Exploiting Multiple Similarity Spaces for Efficient and Flexible Incremental Update of Mobile Apps

Lewei Jin, Wei Dong*, Bowen Jiang, Tong Sun, Yi Gao*
College of Computer Science and Technology, Zhejiang University, China
Email: {jinlw, dongw, jiangbw, tongsun, gaoyi}@zju.edu.cn

***Abstract*—Mobile application updates occur frequently, and they continue to add considerable traffic over the Internet. Differencing algorithms, which compute a small delta between the new version and the old version, are often employed to reduce the update overhead. Transforming the old and new files into the decoded similarity spaces can drastically reduce the delta size. However, this transformation is often hindered by two practical reasons: (1) insufficient decoding (2) long recompression time. To address this challenge, we have proposed two general approaches to transforming the compressed files (more specifically, deflate stream) into the *full* decoded similarity space and *partial* decoded similarity space, with low recompression time. The first approach uses recompression-aware searching mechanism, based on a general full decoding tool to transform deflate stream to the full decoded similarity space with a configurable searching complexity, even when it cannot be recompressed identically. The second approach uses a novel solution to transform a deflate stream into the partial decoded similarity space with differencing-friendly LZ77 token reencoding. We have also proposed an algorithm called MDiffPatch to exploit the full and partial decoded similarity spaces. The algorithm can well balance compression ratio and recompression time by exposing a tunable parameter. Extensive evaluation results show that MDiffPatch achieves lower compression ratio than state-of-the-art algorithms and its tunable parameter allows us to achieve a good tradeoff between compression ratio and recompression time.**

***Index Terms*—Incremental update, differencing algorithm, compression**

I. Introduction

The increasing number of mobile applications and users has resulted in the significant growth of application downloads and updates in markets. According to Statista's report, the number of worldwide mobile application downloads is expected to reach 299 billion by 2023 [32]. To keep up with frequent updates and fixes [12], [14] in applications, leading App market operators, such as Google Play [10], Huawei AppGallery [13] and Galaxy-store [28], have to spend billions of dollars on application updates every year, e.g., pay for the server bandwidth or the CDN services.

Incremental update is a widely-used solution to reduce the data traffic during application updates [11], [20], [22], [33]. It computes the delta between two versions of the application and only transmit the delta during the update. Researchers have proposed many differencing algorithms for computing small delta files over the years [2]–[4], [6], [7], [15], [19], [21], [34]. More sophisticated differencing algorithms have appeared in recent years, e.g., xdelta3 [18], bsdiff [26], HDiffPatch [31], archive-patcher [9], Delta++ [29]. They are generally more suitable for mobile application updates.

Some algorithms focus on how to compute a compact delta given two application files (i.e., APK files), e.g., xdelta3 [18], bsdiff [26], HDiffPatch [31]. These algorithms typically perform difference computation in the *original* similarity space, i.e., the old and new apk files are not changed before difference computation. However, most of binary code and resource files in an APK are independently compressed with various versions of the deflate algorithm [5]. Compression reduces the size of an individual file, but can also damage the similarity between two versions of files. For example, it is possible that two similar files would become quite different after compression. To address this issue, some more advanced algorithms, e.g., Delta++ [29] and archive-patcher [9], utilize the *decompression-before-differencing* technique to drastically reduce the delta size, i.e., they first try to transform the compressed files into the decoded similarity space before difference computation.

However, decompression before differencing is not always possible, resulting in *insufficient* decompression in state-of-the-art algorithms. Note that the reconstructed new app file must remain identical to the original file [35], otherwise it cannot pass the integrity check imposed by the Android installer. By investigating 200 popular apps in the Huawei AppGallery [13], we have found that an average of 24% of bytes in the APK cannot be decompressed with the zlib [1] tool used in archive-patcher, otherwise they cannot be recompressed identically at the mobile side.

Incorporating more versions of deflate algorithm (e.g., 7zip, libdeflate) is one possible solution. However, it faces two severe challenges. First, it is impossible to incorporate all deflate algorithms, especially for those implemented by commercial companies which are not open source [27]. Even it is possible to incorporate many popular deflate algorithms, it would incur excessive large overhead (at the server side) for enumerating all the algorithms and their compression parameters [25]. Second, it may cause huge recompression time at the mobile side. For example, 7zip's recompression time is almost 5x compared with zlib. This is unacceptable as it significantly impairs mobile user's QoE [30].

This work is supported by the National Natural Science Foundation of China under Grant No. 62072396 and 62272407, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under grant No. 2023C01033, and the National Youth Talent Support Program. Wei Dong and Yi Gao are the corresponding authors.

To address the above challenges, we have proposed two general approaches to transforming the deflate stream into the decoded similarity space regardless of the specific version of deflate algorithm, even when it cannot identically recompressed.

- Our first approach tries to transform a deflate stream into the full decoded similarity space. It is required that the recompressed file is similar to the original deflate stream so that a small patch could make it identical to the original deflate stream. Our first approach builds on top of precomp which is an existing tool to decompress the streams using an enhanced zlib algorithm, allowing "re-"compression and reconstruction using a very small patch so that they are bit-to-bit-identical with the original stream. We have proposed recompression-aware searching mechanism with a configurable searching complexity in order to reduce the recompression time at the mobile side.
- Our second approach tries to transform a deflate stream into the partial decoded similarity space. Note that all deflate algorithm first compress the file using the LZ77 algorithm, and then further compress the LZ77 stream using Huffman encoding. While LZ77 performs at the byte level while Huffman encoding performs at the bit level. Transforming the deflate stream into partial decoded space (i.e., LZ77 stream) can also result in significantly larger similarity compared with performing difference computation in the original (e.g., compressed) similarity spaces, since all existing differencing algorithm operates at the byte level.

Generally speaking, computing at the full decoded similarity space results in a small delta but result in additional overhead of the patch and longer recompression time. Computing at the partial decoded similarity space will typically generate a larger delta but incurs a very small recompression time. To strike a reasonable balance between the compression ratio and the reconstruction time, we have devised a flexible algorithm with a tunable parameter α, based on the above mentioned two approaches to exploit both the full decoded similarity space and the partially decoded similarity space for difference computation. Specially, When $\alpha = 1$, the algorithm compares all deflate bytes in at full decoded similarity space, for the highest level of compression. When $\alpha = 0$, the algorithm compares all deflate bytes at partial decoded similarity space for the fastest recompression time at the mobile side.

We implement our algorithm, MDiffPatch, for differencing (at the server side) and reconstruction (at the mobile side) by incorporating the novel techniques and algorithms we have proposed. Extensive evaluation results show that (1) MDiffPatch achieves lower compression ratio than state-of-the-art algorithms. In its most aggressive version (with $\alpha = 1$), MDiffPatch achieves an absolute reduction of 20.61% and relative reduction of 47.01% compared with HDiffPatch and an absolute reduction of 10.91% and relative reduction of 31.96% compared with archive-patcher. (2) its tunable parameter allows us to achieve a good tradeoff between compression ratio and recompression time.

