

```
#!/usr/bin/python

import sys
import zlib

magic_nb="\x1f\x8b\x08"
MAX=6

def uncompress(data,depth):
  if data.startswith(magic_nb) and depth<MAX:
    original=zlib.decompress(data,zlib.
        MAX_WBITS|16)
    uncompress(original,depth+1)
  else:
    if depth>=MAX:
      print "Quine"
      sys.exit()
    else:
      return data

with open(sys.argv[1]) as f:
  data=f.read()
uncompress(data,0)
f.close()
```

We also analyzed a malware analysis tool named `yextend`[3] that is based on YARA[4] — a pattern matching tool to analyze malware files. `yextend` employs a detector identical to Code 1 to identify decompression bombs. The default recursion bound is set to 42 but can be tuned by the end user. It is pertinent to note that YARA is also used by VirusTotal.

Table 4 also provides the result on incident response frameworks. It appears that `Mastiff` recursively decompresses `zip` files (but not `gzip` files). `droste.zip` and `r.zip` are

[3]https://github.com/BayshoreNetworks/yextend
[4]http://virustotal.github.io/yara/

both decompressed infinitely many times by `Mastiff`. The only positive detection obtained was by `Viper` for `r.gz`.

We have also directly studied the behavior of 12 anti-viruses on a computer running Microsoft Windows 10 (see Table 5 for the list).

Table 5: Impact of quines on several anti-viruses running on Microsoft Windows 10. ✗ denotes that the tool could not detect the quine, while ∞ denotes that the tool most probably ran into an infinite recursion.

Anti-viruses	Version	Six quines tested
Ad-Aware	11.12.945.9202	✗
Avast	12.2.2276	✗
Avira	15.0.18.354	✗
AVG	16.101.7752	✗
Baidu	5.4.3.147185	✗
ClamWin	0.99.1	✗
Comodo	8.4.0.5068	✗
FortiClient	5.00233	✗
Panda	16.1.3	∞
Smadav	10.8	✗
Trend Micro	10.0.1150	✗
Windows Defender	1.225.361.0	✗

The results remain very similar to those observed on anti-virus aggregators: quines are largely undetected. The only notable result is that during the analysis of each quine on `Panda`, the anti-virus always indicated a scanned file of around 2000 bytes (but often different at each run) before a complete crash of the software.

In conclusion, quines are largely undetected by most anti-viruses. Several of them including the framework `Mastiff`, anti-viruses `Panda`, `Zoner` (from VirusTotal) and `F-Secure` are still vulnerable to quines. `Sophos` has the most rational method to detect quines while `Qihoo 360` and `ESET` only employ signatures of known quines as a detection tool.

5. QUINE DETECTORS

As seen in the previous section, anti-viruses in general do not have a suitable architecture in place to detect quines. The limited few which do detect quines either employ a signature based technique or recursively decompress with a pre-defined or configurable bound on the depth. Our quine generalization of Table 3 clearly defeats any signature based detector. A naive recursive decompression with bounded depth is however inefficient as it requires several calls to the decompression routine. Moreover, it also entails false positives. In this section, we study several ad-hoc detectors and propose a new one.

5.1 Employing Memoization

Dynamic programming techniques and particularly memoization can improve the efficiency of a recursive decompression detector. In fact, memoization can avoid requiring to decompress multiple times the same file. The idea is to memoize the hash of a compressed file and whenever a new file needs to be decompressed, its hash is first checked in the memo table (which also stores a pointer to the decompressed file). If the hash has been previously seen, the extra decompression can be avoided by replicating the previously decompressed data.

Memoization is ideal for nested bombs such as `42.zip`. This technique has indeed been employed in practice (see [7] for a use case). It can also reduce the number of decompressions to detect a quine in Code 1. However, it increases (depending on the implementation) the memory cost or the CPU cost for non-quine files without any additional benefit (the extra computation are unlikely to be reused). Furthermore, since memoization is based on identifying identical piece of compressed data, it can easily be made useless by creating bombs where internal files differ by a byte.

