applied sciences

MDPI

Article

A Generator for Recursive Zip Files

Ruben Van Mello [1] **and Pieter Audenaert** [2,3,*]

[1] Faculty of Sciences, Ghent University, 9000 Gent, Belgium; ruben.vanmello@ugent.be
[2] Faculty of Engineering, Ghent University, 9000 Gent, Belgium
[3] Interuniversity Microelectronics Centre (IMEC), 9000 Gent, Belgium
* Correspondence: pieter.audenaert@ugent.be

Abstract: This paper explores the concept of zip quines, which are zip files that contain themselves upon extraction, extending the idea of computational self-reference. While only two individuals, Russ Cox and Erling Ellingsen, have created such entities, this study focuses on Cox's method to develop a generator for these files. Overcoming the initial limitations, the generator allows for the inclusion of additional files within the zip quine. Additionally, this research explores the concept of looped zip files, wherein a zip archive contains another archive. This archive then contains the initial zip file. By offering practical methodologies and insights, this study advances the understanding and application of quines in computer science.

Keywords: quine; recursive zip; DEFLATE; LZ77

Citation: Van Mello, R.; Audenaert, P. A Generator for Recursive Zip Files. *Appl. Sci.* **2024**, *14*, 9797. https://doi.org/10.3390/app14219797

Academic Editor: Luis Javier Garcia Villalba

Received: 13 September 2024
Revised: 15 October 2024
Accepted: 16 October 2024
Published: 26 October 2024

1. Introduction

The concept of self-reference has long captivated computer scientists, leading to the development of programs that can reproduce their own source code [1]. These programs are called quines, and a particularly intriguing example is the "zip quine". A zip quine is a program stored within a compressed zip file that contains everything required to recreate itself when extracted. To understand how this works, it is helpful to know how file compression operates. Compression algorithms like those used in zip files reduce the file size by identifying patterns or repeated sections of data, representing them with fewer bytes. This process allows large amounts of information to be stored more efficiently.

In a zip quine, the zip file does not just hold data. It holds the exact instructions necessary to recreate the file itself. This self-replicating behavior represents an elegant intersection of compression and recursion. However, the challenge lies in preserving this property while expanding the contents of the archive. This paper investigates how zip quines can be extended by adding extra files and content, while still maintaining their self-referential nature.

Additionally, this study explores "looped zip files", a novel and more complex concept where one zip file contains another zip file, and this inner zip file also contains the original zip file. This creates a recursive structure, forming an infinite loop of nested archives. Each level of the loop refers back to the previous one, creating a cycle with no clear endpoint. Looped zip files push the boundaries of compression and recursion even further, introducing new technical challenges.

2. Materials and Methods

The creation of zip quines has been largely attributed to two individuals: Russ Cox [2] and Erling Ellingsen [3]. Ellingsen created the most sophisticated quine. His quine did not only encapsulate itself, but also another file. However, the method that he employed to create these entities has been lost over time. Consequently, Ellingsen now refers to Russ Cox's description of the process. Cox's work in this area has provided valuable insights into the creation of zip quines, albeit with certain limitations.

The zip format has some specific headers and footers, which contain important metadata [4]. Consider the names of the files inside the archive, timestamps, checksums, etc. Each file inside the archive has a Local File Header with the metadata, followed by the encoded data using the DEFLATE algorithm. At the end of the archive, there is the so-called Central Directory. This part consists of a Central Directory File Header for each file inside the archive. This contains metadata similar to the Local File Header, but also contains the position of the Local File Header. After this follows a single End of Central Directory Record, whose main purpose is to locate the start of the Central Directory. The general structure is illustrated in Table 1. More information about the individual components can be found in Appendix A.

Table 1. Structure of a zip file.

Local File Header 1
File Data 1
...
Local File Header n
File Data n
Central Directory File Header 1
...
Central Directory File Header n
End of Central Directory Record

2.1. DEFLATE

It is important to note that the zip format, widely used for file compression and archival, employs the DEFLATE algorithm [5] as its primary compression method. This algorithm combines the LZ77 algorithm [6] and Huffman coding [7] to achieve its compression. While the zip format also supports other compression methods, the DEFLATE algorithm is also the algorithm that was used by Cox to create its quine. The significance of using DEFLATE in this context lies in the fact that Russ Cox developed a quine based on the LZ77 algorithm. Our method builds upon his, so LZ77 is essential in generating our zip quines.

2.1.1. LZ77

LZ77 works by replacing repeated occurrences of data with references to previous occurrences, achieving compression. DEFLATE employs a variant of LZ77 known as LZSS [8]. Within LZSS, two token types are utilized: literal tokens and repeat tokens.

- Literal tokens, denoted as $l(x)$, indicate that x is a direct output without modification.
- Repeat tokens, on the other hand, represent recurring sequences. These require two values: y and z. The value y specifies how far back in the output to look (i.e., how many bytes to "backtrack"), while z indicates the length of the sequence to copy from the previous position. Essentially, the repeat token instructs the algorithm to go back y bytes and copy z bytes of data to the current position, which saves space by avoiding redundancy.

An example of the algorithm is given in Figure 1. The DEFLATE specification enforces a lookahead buffer of 258 bytes (length) and a search buffer of 32 KiB (distance).

Search buffer							In	Lookahead buffer										Out
						a	a	b	r	a	c	a	d	a	b	r	a	l(a)
						a	b	r	a	c	a	d	a	b	r	a		l(b)
					a	b	r	a	c	a	d	a	b	r	a			l(r)
				a	b	r	a	c	a	d	a	b	r	a				r(3, 1)
			a	b	r	a	c	a	d	a	b	r	a					l(c)
		a	b	r	a	c	a	d	a	b	r	a						r(2, 1)
	a	b	r	a	c	a	d	a	b	r	a							l(d)
a	b	r	a	c	a	d	a	b	r	a								r(7, 4)

Figure 1. Example of encoding abracadabra with LZSS.

2.1.2. Huffman

In addition to LZ77 compression, another key component of the DEFLATE algorithm is Huffman encoding. Huffman encoding is a variable-length encoding technique that assigns shorter codes to more frequent symbols and longer codes to less frequent symbols, optimizing the compression of the data. Within the DEFLATE algorithm, Huffman encoding is applied to both literal and repeat tokens, enabling the efficient representation of data patterns.

Within the zip format, data are typically divided into blocks for compression, with three block types: stored, fixed and dynamic blocks. Stored blocks simply store raw data without compression. Fixed blocks, on the other hand, use pre-defined Huffman code tables to compress the data efficiently. These tables are fixed and provided by the DEFLATE specification [5]. Dynamic blocks also exist, but they are not needed to create recursive zip files and will not be further discussed.