The contributions of this paper are summarized as follows:

- We have proposed two general approaches to transforming the deflate stream into the *full* decoded similarity space and *partially* decoded similarity space, with low recompression time. The first approach uses recompression-aware searching mechanism based on a general full decoding tool while the second approach uses a novel solution to transform a deflate stream into the partial decoded similarity space with differencing-friendly LZ77 token reencoding.
- We have proposed an algorithm called MDiffPatch to exploit both the full and partial decoded similarity spaces. The algorithm can well balance compression ratio and recompression time by exposing a tunable parameter.
- Extensive evaluation results show that MDiffPatch achieves lower compression ratio than state-of-the-art algorithms and its tunable parameter allows us to achieve a good tradeoff between compression ratio and recompression time.

II. Background

A. APK file format

The APK file format encapsulates the entire content of an Android application, including its code (.dex files), libraries (.so files), and resources (e.g., .png, .jpg and many others), etc. The APK format is basically a compressed archive that adheres to the widely used ZIP file format. Some files in the APK files are compressed, e.g., using different versions of deflate algorithms with different compression parameters. Some files in the APK files are not compressed (i.e., in store mode).

During the installation process, the Android operating system verifies the APK's signature against the embedded public key. If the signature is valid and matches the public key, application is allow to be installed on device. In order to ensure successful installation, it is required that the differencing algorithm should be able to reconstruct the original new version of APK file to ensure the correctness of the signature.

B. Deflate algorithm

Deflate is a lossless data compression algorithm that uses a combination of LZ77 and Huffman coding. Fig. 1 illustrates the overall process of the deflate algorithm (e.g., in zlib) that performs the LZ77 algorithm (①-⑤) at the first stage and performs Huffman encoding (⑥-⑧) at the second stage, with the input bytes stream "BACACACBACA".

LZ77 encoding: LZ77 tokens can be divided into two classes and both can be represented by a three-byte tuple (x, y, z):

- A literal (LIT) token: This is similar to the add instruction in the delta file. For a literal token, x=y=0, and z specifies the character which should be present in the uncompressed file. We also use LIT<'z'> to represent this token.

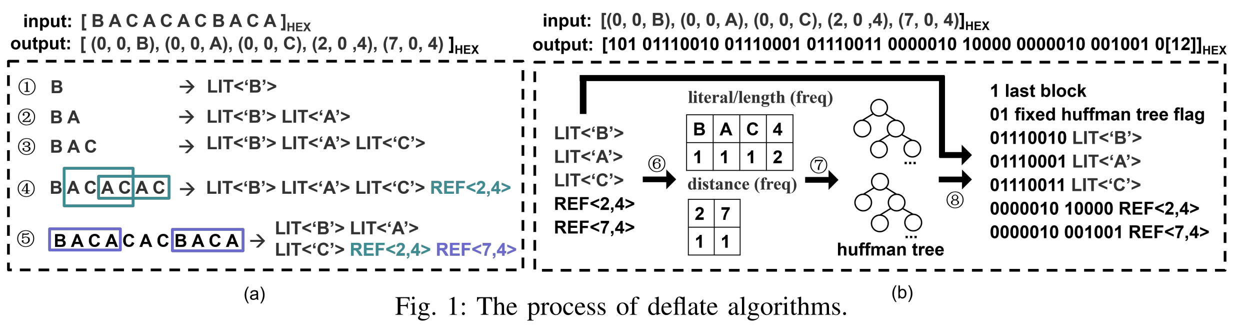

input: [B A C A C A C B A C A]HEX
output: [(0, 0, B), (0, 0, A), (0, 0, C), (2, 0 ,4), (7, 0, 4)]HEX
① B → LIT<'B'>
② B A → LIT<'B'> LIT<'A'>
③ B A C → LIT<'B'> LIT<'A'> LIT<'C'>
④ B A C A C A C → LIT<'B'> LIT<'A'> LIT<'C'> REF<2,4>
⑤ B A C A C A C B A C A → LIT<'B'> LIT<'A'> LIT<'C'> REF<2,4> REF<7,4>
(a)
input: [(0, 0, B), (0, 0, A), (0, 0, C), (2, 0 ,4), (7, 0, 4)]HEX
output: [101 01110010 01110001 01110011 0000010 10000 0000010 001001 0[12]]HEX
LIT<'B'> LIT<'A'> LIT<'C'> REF<2,4> REF<7,4>
⑥ ⑦ ⑧
literal/length (freq)
B A C 4
1 1 1 2
distance (freq)
2 7
1 1
huffman tree
1 last block
01 fixed huffman tree flag
01110010 LIT<'B'>
01110001 LIT<'A'>
01110011 LIT<'C'>
0000010 10000 REF<2,4>
0000010 001001 REF<7,4>
(b)

Fig. 1: The process of deflate algorithms.

- A reference (REF) token: This is similar to the copy instruction in the delta file. For a reference token, z specifies length of the repeated sequence, x and y respectively specifies the lower and higher 8 bits of the distance from the current position back to the start of the repeated sequence, limited to the 32kB window size. We also use REF<distance, length> to represent this token. For example, REF<2,4> in step 4 in Fig. 1 means the reference position is 2 prior to the current position and the length is 4 bytes.

It is worth noting that different deflate algorithms and different compression parameters can result in different mechanisms in how to generate a REF token.

Huffman encoding: First, the number of occurrences for the values in the LIT and REF tokens are counted ⑥. Second, two Huffman trees ⑦ are generated, i.e., one code tree for literals and lengths and a separate code tree for distances. Finally, the deflate stream ⑧ is produced, consisting of Huffman tree code and the values of token encoded by Huffman code. Note that the Huffman encoding process is largely the same for almost all deflate algorithms.

C. Precomp

Precomp is an existing tool to decompress deflate streams to the full decoded similarity space and reconstruct them using an *enhanced* zlib algorithm, allowing reconstruction using a very small metadata so that they are *bit-to-bit-identical* with the original stream. In the decoding phase, precomp tries to reencode the uncompressed stream using an enhanced zlib algorithm. The difference between the original deflate stream and the reencoded stream are encoded as metadata which is stored along with the decoded stream. In the recompression phase, precomp is able to generate the original deflate stream with the help of the metadata. While the size of the metadata is kept small for precomp by using various optimization techniques, the recompression time at the mobile side could be prohibitively large.

A key dominating factor for the large recompression time lies in generating the REF token for the LZ77 stream. As explained earlier, at the first stage of the deflate algorithm, the LZ77 algorithm will identify the repeated segment in the byte stream and encode it into a REF token. Precomp dynamically builds up a hash table in order to efficiently identify the most similar REF token for the LZ77 stream. When building the hash table, the hash value of each three-consecutive characters

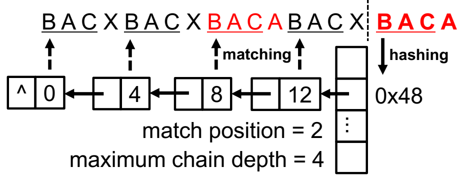


Fig. 2: Example of hash match in precomp.

is calculated. If the hash values for multiple three-character segments are the same, they are chained using a list with each element denoting the positions of the first character. Fig. 2 shows such an example. The hash value for "BAC" is 0x48. There are four appearances of "BAC" preceding the current position and the corresponding positions (i.e., 0,4,8,12) are stored in the element of list. The hash table is used to compactly encode the current characters starting from the current position (denoted using dashed line). To encode the last 4 bytes, precomp first calculates the hash value of "BAC", and obtains the hash value 0x48. Then precomp searches through the corresponding list, and find the position from which common segments with the longest length can be identified. The string "BACA" at position 8 is considered as *the best match* for the unencoded last 4 bytes of "BACA". A REF<8, 4> token will be generated to represent the last 4 bytes, where 8 and 4 means repeating a 4-byte segment at a distance of 8. Note that precomp individually uses a compression strategy. precomp may generate different REF tokens and it encodes these differences in the metadata so that the decompressed stream can be recompressed identically.

