Understanding Differencing Algorithms for Mobile Application Updates

Tong Sun, Bowen Jiang, Lewei Jin, Wenzhao Zhang, Yi Gao, *Member, IEEE*, Zhendong Li, and Wei Dong, *Member, IEEE*

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. Researchers have proposed many differencing algorithms over the years. Unfortunately, it is currently unknown how these algorithms quantitatively perform for different categories of applications. It is also challenging to know the impacts of different techniques and whether a technique in one algorithm can be integrated into another algorithm for further performance improvement.

This paper conducts the first systematic study to understand the performance of four widely used differencing algorithms for mobile application updates, including xdelta3, bsdiff, archive-patcher, and HDiffPatch with respect to five key metrics, including compression ratio, differencing time/memory overhead, and reconstruction time/memory overhead. We perform measurements for 200 mobile applications, and analyze key techniques (such as decompressing-before-differencing, sliding window, and copy instructions merging) that influence the performance of these algorithms. We have provided four important findings which give insights to further optimize for performance improvement. Guided by these insights, we have also proposed a novel algorithm, **sdiff**, which achieves the smallest compression ratio to state-of-the-art algorithms by combining an appropriately chosen set of key techniques.

Index Terms—Differencing algorithms, Mobile applications, Compression.

1 Introduction

Mobile application updates occur frequently and add considerable traffic over the Internet. According to Statista's reports, the number of application downloads worldwide has reached 275 billion in the year 2022 [1]. It is very common that there is an update for a mobile application every few weeks, e.g., adding additional functionalities [2], enhancing user QoE (Quality of Experience) [3], or fixing software bugs. Mobile operators spend billions of dollars for mobile application updates every year, e.g., paying for the server bandwidth of the CDN (Content Delivery Network) services. To reduce the update overhead, they often apply a differencing algorithm that computes a small delta between the new version of the updated application and the version already installed on the user's mobile device.

Researchers have proposed many differencing algorithms over the years [4], [5], [6], [7]. Researchers in the sensor network community have proposed many differencing algorithms for firmware updates [8], e.g., delta++ [9], r2diff [10], r3diff [11], DASA [12], and S2 [13]. DASA [12] and r3diff [11] can generate the *optimal* delta size assuming a given set of commands and cost measures in the delta file [14]. More sophisticated differencing algorithms have appeared in recent years, e.g., xdelta3 [15], bsdiff [16], HDiffPatch [17], and archive-patcher [18]. They are widely used in today's mobile app markets, e.g., the archive-patcher algorithm is employed in Google Play [19], HDiffPatch is used in OPPO's APP Market [20], xdelta3 is utilized in the Xiaomi app store [21], and bsdiff is employed in Yingyong Bao (a widely used third-party app store in China) [22]. We perform measurements on 200 updates for 200 mobile applications, covering both normal (i.e., non-gaming) and game applications.

Unfortunately, it is currently unknown how these algorithms quantitatively perform for different categories of applications, with respect to key metrics such as compression ratio, differencing time (for generating the delta at the server side), and reconstruction time (for reconstructing the new application version by patching the delta to the old application version at the mobile client side). Since these algorithms often adopt inter-correlated techniques, it is also challenging to know the impacts of different techniques and whether a technique in one algorithm can be integrated into another algorithm for further performance improvement.

This paper conducts the first systematic study to understand the performance of four widely used differencing algorithms for mobile application updates, including xdelta3 [15], bsdiff [16], archive-patcher [18], and HDiffPatch [17] with respect to five key metrics including compression ratio, differencing time/memory overhead, and reconstruction time/memory overhead.

Summary of insights. Our measurements lead to four important findings, which we summarize as follows:

- *Tong Sun, Lewei Jin, Wenzhao Zhang, Yi Gao, and Wei Dong are with the College of Computer Science, Zhejiang University, Zhejiang 310027, China. E-mail: {tongsun, jinlw, wz.zhang, gaoyi, dongw}@zju.edu.cn*
- *Bowen Jiang is with the School of Software Technology, Zhejiang University, Zhejiang 315048, China. E-mail: jiangbw@zju.edu.cn*
- *Zhendong Li is with the Huawei Technologies Co., Ltd. Guangdong 518129, China. E-mail: lizhendong8@huawei.com*

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.
(Corresponding author: Wei Dong.)

(i) Our results show that both bsdiff and archive-patcher fail in generating delta files for large APKs. In addition, archive-patcher fails for some cases during reconstruction at the mobile side. While some failures can be addressed by either relaxing the threshold or modifying the temporal directory, it requires a more systematic approach to addressing failures caused by excessive memory consumption during delta generation. Both bsdiff and archive-patcher read the entire old and new files into memory for delta generation, which quickly exhausts server-side memory resources for large APK files. We believe that a sliding window mechanism (like the one used in xdelta3) is necessary to limit the maximum memory usage during delta generation.

(ii) We find that common segments have locality and normal applications have better locality than game applications. For example, 93.8% of common segments can be matched in a 64MB window for WeChat, and 97.2% of common segments can be matched in a 128MB window for Honkai Impact 3. A typical window size of 128MB is appropriate since larger window size would not significantly reduce the delta size.

(iii) A non-negligible portion of APK files is compressed. The archive-patcher algorithm utilizes the technique of decompressing-before-differencing, yielding smaller compression ratio since it preserves a higher similarity. Our results also show that an average of 24% of bytes in the APK cannot be decompressed with the current zlib [23] tool used in archive-patcher. Otherwise, they cannot be recompressed in the same manner at the mobile side. This result indicates that the potential of decompressing-before-differencing is not fully exploited. On the other hand, our study also reveals results that decompressing is not always beneficial: decompressing-before-differencing yields worse results than directly differencing for at least 5.6% files in the APK.

(iv) HDiffPatch performs consistently better than bsdiff, especially for large files. The reason is that HDiffPatch further merges copy instructions, resulting in much fewer copy instructions. We have deeply investigated the optimality problem of merging copy instructions and find that the current merging algorithm in HDiffPatch is still not optimal.

Our findings provide a deeper understanding of the contributing factors and techniques of the performance difference among these differencing algorithms. We hope our results can help researchers identify opportunities for further performance improvement. To illustrate how our findings can be utilized, we have also proposed a novel algorithm, sdiff, which achieves smaller compression ratio than state-of-the-art algorithms by combining a set of existing techniques, including decompressing-before-differencing, sliding window, fast deduplication, and the HDiffPatch algorithm.