Each block in the DEFLATE algorithm is characterized by a three-bit header providing essential information about the block's structure. The first bit denotes whether the block is the final block in the data stream. The second and third bits specify the block type, with a stored block identified by two consecutive 0 bits. Following the three-bit header, the structure of the block varies depending on its type. In the case of a stored block, the byte is padded with 0 bits until reaching the byte boundary. Subsequently, two bytes indicate the number of raw bytes stored within the block. Interestingly, the following two bytes are the bitwise negation of the previous two bytes, ensuring data integrity. Finally, the raw data themselves follow this header structure. If a stored block does not start in the middle of a byte, the header is thus always five bytes. A schematic overview is given in Figure 2.

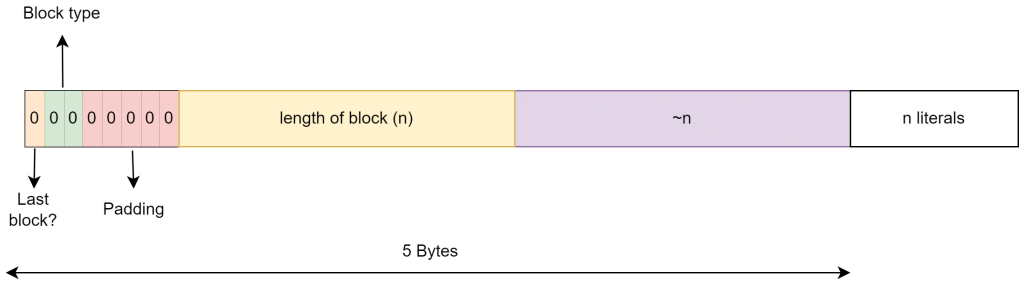

Figure 2. Structure of a stored block.

As stored blocks do not compress data, they can only be used to store the literal tokens from the LZ77 algorithm. In contrast, fixed blocks can encode both literals and repeat tokens. Similar to stored blocks, fixed blocks begin with the same three-bit header, denoting the block type and whether it marks the final block in the stream. However, the distinctive identifier for a fixed block is the sequence "01". Following this header, the compressed data within the fixed block adhere to a predefined encoding structure specified in the DEFLATE specification [5]. To mark the end of the block, seven 0 bits are added. A visual representation is shown in Figure 3.

The tokens are encoded using fixed tables as defined in the specification, employing a predetermined alphabet with codes ranging from 0 to 287. Each code corresponds to a fixed number of bits and a specific binary representation. Table 2 is organized such that

the codes for more frequently occurring values are shorter, while less frequent values have longer codes, thereby enhancing the compression efficiency.

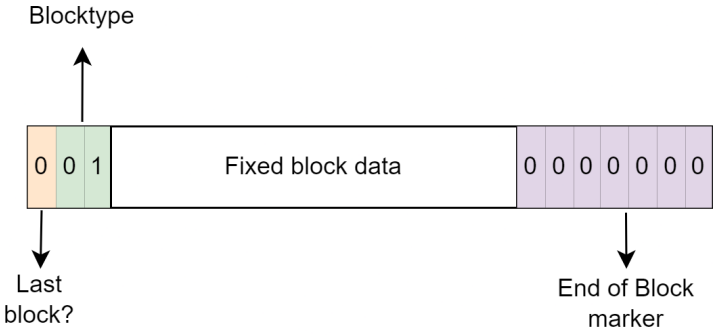

Figure 3. Structure of a fixed block.

Table 2. Fixed block codes and their representations.

Codes	Number of Bits	Binary Representation
0–143	8 bits	00110000 to 10111111
144–255	9 bits	110010000 to 111111111
256–279	7 bits	0000000 to 0010111
280–287	8 bits	11000000 to 11000111

The codes serve to differentiate between literal tokens and repeat tokens.

1. 0 to 255: These codes are assigned to literal tokens.
2. 256: This code indicates the end of the fixed Huffman block, signaled by the seven 0 bits previously mentioned.
3. 257 to 285: These codes represent repeat tokens, specifically the length of the repetition. Additional bits are used depending on the repeat token's length.
4. 286 and 287: Although codes 286 and 287 can be represented within this format, they do not appear in the compressed data. These codes are unused but are part of the overall code design.

The use of 256 codes for literal tokens ensures that all possible bytes can be represented. While the specification allows a repeat token to have a maximum length of 258, there are insufficient codes to represent all possible lengths directly.

To address this issue, each code for the encoding of a repeat token is associated with a base length and a corresponding number of extra bits. A detailed overview of this system is provided in Figure 4.

Code	Base length	Extra bits
257	3	0
258	4	0
259	5	0
260	6	0
261	7	0
262	8	0
263	9	0
264	10	0
265	11	1
266	13	1

Code	Base length	Extra bits
267	15	1
268	17	1
269	19	2
270	23	2
271	27	2
272	31	2
273	35	3
274	43	3
275	51	3
276	59	3

Code	Base length	Extra bits
277	67	4
278	83	4
279	99	4
280	115	4
281	131	5
282	163	5
283	195	5
284	227	5
285	258	0

Figure 4. Table from the DEFLATE specification for the encoding of the length of a repeat token.

This section only addresses the length of a repeat token. The distance is handled similarly. When decoding the compressed data, the distance immediately follows the length once the code associated with a repeat token is encountered. A corresponding table for the distance is provided in the specification and can be found in Figure 5.

Code	Base length	Extra bits	Code	Base length	Extra bits	Code	Base length	Extra bits
0	1	0	10	33	4	20	1025	9
1	2	0	11	49	4	21	1537	9
2	3	0	12	65	5	22	2049	10
3	4	0	13	97	5	23	3073	10
4	5	1	14	129	6	24	4097	11
5	7	1	15	193	6	25	6145	11
6	9	2	16	257	7	26	8193	12
7	13	2	17	385	7	27	12,289	12
8	17	3	18	513	8	28	16,385	13
9	25	3	19	769	8	29	24,577	13

Figure 5. Table from the DEFLATE specification for the encoding of the distance of a repeat token.

The distance code is consistently encoded in a 5-bit binary format. Given that the maximum distance for a repeat token is 32 KiB, up to 13 additional bits are necessary to accommodate all possible distances.

Example: To encode a repeat token with a distance of 24 and a length of 24, the process begins with a 0 bit, indicating that this is not the final block. This is followed by two bits that specify the use of a fixed Huffman block. According to Figure 4, a length of 24 corresponds to a base length of 23, associated with code 270 and requiring two additional bits.