In precomp, maxchainDepth is a key parameter to limit the maximum list length for finding a match for a given hash value. This value will have an impact on both the search time complexity and the optimality of match. For the example shown in Fig. 2, when the maxchainDepth is set to 4, a total of 4 searches will be performed, and the best match the depth of 2 will be found. However, when the maxchainDepth is set to 1, the suboptimal match at the depth of 1 will be identified since only one search will be performed. By default, this value is set to the maximum possible depth in precomp.

III. Exploiting the Full Decoded Similarity Spaces

While it is easy to decompress a deflate stream into the fully decoded space, it is difficult to recompress the decoded byte stream into the original deflate stream. This is because different versions of deflate algorithms and different com-

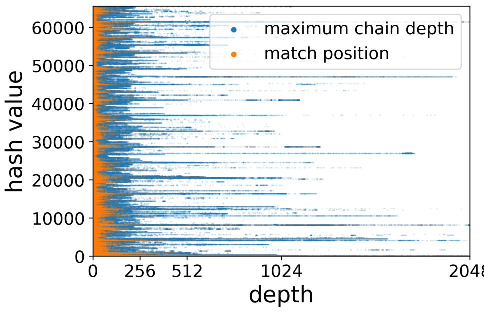


Fig. 3: The maximum chain depths and the match positions for a typical APK file.

pression parameters can be used for compression. Subjected to the specification of deflate stream [5], this information will not be stored. The existing approach—archive-patcher can decode and recompressed it *identically* by enumerating all possible compression parameters of zlib. For the deflate bytes compressed using other popular algorithms (e.g., 7zip, libdeflate, and etc), archive-patcher cannot transform them into the full decoded space.

To address the above issue, we have proposed an approach based on precomp which we have described in detail in Section II-C. As we have mentioned in Section II-C, the maxchainDepth parameter in precomp has a large impact on both the search time complexity and the optimality of match. We have also empirically find that such a default value would cause large searching time during recompression and a non-negligible portion of searching time is unnecessary.

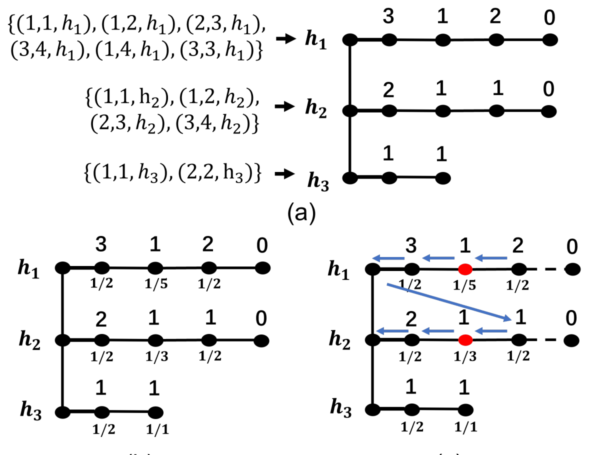
Fig. 3 shows the maximum chain depths and the match positions for each hash value for all match operations for a typical APK file. We can clearly see that

- Most match positions are far below the default maxchainDepth (set at 2048). In other words, we could use a smaller maxchainDepth without influencing the optimality of match positions.
- The actual chain depths are very different for different hash values, suggesting that it might be helpful for each list to use a different maxchainDepth.

Recompression-aware searching mechanism. The above observations motivate us to design a recompression-aware searching mechanism for generating the most appropriate REF token. We use a tuple $(m_i, d_i, h_i), i \in [1, n]$ to denote a match operation, where i denotes the i-th match operation, m_i denotes the match position, d_i denotes the maximum chain depth (note that the length of chain is dynamically updated) when the match is performed and h_i denotes the hash value. The searching time complexity is: $T = \sum_{i=1}^{n} d_i$. If we use L to limit the length of list corresponding to hash value h_i, the search time for the i-th match operation would become

$$t(i, L) = \begin{cases} d_i & d_i \le L_{h_i} \\ L_{h_i} & d_i > L_{h_i} \end{cases} \tag{1}$$

The use of L may incur the penalty of mismatch, i.e., the actual match position m is larger than the maximum allowable



{(1,1,h_1), (1,2,h_1), (2,3,h_1), (3,4,h_1), (1,4,h_1), (3,3,h_1)}
h_1
3
1
2
0
{(1,1,h_2), (1,2,h_2), (2,3,h_2), (3,4,h_2)}
h_2
2
1
1
0
{(1,1,h_3), (2,2,h_3)}
h_3
1
1
(a)
h_1
3
1
2
0
1/2
1/5
1/2
h_2
2
1
1
0
1/2
1/3
1/2
h_3
1
1
1/2
1/1
(b)
h_1
3
1
2
0
1/2
1/5
1/2
h_2
2
1
1
0
1/2
1/3
1/2
h_3
1
1
1/2
1/1
(c)

Fig. 4: The steps of greedy algorithm: (a) Build hash table. (b) Calculate benefit weight ratio. (c) Exclude element step by step until $\sum t(i) \le qT$. The red element is selected to be excluded.

search length, i.e., $m > L$. The penalty for an individual match operation can be calculated as:

$$p(i, L) = \begin{cases} 0 & m_i \le L \\ 1 & m_i > L \end{cases} \tag{2}$$

The problem we would like to solve is to find values of $L_1, ..., L_{h_{max}}$ (where the subscript denotes the hash values with h_{max} being the maximum hash value) in order to

$$\begin{aligned} Minimize \quad & \sum_{i=1}^{n} p(i, L_{h_i}) \\ s.t. \quad & \sum_{i=1}^{n} t(i, L_{h_i}) \le qT \end{aligned} \tag{3}$$

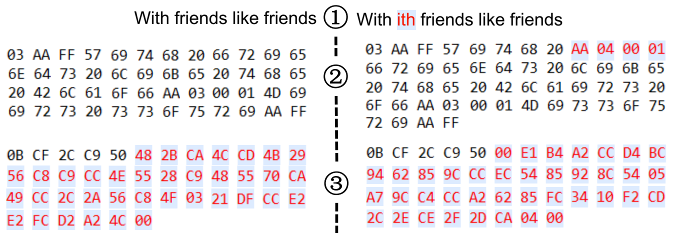
where q is a predefined value (i.e,. percentage) to limit the maximum search time at the mobile side. In our current implementation, $q = 0.5$.