Contributions. In summary, this paper makes the following contributions:

- We conduct the first systematic performance comparison of four differencing algorithms for mobile application updates.
- We investigate and analyze key techniques (such as decompressing-before-differencing, sliding window, copy instructions merging) that influence the performance of these algorithms. We have revealed four key insights that could guide future algorithm design.
- We have proposed **sdiff**, a simple and better algorithm with a smaller compression ratio by combining an appropriately chosen set of key techniques.
- We have publicly made our data and code[1] available to facilitate for further study by other researchers.

2 Related Work

2.1 Binary differencing algorithms

Researchers have proposed many differencing algorithms over the years [4], [5], [6], [7]. The Linux utility `diff` can generate deltas between texts fairly well. The Rsync algorithm [24], operating at the block level, trades optimality for performance and is suitable for synchronizing large files for P2P applications. Researchers in the sensor network community have proposed many differencing algorithms for firmware updates [8], e.g., delta++ [9], r2diff [10], r3diff [11], DASA [12], and S2 [13]. DASA [12] and r3diff [11] can generate the *optimal* delta size assuming a given set of commands and cost measures in the delta file [14]. More sophisticated differencing algorithms have appeared in recent years, e.g., xdelta3 [15], bsdiff [16], HDiffPatch [17], and archive-patcher [18] They are generally more suitable for mobile app updates. The reasons are two-fold: *first*, they handle large app files (i.e., APK files) more efficiently, and *second*, they better exploit the computing power of modern smartphones by applying general compression (e.g., LZMA [25] and bzip2 [26]) to further reduce the delta size.

In this work, we aim to comprehensively understand the major differencing algorithms utilized for mobile application updates, with a specific emphasis on those actually employed in app stores. The technical details of these algorithms are introduced in Section 3.2.

2.2 Hot-patching algorithms for mobile app updates

Hot-patching, also known as live patching or dynamic software updating, refers to the process of applying updates or patches to a running program without the need for a full restart or interruption of service. Recently, there are hot-patching algorithms developed for mobile applications. For instance, WeChat, a renowned social communication application in China, employs the Tinker tool [27] for runtime self-repair. A critical component of Tinker is DexDiff, an differencing algorithm specifically designed for the dex format. However, its use is limited to small updates of dex files in hot-patching scenarios. It is not suitable for differencing (or incremental) updates for app stores because the updates in app store are relatively large and they involves updates for binary libraries and resource files.

2.3 Differencing algorithms besides mobile app updates

Recently, researchers have witnessed developments in differencing algorithms in the fields of program analysis [28], deep neural networks (DNN) [29], [30], and image processing [31], [32], etc. For example, QADroid [28] is a tool for

1. https://github.com/suntong30/sdiff

TABLE 1: Comparison of different differencing algorithms.

Algorithm	Pros	Cons	Main techniques
xdelta3 [15]	(1) The shortest differencing time (2) Low memory footprint in differencing	The delta size of large files are worst than bsdiff.	(1) Uses hash match to find same segments. (2) Uses source and target windows in differencing. (3) Performs LZMA [25] to compress the delta file.
bsdiff [16]	For most cases, the delta size is smaller than xdelta3.	(1) The highest memory footprint in differencing and reconstructing stage. (2) Long execution time	(1) Exploits suffix array by reading all files into memory to find same segments. (2) Uses approximate matching. (3) Uses bzip2 [26] to compress the delta file.
archive-patcher [18]	The delta size of APKs are smallest.	(1) High memory footprint (2) Long execution time	(1) Exploits decompressing-before-differencing (DBD) technique to find more similarity. (2) Uses an improved bsdiff algorithm.
HDiffPatch [17]	(1) For most cases, the delta size is smaller than xdelta3 and bsdiff. (2) The differencing time is shorter than bsdiff.	(1) Differencing memory footprint is lower than bsdiff but higher than xdelta3. (2) High CPU overhead in differencing.	(1) Exploits suffix array (the time and space complexity is better than bsdiff) by reading all files into memory to find same segments. (2) Uses approximate matching. (3) Uses zstd [33] to compress the delta file. (4) Exploits multithreading.

performing regression testing on Android programs. It identifies version differences by constructing call graphs based on FlowDroid and linking events. QD-Compressor [30] is a differencing algorithm aimed at DNN models. It performs quantizing-before-differencing technique on two similar DNN models, thus reduces the network transmission volume for distributing and the storage size of checkpoints. RIDDLE [31] is a differencing algorithm designed for Lidar data in autonomous driving to reduce data storage overhead. imDedup [32] is specifically for JPEG images. It improves JPEG similarities before differencing by first performing Huffman decoding on JPEG images. Each of these algorithms is specifically designed for the unique structures of their processing objects, showcasing distinct methodologies from those applied in mobile app updates.

3 BACKGROUND

In this section, we present relevant technical backgrounds.

3.1 APK file format

The Android application package (APK) file, which is really a ZIP archive file, holds all binary code, resources, and other data required by the Android application. It typically contains *(i)* Java binary codes (.dex); *(ii)* libraries in ELF format (.so); and *(iii)* other resource files such as .png and .jpg; *(iv)* the META-INF directory: contains files related to the signing and verification of the application; *(v)* AndroidManifest.xml: contains essential information about the application for the Android system. It declares the app's package name, permissions, activities, services, etc.

An APK (or ZIP) archive contains many file entries and a central directory. An individual file entry holds data for one file in the APK. The central directory consists of many file headers, which can be used to quickly obtain information about the file list in the APK file. A file header records important information about the corresponding file, e.g., the file's offset in the APK file, whether the data is compressed or uncompressed (i.e., in store mode).

3.2 Differencing algorithms

xdelta3. xdelta3 is a classical differencing algorithm that generates the delta between two files. It uses VCDIFF/RFC 3284 [34] streams, a standardized format for delta compression. There are three instruction types in its delta file, i.e., `copy`, `add`, and `run`. While the former two instructions are widely used in other delta files (for copying a segment from an old file or adding new bytes), the last one indicates that a given byte will be repeated for a given number of times. xdelta3 uses a sliding window mechanism for common segment matching, i.e., it sets up a target window in the new file and a corresponding source window in the old file for common segment matching. This mechanism can significantly reduce the memory consumption for delta generation.

xdelta3 can not only allow copying common segments from the old file but also copying common segments from a partially reconstructed new file. It also employs various techniques to compress the address fields before the default LZMA [25] compression algorithm is applied to the delta file. The xdelta3 algorithm is fast but may produce larger deltas than other algorithms.

bsdiff. The bsdiff algorithm is also a well-known differencing algorithm that focuses on achieving minimal delta size and is specifically optimized for executable files. The bsdiff algorithm uses its own custom delta file format. The bsdiff delta file is composed of four parts, i.e., header, ctrl block, diff block, and extra block. bsdiff generates instructions with the format `(x,y,z)` in the ctrl block: x denotes the length of bytes to copy, y denotes the length of bytes to add (the added bytes reside in the extra block), and z denotes the adjusted pointer address. The bsdiff algorithm uses a suffix array to find the longest common segments between the two files. During the suffix array generation, bsdiff uses qsufsort [35] algorithm to sort the suffixes in lexicographical order.