Given that the base length is 23, an additional 1 is required to achieve the target length of 24 for the repeat token. This is encoded by placing the binary representation of the number 1 within the two extra bits.

Following the length encoding, the distance is addressed. Referring to the distance table (Figure 5), a distance of 24 corresponds to a base distance of 17, which is represented by code 8 and necessitates three extra bits. To obtain the desired distance of 24, an increment of 7 is necessary, represented by three 1 bits.

Finally, seven 0 bits are appended to signify the end of the fixed block. The complete example is illustrated in Figure 6.

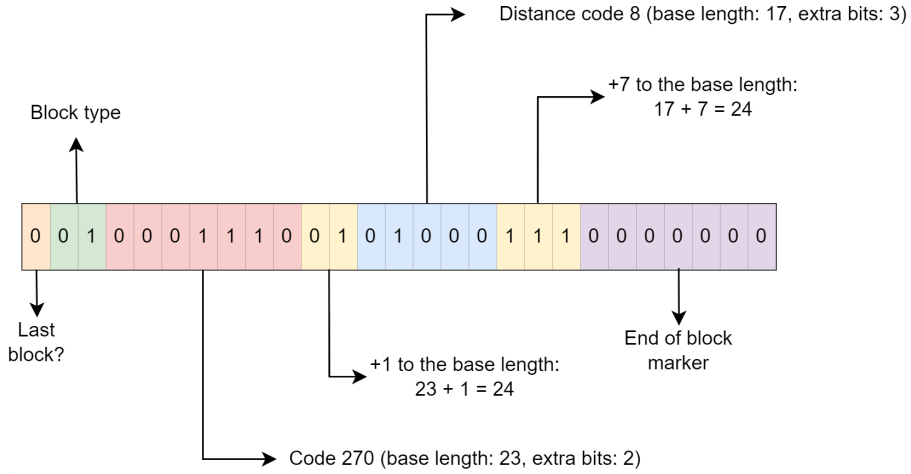

Figure 6. Example: encoding a repeat token in a fixed block.

The relationship between the LZ77, Huffman coding, and DEFLATE algorithms is visually represented in Figure 7. This diagram illustrates how the LZ77 compression and Huffman coding blocks contribute to the DEFLATE process. The DEFLATE block then branches into three types of data representation, namely stored, fixed and dynamic blocks,

which define how compressed data are managed and stored. This visual representation helps to clarify how these components interact within the DEFLATE algorithm, providing a clearer understanding of their roles before we proceed to explore the concept of quines.

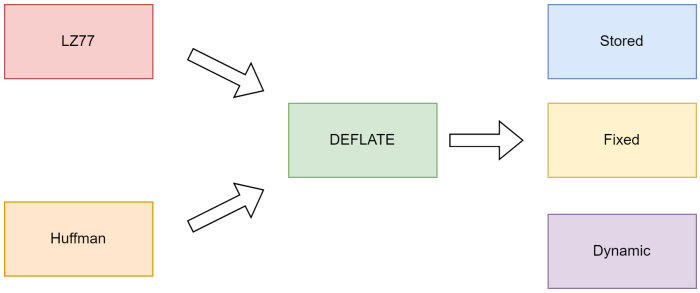

Figure 7. Relationship between LZ77, Huffman, DEFLATE and the different block types.

With the fundamental building blocks of LZ77, Huffman coding and DEFLATE now outlined and their relationships clarified, the discussion can now shift to zip quines.

2.2. Limited Zip Quine

As previously mentioned, Erling Ellingsen's methodology for the creation of zip quines has been lost over time, leaving Russ Cox as the sole reference point for quine creation. Cox's explanation relies on the possibility of constructing a quine solely using LZ77 tokens. Furthermore, Cox demonstrates that it is feasible to construct a quine with arbitrary header and footer bytes. Cox's work is detailed in Table 3.

Table 3. Final LZ77 quine by Russ Cox.

Code	Output
[P]	
Lp+1 [P] Lp+1	[P] Lp+1
Rp+1	[P] Lp+1
L1 Rp+1	Rp+1
L1 L1	L1
L4 Rp+1 L1 L1 L4	Rp+1 L1 L1 L4
R4	Rp+1 L1 L1 L4
L4 R4 L4 R4 L4	R4 L4 R4 L4
R4	R4 L4 R4 L4
L4 R4 L0 L0 Ls+1	R4 L0 L0 Ls+1
R4	R4 L0 L0 Ls+1
L0	
L0	
Ls+1 Rs+1 [S]	Rs+1 [S]
Rs+1	Rs+1 [S]
[S]	

Note: Text in red indicates the quine's header/footer and are not part of the encoded data. The text in blue highlights parts of the code that is straight output.

In Cox's quine, the notation *Lx* signifies that the subsequent *x* bytes are to be output directly without modification (literal tokens). To enhance the clarity, these directly output bytes are highlighted in blue. On the other hand, *Ry* denotes that the preceding *y* bytes in the output should be replicated (repeat tokens). This is shorthand for *repeat*(y, y), i.e., a token with distance *y* and length *y*. The header and footer are symbolized by $[P]$ and $[S]$, colored red, and have size *p* and *s*, respectively. Cox uses these to correspond with the Local File Header and the Central Directory of a zip file, as discussed regarding the zip format. The quine only works if it is assumed that each token is equal in size, meaning that the L and R tokens are equally large. The structure of Cox's zip quine is shown in Table 4.

Table 4. Structure of Cox's zip quine.

Local File Header Quine → [P]
Encoded LZ77 Quine in DEFLATE
Central Directory → [S]

Cox's implementation of the zip quine utilizes stored blocks for literal tokens and fixed blocks for repeat tokens. Dynamic blocks are thus not needed to create zip quines, as shown in Figure 8.

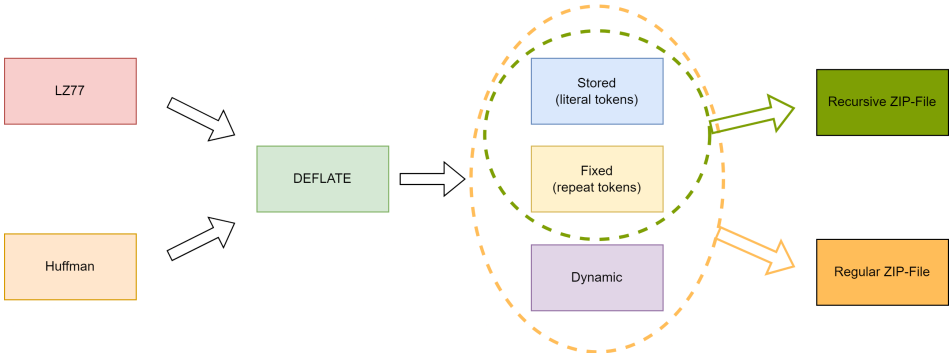

Figure 8. Relationship between LZ77, Huffman, DEFLATE and the different block types specific to quines.

