A better zip bomb

Abstract

We show how to construct a *non-recursive* zip bomb that achieves a high compression ratio by overlapping files inside the zip container. "Non-recursive" means that it does not rely on a decompressor's recursively unpacking zip files nested within zip files: it expands fully after a single round of decompression. The output size increases quadratically in the input size, reaching a compression ratio of over 28 million (10 MB → 281 TB) at the limits of the zip format. Even greater expansion is possible using 64-bit extensions. The construction uses only the most common compression algorithm, DEFLATE, and is compatible with most zip parsers.

1 Introduction

Compression bombs that use the zip format must cope with the fact that DEFLATE, the compression algorithm most commonly supported by zip parsers, cannot achieve a compression ratio greater than 1032 [21]. For this reason, zip bombs typically rely on recursive decompression, nesting zip files within zip files to get an extra factor of 1032 with each layer. But the trick only works on implementations that unzip recursively, and most do not. The best-known zip bomb, 42.zip [1], expands to a formidable 4.5 PB if all six of its layers are recursively unzipped, but a trifling 0.6 MB at the top layer. Zip quines, like those of Ellingsen [11] and Cox [9], which contain a copy of themselves and thus expand infinitely if recursively unzipped, are likewise perfectly safe to unzip once.

This article shows how to construct a non-recursive zip bomb whose compression ratio surpasses the DEFLATE limit of 1032. It works by overlapping files inside the zip container, in order to reference a "kernel" of highly compressed data in multiple files, without making multiple copies of it. The zip bomb's output size grows quadratically in the input size; i.e., the compression ratio gets better as the bomb gets bigger. The construction depends on features of both zip and DEFLATE—it is not directly portable to other file formats or compression algorithms. It is compatible with most zip parsers, the exceptions being "streaming" parsers that parse in one pass without first consulting the zip file's central directory. We try to balance two conflicting goals:

- Maximize the compression ratio. We define the compression ratio as the the sum of the sizes of all the files contained the in the zip file, divided by the size of the zip file itself. It does not count filenames or other filesystem metadata, only contents.
- Be compatible. Zip is a tricky format and parsers differ, especially around edge cases and optional features. Avoid taking advantage of tricks that only work with certain parsers. We will remark on certain ways to increase the efficiency of the zip bomb that come with some loss of compatibility.

The construction we will develop is tunable for different sizes. For the sake of comparison and discussion, we will produce three concrete examples. These examples and others are compared in Table 1 and Figure 6.

zbsm.zip	42 kB →	5.5 GB	Section 5.3
zblg.zip	10 MB →	281.4 TB	Section 5.3
zbxl.zip	46 MB →	4.5 PB	Section 7

2 Structure of a zip file

A zip file consists of a *central directory* which references *files*. Refer to Figure 1.

The central directory is at the end of the zip file. It is a list of *central directory headers*. Each central directory header contains metadata for a single file, like its filename and CRC-32 checksum, and a backwards pointer to a local file header. A central directory header is 46 bytes long, plus the length of the filename.

A file consists of a *local file header* followed by compressed *file data*. The local file header is 30 bytes long, plus the length of the filename. It contains a redundant copy of the metadata from the central directory header, and the compressed and uncompressed sizes of the file data that follows.

Zip is a container format, not a compression algorithm. Each file's data is compressed using an algorithm specified in the metadata—usually DEFLATE [10].

This description of the zip format omits many details that are not needed for understanding the zip bomb. For full information, refer to the file format specification [15], particularly Section 4.3.

3 The first insight: overlapping files

By compressing a long string of repeated bytes, we can produce a *kernel* of highly compressed data. By itself, the kernel's compression ratio cannot exceed the DEFLATE limit of 1032, so we want a way to reuse the kernel in many files, without making a separate copy of it in each file. We can do it by overlapping files: making many central directory headers point to a single file, whose data is the kernel. See Figure 2.

Let's look at an example to see how this construction affects the compression ratio. Suppose the kernel is 1000 bytes and decompresses to 1 MB. Then the first 1 MB of output "costs" 1078 bytes of input: 31 bytes for a local file header (including a 1-byte filename), 47 bytes for a central directory header (including a 1-byte filename), and 1000 bytes for the kernel itself. But every 1 MB of output after the first costs only 47 bytes—we don't need another local file header or another copy of the kernel, only an additional central directory header. So while the first copy of the kernel has a compression ratio of $1\,000\,000/1078 \approx 928$, each additional copy pulls the ratio closer to $1\,000\,000/47 \approx 21\,277$. A bigger kernel raises the ceiling.

The problem with this idea is a lack of compatibility. Because many central directory headers point to a single local file header, the metadata—specifically the filename—cannot match for every file. Some parsers balk at that; see Table 2. Info-ZIP UnZip [13] (the standard Unix `unzip` program) extracts the files, but with warnings:

```
$ unzip overlap.zip
  inflating: A
B:  mismatching "local" filename (A),
        continuing with "central" filename version
  inflating: B
...
```

And the Python zipfile module [17] throws an exception:

```
$ python3 -m zipfile -e overlap.zip .
Traceback (most recent call last):
...
__main__.BadZipFile: File name in directory 'B' \
and header b'A' differ.
```

Next we will see how to modify the construction for consistency of filenames, while still retaining most of the advantage of overlapping files.

4 The second insight: quoting local file headers

We need to separate the local file headers for each file, while still reusing a single kernel. Simply concatenating all the local file headers does not work, because the zip parser will find a local file header where it expects to find the beginning of a DEFLATE stream. But the idea will work, with a minor modification. We'll use a feature of DEFLATE, non-compressed blocks, to "quote" local file headers so that they appear to be part of the same DEFLATE stream that terminates in the kernel. Every local file header (except the first) will be interpreted in two ways: as code (part of the structure of the zip file) and as data (part of the contents of a file).