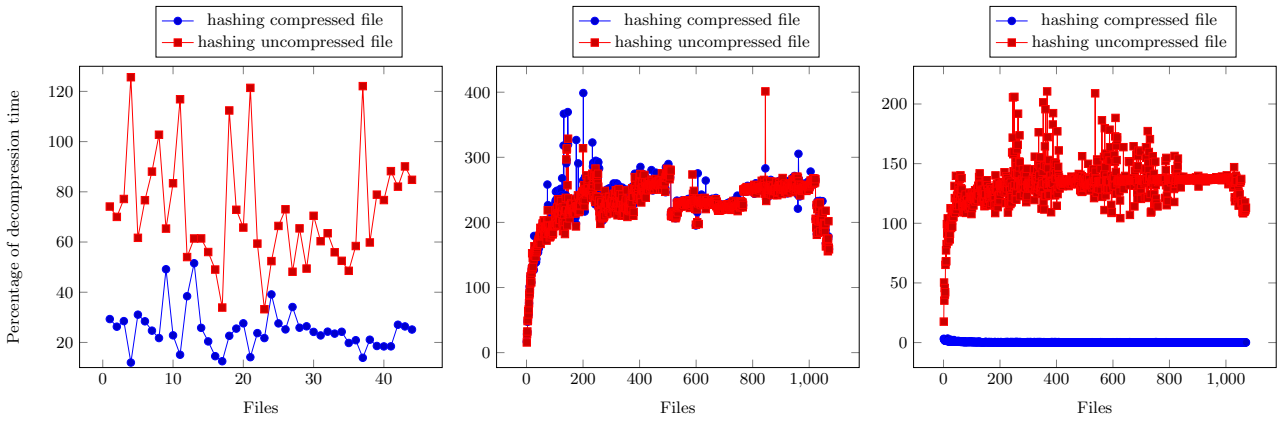
Since hashing is the core primitive of a memoization based detector, we compare the efficiency of decompression and hashing. Our implementation is in Python and it uses `gzip` as the compression routine and SHA-256 as the hash function. The tests were performed on a 64-bit processor laptop computer powered by an `Intel Core i7-4600U CPU` at `2.10GHz` with `4MB cache`, `8GB RAM` and running `Linux 3.13.0-36-generic`. We employ the same machine throughout this paper.

We conduct tests on three datasets. Our first dataset includes two standard lossless data compression corpora namely, *Silesia* [4] and *Canterbury* [1]. These corpora provide datasets of files that cover the typical data types used in practice and hence are useful to test compression algorithms. The Canterbury corpus is an old corpus dating back to 1997. The corpus contains 11 files of sizes less than 1 MB. The more recent Silesia corpus on the other hand contains larger files, between 5 MB to 49 MB. A summary of the contents of these corpora is given in Appendix C. The dataset also includes some other files available on the Canterbury corpus webpage[5]. In total, the first dataset contains 44 files. The second dataset is a set of 1071 files filled with random bytes. Files in the dataset have increasing sizes starting from 1KB to 50MB. The third dataset contains the same number of files having the same sizes, but are filled with `0x00`. In order to compare the efficiency of hashing and decompression, we first compress the files in each dataset using `gzip`, and measure the time required to decompress the output; the time to hash the compressed file and the time required to hash the decompressed file.

In Fig. 1, we plot the time to hash the compressed file and the time to hash the decompressed file as a percentage of the time required for decompression. We observe that 13% of the files from the first dataset require more time to hash the uncompressed file than decompression. Hashing a compressed file is relatively cheaper yet entails a considerable percentage of the decompression time. In fact, around 34% of the uncompressed files require a hashing time which is over 25% of the decompression time. As for the second dataset, all uncompressed files starting from 13 KB require more time to hash than the time required to decompress the file. Similarly, all compressed files starting from 11 KB require more time to hash than the time required to decompress the file. As for the third dataset, we observe that all uncompressed files starting from 15 KB require more time to hash than the time required to decompress the file. Hashing compressed files however requires a negligible percentage of the decompression time.

These experiments clearly demonstrate that hashing often incurs a considerable cost compared to decompression. Hence, any detector should carefully use hashing to avoid introducing any overhead. In the next sections, we study

[5]http://corpus.canterbury.ac.nz



(a) Files from compression corpora. (b) Files filled with random bytes. (c) Files filled with zero bytes.

Figure 1: Comparison of time required to decompress a file, time to compute the hash of a compressed file and the time to compute the hash of a decompressed file. The files in all the three figures are sorted in the increasing order of the size. The reported data is the average over 1000 runs.

two detectors specific to quine detection: *fixed-point detector* and *statistical detector*.

5.2 Fixed-point Detector

The first obvious decompression quine detector consists in checking whether a given compressed file is a fixed point of the decompression function. This requires decompressing the file and checking for equality between the compressed file and the decompressed file. Clearly, a fixed-point detector entails no false positives. A `Python` code for this detector is given in Code 2. It basically decompresses the file and then checks if the compressed and the decompressed files have the same size. If the files have the same size, it computes their SHA-256 digests to check the equality of their contents.