Unlike xdelta3, bsdiff allows approximate matches rather than exact matches, i.e., once two identical segments have been identified by bsdiff, these two segments are further extended, trying to find a match between similar and

longer segments. bsdiff allows a maximum of 50% dissimilarity between the segments, i.e., two segments are considered approximately matched if the number of mismatched bytes is smaller than 50% of the total segment length. The difference between approximately matched segments is encoded in the diff block. Approximate segment match is helpful in reducing the delta size because of two reasons. First, it reduces the number of copy instructions. Second, although additional differences between approximately matched segments are generated, they are compression-friendly, i.e., they can be easily compressed by general lossless compression algorithms such as LZMA [25] or bzip2 [26]. bsdiff uses the default bzip2 algorithm to compress the delta.

HDiffPatch. HDiffPatch is another open-source differencing algorithm that improves bsdiff in many aspects. First, the differences in the diff block are firstly encoded by Run-Length Encoding (RLE) [36]. Second, it uses the DivSufSort [37] algorithm in the suffix array generation process, resulting in better time and space performance compared with qsufsort employed in bsdiff. Third, it also allows approximate segment match, but the maximum allowed dissimilarity ratios differ for different segment lengths. Fourth, HDiffPatch employs a novel copy instruction merging technique. Assume there exist two copy instructions: (1) `copy(l1, x, y)`: copy l1 bytes from location x in old file to location y in the new file (2) `copy(l2, x+n, y+n)`: copy l2 bytes from location x+n in the old file to location y+n in the new file. HDiffPatch will try to merge these two copy instructions into one copy instruction copy(l1+l2+n, x, y), with the n different bytes encoded in the RLE Ctrl block using RLE. Fifth, it supports multi-threads in delta generation, e.g., it uses four threads for suffix sorting, common segment matching, and delta compression by default. Sixth, it employs *fast deduplication* before difference computation, i.e., it first records the identical blocks between the old and new files so that these identical blocks are not involved in the difference computation process. Lastly, it uses the zstd [33] algorithm for delta compression instead of the default bzip2 algorithm in bsdiff.

archive-patcher. archive-patcher is a differencing algorithm specifically designed for ZIP archives, including APK files. Many files in the APK file are compressed using the deflate algorithm [38]. It is beneficial to decompress them before difference computation because the similarity between the old and new files can be better preserved. archive-patcher uses such a technique, i.e., decompressing-before-differencing. A problematic issue is that the reconstructed new file needs to be re-compressed in the same manner as in the original APK file to pass the integrity check. To address this issue, archive-patcher enumerates all possible compression parameters and decompresses the compressed file only when it can be re-compressed identically using appropriate parameters. After decompression, archive-patcher uses an improved bsdiff algorithm for difference computation. As we will find in Section 5, the bsdiff algorithm used in archive-patcher improves the performance of the original bsdiff algorithm.

3.3 Comparison of the algorithms

Table 1 compares the algorithms we have described in the previous section. The pros, cons, and main techniques in Table 1 are concluded based on our evaluation results and code analysis.

3.3.1 Similarities

All the mentioned differencing algorithms operate at the byte level. They all use a mechanism to locate identical or similar segments (for the copy instruction). They all use lossless compression algorithms to compress the delta file further.

3.3.2 Differences

They also differ in some important ways.

Segment matching. xdelta3 uses sliding window while other algorithms read the entire old and new files into memory. xdelta3 uses a hash table to identify common segments and only allows exactly matched segments for copying. Both bsdiff and HDiffPatch use suffix arrays to identify common segments, allowing approximately matched segments for copying.

Compression of the delta. We notice that different differencing algorithms have used different lossless compression algorithms to compress the delta, e.g., xdelta3 uses LZMA [25], bsdiff uses bzip2 [26], and HDiffPatch uses zstd [33]. They have different trade-offs between compression/decompression speed and compression ratio [39], [40]. Typically, *(i)* LZMA tends to outperform bzip2 and zstd in terms of compression ratio but slower compression speed and higher memory usage. *(ii)* zstd tends to have faster compression and decompression speeds than LZMA and bzip2, while still achieving high compression ratios with tunable parameters. It also supports long-range search and deduplication for better performance on large files.

Other important aspects. archive-patcher uses decompressing before differencing for similarity preserving. HDiffPatch employs many other techniques, as we have described in the last subsection, to improve the performance.

4 METHODOLOGY

Figure 1 shows an overview of our methodology, which consists of five steps: (1) Getting APK files of different versions for a given list of apps. (2) Compiling different differencing algorithms. (3) Downloading all APK files to the server and computing the delta for each app update. (4) Dispatching a delta and the corresponding old APK file to a mobile device and reconstructing the new APK file. (5) Performance analysis.

Getting APK files. We first construct the APK dataset used in the study, including normal (i.e., non-gaming) and game apps. The category of normal apps encompasses 17 distinct subcategories, which include but are not limited to finance, social communications, and entertainment, etc. Specifically, we select 150 normal and 50 game apps with most downloads in a popular App Market up until Nov. 14, 2022. The motivation behind selecting apps with the highest downloads is that they are most important for key metrics such as server outbound bandwidth and users' QoE. Each app includes two consecutive recent versions, for a total of 400 APKs. There are a total of 200 update cases. Table 2 shows a selection of 15 representative application updates which are randomly chosen from our dataset. We

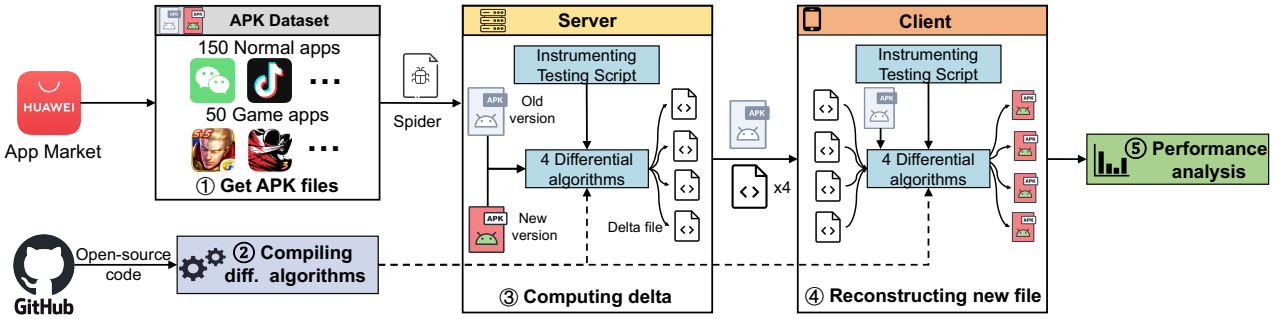


Fig. 1: Methodology overview. It consists of five main steps: (1) Getting APK files (2) Compiling differencing algorithms (3) Computing delta (3) Reconstructing new file (5) Performance analysis

TABLE 2: 15 Representative application updates. ⋆ indicates that it is a game application.