A DEFLATE stream is a sequence of blocks [10 §3.2.3], where each block may be compressed or non-compressed. Compressed blocks are what we usually think of; for example the kernel is one big compressed block. But there are also non-compressed blocks, which start with a 5-byte header [10 §3.2.4] that means simply, "output the next n bytes verbatim." Decompressing a non-compressed block means only stripping the 5-byte header. Compressed and non-compressed blocks may be intermixed freely in a DEFLATE stream. The output is the concatenation of decompressing all the blocks in order. The "non-compressed" notion only has meaning at the DEFLATE layer; the file data still counts as "compressed" at the zip layer, no matter what kind of blocks are used.

It is easiest to understand this quoted-overlap construction from the inside out, beginning with the last file and working backwards to the first. Refer to Figure 3. Start by inserting the kernel, which will form the end of file data for every file. Prepend a local file header LFH_N and add a central directory header CDH_N that points to it. Set the "compressed size" metadata field in LFH_N and CDH_N to the compressed size of the kernel. Now, insert before LFH_N a 5-byte non-compressed block header (colored green in the diagram) whose length field is equal to the size of LFH_N. Prepend a second local file header LFH_{N-1} and add a central directory header CDH_{N-1} that points to it. Set the "compressed size" metadata field in both of the new headers to the compressed size of the kernel, *plus* the size of the non-compressed block header (5 bytes), *plus* the size of LFH_N.

At this point the zip file contains two files, named "Y" and "Z". Let's walk through what a zip parser would see while parsing it. Suppose the compressed size of the kernel is 1000 bytes and the size of LFH_N is 31 bytes. We start at CDH_{N-1} and follow the pointer to LFH_{N-1}. The first file's filename is "Y" and the compressed size of its file data is 1036 bytes. Interpreting the next 1036 bytes as a DEFLATE stream, we first encounter the 5-byte header of a non-compressed block that says to copy the next 31 bytes. We write the next 31 bytes, which are LFH_N, as output to the file "Y". Moving on in the DEFLATE stream, we find a compressed block (the kernel),

	zipped size	non-recursive unzipped size	ratio	recursive unzipped size	ratio
Cox quine [9]	440	440	1.0	∞	∞
Ellingsen quine [11]	28 809	42 569	1.5	∞	∞
42.zip [1]	42 374*	558 432	13.2	4 507 981 343 026 016	106 billion
zbsm.zip (this method)	42 374	5 461 307 620	129 thousand	5 461 307 620	129 thousand
zblg.zip (this method)	9 893 525	281 395 456 244 934	28 million	281 395 456 244 934	28 million
zbxl.zip (this method, Zip64)	45 876 952	4 507 981 427 706 459	98 million	4 507 981 427 706 459	98 million

Table 1: Comparison of zip bomb compression ratios.

* There are two versions of 42.zip, an older version of 42 374 bytes, and a newer version of 42 838 bytes. The difference is that the newer version requires a password before unzipping. We compare only against the older version.

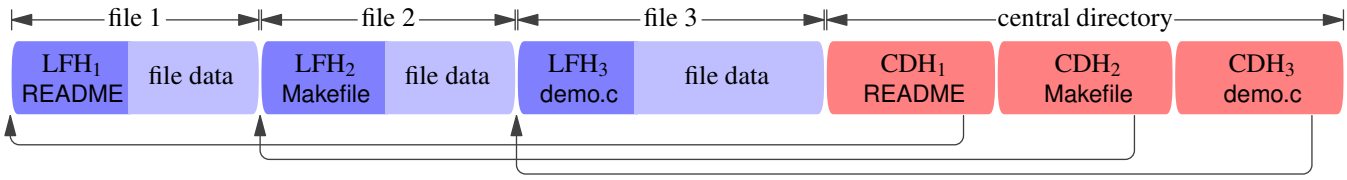

Figure 1: A normal zip file (Section 2).

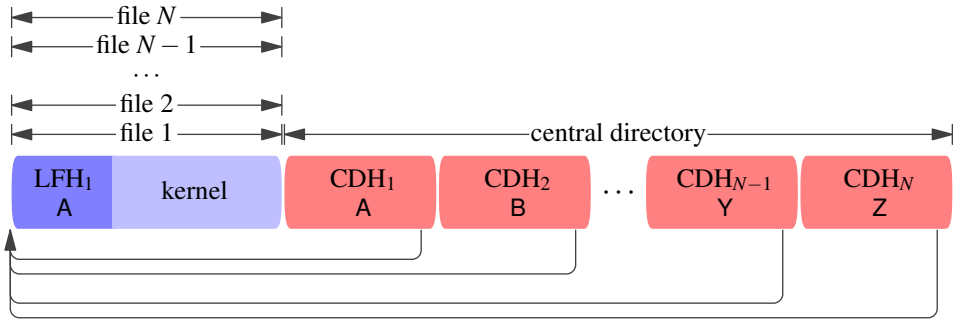

Figure 2: Full-overlap zip bomb construction (Section 3). This construction has problems with compatibility, because filenames do not agree between the central directory headers and the local file headers. The "kernel" is a block of highly compressed data, reused in every file.