Code 2: The fixed-point detector

```
#!/usr/bin/python

import sys
import gzip
import hashlib

with open(sys.argv[1],'rb') as f:
  compress=f.read()
with gzip.open(sys.argv[1],'rb') as g:
  uncompress=g.read()
if len(compress)==len(uncompress):
  dc=hashlib.sha256(compress).digest()
  du=hashlib.sha256(uncompress).digest()
  if dc==du:
    print "Quine"
  else:
    print "Safe"
else:
  print "Safe"
```

We note that the step in Code 2 that consists in comparing the file sizes is critical for the performance of the detector. In fact, due to the efficiency of the compression algorithm, the decompressed file can be several order larger than the compressed one. For instance, in case of `gzip`, the compression ratio can be as high as 1032:1. Therefore, the computation of the SHA-256 digest of the decompressed file can be very costly. The a priori comparison of the file sizes eliminates costly digest computations for compressed files which are not quines.

The detector is very simple and performs well on most of the files it may encounter. However, there exists a class of compressed files for which it is rather inefficient. It is a class of large files for which the compressed and decompressed files are of the same size. In this case, the computation of the SHA-256 digests is unfortunately costly. Such files can be easily generated in practice by taking an arbitrarily large non-compressed file say `example.txt`. The file is then compressed to obtain `example.gz`. Let us assume that the size of `example.gz` is smaller than that of `example.txt` by k bytes. One can then change the field `FNAME` that stores the uncompressed filename in `example.gz` to add the remaining k bytes (see Appendix A for further details on the `FNAME` field). It is to note that such a file remains a valid `.gz` file. We followed this strategy to generate a 10 GB file. Hashing the compressed and the uncompressed files took around 1m40s using the bash `sha256sum` utility. Clearly, the time spent by the detector can be made arbitrarily large by choosing the initial file of suitable size.

We also note that the fixed-point detector is not capable of detecting quine variants such as `rec_dup.gz` — a file that recursively doubles its size upon decompression.

In the next section, we study another detector that does not require a hash function and takes a decision based on the ratio of the number of different block types (`stored`, `static` and `dynamic`). It is also capable of detecting quine variants such as `rec_dup.gz`.

5.3 Statistical Detector

We propose a new detector which assumes that a legitimate compression software cannot produce a quine. In case of a quine the block structure (the type of the block — stored, static or dynamic, and their order) is manually chosen by a human, while, compression algorithms apply their own heuristics to decide on the blocks.

In order to illustrate the idea, let us consider the case of the quine `rec_fix.gz`. The file has a size of 130 bytes. If we compress it using either `gzip 1.2.4` or `zlib 1.2.8`, we obtain a compressed file with a single `static` block. However, `rec_fix.gz` itself is composed of 8 blocks as shown in Fig. 2, where, `stored` (uncompressed) and `static` blocks alternate. Moreover, there are no dynamic blocks. The block structure of other quines is left to Appendix B.

Figure 2: Structure of `rec_fix.gz`. The black blocks represent the header and the footer of the `.gz` file. The gray blocks correspond to `stored` (uncompressed) blocks and the light gray are compressed with the static Huffman table (`static` blocks).

To exploit the difference between the output of a legitimate compression algorithm and the manual design of quines, we directly analyze their block structure and study the statistics of block types. We use the following notation to denote the number of different block types:

- U : number of uncompressed blocks;
- S : number of static Huffman blocks;
- D : number of dynamic Huffman blocks;
- T : the total number of blocks (*i.e.*, $T = U + S + D$).

In order to study the block structure of compressed files, we use an instrumented version of the `gzip` compression routine. The instrumented version is obtained by modifying the source code such that the decompression routine jumps to the start of each block, and decodes the block type from the block header. It is to note that the instrumented code does not need to write the decompressed data to the disk. All modifications are limited to the `inflate.c` file available in the `gzip` source code package (version 1.2.4) for Linux. A similar instrumented library was obtained from the `zlib` library for Linux (version 1.2.8).

Using our instrumented `gzip` routine, we obtain the number of each block type in all the existing quines. Table 6 summarizes our findings. It is worth noticing that dynamic blocks are quite rare.

Table 6: Block structures of quines.

Quine	U	S	D	T	U/T	S/T	D/T
`droste.zip`	27	75	1	102	0.26	0.73	0.01
`r.zip`	12	6	0	18	0.67	0.33	0
`rec_fix.gz`	4	4	0	8	0.5	0.5	0
`rec_tst.gz`	11	6	0	17	0.64	0.36	0
`r.gz`	11	6	0	17	0.64	0.36	0
`r.tar.gz`	13	7	0	20	0.65	0.35	0
`rec_dup.gz`	6	5	0	20	0.55	0.45	0

Table 6, also shows the different ratios for quines, namely U/T, S/T and D/T. As for U/T, results vary from 0.5 to 0.66 for all files except one, `droste.zip`. In a similar vein, all files except `r.zip` have a S/T ratio larger than 0.35, while all files except `droste.zip` have a D/T ratio of 0. Clearly, these ratios are less homogeneous for `.zip` files, since they also perform archiving and hence the headers are much more complex than `.gz` files.