It is important to note that Cox's quine is an LZ77 quine and thus might be able to be used in other file formats that use LZ77 or DEFLATE. However, translating the LZ77 quine to other formats may require some adjustment. For example, in Figure 3, there is a five-byte header in front of the raw data. This header corresponds with the meaning of a literal token. Cox thus uses a stored block to encode a token *Lx*. However, in DEFLATE, both repeat tokens and literal tokens must occupy the same size. Consequently, both types of tokens will consist of five bytes in a DEFLATE quine. This equivalence implies that any adjustment or increment mentioned in Cox's quine, such as the $+1$ indicated in Table 3, corresponds to a $+5$ adjustment in the DEFLATE context.

Limitations

A significant constraint arises from the manner in which the number of bytes following a stored block is encoded in two bytes. Consequently, a literal token (*Lx*) can only be followed by a maximum of 2^{16} bytes of data. This limitation imposes a constraint on the sizes of the header and footer, denoted as $[P]$ and $[S]$, respectively, which can maximally be $2^{16} - 5 = 65,531$ KiB. However, the utilization of a search buffer of 32 KiB for repeat tokens further diminishes this maximum, resulting in a more restrictive limit of $2^{15} - 5 = 32,763$ bytes. This is due to the necessity of tokens *Rp+1* and *Rs+1*. Additionally, the maximum length of a repeat token is 258 bytes. This limitation can be circumvented by utilizing multiple repeat tokens consecutively.

However, the most stringent limitation arises from the requirement that a repeat token must consist of five bytes. According to Cox, only repeat tokens *R9* to *R64* can fit within this constraint. Consequently, with the $+1$ adjustment mentioned in the table, this restricts the total size of $[P]$ and $[S]$ to $64 - 5 = 59$ bytes. This limitation effectively makes it impossible for Cox's quine to accommodate other files in the archive, as even the Local File Header and Central Directory of the quine itself barely fit within the allocated 59 bytes each.

Finally, both the Central Directory and Local File Header within a zip file include CRC-32 checksums, serving as a means to verify the integrity of the archived files [9]. However, in the case of a zip quine, where the file includes itself, modifying the CRC-32 checksum to match the value of the total zip file introduces a new challenge. This is because altering the CRC-32 checksum once again changes the value of the total zip file, creating a

recursive loop. To address this challenge, Cox employs a brute-force approach, attempting all possible CRC-32 values until a match is found. Despite this method, it is important to note that there exists the possibility that no such matching CRC-32 value can be found.

3. Results

The previous section reviewed the existing methodologies for recursive zip file creation. This section builds on Cox's foundational work to address the limitations of the current quine implementation. Furthermore, an analysis of the implications of these modifications will be conducted, highlighting the newly identified constraints.

3.1. Beyond the Limitations

The primary objective is to go beyond these limitations and enable the inclusion of additional files in the archive, akin to Ellingsen's achievement. The limits caused by the DEFLATE specification are not aspects that can be changed easily. While they are limits, they do not prohibit the ability to add extra files to the archive. Expanding *Rp+1* and *Rs+1* past the five-byte size limit is sufficient to add extra files in the archive.

Expanding the size of *Rs+1* is relatively straightforward. If *Rs+1* has a size *y*, then *Ls+1* should be adjusted to *Ls+y*, and, similarly, *Rs+1* should become *Rs+y*. In this case, *Rs+y* is not necessarily a single repeat token. It can also represent multiple repeat tokens following each other. However, this adjustment introduces a complication, namely that $[S]$ may begin in the middle of a byte. To mitigate this issue, a workaround can be implemented. By incorporating two *L0* tokens directly before $[S]$, $[S]$ is guaranteed to commence at the start of a byte. A single *L0* token is insufficient, as it would imply that the stored block is the final block and thus would begin with a 1 bit.

Addressing the size limit of *Rp+1* presents a somewhat greater challenge. Cox's quine relies on the ability for the repeat token to fit within five bytes. Specifically, the utilization of *L4* and *R4* tokens in the middle of the quine is critical to its functionality. If *Rp+1* were to equal a size *x*, the quine would no longer function as intended. Numerous attempts to modify the quine to accommodate the size *x* have proven unsuccessful.

A repeat token's size is fixed, but it can produce a larger output. For instance, a *R258* token is only 30 bits in size but produces an output of 258 bytes. Leveraging this principle, if *Rp+1* is *x* bytes, it can be compressed further. Through the iterative application of this principle in a loop, a repeat token that fits within five bytes can be achieved. However, a condition for this approach is that the repeat token already exists in the output. Table 5 provides an illustrative example of this principle.

Table 5. Example of rendering a repeat token smaller.

	Code	**Output**
	... R506 → size = 80 bits	...
	L10 R506 L5 L10 L25 R506 L5 L10 L15 R25 → size = 32 ≤ 80 bits	R506 L10 R506 L5 L10 L25 R506 L5 L10 L25
	L4 R25 L5 L4 L19 R25 L5 L4 L19 R19 → repeat	R25 L4 R25 L5 L4 L19 R25 L5 L4 L19
	...	...

Note: Text in blue highlights parts of the code that is straight output.

These solutions can now be applied to Cox's quine, resulting in the quine shown in Table 6. Moving forward, all quines will be standardized to the DEFLATE format. This entails replacing all instances of +1 in Cox's quine with +5 to ensure alignment with the

DEFLATE specifications. Additionally, other adjustments involve multiplying the token values by 5. For instance, *R4* would be altered to *R20*. The segment between two horizontal lines in Table 6 represents the portion that needs to be iteratively repeated until *Rx+15* can be compressed to fit within five bytes.

Table 6. Reworked quine in DEFLATE format.