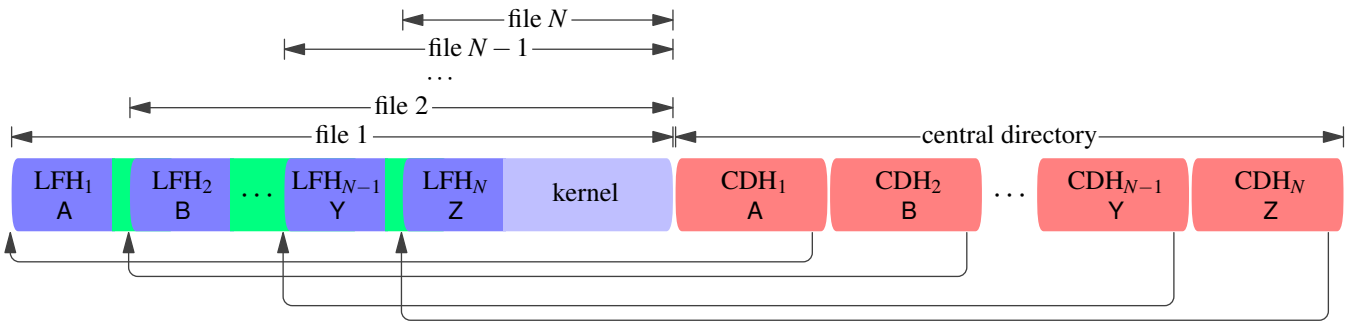

Figure 3: Quoted-overlap zip bomb construction (Section 4). Each file contains the local file headers of all the files which follow it, as well as the kernel. The green parts stand for DEFLATE non-compressed blocks.

which we decompress and append to file "Y". Now we have reached the end of the compressed data and are done with file "Y". Proceeding to the next file, we follow the pointer from CDH_N to LFH_N and find a file named "Z" whose compressed size is 1000 bytes. Interpreting those 1000 bytes as a DEFLATE stream, we immediately encounter a compressed block (the kernel again) and decompress it to the file "Z". Now we have reached the end of the final file and are done. The output file "Z" contains the decompressed kernel; the output file "Y" is the same, but additionally prefixed by the 31 bytes of LFH_N.

We complete the construction by repeating the quoting procedure until the zip file contains the desired number of files. Each new file adds a central directory header, a local file header, and a non-compressed block to quote the immediately succeeding local file header. Compressed file data is generally a chain of DEFLATE non-compressed blocks (the quoted local file headers) followed by the compressed kernel. Each byte in the kernel contributes about $1032N$ to the output size, because each byte is part of all N files. The output files are not all the same size: those that appear earlier in the zip file are larger than those that appear later, because they contain more quoted local file headers. The contents of the output files are not particularly meaningful, but no one said they had to make sense.

This quoted-overlap construction has better compatibility than the full-overlap construction of Section 3, but the compatibility comes at the expense of the compression ratio. There, each added file cost only a central directory header; here, it costs a central directory header, a local file header, and another 5 bytes for the quoting header.

5 Optimization

Now that we have the basic zip bomb construction, we will try to make it as efficient as possible. We want to answer two questions:

- For a given zip file size, what is the maximum compression ratio?
- What is the maximum compression ratio, given the limits of the zip format?

5.1 Kernel compression

It pays to compress the kernel as densely as possible, because every decompressed byte gets magnified by a factor of N. To that end, we use a custom DEFLATE compressor called bulk_deflate, specialized for compressing a string of repeated bytes.

All decent DEFLATE compressors will approach a compression ratio of 1032 when given an infinite stream of repeating bytes, but we care more about specific finite sizes than asymptotics. Figure 4 shows the maximum amount of output that can result from decompressing a DEFLATE stream of a given size, for bulk_deflate and other implementations. bulk_deflate compresses more data into the same space than the general-purpose compressors: about 26 kB more than zlib and Info-ZIP, and about 15 kB more than Zopfli [4], a compressor that trades speed for density.

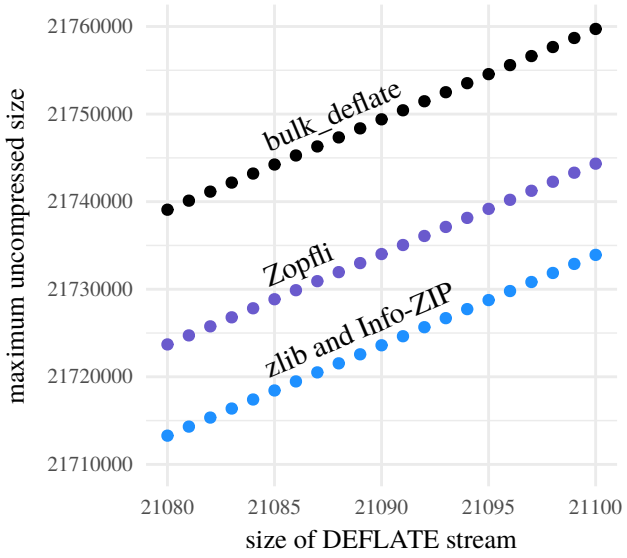

Figure 4: Comparison of DEFLATE compressors on a string of repeated bytes.

The price of bulk_deflate's high compression ratio is a lack of generality. bulk_deflate can only compress strings of a single repeated byte, and only those of specific lengths, namely $517 + 258k$ for integer $k \geq 0$. Besides compressing densely, bulk_deflate is fast, doing essentially constant work regardless of the input size, aside from the $O(n)$ work of actually writing out the compressed string.

5.2 Filenames

For our purposes, filenames are mostly dead weight. While filenames do contribute something to the output size by virtue of being part of quoted local file headers, a byte in a filename does not contribute nearly as much as a byte in the kernel. We want filenames to be as short as possible, while keeping them all distinct, and subject to compatibility considerations.

The first compatibility consideration is character encoding. The zip format specification states that filenames are to be interpreted as CP 437, or UTF-8 if a certain flag bit is set [15 Appendix D]. But this is a major point of incompatibility across zip parsers, which may interpret filenames as being in some fixed or locale-specific encoding. So for compatibility, we must limit ourselves to characters that have the same encoding in both CP 437 and UTF-8; namely, the 95

printable characters of US-ASCII.

We are further restricted by filesystem naming limitations. Some filesystems are case-insensitive, so "a" and "A" do not count as distinct names. Common filesystems like FAT32 prohibit certain characters like '*' and '?' [19 §Limits].