Solving this problem using existing solvers would incur a large computation overhead since the number of decision variables is large, i.e., 65535, the number of possible hash values. To address this issue, we propose a greedy algorithm which works as follows:

(1) Build hash table. The algorithm first scans each match operation, builds a hash table. Each element in the list (corresponding to a specific hash value) is associated with a weight denoting the number of matches. For example, the match operations, i.e., (1,1,h_2), (1,2,h_2) (2,3,h_2) (3,4,h_2), lead to a list corresponding to h_2 with 4 elements shown in Fig. 4(a).

(2) Calculate benefit weight ratio. We use e to denote an element in the lists. $h(e)$ denotes its hash value. $d(e)$ denotes e's position (or depth) in the list. $w(e)$ denotes its weight, i.e., the number of matches of this element. The benefit in search time reduction for e is calculated as

$$\Delta t(e) = \sum_{i:h_i=h(e)} t(i, d(e)) - \sum_{i:h_i=h(e)} t(i, d(e) - 1) \tag{4}$$



```
03 AA FF 57 69 74 68 20 66 72 69 65
6E 64 73 20 6C 69 6B 65 20 74 68 65
20 42 6C 61 6F 66 AA 03 00 01 4D 69
69 72 73 20 73 73 6F 75 72 69 AA FF

0B CF 2C C9 50 48 2B CA 4C CD 4B 29
56 C8 C9 CC 4E 55 28 C9 48 55 70 CA
49 CC 2C 2A 56 C8 4F 03 21 DF CC E2
E2 FC D2 A2 4C 00
```

```
03 AA FF 57 69 74 68 20 AA 04 00 01
66 72 69 65 6E 64 73 20 6C 69 6B 65
20 74 68 65 20 42 6C 61 69 72 73 20
6F 66 AA 03 00 01 4D 69 73 73 6F 75
72 69 AA FF

0B CF 2C C9 50 00 E1 B4 A2 CC D4 BC
94 62 85 9C CC EC 54 85 92 8C 54 05
A7 9C C4 CC A2 62 85 FC 34 10 F2 CD
2C 2E CE 2F 2D CA 04 00
```

Fig. 5: Differencing results of the case where a few characters are added into an existing file, i.e., “With friends like friends” → “With ith friends like friends”. ① indicates the decompressed stream, ② indicates the LZ77 stream, ③ indicates the deflate stream. Different bytes are indicated in red color.

The benefit weight ratio for e is simply $r(e) = \Delta t(e)/w(e)$.

(3) Exclude element step by step. The algorithm tries to exclude the minimum weighted number of elements from the lists while satisfying the maximum allowable search time at the same time. The algorithm exclude the trailing element with a weight of 0 for each list. Then the algorithm operates from top to down, and right to left. In each step, the algorithm tries to exclude an element with the highest benefit weight ratio, i.e., the minimum weight and maximum benefit in search time reduction. The algorithm ends up until $\sum t(i) \leq qT$. Fig. 4 shows each execution steps of this algorithm.

IV. Exploiting the Partial Decoded Similarity Spaces

As we have mentioned in Section II-B, all deflate algorithms first compress the file using LZ77, and then further compress the LZ77 stream using huffman encoding. LZ77 performs at the byte level while huffman encoding performs at the bit level. Transforming the deflate stream into partial decoded space (i.e., LZ77 stream) can result in significantly larger similarity compared with performing comparison in the original (e.g., compressed) similarity space, since all existing differencing algorithms operates at the byte level.

Fig. 5 shows an example change case where we add a few characters into an existing file. We can see that the delta would be very large if comparing the compressed files (e.g., deflate streams) although the actual changes are very small. However, if we compare the files in the partial decoded spaces (i.e., decompressing the compressed files into LZ77 streams), we retain a much smaller delta compared with computing the delta of two deflate streams.

Only huffman encoding should be performed when transforming a LZ77 stream (partial decoded stream) to the original deflate stream. Therefore, operating in the partial decoded spaces has two additional benefits. First, it is easy to recompress the LZ77 stream to the original deflate stream with the saved huffman trees regardless of the specific deflate algorithms. Second, computing at the partial decoded space can result in significantly smaller recompression and reconstruction time at the mobile side, because only huffman encoding is performed during the recompression stage.

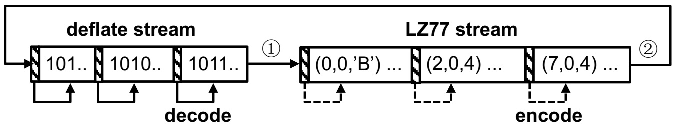


Fig. 6: Workflow of partial decode and partial encode.

Fig. 6 depicts the workflow of partial decode and partial encode. When performing partial decoding ①, we use the Huffman tree code at the head of blocks to decode the bit stream in the block to obtain the LZ77 stream. The Huffman tree code will remain in the partial decoded space as metadata. When performing partial encode ②, we encode the LZ77 stream to Huffman codes through the saved metadata to obtain the original deflate stream.

Differencing-friendly LZ77 tokens reencoding. A LZ77 stream in the partial decoded space consists of a series of LZ77 tokens. As we have described in Section II-B, LZ77 tokens can be divided into LIT and REF tokens and both can be represented by a three-byte tuple (x, y, z).

While the byte form of the LZ77 tokens is friendly for Huffman encoding, it is not originally designed and fully optimized for difference computation. Fig. 7 shows an example change case when we change an old file “BBBBACACACBACA” to a new file “BBBBACXCACBACA”. When encoded using LZ77, the old file will be transformed to LIT<‘B’>REF<1,3>LIT<‘A’>LIT<’C’>REF<2,4>REF<7,4> and the new file will be transformed to LIT<‘B’>REF<1,3>LIT<‘A’>LIT<‘C’>LIT<‘X’>LIT<‘C’>LIT<‘A’>LIT<‘C’>REF<7,3>, as indicated in Fig. 7. Comparing the old and new files in the LZ77 space would generate two copy instructions and two add instructions. Note that the copy instructions (e.g., in HDiffPatch as well as bsdiff) can copy not only identical segments but also similar segments, and the additional segment difference (encoded using Run-Length-Encoding, i.e., RLE) is added to the copied segment to generate the final segment in the new file. For example, the cost of the segment difference of (0,0,0,0,0,0) is 2 bytes when RLE is applied (i.e., five zeros). The total cost of the four instructions will be (3+2)+6+(3+2)+6=22 bytes. We can see that a very small change in the old file would generate a relatively large difference in the LZ77 space: the three byte representation of literal tokens is not compact enough and it would cause large overhead in the add instruction.