Name	Version	Old size (B)	New size (B)
WeChat	8.0.27→8.0.28	276,602,366	266,691,829
Baidu	13.19.5.10→13.21.0.11	137,200,701	137,805,661
Douyin	22.9.0→23.0.0	168,405,402	169,354,581
Weibo	12.10.2→12.11.0	206,500,685	207,162,286
Bilibili	7.3.0→7.4.0	102,429,902	101,626,272
QQ	8.9.15→8.9.18	311,322,716	307,940,064
Alipay	10.3.0.8000→10.3.10.8310	122,073,135	116,138,881
Youku	10.2.57→10.2.59	65,399,355	65,639,977
Zhihu	8.38.0→8.39.0	69,060,906	70,652,038
Jingdong	11.3.0→11.3.2	100,750,185	97,412,360
Harry Potter⋆	1.20.211450→1.20.212190	1,961,481,727	2,131,374,029
Honkai Impact 3⋆	6.0.0→6.1.0	613,738,862	634,074,809
PUBG Mobile⋆	1.19.3→1.20.13	2,056,739,892	2,037,120,844
Ace Racer⋆	4.0.6→4.1.0	2,030,659,990	2,032,064,158
Ninja Must Die 3⋆	2.0.19→2.0.20	1,813,459,454	1,832,093,657

TABLE 3: Compilers and optimization levels used for the differencing algorithms.

Algorithm	Version	Implem-entation	Compiler (Server)	Compiler (Mobile)	Optim-ization
xdelta3 [15]	3.1.0	C	gcc 11.3.0	Clang 14.0	-O3
bsdiff [16]	4.3	C	gcc 11.3.0	Clang 14.0	-O3
HDiffPatch [17]	4.4.0	C++	g++ 11.3.0	Clang++ 14.0	-O3
archive-patcher [18]	1.1	Java	OpenJDK 1.8.0	OpenJDK 1.8.0	/

investigate the reasons for updates and observe that most normal apps (e.g., WeChat, Baidu, and Alipay) typically update for bug fixes. Conversely, most gaming apps (Harry Potter and PUBG Mobile) update to add new features.

Compiling differencing algorithms. We download the source codes of different differencing algorithms and compile them on the server (CPU @2.10GHz with 20 cores, 16GB DDR4 RAM @3200 MT/s) with Ubuntu 22.04 LTS. Table 3 shows the compilers and the optimization levels we have used. Due to the Java implementation of archive-patcher, we use OpenJDK 1.8.0 to compile and execute Java bytecodes on the server. To run the archive-patcher on the mobile side, we first compile and archive the Java source code to a JAR file with OpenJDK 1.8.0. Then we use Android 10 build-tools dx to convert the JAR file into a .dex file. Finally, we run the .dex file of archive-patcher with Dalvik VM on the mobile phone.

Computing delta at the server side. All APK files are downloaded to the server, where different algorithms compute the delta files. Without otherwise specified, we use the default parameter settings in these algorithms. For example, we use the default source window size of 64MB and target window size of 8MB in xdelta3. We also use a default thread number of four in HDiffPatch.

For performance measurements, we write scripts that use the tool of `/usr/bin/time` to capture various metrics related to algorithm execution, including user time, system time, and maximum resident set size. The execution time of the differencing algorithm is calculated as the sum of user and system time, while the maximum resident set size provides insight into peak memory usage. For assessing average memory usage, we utilize the `mprof` tool, which records memory consumption at regular 100ms intervals.

Reconstructing the new file at the mobile side. We run reconstructing algorithms on a mobile phone, ZTE Axon 10 (Snapdragon 855 CPU@2.8GHz with 8 cores, 6GB RAM). The mobile device uses Android 10 with kernel 4.14.117. After receiving a delta and the corresponding old APK file, the mobile device reconstructs the new APK file using the compiled algorithms. We use the `simpleperf` to record the task clocks of the CPU. We also employ the `mprof` tool that records memory usage at regular 10ms intervals.

To minimize interference from other background tasks running on either the server or the mobile, we clear the extraneous background tasks of the device before each evaluation. Besides, we disable wireless and cellular connections to avoid influencing the mobile device. To ensure that the results on the phone are not impacted by the throttle, we lock the CPU frequency with the same value for each tested algorithm. In addition, we conduct experiments under conditions of ventilation and heat dissipation. An external radiator is also attached to the back cover of the mobile phone to enhance cooling. Furthermore, we ensure there were sufficient time gaps between each test, setting a minimum interval of one minute. During the experiment, we monitor the mobile phone's temperature to asses heating.

Performance analysis. We analyze the performance of differencing algorithms in terms of five 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.
- *Differencing time*. It is time to generate a delta on the server side.
- *Differencing memory overhead*. We measure both average and peak memory usage during the differencing phase

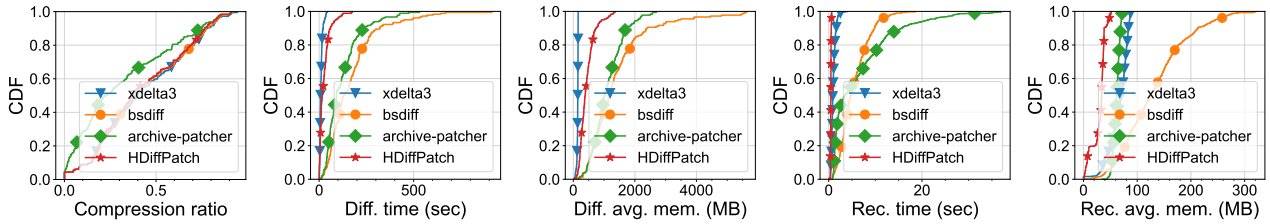


Fig. 2: Comparison of different algorithms in terms of five key metrics for 200 app updates.