We hence propose a statistical detector that consists in using the instrumented code to compute the ratios and reporting a compressed file as potentially dangerous if it yields a U/T ratio larger than or equal to 0.5, or an S/T ratio larger than or equal to 0.35. The detector comes with the eventual possibility of having false positives.

In the rest of this section, we evaluate our statistical detector for false positives. To this end, we conduct experiments to study the block structure of safe files. In order to have a set of safe files, we have built a dataset of 3655 files from gnu.org. It includes the source code of the different versions of 338 software packages. We also include the data compression corpora Silesia and Canterbury.

All the safe files and the files in the corpora were first compressed using `gzip v1.2.4` with default parameters and then their block structure was determined using the instrumented version of the `gzip` source code. Fig. 3 provides the statistics on the block types for these files. The first three plots Fig. 3a, Fig. 3b and Fig. 3c present results on the 3655 files harvested from gnu.org, while the last three plots Fig. 3d, Fig. 3e and Fig. 3f present combined results for the Canterbury and Silesia corpora.

We observe that only three files from the GNU dataset (Fig. 3a, Fig. 3b and Fig. 3c) have a U/T ratio larger than 0.5. One of these files has a U/T ratio as high as 0.64, the same as that of `rec_tst.gz` and `r.gz`. Moreover, 89% of the files from the GNU dataset have a U/T ratio of 0. This shows that the `gzip` compression routine rarely includes an uncompressed block. As for the S/T ratio, only one file surpasses the quine bound of 0.35. This file yields an S/T ratio of 0.5. Again, 99.7% of the files do not contain any `static` block. As for D/T, all files have a ratio larger than 0.37. Hence, the overall false positive rate for this dataset is $4/3655 = 0.001$.

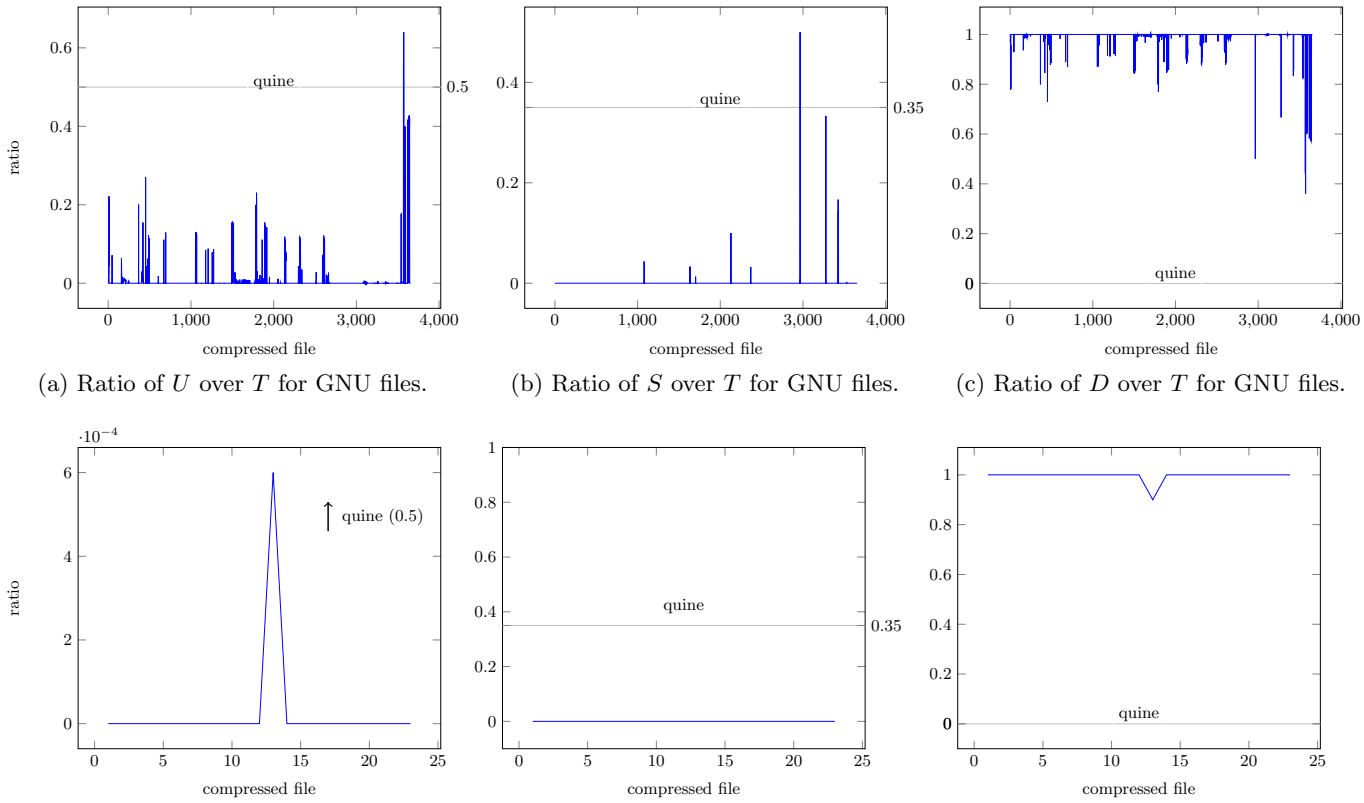
We observe even better results with Canterbury and Silesia corpora (Fig. 3d, Fig. 3e and Fig. 3f). Only one file out of a total of 23 has a non-zero U/T ratio of 0.0006 (yet much smaller than the quine bound of 0.5). As for S/T, all files yield a ratio of 0. And all files except one have a D/T ratio of 1; the file has a D/T ratio of 0.999. These results show that all the files from the corpora except one generate a single dynamic block that contains all the compressed data. Hence, the overall false positive rate is 0.