Code	Output
[P]	
Lp+5 [P] Lp+5	[P] Lp+5
Rp+5 → size = x	[P] Lp+5
Lx Rp+5	Rp+5
L5 Lx	Lx
Lx+15 Rp+5 L5 Lx Lx+15	Rp+5 L5 Lx Lx+15
Rx+15 → until size = 5 bytes	Rp+5 L5 Lx Lx+15
L5 Rx+15	Rx+15
L5 L5	L5
L20 Rx+15 L5 L5 L20	Rx+15 L5 L5 L20
R20	Rx+15 L5 L5 L20
L20 R20 L20 R20 L20	R20 L20 R20 L20
R20	R20 L20 R20 L20
L20 R20 L0 L0 Ls+y+10	R20 L0 L0 Ls+y+10
R20	R20 L0 L0 Ls+y+10
L0	
L0	
Ls+y+10 Rs+y+10 L0 L0 [S]	Rs+y+10 L0 L0 [S]
Rs+y+10 → size = y	Rs+y+10 L0 L0 [S]
L0	
L0	
[S]	

Note: Text in blue highlights parts of the code that is straight output.

These changes now allow the addition of extra files inside the archive. The structure of a zip file is then as shown in Table 7.

Table 7. Structure of zip quine with extra files.

Local File Header 1	[P]
File Data 1	
...	
Local File Header n	
File Data n	
Local File Header Quine	
Encoded DEFLATE Quine	
Central Directory File Header 1	[S]
...	
Central Directory File Header n	
Central Directory File Header Quine	
End of Central Directory Record	

Note that the structure shown in Table 7 is not the only possible structure. The Local File Header of the zip file itself and the corresponding encoded data do not need to come after the other Local File Headers and data. An alternative structure is shown in Table 8.

Table 8. Alternative structure of a zip quine with extra files .

Local File Header 1	[P]
File Data 1	
Local File Header 2	
File Data 2	
...	
Local File Header m	
File Data m	
Local File Header Quine	
Encoded DEFLATE Quine	
Local File Header m+1	[S]
File Data m+1	
...	
Local File Header n	
File Data n	
Central Directory File Header 1	
Central Directory File Header 2	
...	
Central Directory File Header m	
Central Directory File Header Quine	
Central Directory File Header m+1	
...	
Central Directory File Header n	
End of Central Directory Record	

Limitations of the Adapted Quine

The standard quine proposed by Cox has been modified to allow for the expansion of the header and footer sections, denoted as [P] and [S]. This modification facilitates the inclusion of additional files within the zip archive, thereby enhancing the quine's functionality. However, it is essential to recognize that these additional files are subject to strict size limitations, which prevents them from being infinitely large.

A significant constraint arises from the manner in which the number of bytes following a stored block is encoded in two bytes. Consequently, a literal token (*Lx*) can only be followed by a maximum of 2^{16} bytes of data. This limitation imposes a constraint on the sizes of the header and footer, denoted as $[P]$ and $[S]$, respectively, which can maximally be $2^{16} - 5 = 65,531$ bytes. However, the utilization of a search buffer of 32 KiB for repeat tokens further diminishes this maximum, resulting in a more restrictive limit of $2^{15} - 5 = 32,763$ bytes. This is due to the necessity of tokens *Rp+1* and *Rs+1*.

Moreover, the maximum length of a repeat token is capped at 258 bytes, but it is possible to work around this limitation by using multiple repeat tokens consecutively.

The implications of these limitations are significant. They constrain the size of the embedded data, potentially limiting the quine's practical applications. As a result, while the adaptation allows for the inclusion of extra content, the effectiveness and utility of the quine remain constrained by these inherent limitations.

3.2. Loopy Zip Files

Now that it is possible to create standard zip quines with additional files, a new concept is explored: creating a zip file that contains another zip file, which in turn contains the first zip file. This concept, to the best of our knowledge, has not been previously implemented. However, analogous concepts have been explored in other programming languages. For example, there exists a Java program that outputs a C++ program [10], which outputs the original Java program [11]. In these examples, both the Java 8 and C++17 source codes are included within the program.

The source code that defines a zip file consists of the header $[P]$ and the footer $[S]$, as they include the encoded data and metadata of everything in the zip file. For a loopy zip file with a depth of two, the headers and footers $[P_1]$, $[P_2]$, $[S_1]$, and $[S_2]$ are required. Specifically, $[P_1]$ and $[S_1]$ define one zip file, while $[P_2]$ and $[S_2]$ define the other.

To achieve a loopy zip file, it is necessary to alternate the headers $[P_1]$ and $[P_2]$, as well as the footers $[S_1]$ and $[S_2]$. Table 9 illustrates this principle.

Table 9. Alternating $[P_1]$ and $[P_2]$.

	Code	**Output**
	[P1]	
	Lp2+5 [P2] Lp1+5	[P2] Lp1+5
	Lp1+5 [P1] Lp2+5	[P1] Lp2+5
size x {	R5	Lp1+5
	Rp1+p2+15,p2+5	[P2] Lp1+5
	Lx R5 Rp1+p2+15,p2+5	R5 Rp1+p2+15,p2+5
	...	...

Note: Text in blue highlights parts of the code that is straight output.

The repeat token $R_{p_1+p_2+15,p_2+5}$ has a length determined solely by the size of $[P_2]$. Therefore, $[P_1]$ and $[P_2]$ must be equal in size. This requirement is manageable because the Local File Header and Central Directory File Headers contain an Extra Field section, which can accommodate up to 64 KiB. By utilizing this field, $[P_1]$ and $[P_2]$ can be adjusted to be of equal size. The same approach applies to $[S_1]$ and $[S_2]$.

Alternating the footers is a challenge that cannot be solved using the same approach. Alternating the headers works because the initial header does not belong to the encoded data of the quine itself. It only includes the data of other files in the archive and the Local File Header of the zip file. However, once the middle part of the quine is reached, using *L20* and *R20*, all data within the literal token occupy the same location in the output. This makes it impossible to alternate $[S_1]$ and $[S_2]$ within this segment.

Some creativity leads to a solution: what if $[S_1]$ and $[S_2]$ are placed immediately after $[P_1]$ and $[P_2]$ at the beginning? This arrangement means that the first footer to be output does not necessarily belong to the essential data required by unzip programs. It is found that bytes following the last block of encoded data and preceding the next Local File Header are ignored. The advantage of this approach is that the footers can be alternated along with the headers. To ensure that the footer is also output at the end of the zip file, a repeat token can be used to replicate the footer from the beginning. Both solutions to handle the alternation of headers and footers are illustrated in Table 10.

Table 10. Final quine for loopy zip files.