As a safe but not necessarily optimal compromise, our zip bomb will use filenames consisting of characters drawn from a 36-character alphabet that does not rely on case distinctions or use special characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F G H
I J K L M N O P Q R S T U V W X Y Z

Filenames are generated in the obvious way, cycling each position through the possible characters and adding a position on overflow:

"0", "1", "2", …, "Z",
"00", "01", "02", …, "0Z",
…,
"Z0", "Z1", "Z2", …, "ZZ",
"000", "001", "002", …

There are 36 filenames of length 1, 36^2 filenames of length 2, and so on. The length of the nth filename is $\lfloor \log_{36}((n+1)/\frac{36}{35}) \rfloor + 1 = O(\log n)$. Four bytes are enough to represent 1 727 604 distinct filenames.

Given that the N filenames in the zip file are generally not all of the same length, which way should we order them, shortest to longest or longest to shortest? A little reflection shows that it is better to put the longest names last, because those names are the most quoted. Ordering filenames longest last adds over 900 MB of output to the zblg.zip we will see in Section 5.3, compared to ordering them longest first. It is a minor optimization, though, as those 900 MB comprise only 0.0003% of the total output size.

5.3 Kernel size

The quoted-overlap construction allows us to place a compressed kernel of data, and then cheaply copy it many times. For a given zip file size X, how much space should we devote to storing the kernel, and how much to making copies?

To find the optimum balance, we only have to optimize the single variable N, the number of files in the zip file. Every value of N requires a certain amount of overhead for central directory headers, local file headers, quoting block headers, and filenames. All the remaining space can be taken up by the kernel. Because N has to be an integer, and you can only fit so many files before the kernel size drops to zero, it suffices to test every possible value of N and select the one that yields the most output.

Applying the optimization procedure to $X = 42\,374$, the size of 42.zip, finds a maximum at $N = 250$. Those 250 files require 21 195 bytes of overhead, leaving 21 179 bytes for the kernel. A kernel of that size decompresses to 21 841 249 bytes (a ratio of 1031.3). The 250 copies of the decompressed kernel, plus the little bit extra that comes from the quoted local file headers, produces an overall unzipped output of 5 461 307 620 bytes and a compression ratio of 128 thousand. This zip bomb is zbsm.zip—refer to Table 1.

Optimization produced an almost even split between the space allocated to the kernel and the space allocated to file headers. It is not a coincidence. Let's look at a simplified model of the quoted-overlap construction. In the simplified model, we ignore filenames, as well as the slight increase in output file size due to quoting local file headers. Analysis of the simplified model will show that the optimum split between kernel and file headers is approximately even, and that the output size grows quadratically when allocation is optimal.

Define some constants and variables:

X	zip file size (take as fixed)
N	number of files (variable to optimize)
$\mathrm{CDH} = 46$	size of a central directory header
$\mathrm{LFH} = 30$	size of a local file header
$\mathrm{Q} = 5$	size of a quoting block header
$\mathrm{C} \approx 1032$	compression ratio of the kernel

Let $H(N)$ be the amount of header overhead required for N files. Refer to Figure 3 to understand where this formula comes from.

$$H(N) = N \cdot (\mathrm{CDH} + \mathrm{LFH}) + (N-1) \cdot \mathrm{Q}$$

The space remaining for the kernel is $X - H(N)$. The total unzipped size $S_X(N)$ is the size of N copies of the kernel, decompressed at ratio C. (In this simplified model we ignore the minor additional expansion from quoted local file headers.)

$$\begin{aligned} S_X(N) &= (X - H(N))\,\mathrm{C}\,N \\ &= (X - (N \cdot (\mathrm{CDH} + \mathrm{LFH}) + (N-1) \cdot \mathrm{Q}))\,\mathrm{C}\,N \\ &= -(\mathrm{CDH} + \mathrm{LFH} + \mathrm{Q})\,\mathrm{C}\,N^2 + (X + \mathrm{Q})\,\mathrm{C}\,N \end{aligned}$$

$S_X(N)$ is a polynomial in N, so its maximum must be at a place where the derivative $S'_X(N)$ is zero. Taking the derivative and finding the zero gives us N_{OPT}, the optimal number of files.

$$\begin{aligned} S'_X(N_{\mathrm{OPT}}) &= -2(\mathrm{CDH} + \mathrm{LFH} + \mathrm{Q})\,C\,N_{\mathrm{OPT}} + (X + \mathrm{Q})\,\mathrm{C} \\ 0 &= -2(\mathrm{CDH} + \mathrm{LFH} + \mathrm{Q})\,C\,N_{\mathrm{OPT}} + (X + \mathrm{Q})\,\mathrm{C} \\ N_{\mathrm{OPT}} &= \frac{X + \mathrm{Q}}{2(\mathrm{CDH} + \mathrm{LFH} + \mathrm{Q})} \end{aligned}$$

$H(N_{\text{OPT}})$ gives the optimal amount of space to allocate for file headers. It is independent of CDH, LFH, and C, and is close to $X/2$.

$$\begin{aligned} H(N_{\text{OPT}}) &= N_{\text{OPT}} \cdot (\text{CDH} + \text{LFH}) + (N_{\text{OPT}} - 1) \cdot \text{Q} \\ &= \frac{X - \text{Q}}{2} \end{aligned}$$

$S_X(N_{\text{OPT}})$ is the total unzipped size when the allocation is optimal. From this we see that the output size grows quadratically in the input size.

$$S_X(N_{\text{OPT}}) = \frac{(X + \text{Q})^2 \, \text{C}}{4(\text{CDH} + \text{LFH} + \text{Q})} \tag{1}$$

As we make the zip file larger, eventually we run into the limits of the zip format. A zip file can contain at most $2^{16} - 1$ files, and each file can have an uncompressed size of at most $2^{32} - 1$ bytes. Worse than that, some implementations (see Table 2) take the maximum possible values as an indicator of the presence of 64-bit extensions (Section 7), so our limits are actually $2^{16} - 2$ and $2^{32} - 2$. It happens that the first limit we hit is the one on uncompressed file size. At a zip file size of 8 319 377 bytes, naive optimization would give us a file count of 47 837 and a largest file with an impossible size of $2^{32} + 311$ bytes.