The experiments with the datasets show that the detector is quite efficient in terms of false positives. In the following section, we present a limitation that creates a potential to influence the false positive rate of the detector.

5.3.1 Limitation

The DEFLATE specification [5] allows a compression routine implementing the algorithm to choose a heuristic to decide on the block structure. That is, each compression routine may decide on 1) when to create a new block 2) the type of block to create. Clearly, compressing using different heuristics may produce different statistics on the blocks. Hence, it is possible to come up with a heuristic that may influence the false positive rate of our statistical detector. For instance, one may apply a heuristic to increase the number of uncompressed blocks such that the ratio U/T surpasses the quine bound of 0.5. However, such a heuristic is clearly irrational.

In order to study this limitation, we compare the impact of the heuristics used in two different compression routines



(a) Ratio of U over T for GNU files. (b) Ratio of S over T for GNU files. (c) Ratio of D over T for GNU files.

(d) Ratio of U over T for corpora. (e) Ratio of S over T for corpora. (f) Ratio of D over T for corpora.

Figure 3: Distribution of blocks for safe files. The first three plots present results on the 3655 files harvested from gnu.org, while the last three plots present combined results for the Canterbury and Silesia corpora.

namely, `zlib` and `gzip`. We also study the impact of modifying certain compression parameters in `gzip` and `zlib` on the statistics of blocks. To this end, we note that `zlib` provides a parameter `memLevel` that specifies how much memory should be allocated for the internal compression state. The parameter partly determines when a new block is created and which type of block is created. The parameter takes values between 1 and 9. With `memLevel=1`, the algorithm uses minimum memory but is slow and yields the smallest compression ratio; `memLevel=9` uses maximum memory for optimal speed. The default value is 8. The value of `memLevel` can be set to a desired value using the `MAX_MEM_LEVEL` macro defined in the source code. `gzip` also provides a similar parameter as a command line option.

Fig. 4 and Fig. 5 present the result of modifying the `memLevel` parameter in `zlib` and `gzip` respectively. The first observation being that the two compression routines do not produce the same distribution of block types. We also observe that in both the cases, the increase in the value of `memLevel`, systematically decreases the ratios U/T and S/T, while increasing the ratio D/T. Since the increase in the value of `memLevel` leads to higher compression ratios, more and more blocks with dynamic Huffman coding are created. A comparison of the two figures shows that for `memLevel=1` and `memLevel=2`, a `zlib` compressed file yields an S/T ratio which is higher than the quine bound of 0.35. However, for these values of `memLevel`, the `gzip` compressed file has ratios much within the quine bound.

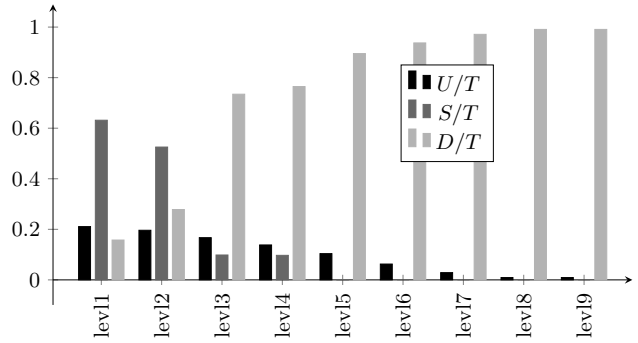


Figure 4: Results on the `mozilla` file from the Silesia corpus. The file was compressed using `zlib` with different values of `memLevel`.

These results show that it is possible to choose a specific parameter setting to skew the distribution of blocks. However, it would generally yield bad compression ratios.