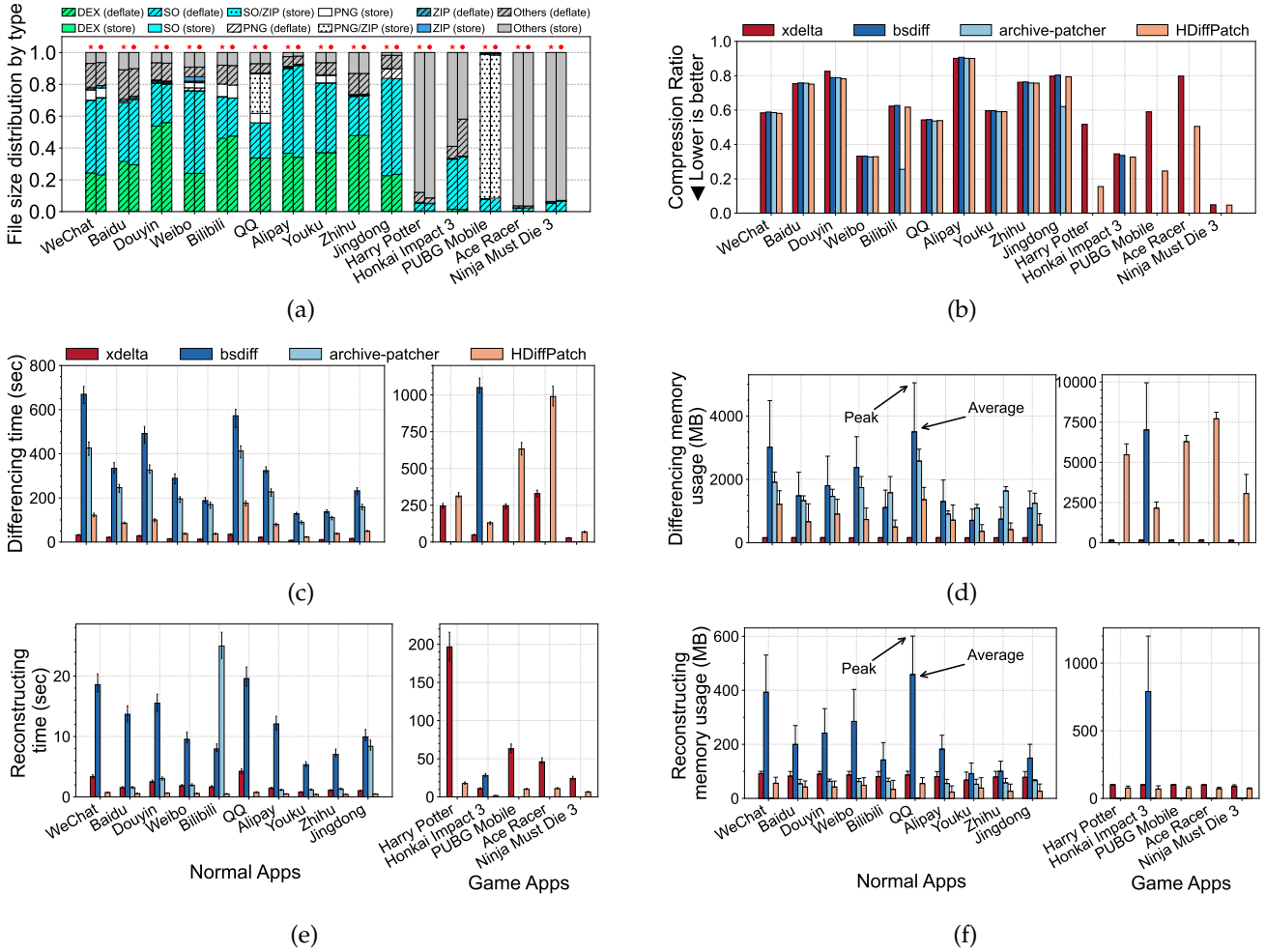


Fig. 3: APK file decomposition and overall performance in terms of five key metrics for 15 representative app updates. (a) ⋆ indicates the old version, and • indicates the new version. The A/B means the file extension name is A, but the actual file type (according to the magic number) is B. (b) Compression ratio. (c) Differencing time. (d) Differencing memory consumption. (e) Reconstructing time. (f) Reconstructing memory consumption.

on the server side.

- *Reconstruction time*. It is time to reconstruct the new file from the delta and the corresponding old file on the mobile side.
- *Reconstruction memory overhead*. We measure both average and peak memory usage during the reconstruction phase on the mobile side.

5 Overall Measurement Results

Figure 2 compares four differencing algorithms with respect to the five metrics. We can see that: *(i)* archive-patcher is much better than other algorithms in terms of compression ratio, while other algorithms achieves similar performance in terms of compression ratio. *(ii)* At the differencing stage, the average memory usage of xdelta3 is universally better than other algorithms at the differencing stage, due to its sliding window mechanism. *(iii)* At the reconstruction stage, the execution times of HDiffPatch and xdelta3 are similar, while the average memory overhead of HDiffPatch is smaller than xdelta3.

To carefully study how each algorithm performs on each individual application, we show the detailed performance for 15 representative applications. We conduct 10 runs on

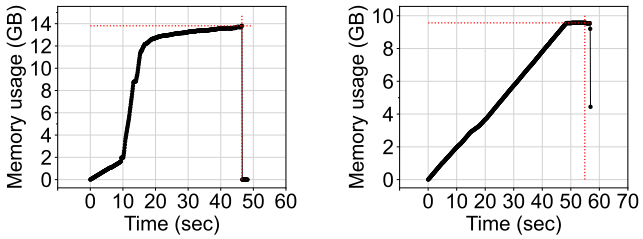
the same device and environments and take the average value of multiple runs. The error bars in Figure 3(c) and Figure 3(e) are 95% confidence intervals under the student's t-distribution. Figure 3(a) shows the fraction of different file types in the APK files for 15 representative applications. Three observations can be made: *(i)* Normal applications are mainly composed of .dex, .so, and .png files, while game applications are quite different: we find that a large portion of game applications are specific resource files, e.g., .mpk files and .npk files. *(ii)* A large component of the applications (especially for normal applications) is compressed using the deflate algorithm. *(iii)* Occasionally, a file extension may not precisely indicate the actual file type. For example, some files with the extension .png are actually ZIP files, as indicated by their file magic number.

Figure 3(b) shows the compression ratio for representative app updates. We can see that *(i)* For normal applications, the compression ratios of xdelta3, bsdiff, and HDiffPatch are comparable. However, for game applications, HDiffPatch outperforms xdelta3 and bsdiff significantly. *(ii)* bsdiff and archive-patcher encounter failures in some cases, e.g., for Harry Potter, PUBG Mobile, and Ace Racer. *(iii)* archive-patcher is substantially better than other algorithms for some applications, such as Bilibili and Jingdong.