Accepting that we cannot increase N nor the size of the kernel without bound, we would like find the maximum compression ratio achievable while remaining within the limits of the zip format. The way to proceed is to make the kernel as large as possible, and have the maximum number of files. Even though we can no longer maintain the roughly even split between kernel and file headers, each added file *does* increase the compression ratio—just not as fast as it would if we were able to keep growing the kernel, too. In fact, as we add files we will need to *decrease* the size of the kernel to make room for the maximum file size that gets slightly larger with each added file.

The plan results in zblg.zip, a zip file that contains $2^{16} - 2$ files and a kernel that decompresses to $2^{32} - 2\,178\,825$ bytes. Refer to Table 1. Files get longer towards the beginning of the zip file—the first and largest file decompresses to $2^{32} - 56$ bytes. That is as close as we can get using the coarse output sizes of bulk_deflate—encoding the final 54 bytes would cost more bytes than they are worth. (The zip file as a whole has a compression ratio of 28 million, and the final 54 bytes would gain at most $54 \cdot 1032 \cdot (2^{16} - 2) \approx 36.5$ million bytes, so it only helps if the 54 bytes can be encoded in 1 byte—we could not do it in less than 2.) The output size of this zip bomb, 281 395 456 244 934 bytes, is 99.97% of the theoretical maximum $(2^{32} - 1) \cdot (2^{16} - 1)$. Any major improvements to the compression ratio can only come from reducing the input size, not increasing the output size.

6 Efficient CRC-32 computation

Among the metadata in the central directory header and local file header is a CRC-32 checksum of the uncompressed file data. This poses a problem, because directly calculating the CRC-32 of each file requires doing work proportional to the total *unzipped* size, which is large by design. (It's a zip bomb, after all.) We would prefer to do work that in the worst case is proportional to the *zipped* size. Two factors work in our advantage: all files share a common suffix (the kernel), and the uncompressed kernel is a string of repeated bytes. We will represent CRC-32 as a matrix product—this will allow us not only to compute the checksum of the kernel quickly, but also to reuse computation across files. The technique described in this section is a slight extension of the `crc32_combine` function in zlib, which Mark Adler has explained [3].

You can model CRC-32 as a state machine that updates a 32-bit state register for each incoming bit. The basic update operations for a 0 bit and a 1 bit are:

```
uint32 crc32_update_0(uint32 state) {
    // Shift out the least significant bit.
    bit b = state & 1;
    state = state >> 1;
    // If the shifted-out bit was 1, XOR
    // with the CRC-32 constant.
    if (b == 1)
        state = state ^ 0xedb88320;
    return state;
}
uint32 crc32_update_1(uint32 state) {
    // Do as for a 0 bit, then XOR
    // with the CRC-32 constant.
    return crc32_update_0(state) ^ 0xedb88320;
}
```

If you think of the state register as a 32-element binary vector, and use XOR for addition and AND for multiplication, then `crc32_update_0` is a linear transformation; i.e., it can be represented as multiplication by a 32×32 binary transformation matrix. To see why, observe that multiplying a matrix by a vector is just summing the columns of the matrix, after multiplying each column by the corresponding element of the vector. The shift operation `state >> 1` is just taking each bit i of the state vector and multiplying it by a vector that is 0 everywhere except at bit $i - 1$ (numbering the bits from right to left). The conditional final XOR `state ^ 0xedb88320` that only happens when bit `b` is 1 can instead be represented as first multiplying `b` by 0xedb88320 and then XORing it into the state.

Furthermore, `crc32_update_1` is just `crc32_update_0` plus (XOR) a constant. That makes `crc32_update_1` an affine transformation: a matrix multiplication followed by a translation (i.e., vector addition). We can represent both the matrix multiplication and the translation in a single step if we enlarge the dimensions of the transformation matrix to 33×33 and append an extra element to the state vector

$$M_0 = \begin{bmatrix}
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

$$M_1 = \begin{bmatrix}
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

Figure 5: The 33×33 transformation matrices M_0 and M_1 that compute the CRC-32 state change effected by a 0 bit and a 1 bit respectively. Column vectors are stored with the most significant bit at the bottom: reading the first column from bottom to top, you see the CRC-32 polynomial constant $\text{edb88320}_{16} = 11101101101110001000001100100000_2$. The two matrices differ only in the final column, which represents a translation vector in homogeneous coordinates. In M_0 the translation is zero and in M_1 it is edb88320_{16}, the CRC-32 polynomial constant. The 1's just above the diagonal represent the shift operation `state >> 1`.

that is always 1. (This representation is called homogeneous coordinates.)