5.3.2 *Performance Comparison*

In this section, we study the performance of three detectors: recursive decompression with bounded depth, fixed-point detector and statistical detector. The detectors use the `gunzip` decompression routine. The hash function implementation in the fixed-point detector comes from the bash `sha256sum` utility. In the rest of this section, we report the

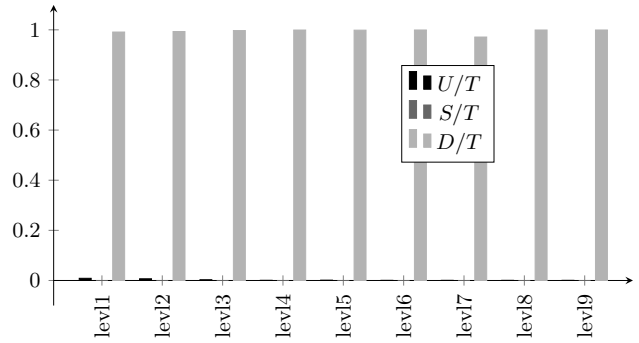


Figure 5: Results on the `mozilla` file from the Silesia corpus. The file was compressed using `gzip` with different values of `memLevel`. All U/T and S/T ratios are very close to 0.

time required by the detectors on quines and safe files from the Silesia corpus (since it is the most recent one). The reported time is the average over 1000 runs.

In Table 7, we present the result on two quines. The detector based on recursive decompression with bounded depth is tested against three values for the bound: 2, 6 and 42. The first value corresponds to the base test, while the other two correspond to values currently used in `Sophos` and `yextend` respectively. Clearly, the detection time increases with the bound and the size of the quine. This is essentially due to the cost of the underlying decompression algorithm. We also observe that the statistical detector performs better than the fixed-point detector for both the quines. There are two reasons to explain this phenomenon: 1) Since the files under scrutiny are quines, the fixed-point detector needs to compute the SHA-256 digest of the compressed and uncompressed files. The statistical detector on the other hand does not require any hashing. 2) Since the fixed-point detector needs to compute the hash, it has to write the decompressed data. Writing decompressed files to the disk is an overhead that is clearly absent in our statistical detector. Comparing the results on the recursive detector with bounded depth and the statistical detector, we observe that even for the bound 2, the statistical detector performs better than the recursive decompression. This is again due to the fact that the recursive decompression writes the decompressed data while the statistical detector does not.

Table 7: Results on two quines. Recursive decompression is tested for three bounds: 2, 6 and 42.

	Time (ms)				
		recursive			
File (size)	**fixed-pt.**	2	6	42	**stat.**
`rec_fix.gz` (130B)	5.2	5.3	11.6	65.6	4.7
`r.gz` (250B)	5.3	6.3	13.0	72.3	3.8

Table 8 presents the results on the detectors for safe files from the Silesia corpus. For safe files, the recursive decompression is able to detect the file as safe after the first decompression. The fixed-point detector needs to write the decompressed file and compute the size of the compressed and decompressed files. Again, since the statistical detector does not require writing the decompressed file, it performs the best among the three detectors. We observe that the statistical detector on an average required only 55% of the time required by the fixed-point detector and only 57% of that required by the detector based on bounded recursion.

Table 8: Results on files from the Silesia corpus.

	Time (ms)		
File (size)	**fixed-point**	**recursive**	**statistical**
`xml` (676K)	52.8	38.0	30.8
`reymont` (1.8M)	134.1	58.2	45.1
`ooffice` (3.0M)	108.2	70.1	59.7
`nci` (3.1M)	367.7	221.1	131.3
`mr` (3.6M)	158.7	121.1	76.0
`osdb` (3.6M)	100.8	124.8	74.6
`dickens` (3.7M)	106.8	151.8	82.1
`sao` (5.1M)	111.7	126.8	66.8
`samba` (5.3M)	309.2	407.3	117.6
`x-ray` (5.8M)	109.4	202.4	94.2
`webster` (12M)	527.6	606.7	257.8
`mozilla` (19M)	800.0	825.5	354.4

In conclusion, our statistical detector performs the best among all the detectors on both the quines and safe files.

6. CONCLUSION

In this paper, we showed that DEFLATE has an underlying pseudo-language that is sufficiently rich to produce quines. Moreover, the DELFATE specification gives several degrees of freedom such as the possibility to insert any arbitrary filename and create any block type of choice at any stage of compression among many others. Quines clearly exploit these liberties. Hence, it is much needed to redesign a new unified specification that reduces the attack surface available in DEFLATE.

Our tests with several anti-viruses and incident response frameworks show that most of them lack adequate architecture to efficiently detect quines. In face with this, we studied two ad-hoc detectors: the recursive decompression with bounded depth and the fixed-point detector. We also proposed a new statistical detector. Recursive detector is inefficient since it requires several calls to the decompression routine. Moreover, it has false positives as any archive file which has a larger number of compressed files that the hard-coded bound will be declared as unsafe. For some anti-viruses the bound is as low as 6. The fixed-point detector has no false positives and is more efficient on quines than the recursion based detector. However it performs poorly on a certain class of files. We have shown that such files are very easy to construct due to the flexible nature of the DELFATE specification. The proposed statistical detector is the most efficient of all but entails false positives. Again to the DEFLATE specification, it is possible to somewhat influence the false positive rate. However, tests with several datasets show that the false positive rate is very low.