To address this issue, we have designed a differencing-friendly encoding scheme to reencode the LZ77 stream. The revised reencoding scheme works as follows: First, we make the LIT token representation more compact by using only one byte to represent a LIT token. Second, we use an additional escape character to precede a REF token to make it distinguishable from LIT tokens. We have chosen 0xAA as the escape character since it has a low frequency in .so and .dex files from our observation. When the value of LIT token is also 0xAA, we will use two 0xAA to represent it. Finally, considering that REF tokens of length 3 often occupy the largest proportion of REF tokens, we omit the length byte

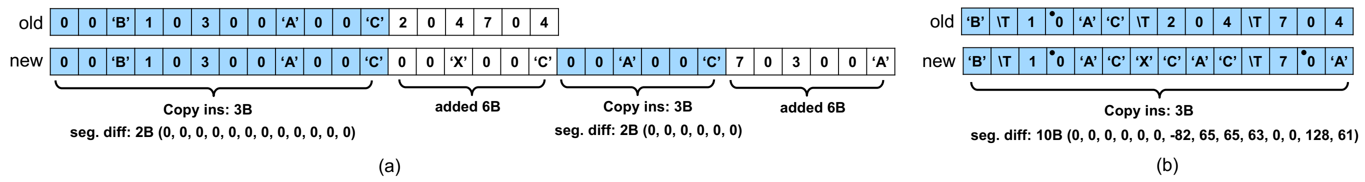

old
new
Copy ins: 3B
seg. diff: 2B (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
added 6B
Copy ins: 3B
seg. diff: 2B (0, 0, 0, 0, 0, 0)
added 6B
(a)
old
new
Copy ins: 3B
seg. diff: 10B (0, 0, 0, 0, 0, 0, -82, 65, 65, 63, 0, 0, 128, 61)
(b)

Fig. 7: The example of differencing the LZ77 stream in partial decoded space for a pair of files, i.e., BBBBACACACBACA $\rightarrow$ BBBBACXCACBACA. The LZ77 stream of old file and new file are LIT<'B'>REF<1,3>LIT<'A'>LIT<'C'> REF<2,4>REF<7,4> and LIT<'B'>REF<1,3>LIT<'A'>LIT<'C'>LIT<'X'>LIT<'C'>LIT<'A'>LIT<'C'>REF<7,3> LIT<'X'> respectively. The "\T" indicates escape character 0xAA. The value with $\cdot$ acutally means the value + 128.

and set the highest bit of the "y" field in the three byte tuple representation to 1. It is possible since the highest bit of "y" is currently unused because the distance byte is limited to a 32k window.

The above reencoding scheme has a more compact representation in the byte stream and it is also more differencing friendly than LZ77's original encoding scheme. Fig. 7 shows the same example using our scheme. We can see that comparing the reencoded LZ77 streams will only generate one single copy instruction with the segment difference being 10B. This is because the cost of this segment difference is 2B for encoding six zeros using RLE and 8B without using RLE. Therefore, the total cost of the copy instruction is 3+10=13 bytes.

V. A tunable algorithm exploiting both similarity spaces

We have described two approaches exploiting the full and partial decoded similarity spaces for difference computation. Generally speaking, computing at the full decoded similarity space results in a small delta. However, it can result in additional overhead of the patch and much longer reconstruction time. Computing at the partial decoded similarity space will typically generate a larger delta but incurs a small reconstruction time.

Fig. 8a shows a typical change case in which comparing in the full decoded similarity spaces generate the smallest delta, at the cost of longer recompression time at the mobile side. However, it is also possible that comparing in the partial decoded similarity spaces generate a smaller delta. Fig. 8b shows such a case. This can be due to two reasons. First, the added bytes in the full decoded space can be compressed by REF tokens in LZ77 streams. Second, the full decoded file contains additional metadata for recompression. Partial decoding will be preferred in the cases where partial decoding yields a smaller delta size, as partial decoding always has a lower recompression time.

Based on these observations, we have devised MDiffPatch, which combines the approaches we have described above with a tunable parameter α. The algorithm tries to compares at the full decoded similarity spaces for a total of $F_\alpha = \alpha \cdot S_{\text{deflate}}$, where S_{deflate} denotes the total number of deflate bytes in APK, and make the optimal strategy for differencing with α, that is, to get the smallest delta file size with the limited number of bytes transformed into full decoded space.

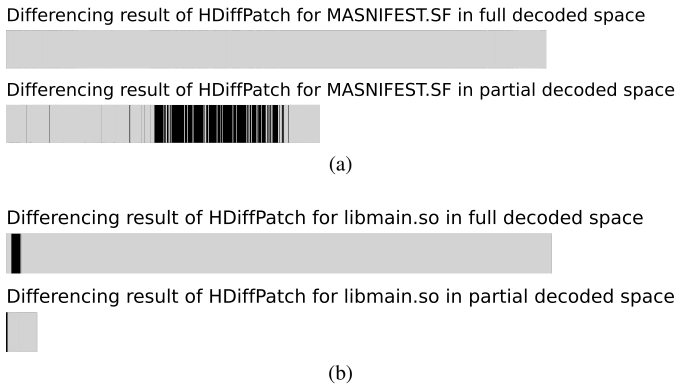


Fig. 8: Differencing results of HDiffPatch for two pairs of files in APK. In the rectangle, the added bytes are indicated in black color, and the copied bytes are indicated in grey color. (a) The case where full decoding is most beneficial. (b) The case where partial decode is most beneficial.

Algorithm 1 shows the pseudocode for MDiffPatch at the differencing phase. The algorithm tries to select a deflate file f_i in APK with size s_i and benefit b_i being the reduced size of delta files computing in the full decoded space compared with computing in the partial decoded space. The algorithm should have a set of files in APK for possible full decoding until the full decoded bytes reaches F_α or the remaining files have negative benefits (i.e. partial decoding is better). This is a classical 0-1 knapsack problem with the capacity being the F_α and each item being the file (selected or not selected), which can be solved by dynamic programming that select the deflate files to reach the maximum benefit, i.e., lowest compression ratio, while keeping the full decoded bytes within F_α to restrict the high recompression time caused by full decoding.

It is relatively simple for MDiffPatch in the reconstruction phase since we have recorded necessary metadata information in the delta file (we omit the details due to space limit).

VI. Evaluation

We implement MDiffPatch by integrating techniques and algorithms we have proposed in Section III-V, based on open-source software including precomp, zlib, HDiffPatch. The new lines of code is approximately 2000 in C. In this section, we evaluate performance of MDiffPatch compared with state-of-the-art algorithms.

Algorithm 1: MDiffPatch (differencing phase)

```
Input  : old source APK $A_{old}$, new target APK $A_{new}$, the ratio $\alpha$
Output: delta file $\Delta$
1  Get a list of pairs of files in APK that have changed
2  $L \leftarrow \{(f_{old}(i), f_{new}(i))\}_{i=1}^{n}$, $f_{new}(i) \in A_{new}$ and $f_{old}(i)$ is the
     corresponding file in $A_{old}$, $f_{new}(i) \neq f_{old}(i)$
3  for each $(f_{old}(i), f_{new}(i))$ in $L$ do
4      Differencing $f_{old}(i)$ and $f_{new}(i)$ in partial decoded space by
         HDiffPatch, get the delta file $\Delta_p$
5      Differencing $f_{old}(i)$ and $f_{new}(i)$ in full decoded space by
         HDiffPatch, get the delta file $\Delta_f$
6      $b(i) \leftarrow size(\Delta_p) - size(\Delta_f)$
7      if $b_i > 0$ then
8          $s(i) \leftarrow$ size of $f_{new}(i)$
9      else
10         $s(i) \leftarrow maximum$
11 $F_\alpha \leftarrow \alpha\cdot$ size of $\{file_{new}(i)\}_{i=1}^{n}$
12 /* 0/1 Knapsack Problem Solver seekMax */
13 F[1..n] = seekMax(n,$F_\alpha$,b[1..n],s[1..n])
14 for i = 1..n do
15     if $F[i] == 1$ then
16         Full decode $f_{old}(i), f_{new}(i)$, update $A_{old}$ and $A_{new}$
17     else
18         Partial decode $f_{old}(i), f_{new}(i)$, update $A_{old}$ and $A_{new}$
19 Perform HDiffPatch between $A_{old}$ and $A_{new}$ and write instructions
     to $\Delta$
20 Add metadata $L$ to $\Delta$
21 Perform zstd compression on $\Delta$
```

TABLE I: The 10 representative application updates and the 400 updates. $\star$ indicates that it is a game application.