Figure 3(c) shows the differencing time at the server side. The results show that: *(i)* xdelta3's differencing time is significantly shorter than other algorithms mainly due to its sliding window and hash-based matching mechanism. *(ii)* HDiffPatch is better than both bsdiff and archive-patcher due to its additional mechanisms, e.g., fast deduplication and multithreading. *(iii)* it is surprising that archive-patcher is even better than bsdiff as archive-patcher uses bsdiff for difference computation after it decompresses files in APK. After a closer look, we find that archive-patcher uses an optimized version of bsdiff, e.g., it uses the DivSufSort [37] suffix array sorting algorithm, which is faster than the qsufsort [35] algorithm used by the original bsdiff algorithm.

Figure 3(d) compares the differencing memory consumption of different algorithms. We have the following observations: *(i)* xdelta3 has the lowest memory usage because it utilizes a sliding window mechanism. *(ii)* The lower memory usage of HDiffPatch compared to bsdiff and archive-patcher is mainly to its use of fast deduplication. *(iii)* For the WeChat app, the memory usage of archive-patcher is lower than bsdiff, while the opposite is true for the Zhihu app. It is worth noting that there are two contradicting factors. One factor is that archive-patcher's decompressing-before-differencing technique increases the input size for difference computation. The other factor is that archive-patcher uses an optimized bsdiff with a lower space complexity than the original bsdiff. Therefore, the ultimate outcome is determined by which of two factors plays a more important role.

Figure 3(e) compares the reconstruction time of different algorithms at the mobile side. Our observations are as follows: *(i)* HDiffPatch exhibits the shortest execution time on mobile devices, since it uses much fewer instructions in the delta file. *(ii)* xdelta3 demonstrates significantly slower reconstruction time for large files, especially for the Harry Potter game app. *(iii)* archive-patcher encounters failures during the reconstruction process on mobile devices, such as WeChat and QQ applications.



(a) (b)

Fig. 4: Memory consumption during delta generation. (a) bsdiff for the PUBG Mobie app. (b) HDiffPatch for artificially constructed files.

Figure 3(f) shows the average and peak memory consumption of reconstruction for different algorithms. We can observe that bsdiff has significantly higher memory usage for reconstructing all applications than other algorithms, especially for WeChat, QQ, and Honkai Impact 3 apps. This is because bsdiff reads the entire delta files into memory for reconstruction. All other algorithms allocate a fixed buffer for stream-based reconstruction, effectively reducing the memory usage at the mobile side.

6 Deeper Analysis

In this section, we aim to answer the underlying reasons for some important research questions (RQs) revealed by the observations described in the previous section.

- *RQ1* (Section 6.1): Why do bsdiff and archive-patcher fail in some cases in the differencing stage, and why archive-patcher fails in some cases in the reconstruction stage?
- *RQ2* (Section 6.2): While the sliding window mechanism effectively reduces the differencing memory consumption, it may sacrifice performance in terms of the compression ratio. How to appropriately set the window size?
- *RQ3* (Section 6.3): Why does the decompressing-before-differencing technique employed in archive-patcher result in a significantly better compression ratio than other algorithms? Why is it less effective for some other applications?
- *RQ4* (Section 6.4): Why HDiffPatch has a consistently better compression ratio than bsdiff?

We intend to answer these questions in the following subsections, respectively.

6.1 Failure analysis

Server-side bsdiff failure analysis. In Figure 3(b), we find that bsdiff fails for four game applications. To investigate the underlying reasons, we consider the PUBG Mobile app as an example. Figure 4(a) shows bsdiff's memory consumption during its delta generation stage. We find that at the time of 50 sec, bsdiff has already consumed ∼14GB of memory and is still requesting more memory from the system, exceeding the maximum memory allowed for a single program in our operating system and is thus directly killed by the system.

The underlying reason is that bsdiff reads the entire old and new files into memory during delta generation. This may cause failures due to excessive memory consumption, especially for game apps with large APK files.

Server-side HDiffPatch potential failure analysis. We note that HDiffPatch adopts a similar approach of reading the entire old and new files. However, it does not cause failures in our cases. This is due to additional techniques, such as fast deduplication, which mitigate memory consumption in the delta generation process. For huge files, we conjecture that HDiffPatch may also fail due to the same reason. We manually construct 10GB of old and 10GB of new files and let HDiffPatch generate the delta. Figure 4(b) shows the memory consumption during the execution. We notice that at around 60 seconds, HDiffPatch already consumes 10GB of memory. Hence, the algorithm fails when it requests additional memory afterward.

Server-side archive-patcher failure analysis. For archive-patcher, it is expected that it fails when bsdiff fails since archive-patcher employs bsdiff for difference computation. Figure 3(b) shows that bsdiff succeeds and archive-patcher fails for one game application (Honkai Impact 3 app). After a careful check, we find that this was due to the implementation of the Google source code, which checks the size of the decompressed files and throws an exception if they exceed 512 MB. This threshold should be relaxed for game applications to ensure the successful execution of the differencing algorithm on the server side.

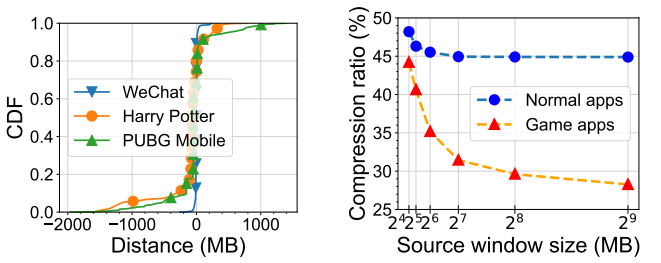
Mobile side archive-patcher failure analysis. Figure 3(e) shows that archive-patcher encounters failures during the reconstruction process on mobile devices, even for normal applications, such as WeChat and QQ. After a careful check, we find that the algorithm places the temporally reconstructed new files in the default `/tmp/` directory [41], which occasionally encounters failures when there is no sufficient space. To address this problem, we can modify the source code to place the temporal files in another dedicated directory with sufficient space and remove them when they are no longer needed.

Insight ❶: *While some failures can be addressed by either relaxing the threshold or modifying the temporal directory, it requires a more systematic approach to addressing failures caused by excessive memory consumption during delta generation. We believe that a sliding window mechanism (like the one used in xdelta3) is necessary to limit the maximum memory usage during delta generation.*

6.2 The sliding window mechanism

While the sliding window mechanism can help address the excessive memory usage during the delta generation process, it inevitably sacrifices the compression ratio since common segments may not be found outside the window. Therefore, how to set up an appropriate window size becomes a critical issue.