7. ACKNOWLEDGMENTS

This research was partially supported by the Labex PERSYVAL-LAB (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

8. REFERENCES

[1] R. Arnold and T. C. Bell. A Corpus for the Evaluation of Lossless Compression Algorithms. In

Proceedings of the 7th Data Compression Conference (DCC '97), pages 201–210, 1997.
[2] R. Cox. Zip Files All The Way Down , March 2010. http://research.swtch.com/zip.
[3] P. L. Daniels. Arbomb, 2002. http://www.pldaniels.com/arbomb/.
[4] S. Deorowicz. Silesia Compression Corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.
[5] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.
[6] P. Deutsch. GZIP File Format Specification Version 4.3. RFC 1952 (Informational), May 1996.
[7] S. L. Garfinkel. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56–72, 2013.
[8] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[9] Joxlean Koret. Breaking Anti-Virus Software. In *Symposium on Security for Asia Network - Syscan 2014*, Singapore, Singapore, April 2014.
[10] K. Kent, S. Chevalier, T. Grance, and H. Dang. Guide to Integrating Forensic Techniques into Incident Response. Technical Report 800-68, NIST, 2006.
[11] J. Koret and E. Bachaalany. *The Antivirus Hacker's Handbook*. Wiley, 2015.
[12] K. Lansing. *Fidonet: A Study of Computer Networking*. Texas Tech University, 1991.
[13] Marie Cara. I Came to Drop Bombs: Auditing the Compression Algorithm Weapons Cache. In *Blackhat USA 2016*, Las Vegas, NV, USA, July–August 2016.
[14] G. Pellegrino, D. Balzarotti, S. Winter, and N. Suri. In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services. In *24th USENIX Security Symposium, USENIX Security 15*, pages 801–816, Washington, D.C., USA, August 2015. USENIX Association.
[15] Peter Bieringer. Decompression bomb vulnerabilities. Technical report, AERAsec Network Services and Security GmbH, 2004. http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html.
[16] PKWARE Inc. APPNOTE.TXT - .ZIP File Format Specification, 2014. version 6.3.4.
[17] K. Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, 1984.
[18] Usage of Compression for websites, July 2016. https://w3techs.com/technologies/details/ce-compression/all/all.
[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

APPENDIX

A. FILE FORMAT SPECIFICATIONS

The DEFLATE algorithm is implemented in three popular compression routines, namely, `gzip`, `zlib` and `zip`. The first two compression routines are very similar, while, `zip`, apart from being a compression routine is also an archiver, *i.e.*, it aggregates a collection of files and directories by storing them into a single container. In this appendix, we present the file format specification for each of these routines.

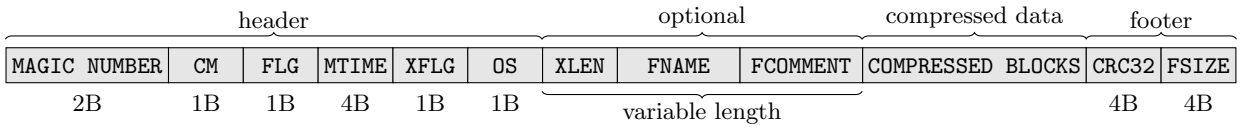
GZIP: A `gzip` compressed file as specified in RFC 1952 [6] is composed of three mandatory parts and an optional part (see Fig. 6a). The mandatory parts include a header followed by the compressed data and a footer. The header is collectively 10 bytes long and stores the `gzip` magic number (two bytes for `MAGIC NUMBER` `0x1F8B`), a 1-byte field to identify the compression method used (`CM`)[6], another 1-byte field for a flag meant to activate options (`FLG`), a 4-byte field for the unix time at which the file was last modified (`MTIME`), a one byte field for extra flags (`XFLG`) that is set depending on the compression method and finally the last byte reserved to indicate the operating system (`OS`). Depending on whether some of the bits of `FLG` are set or not, there can be an optional part which consists of three additional fields: `XLEN` gives the length of the optional field, `FNAME` is the filename and `FCOMMENT` is the file comment. `FNAME` and `FCOMMENT` terminate with a `0x00` byte. The optional part is followed by the actual compressed data blocks (`COMPRESSED BLOCKS`). A `gzip` file ends with a mandatory footer composed of a 4-byte cyclic redundancy code (`CRC32`) and the size of the uncompressed file modulo 2^{32} (`FSIZE`).