Name	Version	Old size (B)	New size (B)
Douyin	22.9.0→23.0.0	168,405,402	169,354,581
Zhihu	8.38.0→8.39.0	69,060,906	70,652,038
WeChat	8.0.27→8.0.28	276,602,366	266,691,829
Weibo	12.10.2→12.11.0	206,500,685	207,162,286
QQ	8.9.15→8.9.18	311,322,716	307,940,064
Baidu	13.19.5.10→13.21.0.11	137,200,701	137,805,661
Bilibili	7.3.0→7.4.0	102,429,902	101,626,272
Youku	10.2.57→10.2.59	65,399,355	65,639,977
PUBG Mobile $\star$	1.19.3→1.20.13	2,056,739,892	2,037,120,844
Honkai Impact 3 $\star$	6.0.0→6.1.0	613,738,862	634,074,809
......			
Sum	400 updates	98.69 GB	99.13 GB

A. Methodology

Evaluation platform. We run our differencing algorithm on a sever with Intel i7-12700 CPU @2.10GHz with 20 cores, 16GB DDR4 RAM @3200 MT/s with Ubuntu 22.04 LTS. We run our reconstruction (including recompression) algorithm on smartphone ZTE Axon 10. The CPU is a Snapdragon 855 with AArch64 Cortex-A76 based Kryo 485 architecture, 8 cores at 2.84 GHz. and 6 GB of RAM.

Dataset. We select 150 normal and 50 game apps with the highest downloads in a popular App Market up until November 14, 2022. Each app includes three consecutive recent versions, for a total of 600 APKs. There are a total of 400 update cases (i.e., update to the newest version) in our dataset. We also select 10 representative APK and 10 representative update cases with the top downloads for a detailed study. Table I shows the statistics about the 10 update cases in our dataset.

Algorithms for comparison. We compare the following algorithms:

- *HDiffPatch*. To the best of our knowledge, HDiffPatch is the most efficient algorithm in terms of compression ratio and reconstruction time. It is actually used in well-known app markets (e.g., OPPO's App Market) for app updates. We configure HDiffPatch by using the default settings, e.g., segment matching using suffix array and delta compression using zstd-21 [8].
- *archive-patcher*. It is the algorithm developed by Google and it uses zlib for decompression and recompression. We configure archive-patcher by using the default settings, e.g., delta compression using zlib-9 [1].
- *MDiffPatch*. It is the algorithm we have developed in this paper. MDiffPatch has a tunable parameter α which controls how to transforming the deflate bytes into different similarity spaces.
- *MDiffPatch-precomp*. It is the algorithm which directly builds on top of precomp without recompression-aware searching. This algorithm always tries transform all deflate bytes into the full decoded similarity space.

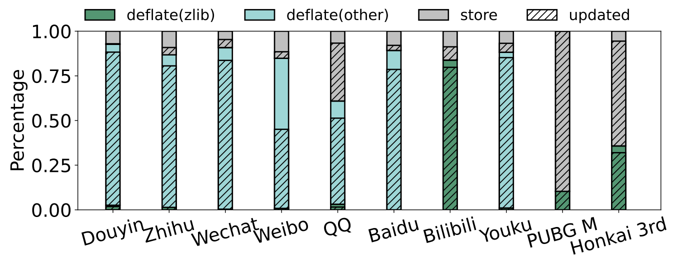
Metrics. Evaluations focus on the three key metrics:

- *Compression ratio*. The compression ratio is defined as the ratio between the delta size and the size of the new file. A smaller compression ratio is preferred as it saves more network bandwidth.
- *Reconstruction time*. It is time to reconstruct the new file from the delta and the corresponding old file on the mobile side. Recompression (to the original APK) occupies a large fraction in the reconstruction time, for algorithms which employ decompression-before-differencing.
- *Reconstruction memory usage*. We measure average memory usage during the reconstruction phase on the mobile side.

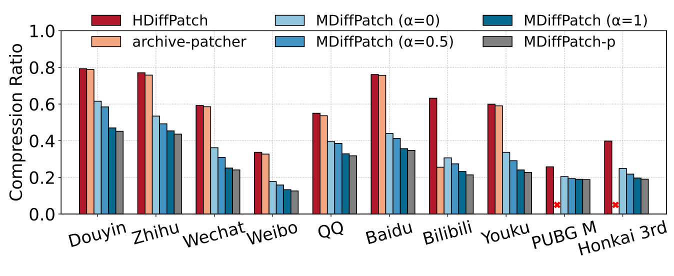
B. Main results

Detailed results for representative updates. Fig. 9a shows the composition of APK (new version). Different colors show information whether the file in APK is deflated or not (i.e., in store mode), whether it is deflated using zlib or some other deflate algorithms, and whether the file is updated. We can see that: *(i)* Most of the files in normal APK are compressed as deflate stream, while most of the files of game applications are stored without compression (see PUBG M, Honkai 3rd in Fig. 9a). *(ii)* The number of updated files account for a large proportion of the total files. However, it doesn't mean a large amount of changed bytes since there may exist only a small change in the updated file. *(iii)* Only three representative APKs mainly consist of zlib-compressed deflate streams, i.e., Bilibili, PUBG M and Honkai 3rd, indicating that many popular applications tend to use other more specific deflate algorithms with higher compression ratio. Note that deflate stream using other algorithms (i.e., not zlib) cannot handled by state-of-the-art algorithms, e.g., archive-patcher.

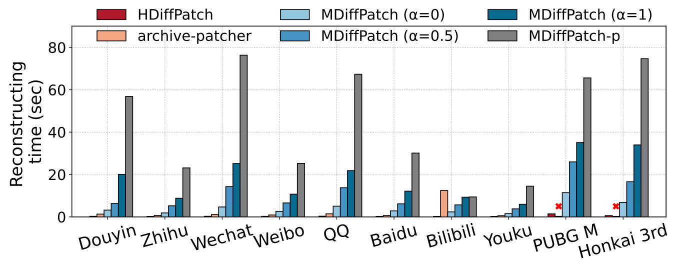
Fig. 9b shows the comparison of compression ratio of different algorithms. We can see that: *(i)* MDiffPatch (α=1) is performs the best in terms of compression ratio. *(ii)* MDiffPatch (α=0) is also much better than other algorithms.