	Code	Output
Rz with size x	[P1]	
	[S1]	
	Lp2+s2+5 [P2] [S2] Lp1+s1+5	[P2] [S2] Lp1+s1+5
	Lp1+s1+5 [P1] [S1] Lp2+s2+5	[P1] [S1] Lp2+s2+5
	R5	Lp1+s1+5
	Rp1+p2+s1+s2+15,p2+s2+5	[P2] [S2] Lp1+s1+5
	Lx Rz	Rz
	L5 Lx	Lx
	Lx+15 Rz L5 Lx Lx+15	Rz L5 Lx Lx+15
	Rx+15 → until size = 5 bytes	Rz L5 Lx Lx+15
	L5 Rx+15	Rx+15
	L5 L5	L5
	L20 Rx+15 L5 L5 L20	Rx+15 L5 L5 L20
	R20	Rx+15 L5 L5 L20
	L20 R20 L20 R20 L20	R20 L20 R20 L20
	R20	R20 L20 R20 L20
	L20 R20 L0 L0 Ly+w+4	R20 L0 L0 Ly+w+4
	R20	R20 L0 L0 Ly+w+4
	L0	
	L0	
	Ly+w+4 00 00 00 00 Rw Ry	00 00 00 00 Rw Ry
size w	Rw	Rw Ry
	Ry → repeats earlier [S2]	[S2]
	[S1]	

Note: Text in blue highlights parts of the code that is straight output. The text in orange are reserved (see Section 3.2.1).

3.2.1. CRC-32

One should note the four 0 bytes in orange. These provide the solution to the final problem that needs to be addressed. Each zip file in our loop has its own CRC-32 checksum in the header and footer. Changing one checksum also changes the other, and attempting all possibilities would mean testing $2^{32} \times 2^{32}$ options, which is unfeasible.

However, it is found that adding a CRC-32 value to the end of the data used to compute the checksum results in a checksum of a constant value [12,13]. This can be exploited to reset the CRC-32 value from this point onward, as demonstrated in Listing 1.

Listing 1. Reset CRC-32.

```
val input = inputFile.readBytes()
// CRC-32: C6 39 74 44
val crc = CRC32(input)
val reset = intToBytes(crc.inv())
// CRC-32: FF FF FF FF
val newCrc = CRC32(input + reset)
```

The bytes highlighted in yellow act as the reset bytes. By strategically placing these bytes, it becomes possible to calculate the CRC-32 value of one zip file based on everything in following the reset bytes. This enables the use of the pseudocode outlined in Listing 2. With this approach, only 2^{32} possibilities need to be tested, as the CRC-32 value for the other zip file can be calculated based on the reset point.

Listing 2. Calculate CRC-32 checksums.

```
for crc in all_crc:
  Fill crc in [P1] and [S1] (all places)
  Calculate CRC of bytes after reset bytes
  Fill this in [P2] and [S2] (all places)
  Calculate CRC of bytes before the reset
  Use this CRC to set the reset bytes
  if crc == CRC32(zip2):
    print("Solution found!")
```

With this final aspect resolved, the creation of the first-ever loopy zip file has been achieved, to the best of our knowledge.

3.2.2. Limitations of Loopy Zip Files

The concept of loopy zip files, introduced in Section 3.2, represents a novel and complex advancement in the realm of zip quines. To date, no such structures have been documented, marking this as a significant development in the field. However, despite this innovation, it is crucial to recognize that loopy zip files come with their own set of limitations.

One of the most stringent constraints involves the maximum distance of a repeat token, which is capped at 32 KiB. In the context of loopy zip files, the repeat token with the greatest distance can be expressed mathematically as $p1 + p2 + s1 + s2 + 15$, where $p1$ and $p2$ correspond to the sizes of the headers for each zip file, and $s1$ and $s2$ denote the sizes of the footers. This configuration highlights how the interdependencies between different components of the archive affect its overall structure.

To better understand this limitation, it can also be expressed as $2x + 15$, where x represents the size of $[P_1][S_1]$ and $[P_2][S_2]$. Consequently, both $[P_1][S_1]$ and $[P_2][S_2]$ must adhere to a maximum size of $(2^{15} - 15)/2 = 16,376.5 \approx 16,376$ bytes each. This limitation significantly restricts the number of data that can be contained within each zip file in the loop, which may limit their practical applications.

3.3. The Generator

The zip quine generator discussed in this paper is a practical implementation of the technical concepts outlined earlier, fully written in Kotlin. It serves as a tool that allows users to experiment with recursive zip files, providing a framework for the creation of zip quines with the ability to include additional files within the archive, while adhering to the limitations previously discussed.

In contrast to earlier work, where Cox and Ellingsen created only one working example each, this generator represents a significant advancement. Cox did not include extra files in his implementation, while Ellingsen succeeded in adding a single additional file but the methodology that he used was lost. The generator and methods presented here overcome these hurdles, offering a more flexible and accessible approach to recursive zip file creation, enabling further experimentation within the constraints identified in the paper.

Additionally, the generator supports the creation of looped zip files. This capability has never been achieved before and represents a novel advancement in the field, expanding the possibilities of recursive file compression.

The general workflow of the generator is illustrated in Figure 9. This diagram provides a visual overview of the process, demonstrating the steps that the generator takes to create recursive zip files.

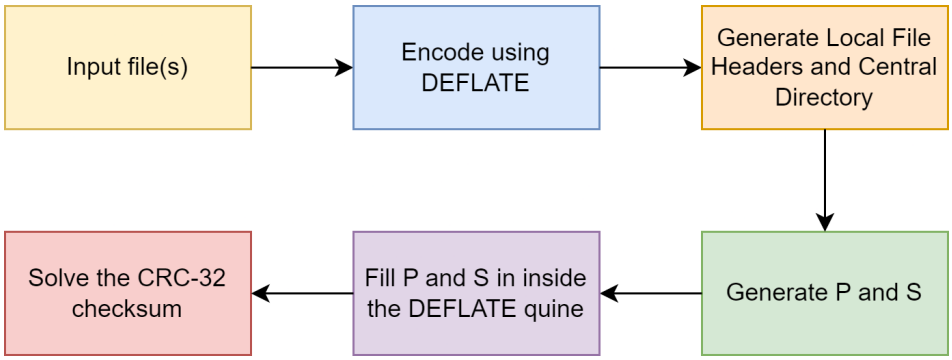

Figure 9. General workflow of the generator.

4. Discussion

The research presented in this paper offers a comprehensive exploration of the creation and encoding of zip quines and looped zip files.

4.1. Advancements

The primary technical advancement achieved in this work lies in the ability to incorporate more data into a recursive zip file than was previously possible. Cox's quine, which was limited by constraints on the repeat token lengths and buffer sizes, could not fit anything beyond the Local File Header and Central Directory. By addressing this limitation, the current study pushes the boundaries of what zip quines can achieve, particularly by carefully managing DEFLATE's stored and fixed blocks. The generator introduced in this research enables the creation of recursive zip files that can include additional files up to 32,763 bytes in size, a clear improvement over the 59-byte limitation faced by Cox's implementation.