To answer this question, we employ the common segment matching algorithm used in HDiffPatch, i.e., reading the entire old and new files into memory and using a suffix array to find the common segments. We investigate the relative distance of common segments, i.e., the difference between the starting address of the segment in the new file and the starting address of the corresponding segment in the old file. Figure 5(a) shows the CDF of the distances for all common segments. We find that common segments have locality, and normal application has better locality than game applications. For example, 93.8% of common segments in WeChat are within a 64MB (i.e., [-32MB, 32MB]) window; 97.2% of common segments in Honkai Impact 3 are within a 128MB (i.e., [-64MB, 64MB]) window.



(a) (b)

Fig. 5: (a) CDF of the distances for all common segments. (b) xdelta3's average compression ratios with different window sizes.

To further investigate which size the sliding window should be, we set different window sizes in the xdelta3 algorithm. Figure 5(b) shows the average compression ratio with different window sizes for 200 app updates. We can see that: *(i)* For normal applications, a window size of 64MB is appropriate since a larger window size would not significantly reduce the delta size; *(ii)* For game applications, a windows size of 128MB is appropriate for the same reason.

Insight ❷: *Common segments have locality, and normal applications have better locality than game applications. A typical window size of 128MB is appropriate since a larger one would not significantly reduce the delta size.*

6.3 Decompressing before differencing

The archive-patcher algorithm employs the technique of decompressing-before-differencing, and we find that it significantly reduces the delta sizes for some apps, e.g., Bilibili and Jingdong. To investigate the underlying reasons, we calculate the decompression ratio, which is defined as the ratio between the number of bytes that can be decompressed and the total number of bytes that are compressed in an APK file. Note that not all bytes can be decompressed since they should be re-compressed in the same manner on the mobile side. Otherwise, they cannot pass the integrity check.

Figure 6 shows the decompression ratios for the representative applications. We make three observations. *(i)* For normal applications, only Bilibili and Jingdong show a high decompression ratio. This observation is consistent with the results observed in Figure 3(b), where archive-patcher results in significantly smaller delta sizes. *(ii)* For other normal apps, the decompression ratios are low. This indicates that a large portion of compressed files cannot be decompressed because the specific compression parameters (e.g., compression levels) cannot be guessed using the current zlib library employed in archive-patcher (i.e., `Java.util.zip`). *(iii)* For the game apps, although the decompression ratio

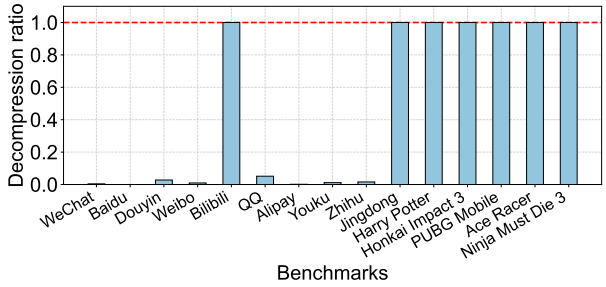


Fig. 6: archive-patcher's decompression ratios for representative apps.

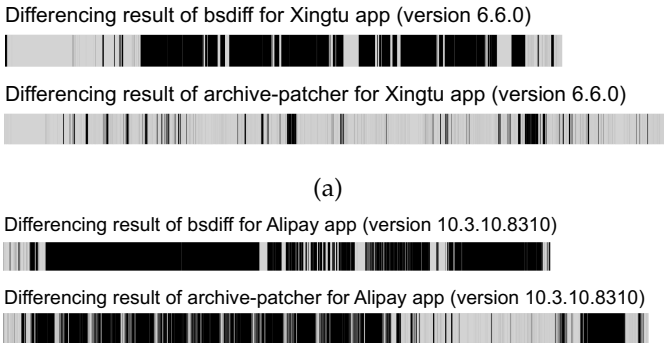


Fig. 7: Differencing results of bsdiff and archive-patcher for two app updates. In the rectangle, the added bytes are indicated in black color, and the copied bytes are indicated in grey color. (a) The case where decompressing-before-differencing is beneficial. (b) The case where decompresing-before-differencing is not beneficial.

is high, archive-patcher does not show benefits because it encounters failures due to reasons described in Section 6.1.

Decompressing-before-differencing results in a smaller delta file since it can preserve a significant similarity between the old and new files. Figure 7(a) shows a case where the top rectangle visualizes differencing results of bsdiff and the bottom rectangle visualizes differencing results of archive-patcher. In the rectangle, the added bytes are indicated in black color, and the copied bytes are indicated in grey color. We can see that: *(i)* the number of added bytes is relatively high when comparing the original APK files; *(ii)* the number of added bytes drastically decreases when comparing decompressed APK files. On the other hand, decompressing-before-differencing is not always effective. Figure 7(b) shows such a case. We can see that the number of added bytes remains almost the same whether the files in APK are decompressed or not.

To quantify the benefits of decompressing, we compare the sizes of delta generated by bsdiff and archive-patcher for a pair of compressed files (e.g., for a compressed .dex) with the same name in the APK files. Specifically, the number of reduced bytes for a pair of files RB is defined as the difference between the delta size generated by bsdiff and the delta size generated by archive-patcher. A positive value of RB indicates that decompressing-before-differencing is beneficial, while a negative value of RB indicates that this technique is not beneficial. We calculate the number of reduced bytes for a total of 318,221 updated pairs of

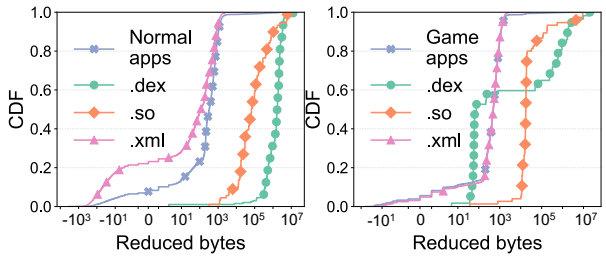


Fig. 8: CDF of reduced bytes (RB). (a) normal apps (b) game apps.

compressed files in our APK files. Figure 8 shows the CDF of the reduced bytes for normal apps and game apps. We can see that 8.3% (for normal apps) and 5.6% (for game apps) of the files exhibit negative benefits when decompressing-before-differencing is applied. One possible reason is that the modifications made to the file are too substantial, resulting in small similarity even if they are decompressed.