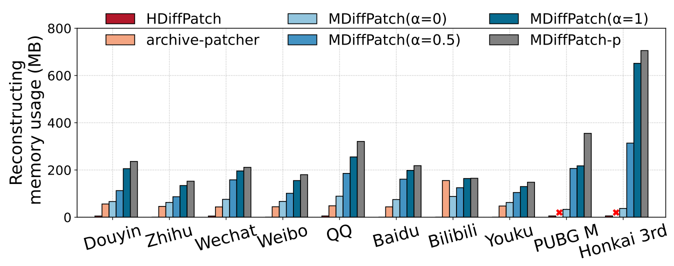
(a)



(b)

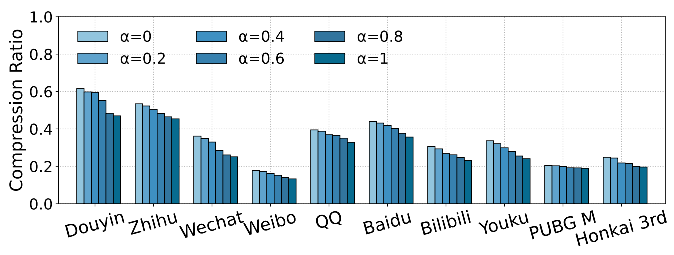


(c)

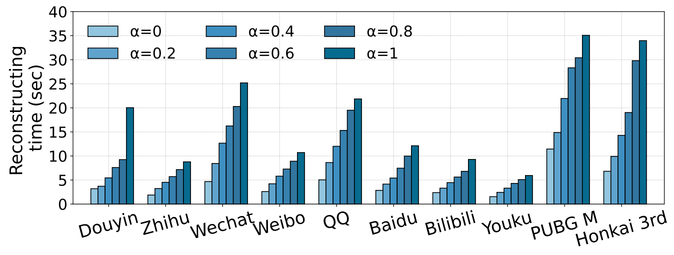


(d)

Fig. 9: Comparison of different algorithms performance for 10 representative app updates.



(a)



(b)

Fig. 10: MDiffPatch performance with different α for 10 representative app updates.

The exception is Bilibili which uses zlib for compression and thus can well be handled by existing algorithms such as archive-patcher. *(iii)* The compression ratio of MDiffPatch decreases with the increase of α. *(iv)* The compression ratio of MDiffPatch ($\alpha = 1$) is only slightly larger than MDiffPatch-precomp, which means that the recompression-aware searching mechanism only incurs a very small impact on the compression ratio. Note that archive-patcher fails to diff game applications, e.g., PUBG M and Honkai 3rd, because of their excessive size.

Fig. 9c shows the reconstruction time of different algorithms. We can see that: *(i)* The reconstruction time of MDiffPatch (α=0) is relatively small while the reconstruction time of MDiffPatch (α=1) is relatively high. When $\alpha = 0$, all deflate is transformed into partial decoded space, which make the reconstruction of deflate streams fast because only the Huffman encoding is performed. When $\alpha = 1$, all deflate is transformed into full decoded space, which make the reconstruction of deflate streams slow because of the process of LZ77 algorithm. *(ii)* The reconstruction time of MDiffPatch increases with the increase of α. *(iii)* The reconstruction time of MDiffPatch (α=1) is much smaller than that of MDiffPatch-precomp due to the novel optimization technique we have proposed in Section III.

Fig. 9d shows the reconstruction memory usage of different algorithms. We can see that: *(i)* HDiffPatch has the lowest memory usage. *(ii)* MDiffPatch-precomp has the highest memory usage due to the use of large hash table. *(iii)* The reconstruction memory usage of MDiffPatch increases with the increase of α. It is reasonable since more memory will be used in order to compress more fully decoded bytes.

Impacts of the tunable parameter α. Fig. 10a shows the comparison of compression ratio of MDiffPatch with different α for the representative app updates. We can see that: *(i)* The compression ratio decreases as α increases. *(ii)* The compression ratio of normal APKs is more sensitive to α, compared with game app. This is because the deflate stream only occupies a small fraction of the entire game application which is illustrated in Fig. 9a.

Fig. 10b shows the comparison of reconstruction time of

TABLE II: Comparison of MDiffPatch with other algorithms for 400 app updates. Ver. rate denotes the success rate for the verification of the reconstructed apps.

Algorithm	Compression ratio	Recons. time	Ver. rate
HDiffPatch	43.84%	0.46 sec	100%
archive-patcher	34.14%	6.31 sec	100%
MDiffPatch (α=0)	28.05%	2.79 sec	100%
MDiffPatch (α=0.5)	25.91%	6.45 sec	100%
MDiffPatch (α=1)	**23.23**%	10.80 sec	100%

MDiffPatch with different α for the representative app updates. We can see that *(i)* The reconstruction time increases as α increases. *(ii)* The reconstruction time of normal APKs is less sensitive to α, compared with game app.

To summarize, MDiffPatch can strike a reasonable balance between compression ratio and reconstruction time, thus achieving the required level of flexibility by exposing the tunable parameter α.

Results for 400 app updates. Table II shows the evaluation results for 400 app updates with respect to two key metrics including compression ratio and reconstruction time. We can see that: *(i)* When α = 1, MDiffPatch achieves the best compression ratio, i,e., 23.23%. It means an absolute reduction of 20.61% and relative reduction of 47.01% compared with HDiffPatch and an absolute reduction of 10.91% and relative redunction of 31.96% compared with archive-patcher. The reconstruction time is kept within 1.5x compared with archive-patcher. *(ii)* When α = 0, MDiffPatch achieves an absolute reduction of 15.79% and relative reduction of 36.01% compared with HDiffPatch in terms of compression ratio. It achieves an absolute reduction of 6.09% and relative reduction of 17.84% compared with archive-patcher in terms of compression ratio. The reconstruction time is only 44.21% compared with archive-patcher since partial decoding and reencoding is significanly faster than full decoding and reencoding. *(iii)* The verification success rate (Ver. rate) is 100%, i.e., all the reconstructed new apps successfully pass the verification of signatures.

VII. Related work

We divide related work into two categories: differencing algorithms and decompression-before-differencing techniques.

Differencing algorithms. Many differencing algorithms have been proposed by researchers throughout the years [2], [6], [7], [15], [16], [19], [21], [23]. The well-known `diff` [17] utility in Linux exhibits a proficient ability in generating deltas between textual data. More sophisticated binary differencing algorithms have emerged recently, e.g., xdelta3 [18], bsdiff [26], and HDiffPatch [31]. xdelta3 uses a hash match to identify identical segments, while bsdiff and HDiffPatch employ suffix-arrays to find approximate matching segments. These algorithms only perform differencing in the original similarity space. However, the majority of binary code and resource files within the APK are independently compressed using the deflate algorithm, which drastically reduces the similarity between similar files. Therefore, their performance degrades when a large portion of files in APK are compressed.

MDiffPatch is based on HDiffPatch. The difference is that MDiffPatch employs decompressing-before-differencing in order to achieve a higher level of compression ratio.

Decompressing-before-differencing techniques. To preserve a large similarity when comparing two APKs, Delta++ [29] and archive-patcher [9] employ the decompression-before-differencing technique to significantly reduce the delta size. The key idea of these algorithm is trying to decompress the compressed files before perform difference computation so as to retain a much larger similarity. These algorithms build on top of zlib, and try to guess the correct compression parameters so that they can recompress the decoded bytes to the original deflate stream.