Both operations `crc32_update_0` and `crc32_update_1` can be represented by a 33×33 transformation matrix. The matrices M_0 and M_1 are shown in Figure 5. The benefit of a matrix representation is that matrices compose. Suppose we want to represent the state change effected by processing the ASCII character 'a', whose binary representation is 01100001_2. We can represent the cumulative CRC-32 state change of those 8 bits in a single transformation matrix:

$$M_\text{a} = M_0 M_1 M_1 M_0 M_0 M_0 M_0 M_1$$

And we can represent the state change of a string of repeated 'a's by multiplying many copies of M_a together—matrix exponentiation. We can do matrix exponentiation quickly using a square-and-multiply algorithm, which allows us to compute M^n in only about $\log_2 n$ steps. For example, the matrix representing the state change of a string of 9 'a's is

$$\begin{aligned}
(M_\text{a})^9 &= M_\text{a} M_\text{a} M_\text{a} M_\text{a} M_\text{a} M_\text{a} M_\text{a} M_\text{a} M_\text{a} \\
&= (M_\text{a} M_\text{a} M_\text{a} M_\text{a})^2 M_\text{a} \\
&= ((M_\text{a} M_\text{a})^2)^2 M_\text{a} \\
&= (((M_\text{a})^2)^2)^2 M_\text{a}
\end{aligned}$$

The square-and-multiply algorithm is useful for computing M_kernel, the matrix for the uncompressed kernel, because the kernel is a string of repeated bytes. To produce a CRC-32 checksum value from a matrix, multiply the matrix by the zero vector. (The zero vector in homogeneous coordinates, that is: 32 0's followed by a 1. Here we omit the minor complication of pre- and post-conditioning the checksum.) To compute the checksum for every file, we work backwards. Start by initializing $M := M_\text{kernel}$. The checksum of the kernel is also the checksum of the final file, file N, so multiply M by the zero vector and store the resulting checksum in CDH_N and LFH_N. The file data of file $N-1$ is the same as the file data of file N, but with an added prefix of LFH_N. So compute M_{LFH_N}, the state change matrix for LFH_N, and update $M := M M_{\text{LFH}_N}$. Now M represents the cumulative state change from processing LFH_N followed by the kernel. Compute the checksum for file $N-1$ by again multiplying M by the zero vector. Continue the procedure, accumulating state change matrices into M, until all checksums have been computed.

7 Extension: Zip64

In Section 5.3 we hit a wall on expansion due to limits of the zip format—it was impossible to produce more than about 281 TB of output, no matter how cleverly packed the zip file. It is possible to surpass those limits using Zip64, an extension to the zip format that increases the size of certain header fields

to 64 bits. Support for Zip64 is by no means universal, but it is one of the more commonly implemented extensions—see Table 2. As regards the compression ratio, the effect of Zip64 is to increase the size of a central directory header from 46 bytes to 58 bytes, and the size of a local directory header from 30 bytes to 50 bytes. Referring to Equation 1, we see that a zip bomb in Zip64 format still grows quadratically, but more slowly because of the larger denominator—this is visible in Figure 6 in the Zip64 line's slightly lower vertical placement. In exchange for the loss of compatibility and slower growth, we get the removal of all practical file size limits.

Suppose we want a zip bomb that expands to 4.5 PB, the same size that 42.zip recursively expands to. How big must the zip file be? Using binary search, we find that the smallest zip file whose unzipped size exceeds the unzipped size of 42.zip has a zipped size of 46 MB. See the zbxl.zip row in Table 1.

With Zip64, it's not practically interesting to consider the maximum compression ratio, because we can just keep increasing the zip file size, and the compression ratio along with it, until even the compressed zip file is prohibitively large. An interesting threshold, though, is 2^{64} bytes (18 EB or 16 EiB)—that much data will not fit on most filesystems [19 §Limits]. Binary search finds the smallest zip bomb that produces at least that much output: it contains 12 million files and has a compressed kernel of 1.5 GB. The total size of the zip file is 2.9 GB and it unzips to $2^{64} + 11\,727\,895\,877$ bytes, having a compression ratio of over 6.2 billion.

8 Extension: bzip2

DEFLATE is the most common compression algorithm used in the zip format, but it is only one of many options [15 §4.4.5]. bzip2 [18], while not nearly as compatible as DEFLATE (see Table 2), is probably the second most commonly supported compression algorithm. Empirically, bzip2 has a maximum compression ratio of about 1.4 million, which allows for denser packing of the kernel. Ignoring the loss of compatibility, does bzip2 enable a more efficient zip bomb?

Yes—but only for small files. The problem is that bzip2 does not have anything like the non-compressed blocks of DEFLATE that we used in Section 4 to quote local file headers. So it is not possible to reuse the kernel—each file must have its own copy, and therefore the overall compression ratio is not better than the ratio of any single file. (See the paragraph on extra-field quoting in the next section for a possible workaround.) In Figure 6 we see that bzip2 outperforms DEFLATE-with-quoting only for zip bombs under about a megabyte.

If you happen to know that a certain zip parser both supports bzip2 and tolerates mismatched filenames, then you can use the full-overlap construction of Section 3. There is no need for quoting in that construction, so it is compatible with bzip2.