Ellingsen's work, which managed to include one additional file, was a step forward, but, without his methodology, the field was without a consistent process for the expansion of zip quines. The generator presented in this study provides a reliable and replicable method for the creation of recursive zip quines with additional files, thus filling this gap.

In addition to extending the work on zip quines, this research introduces a new concept: loopy zip files. These files create an infinite recursive structure where a zip file contains another zip file, which in turn references the original file, forming a continuous loop. To the best of our knowledge, this type of quine has not been previously achieved, marking a novel advancement in the field.

4.2. Applications

The ability to experiment with these recursive structures under the discussed limitations allows for a deeper exploration of file compression and recursion, opening new avenues for practical application and theoretical research.

The broader implications of this research encompass several areas, including file compression, software robustness, and cybersecurity. Recursive zip files, particularly the looped zip files introduced in this study, represent a new category of archival structures with unique characteristics. The possibility of creating an infinite loop within a zip file raises intriguing questions about file handling, extraction processes, and potential vulnerabilities in software designed to manage compressed archives.

From a practical standpoint, recursive zip quines can be used as a tool for testing. This is especially relevant in the context of security testing, where recursive structures can serve as stress tests for the detection of weaknesses in file extraction software. Recursive archives may expose vulnerabilities that could be exploited in denial-of-service (DoS) attacks, where the software becomes trapped in an infinite loop during the extraction process. This mirrors the threat posed by zip bombs, which have been studied in depth in the security literature [14,15]. While zip bombs rely on extreme compression ratios to crash systems, looped zip files create infinite recursion, potentially leading to similar outcomes. The use

of recursive zip files as a means to test and improve the robustness of software handling compressed archives represents a valuable contribution to the field of cybersecurity.

4.3. Future Research

As previously discussed, the presence of CRC-32 checksums in zip quines introduces challenges because modifying the quine's content to reflect the correct checksum creates a recursive problem, as the checksum itself becomes part of the content. Cox's brute-force approach to finding a matching CRC-32 value highlights this challenge, but future work could explore more efficient algorithms to handle CRC calculations in recursive file structures. The recursive nature of zip quines makes them an interesting case study for an understanding of the limits of CRC-based integrity checks and opens the door for the further exploration of advanced validation techniques [16].

While this paper focused on stored and fixed blocks in DEFLATE compression, future work could explore the use of dynamic blocks. While dynamic blocks are not necessary for the quine itself, they may offer opportunities for further optimization, particularly in terms of the compression efficiency. Investigating how dynamic Huffman coding could be applied to recursive archives may yield new methods for the management of more complex or larger zip quines.

Another area for future research lies in the further exploration of looped zip files. While this study presented the first known implementation of looped recursive zip files, their practical applications and limitations warrant further investigation. In particular, it would be valuable to explore methods of safely extracting and analyzing looped zip files without triggering infinite recursion or crashing decompression software. Techniques for the detection and mitigation of the risks associated with recursive loops in compressed archives could be a critical area of focus.

Finally, the looped zip files presented in this article consist of a loop between two zip files. Further research is needed to investigate the possibility of creating even deeper loops, involving three or more zip files within the recursive structure. In general, any directed graph might be interpreted as a zip quine, with nodes corresponding to files and edges reflecting which zip files are contained in which. Such exploration could reveal new limitations, challenges, and potential applications.

Author Contributions: Conceptualization, R.V.M.; methodology, R.V.M.; software, R.V.M.; formal analysis, R.V.M.; investigation, R.V.M.; writing—original draft preparation, R.V.M.; writing—review and editing, R.V.M. and P.A.; project administration, R.V.M.; funding acquisition, P.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Ghent University, grant number BOF/STA/202009/039, and IMEC.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The code is available in the publicly accessible repository https://github.com/ruvmello/zip-quine-generator, accessed on 15 October 2024.

Acknowledgments: We would like to thank Ghent University and IMEC for supporting this work. We also thank A.V., L.A., M.C., L.A., Th.A., K.A. and H.B.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CRC	Cyclic Redundancy Check
CD	Central Directory
CDFH	Central Directory File Header
EOCDR	End of Central Directory Record

Appendix A

A zip file consists of several key structures that enable the compression and storage of files within a single archive. Below is an overview of the primary components: the Local File Header, Central Directory File Header, and End of Central Directory Record.

Appendix A.1. Local File Header

The structure of the Local File Header can be found in Table A1.

Table A1. Structure of the Local File Header.

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
Signature				Version		Flags		Method		Mod Time		Mod Date		CRC-32	
CRC-32		Compressed Size				Uncompressed Size				Name Len.		Extra Len.		Name	
Name (variable length)															
Extra Fields (variable length)															

Signature: A 4-byte signature that indicates that the file is a Local File Header. This is represented by the bytes '0x504b0304'.

Version: The version of the compression format used for the file. In the implementation of this work, the bytes '0x1400' are used, which correspond to the decimal value 20 (in little endian), representing version 2.0 of the zip format.

Flags: A series of bits that indicate various properties of the file, such as whether it is encrypted. For the creation of recursive zip files, this is not necessary, and two zero bytes are used.

Method: The compression method used for the file. The zip format supports various methods, such as store (no compression) or DEFLATE (standard compression for ZIP files). In this work, the standard algorithm used for zip archives is applied, as not every method is suitable for creating a quine. This method is indicated by '0x0800'.

Modification Time: The hour, minute, and second of the last modification to the file. The first 5 bits represent the hour, followed by 6 bits for the minutes, and the last 5 bits for the seconds. According to the specification, the seconds must be divided by two before encoding and multiplied by two during decoding.
Example: '0x7d1c' = 0111110100011100

- Hour: (01111)10100011100 = 15
- Minutes: 01111(101000)11100 = 40
- Seconds: 01111101000(11100) = 28 $\Rightarrow$ 56 s

Modification Date: The day, month, and year of the last modification to the file. The first 7 bits are used to represent the year, counting from 1980. Therefore, the year 2024 is represented as 44. The next 4 bits are used for the month, and the last 5 bits are for the day.
Example: '0x5871' = 0101100001110001

- Year: (0101100)001110001 = 44 $\Rightarrow$ 2024
- Month: 0101100(0011)10001 = 3
- Day: 01011000011(10001) = 17

CRC-32: A 4-byte cyclic redundancy check (CRC) value used to verify the integrity of the file [9]. The checksum is calculated before compression. When the file is later decompressed, the checksum can be recalculated. By comparing the two, the integrity of the file can be verified.