A variety of compression tools are available for the deflate algorithms. Notably, zlib [1] and 7zip [24] are extensively utilized for a wide range of compression tasks. They can transform uncompressed files into compressed files through the deflate algorithm which consists of two main stages including LZ77 encoding and Huffman encoding. Precomp is an general tool to decompress deflate streams to the full decoded similarity space and reconstruct them using an *enhanced* zlib algorithm, allowing reconstruction using a very small metadata so that they are *bit-to-bit-identical* with the original stream. Precomp is mainly used for more compact compression (e.g., compression using more advanced tools on the decoded stream) and is not tailored for use in combination with differencing algorithms.

MDiffPatch borrows idea of decompression-before-differecing and builds on top of precomp. The difference is that MDiffPatch explores the decompression-before-differencing technique much further. (1) It addresses the insufficient decompression problem in existing differencing algorithms. Both the full decoding and partial decoding approaches in MDiffPatch can handle all deflate streams while start-of-the-art algorithms (e.g. archive-patcher) can only handle zlib-compressed deflate streams. (2) It addresses the problem of high recompression time at the mobile side, by employing novel techniques such as recompression-aware searching (see Section III) and partial decoding (Section IV).

VIII. Conclusion

In this paper, we exploit two similarity spaces, i.e., the full decoded similarity space and the partially decoded similarity space, for efficient and flexible incremental updates of mobile applications. The first approach uses recompression-aware searching mechanism based on a general full decoding tool to transform deflate stream to the full decoded similarity space with a configurable searching complexity. The second approach uses a novel solution to transform a deflate stream into the partial decoded similarity space with differencing-friendly LZ77 token reencoding. We have also proposed an algorithm MDiffPatch exploiting both similarity spaces. Results shows that MDiffPatch achieves lower compression ratio than state-of-the-art algorithms and its tunable parameters allows us to achieve a good tradeoff between compression ratio and recompression time.

References

[1] Adler, M. zlib. https://www.zlib.net/, 2022.

[2] Ajtai, M., Burns, R., Fagin, R., Long, D. D., and Stockmeyer, L. Compactly Encoding Unstructured Inputs with Differential Compression. *Journal of the ACM (JACM) 49*, 3 (2002), 318–367.

[3] Apiwattanapong, T., Orso, A., and Harrold, M. J. A Differencing Algorithm for Object-oriented Programs. In *Proc. of IEEE/ACM ASE* (2004), pp. 2–13.

[4] Burns, R., Stockmeyer, L., and Long, D. D. In-place Reconstruction of Version Differences. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 15*, 4 (2003), 973–984.

[5] Deutsch, P. Deflate. RFC 1951: Deflate compressed data format specification version 1.3,, 1996.

[6] Dong, W., Chen, C., Bu, J., and Liu, W. Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems. *ACM Transactions on Sensor Networks (TOSN) 11*, 2 (2014), 1–34.

[7] Dong, W., Liu, Y., Chen, C., Bu, J., Huang, C., and Zhao, Z. R2: Incremental Reprogramming using Relocatable Code in Networked Embedded Systems. *IEEE Transactions on Computers (TC) 62*, 9 (2012), 1837–1849.

[8] Facebook. zstd. https://github.com/facebook/zstd, 2023.

[9] Google. archive-patcher. https://github.com/google/archive-patcher, 2023.

[10] Google. Googleplay. https://play.google.com, 2023.

[11] Google. Smart app updates. https://www.engadget.com/2012-06-27-google-brings-incremental-app-updates-to-android-added-encrypti.html, 2023.

[12] He, B., Xu, H., Jin, L., Guo, G., Chen, Y., and Weng, G. An Investigation into Android In-App Ad Practice: Implications for App Developers. In *Proc. of IEEE INFOCOM* (2018), pp. 2465–2473.

[13] Huawei. Huaweiappgallery. https://appgallery.huawei.com, 2023.

[14] Kaul, S., Yates, R., and Gruteser, M. Real-time status: How often should one update? In *Proc. of IEEE INFOCOM* (2012), pp. 2731–2735.

[15] Li, B., Tong, C., Gao, Y., and Dong, W. S2: a Small Delta and Small Memory Differencing Algorithm for Reprogramming Resource-constrained IoT Devices. In *Proc. of IEEE INFOCOM (WKSHPS)* (2021), pp. 1–2.

[16] Li, Q., Feng, X., Wang, R., Li, Z., and Sun, L. Towards fine-grained fingerprinting of firmware in online embedded devices. In *Proc. of IEEE INFOCOM* (2018), pp. 2537–2545.

[17] linuxize. diff-command-in-linux. https://linuxize.com/post/diff-command-in-linux/, 2023.

[18] Macdonald, J. xdelta3. http://xdelta.org/, 2023.

[19] May, M. J. Donag: Generating Efficient Patches and Diffs for Compressed Archives. *ACM Transactions on Storage (TOS) 18*, 3 (2022), 1–41.

[20] Mi. autoupdate. https://dev.mi.com/console/appservice/autoupdate.html, 2023.

[21] Mo, B., Dong, W., Chen, C., Bu, J., and Wang, Q. An Efficient Differencing Algorithm Based on Suffix Array for Reprogramming Wireless Sensor Networks. In *Proc. of IEEE ICC* (2012), pp. 773–777.

[22] OPPO. Oppo incremental update. https://open.oppomobile.com/, 2023.

[23] Panta, R. K., and Bagchi, S. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *IEEE INFOCOM* (2009), pp. 639–647.

[24] Pavlov, I. 7zip. https://www.7-zip.org/, 2023.

[25] Pavlov, I. attemp to recompress files with deflate of 7zip. https://sourceforge.net/p/sevenzip/discussion/45797/thread/7303ced019/, 2023.

[26] Percival, C. Binary diff/patch utility. http://www.daemonology.net/bsdiff/, 2023.

[27] rarlab. Winrar. https://www.rarlab.com/download.htm, 2023.

[28] samsung. Huaweiappgallery. https://galaxystore.samsung.com, 2023.

[29] Samteladze, N., and Christensen, K. DELTA++: Reducing the Size of Android Application Updates. *IEEE Internet Computing 18*, 2 (2013), 50–57.

[30] Shang, X., Huang, Y., Mao, Y., Liu, Z., and Yang, Y. Enabling qoe support for interactive applications over mobile edge with high user mobility. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (2022), pp. 1289–1298.

[31] sisong. HDiffPatch. https://github.com/sisong/HDiffPatch, 2023.

[32] Statista. Number of mobile app downloads worldwide from 2018 to 2023. https://www.statista.com/statistics/241587/number-of-free-mobile-app-downloads-worldwide/, 2023.

[33] Tencent. Yingyongbao incremental update. https://36kr.com/p/1640964784129, 2023.

[34] Tridgell, A., Mackerras, P., et al. The rsync algorithm.

[35] Xu, Q., Liao, Y., Miskovic, S., Mao, Z. M., Baldi, M., Nucci, A., and Andrews, T. Automatic generation of mobile app signatures from traffic observations. In *2015 IEEE Conference on Computer Communications (INFOCOM)* (2015), pp. 1481–1489.