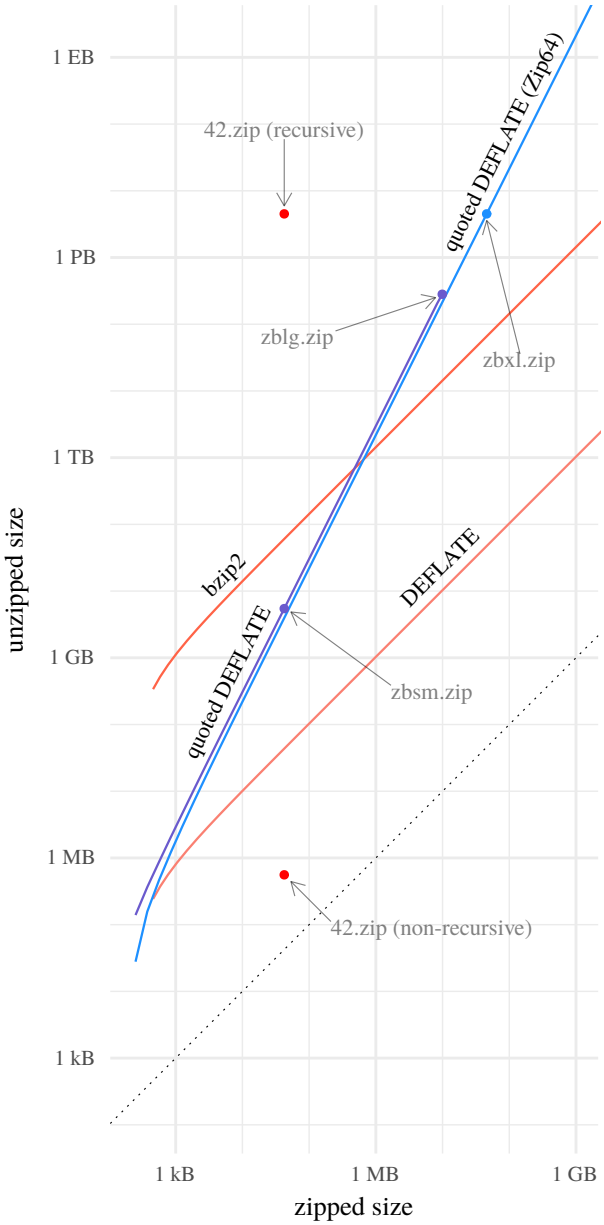

Figure 6: Zipped size versus unzipped size for various zip bomb constructions. Note the log–log scales. The red and orange lines are naive zip bombs without overlapping files. They have a linear rate of growth, as evidenced by their 1:1 slope. The vertical offset of the bzip2 line shows that its compression ratio is about a thousand times greater than that of DEFLATE. The blue lines are quoted-overlap zip bombs that use DEFLATE. They have a quadratic rate of growth, as shown by their 2:1 slope. Zip64 has a slightly lower compression ratio for a zip file of the same size, but it permits output in excess of 281 TB. Labeled dots indicate specific zip bombs that also appear in Table 1.

	Info-ZIP UnZip 6.0 [13]	Python 3.7 zipfile [17]	Go 1.12 archive/zip [12]	yauzl 2.10.0 [20] (Node.js)	Nail [7] examples/zip [6]	Android 9.0.0 r1 libziparchive [5]	sunzip 0.4 [2] (streaming)
DEFLATE	✓	✓	✓	✓	✓	✓	✓
Zip64	✓	✓	✓	✓	✗	✗	✓
bzip2	✓	✓	✗	✗	✗	✗	✓
permits mismatched filenames	warns	✗	✓	✓	✓	✗	✓
permits incorrect CRC-32	warns	✗	if zero	✓	✗	✓	✗
permits file size of $2^{32}-1$	✓	✓	✓	✗	✓	✓	✓
permits file count of $2^{16}-1$	✓	✓	✓	✗	✓	✓	✓
unzips full overlap (Section 3)	warns	✗	✓	✓	✓	✗	✗
unzips quoted overlap (Section 4)	✓	✓	✓	✓	✓	✓	✗
unzips quoted overlap Zip64 (Section 7)	✓	✓	✓	✓	✗	✗	✗

Table 2: Compatibility of selected zip parsers with various zip features, edge cases, and zip bomb constructions. The background colors indicate a scale from less restrictive to more restrictive. For best compatibility, use DEFLATE compression without Zip64, match names in central directory headers and local file headers, compute correct CRCs, and avoid the maximum values of 32-bit and 16-bit fields.

9 Discussion

In related work, Plötz et al. [16 §4] used overlapping files to create a near-self-replicating zip file. Gynvael Coldwind [8 p47] has previously suggested overlapping files in the style of Section 3.

We have designed the quoted-overlap zip bomb construction for compatibility, taking into consideration a number of implementation differences, some of which are shown in Table 2. The resulting construction is compatible with zip parsers that work in the usual back-to-front way, first consulting the central directory and using it as an index of files. Among these is the example zip parser included in Nail [7], which is automatically generated from a formal grammar. The construction is not compatible, however, with "streaming" parsers, those that parse the zip file from beginning to end in one pass without first reading the central directory. By their nature, streaming parsers do not permit any kind of file overlapping. The most likely outcome is that they will extract only the first file. They may even raise an error besides, as is the case with sunzip [2], which parses the central directory at the end and checks it for consistency with the local file headers it has already seen.

If you need the extracted files to start with a certain prefix other than the bytes of a local file header, you can insert a DEFLATE block before the non-compressed block that quotes the next header. Not every file in the zip file has to participate in the bomb construction: you can also include ordinary files if needed to conform to some special format. Many file formats use zip as a container; examples are Java JAR, Android APK, and LibreOffice documents.

There's another quoting trick possible. At the end of each local file header there is a variable-length *extra field* whose purpose is to store information that doesn't fit into the ordinary fields of the header. This extra information may include, for example, a high-resolution timestamp or a Unix uid/gid; Zip64 sizes are stored in the extra field. Because the extra field is at the very end of the header, if we increase its apparent length, it will grow to include whatever bytes follow—such as the next local file header. The extra field therefore provides an alternative means of quoting. The benefits of extra-field quoting, as compared to DEFLATE non-compressed block quoting, are threefold:

1. extra-field quoting requires only 4 bytes of overhead, not 5, leaving more room for the kernel;
2. extra-field quoting does not increase the size of files, which leaves more headroom for a bigger kernel when operating at the limits of the zip format;
3. extra-field quoting provides a way of combining quoting with bzip2.

Despite these benefits, extra-field quoting is less flexible than DEFLATE quoting. It does not chain the way that DEFLATE quoting does: each local file header must quote not only the immediately following header but *all* following headers. Because the extra field has a length limit of $2^{16}-1$ bytes, it is only possible to quote up to 1808 local file headers, or 1170 with Zip64 (assuming filenames are allocated as in Section 5.2). You can use extra-field quoting for the first (shortest)

local file headers, but then must switch to DEFLATE quoting for the rest. Another problem is that we must select a 16-bit type tag [15 §4.5.2]. We want to avoid using a type tag that has already been allocated and has meaning to parsers, or else the quoted bytes will be interpreted as meaningful extra data. Zip parsers are supposed to ignore unknown type tags, so we could choose a type tag at random, but there is the risk that a parser now or in the future will try to interpret it. Extra-field quoting increases the compression ratio of zbsm.zip by 1.2%; that of zblg.zip by 0.019%; and that of zbxl.zip by 0.0025%.