ZLIB: A `zlib` compressed file [5] has the same format as that of a `gzip` compressed file, except that the magic number is different (`0x789C`) and that the `CRC32` checksum is replaced by `ADLER32` which trades reliability for speed.

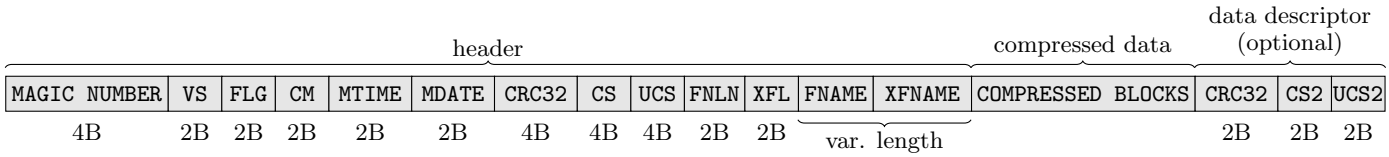
ZIP: In the description provided here, we do not take into account the archiving ability of `zip`. In other words, we assume that only a single file is zipped. The `zip` file format as specified in [16] is shown in Fig. 6b. Under the stated restriction, a `zip` compressed file is identified by a `MAGIC NUMBER`, *aka* a *file header signature* which is a 4-byte field (`0x04034b50`). It is followed by five 2-byte fields, namely, a field that identifies the version needed to extract the data (`VS`), a flag to store options (`FLG`), a field to identify the compression method used (`CM`)[7], a field to indicate the last modification time of the file (`MTIME`) and another to indicate the last modification date of the file (`MDATE`). This is followed by three 4-byte fields, namely, a CRC32 checksum to check file integrity (`CRC32`), a field to store the compressed file size (`CS`) and another to store the uncompressed file size (`UCS`). These size-related fields are followed by two 2-byte fields. The first stores the filename length (`FNLN`), while the other stores the length of the extra field (such as a file comment) (`XFL`). Then, there are two fields of variable length to store the filename and the extra field name (`XFLNAME`). This is followed by the actual compressed data (`COMPRESSED BLOCKS`). Finally, depending upon the bits of `FLG`, the file may have three 4-byte fields, namely, `CRC32`, `CS2` — a repetition of the compressed file size `CS` and `UCS2` — a repetition of the uncompressed file size `UCS`.

B. BLOCK STRUCTURE OF KNOWN QUINES

[6]Apart from the DEFLATE compression, other methods such as LZH and `pack` (a deprecated compression algorithm based on Huffman coding) are also supported.
[7]`zip` also implements LZMA, IBM LZ77, BZIP2.



(a) A `gzip` compressed file.



(b) A `zip` compressed file.

Figure 6: File format of a `gzip` and `zip` compressed file.

(a) `rec_tst.gz`.

(b) `r.gzip`.

(c) `r.gz`.

(d) `r.tar.gz`.

Figure 7: Block structure for different quines. The gray blocks correspond to uncompressed (`stored`) blocks and the light gray are compressed with the static Huffman table (`static` blocks).

Table 9: Canterbury corpus.

Filename	Description	Type	Filesize (KB)
`alice29.txt`	Alice in Wonderland	English text	148
`asyoulik.txt`	Shakespeare's play	English text	122
`cp.html`	sample HTML	HTML	24
`fields.c`	sample C source	C	10
`grammar.lsp`	list	LISP source	3
`kennedy.xls`	sample Excel	Excel spreadsheet	1005
`lcet10.txt`	technical writing	text	416
`plrabn12.txt`	poem	text	470
`pt5`	fax	CCITT	501
`sum`	SPARC	SPARC executable	37
`xargs.1`	GNU man page	man	4

Table 10: Silesia corpus.

Filename	Description	Type	Filesize (MB)
`dickens`	collected works of Charles Dickens	English text	9.7
`mozilla`	tarred executable of Mozilla 1.0	exe	48.8
`mr`	medical magnetic resonance image	picture	9.5
`nci`	chemical database	database	32
`ooffice`	dll from Open Office.org 1.01	exe	5.9
`osdb`	database in MySQL format from Open Source Database Benchmark	database	9.6
`reymont`	text of the book Chłopi by Władysław Reymont	Polish pdf	6.32
`samba`	tarred source code of Samba 2-2.3	src	20.6
`sao`	SAO star catalog	binary	6.9
`webster`	Webster Unabridged Dictionary (1913)	html	39.5
`xml`	XML files	html	5
`x-ray`	X-ray image	image	8

Fig. 7a, Fig. 7b, Fig. 7c, Fig. 7d, present the block structure for `rec_tst.gz`, `r.gzip`,`r.gz` and `r.tar.gz` respectively.

We present details on the Canterbury and Silesia compression corpora in Table 9 and Table 10 respectively.

C. COMPRESSION CORPORA