Compressed Size: The size of the compressed file, represented in 4 bytes. This indicates the file's size after compression by the chosen method.

Uncompressed Size: The size of the uncompressed file, represented in 4 bytes. This indicates the file's size before compression. When the file is decompressed, its size should match this value.

Name Length: The size of the file name in bytes, which corresponds to the number of characters in the file name, including the extension.

Extra Length: The size of any additional fields appended to the file.

Name: The actual file name, with a length corresponding to the previously stored value.

Extra Fields: Any additional fields, with a length corresponding to the previously stored value.

Appendix A.2. Central Directory File Header

The structure of the Central Directory File Header can be found in Table A2. Only the components in bold are different from the Local File Header. Therefore, only these will be discussed.

Table A2. Structure of the Central Directory File Header.

0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**	**0x08**	**0x09**	**0x0A**	**0x0B**	**0x0C**	**0x0D**	**0x0E**	**0x0F**
Signature				Version		**Min. Version**		Flags		Method		Mod Time		Mod Date	
CRC-32				Compressed Size				Uncompressed Size				Name Len.		Extra Len.	
Comm. Len.		**Disk #**[1]		**Int. Attr.**		**External Attr.**				**Local File Header Offset**				Name	
Name (variable length)															
Extra Fields (variable length)															
Comment (variable length)															

[1]#: number

Signature: A 4-byte signature indicating that the file is a Central Directory File Header. This is represented by the bytes '0x504b0102'.

Minimum Version: The minimum version of the software required to decompress the zip file. The value of this field usually matches the version specified in the Local File Header.

Comment Length: The length of the comment added to the compressed file.

Disk Number: The number of the disk on which the file is located. This field originates from the time when zip files were distributed across multiple physical disks. Larger files could not be stored on a single floppy disk, so the zip specification provided a solution by allowing a zip archive to be split across multiple disks. This disk number indicates on which disk the Local File Header of the encoded file is located.

Internal Attributes: Internal attributes of the file. These are rarely used and will be represented by four zero bytes.

External Attributes: External attributes of the file. These are also rarely used and will be represented by four zero bytes.

Local File Header Offset: The position (offset) of the corresponding Local File Header within the zip archive. This indicates where the Local File Header begins, followed by the compressed data.

Comment: An optional comment or note for the file, with a length equal to the previously specified value.

Appendix A.3. End of Central Directory Record

The structure of the End of Central Directory Record can be found in Table A3.

Table A3. Structure of the End of Central Directory Record.

0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**	**0x08**	**0x09**	**0x0A**	**0x0B**	**0x0C**	**0x0D**	**0x0E**	**0x0F**
Signature				Disk #[1]		Disk # CD		Disk Entries		Total Entries		Central Directory Size			
Offset of CD				Comm. Len		Zip file comment (variable length)									

[1]#: number

Signature: A 4-byte signature indicating that the file is an End of Central Directory Record (EOCDR). This is represented by the bytes '0x504b0506'.

Disk Number: The number of the disk on which the EOCDR is located.

Disk Number CD: The number of the disk where the Central Directory File Headers (CDFHs) begin.

Disk Entries: The number of CDFHs on the disk where the EOCDR is located.

Total Entries: The total number of CDFHs across all disks.

Central Directory Size: The size of the Central Directory (CD) in bytes.

Central Directory Offset: The position where the Central Directory begins. For a zip file spanning multiple disks, this indicates the position on the disk where the Central Directory starts.

Comment Length: The length of the comment associated with the zip file.

Zip File Comment: The comment itself, with a length equal to the previously specified value.

References

1. Hofstadter, D.R. *Gödel, Escher, Bach: An Eternal Golden Braid*; Basic Books: New York, NY, USA, 1980.
2. Cox, R. Zip Files All The Way Down. Available online: https://research.swtch.com/zip (accessed on 1 September 2024).
3. Ellingsen, E. Zip Quine. Available online: https://alf.nu/ZipQuine (accessed on 1 September 2024).
4. .ZIP File Format Specification. Available online: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT (accessed on 1 September 2024).
5. Deutsch, L.P. DEFLATE Compressed Data Format Specification version 1.3. *Req. Comments* **1996**, RFC 1951. . [CrossRef]
6. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343. [CrossRef]
7. Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE* **1952**, *40*, 1098–1101. [CrossRef]
8. Storer, J.A.; Szymanski, T.G. Data Compression via Textual Substitution. *J. ACM* **1982**, *29*, 928–951. [CrossRef]
9. Peterson, W.W.; Brown, D.T. Cyclic Codes for Error Detection. *Proc. IRE* **1961**, *49*, 228–235. [CrossRef]
10. Quine: Java-C++. Available online: https://web.stanford.edu/class/cs208e/cgi-bin/main.cgi/static/lectures/18-ReflectionsOnTrustingTrust/code/Quine.java (accessed on 1 September 2024).
11. Quine: C++-Java. Available online: https://web.stanford.edu/class/cs208e/cgi-bin/main.cgi/static/lectures/18-ReflectionsOnTrustingTrust/code/Quine.cpp (accessed on 1 September 2024).
12. Stigge, M.; Plötz, H.; Müller, W.; Redlich, J.-P. Reversing CRC: Theory and Practice. Available online: https://api.semanticscholar.org/CorpusID:17886305 (accessed on 1 September 2024).
13. Adler, M. Append calculated CRC to data and recalculate does not yield zero. *Electr. Eng. Stack Exch.* **2019**. Available online: https://electronics.stackexchange.com/questions/450297/append-calculated-crc-to-data-and-recalculate-does-not-yield-zero (accessed on 1 September 2024).
14. Canet, M.; Kumar, A.; Lauradoux, C.; Rakotomanga, M.-A.; Safavi-Naini, R. Decompression Quines and Anti-Viruses. In Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy (CODASPY '17), New York, NY, USA, 22–24 March 2017; pp. 23–34. [CrossRef]
15. Wani, M.A.; AlZahrani, A.; Bhat, W.A. File System Anti-Forensics—Types, Techniques and Tools. *Comput. Fraud. Secur.* **2020**, *2020*, 14–19. [CrossRef]
16. Maxwell, B.; Thompson, D.R.; Amerson, G.; Johnson, L. Analysis of CRC Methods and Potential Data Integrity Exploits. In Proceedings of the International Conference on Emerging Technologies, Minneapolis, MN, USA, 25–26 August 2003; pp. 25–26.