> *Note to reviewers: we are considering whether extra-field quoting meets our compatibility goals. We may decide to work the preceding paragraph into Section 4 as a primary technique.*

Detecting the type of zip bomb we have developed is easy: just look for overlapping files. Parse the central directory, then sort the central directory entries in order of the local file headers the point to. Reject the zip file if any file's "compressed size" metadata field would make it overlap the following file. In general, though, rejecting overlapping files is not sufficient against all kinds of zip bombs. Section 8 and Figure 6 show that at small sizes, a non-overlapped bzip2 zip bomb is even more effective than the overlapped construction with DEFLATE. Trying to predict a zip file's total uncompressed size by peeking into the metadata and summing the "uncompressed size" fields of all files does not work, in general, because the value of the metadata field may not agree with the actual uncompressed size of the file. In general, trying to predict in advance whether a zip file is safe to open or not is fraught, because there are so many zip parser implementations that differ in behavior. Robust protection against zip bombs involves placing time, memory, and disk space limits on the zip parser while it operates. Handle zip parsing, as any complex operation on untrusted data, with caution.

Availability

> *For anonymous submission, we have uploaded source code and the three example zip bombs to an anonymous hosting service. We will of course make the source code and zip bombs publicly available. Do be careful with these—at least, be sure to "Save As" to avoid automatic unzipping by your web browser.*

source.zip SHA256:965bda4ed4535956fbd8880efe78d64fbc6e31a835306b85f6bb551bfbc106ef
https://framadrop.org/r/ZZ9ahctElh#11Gnm6QqzXMaM5XR20RUhre4+6Sz5I3CPGUoWj3TsrY=

zbsm.zip SHA256:fb4ff972d21189beec11e05109c4354d0cd6d3b629263d6c950cf8cc3f78bd99
https://framadrop.org/r/UIVszQVrjt#6vTkGCBCSJd50sz9oS+ZiK1SNYbGaX2hOdUPAgckp88=

zblg.zip SHA256:f1dc920869794df3e258f42f9b99157104cd3f8c14394c1b9d043d6fcda14c0a
https://framadrop.org/r/7xWQcduGPR#KyD7bm7CB4PQwTwXW5J6WEIsqilwZWx+6L/Tgcmk0L0=

zbxl.zip SHA256:eafd8f574ea7fd0f345eaa19eae8d0d78d5323c8154592c850a2d78a86817744
https://framadrop.org/r/6JET9-Ieb-#mreNPDbTk28ouqDm+x02dHWOSWf0QwemE9jCugPyM3Q=

References

[1] 42.zip. https://www.unforgettable.dk/. *Note to reviewers: we would be grateful for any additional information about the provenance of 42.zip. Pellegrino et al. [14 §3.1] call it a "classic zip bomb" but do not trace its source.*

[2] Mark Adler. sunzip. https://github.com/madler/sunzip.

[3] Mark Adler. Re: How should/could I combine CRCs? sci.crypt, September 2008. https://groups.google.com/d/msg/sci.crypt/SHyr5bp5rtc/UGlf4tK3RPMJ, https://stackoverflow.com/a/23126768.

[4] Jyrki Alakuijala and Lode Vandevenne. Data compression using Zopfli. Technical report, Google, February 2013. https://web.archive.org/web/20160629205704/http://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf.

[5] Android Open Source Project. libziparchive. https://android.googlesource.com/platform/system/core/+/refs/tags/android-9.0.0_r1/libziparchive.

[6] Julian Bangert and Nickolai Zeldovich. Nail examples/zip. https://github.com/jbangert/nail/tree/4bd9cc29c4092abe7a77f8294aff2337bba02ec5/examples/zip.

[7] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert.

[8] Gynvael Coldwind. Ten thousand security pitfalls: The ZIP file format, April 2018. https://gynvael.coldwind.pl/?id=682.

[9] Russ Cox. Zip files all the way down. March 2010. https://research.swtch.com/zip.

[10] L. Peter Deutsch. RFC 1951. DEFLATE compressed data format specification version 1.3, May 1996. https://tools.ietf.org/html/rfc1951.

[11] Erling Ellingsen. ZIP file quine. Before 2005. https://web.archive.org/web/20160130230432/http://www.steike.com/code/useless/zip-file-quine/.

[12] Go 1.12. archive/zip. https://golang.org/pkg/archive/zip/.

[13] Info-ZIP. UnZip. http://infozip.sourceforge.net/UnZip.html.

[14] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, and Neeraj Suri. In the compression hornet's nest: A security study of data compression in network services. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pellegrino.

[15] PKWARE Inc. APPNOTE.TXT - .ZIP file format specification version 6.3.6, April 2019. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

[16] Henryk Plötz, Martin Stigge, Wolf Müller, and Jens-Peter Redlich. Self-replication in J2ME MIDlets. Technical Report SAR-PR-2006-04, Humboldt University Berlin, March 2006. http://sar.informatik.hu-berlin.de/research/publications/index.htm#SAR-PR-2006-04.

[17] Python 3.7. zipfile module. https://docs.python.org/3/library/zipfile.html.

[18] Julian Seward. bzip2, 2018. https://sourceware.org/bzip2/.

[19] Wikipedia contributors. Comparison of file systems. May 2019. https://en.wikipedia.org/w/index.php?title=Comparison_of_file_systems&oldid=896966260.

[20] Josh Wolfe. yauzl. https://github.com/thejoshwolfe/yauzl.

[21] zlib technical details. May 2006. https://www.zlib.net/zlib_tech.html.