It is also worth noting that decompressing will cause additional overhead on both the server and mobile sides. On the server side, the algorithm should guess the possible compression parameters and decompress the compressed file only when it can be re-compressed identically using the guessed parameters. On the mobile side, the algorithm should re-compress the newly constructed files if they are decompressed on the server side. Considering these factors, a file may not deserve decompression if it shows a small improvement in the number of reduced bytes.

Insight ❸: *Decompressing-before-differencing can preserve the file similarity and effectively reduce delta file size. On one hand, a large portion of files is left uncompressed in APK files, indicating that the potential of this technique is not fully exploited. On the other hand, decompressing is not always beneficial.*

6.4 Copying instruction merging

We observe that HDiffPatch consistently outperforms bsdiff in terms of delta size. For some applications, the reduction in delta size is significant.

To investigate the underlying reasons, we analyze different components in the delta files. The delta formats of bsdiff and HDiffPatch are similar. There are ctrl block, diff block, and extra block. The ctrl block stores instructions, the diff block stores the difference between the new segment and the copied segment, and this difference is encoded using a RLE scheme, and the extra block stores the added bytes.

Figure 9(a) shows the difference between bsdiff and HDiffPatch for three different components described above. A positive value indicates that HDiffPatch yields a smaller component size. We can see that in our representative applications, the ctrl blocks generated by HDiffPatch are consistently smaller than those generated by bsdiff, while this is not necessarily true for the remaining two blocks. As long as the sum of the three components remains positive, HDiffPatch yields a smaller delta file.

Figure 9(b) shows the number of copy instructions in the ctrl block for bsdiff and HDiffPatch, respectively. We can see that HDiffPatch results in a significant reduction in the number of copy instructions: The instruction count of bsdiff is ∼2x that of HDiffPatch. After a careful study of the internal mechanisms of HDiffPatch, we find that HDiffPatch

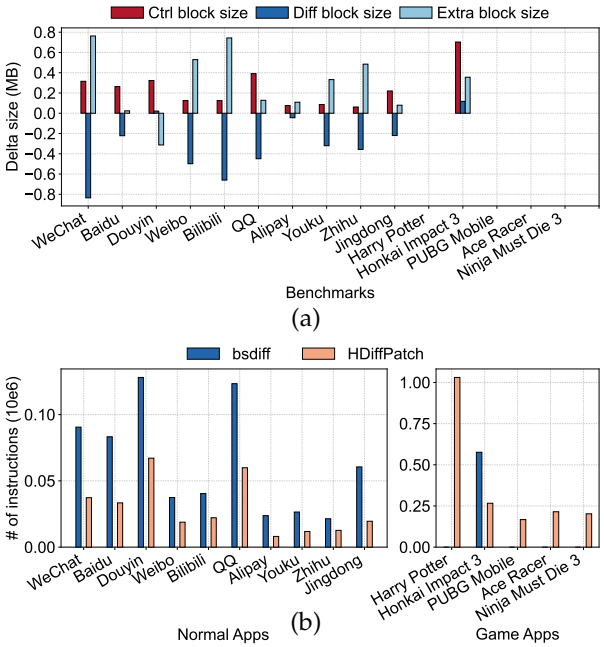


Fig. 9: (a) the difference between bsdiff and HDiffPatch for three different components in delta file. A positive value indicates that HDiffPatch yields a smaller component size. (b) the number of copy instructions in the ctrl block.

employs a copy instruction merging technique to reduce the number of instructions to reduce the final delta file size.

HDiffPatch calculates a cost for reconstructing the new file, which can be approximately regarded as the delta size without final compression. We use the example shown in Figure 10 to explain the copy instruction merging technique used in HDiffPatch. This technique works as follows:

(1) Find common segments. In Figure 10(a), the algorithm has identified three common segments in the orange color. Calculate the cost of directly copying these segments and adding the rest of bytes. We can assume the cost of copy instruction itself is three bytes (see details of HDiffPatch's copy instruction in Section 3.2). The cost of add instruction itself is none because we can identify what data is carried by the add instruction only by the length of the copy instruction. Additionally, the cost of data in copy instruction is the encoded size after RLE processing while in add instruction is the added byte size. As shown in Figure 10(a), the total cost is (3+2)+3+(3+2)+12+(3+2)=30 bytes. Note that HDiffPatch (as well as bsdiff) can copy not only identical segments but also similar segments, and the additional segment difference (encoded using 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) is 2 bytes when RLE is applied (i.e., five zeros, one byte encodes five and another byte encodes zero).

(2) For each common segment, seek for opportunities whether it can be merged with the next copy instruction. In Figure 10(b), the algorithm tries to merge the first copy instruction and the second copy instruction. HDiffPatch will perform this operation since the newly calculated cost is (3+6)+12+(3+2)=26 bytes, smaller than without this merging operation.

(3) Repeat step (2) until there is no improvement. Figure 10(c) shows that the algorithm tries to merge the first copy instruction and the last copy instruction. Note that merging of two copy instructions is only possible when the lengths of bytes in between are identical in the old and new files. Therefore, HDiffPatch needs to move the first common segment right for one byte in the new file, as illustrated in Figure 10(c). Finally, HDiffPatch ends up in the situation illustrated in Figure 10(d), with one added byte and one large copy instruction. The final cost is 1+(3+20)=24 bytes.

However, we find that HDiffPatch does not necessarily generate the optimal results after its copy instruction merging mechanism. Figure 10(right) shows such a situation. A better result can be achieved by trying to merge the second and third copy instructions. For this purpose, the second common segment needs to be moved to the right for one byte in the new file in the first place as illustrated in Figure 10(e). In this situation, the final cost is (3+2)+4+(3+4)=16 bytes after merging, as illustrated in Figure 10(f). Note that HDiffPatch cannot find this solution since it adopts a greedy algorithm that sequentially checks whether a copy instruction can be merged with the next copy instruction.

Insight ❹: *HDiffPatch outperforms bsdiff mainly due to the technique of copy instruction merging. However, the current merging algorithm in HDiffPatch is still not optimal.*

7 New Algorithm with Smaller Compression Ratio

In this section, we demonstrate that we can easily build a new algorithm with a smaller compression ratio, guided by the insights revealed by our measurement study. Our algorithm, called **sdiff**, combines several existing techniques for reducing the delta size. First, it uses the fast deduplication technique used in HDiffPatch to preprocess the old and new files so as to improve the time and memory performance in the differencing stage. Second, it uses the sliding window mechanism employed in xdelta3 to limit the maximum memory usage so as to avoid possible failures during delta generation as well as reducing the memory in the differencing stage. Third, it adopts the decompressing-before-differencing technique employed in archive-patcher since this technique preserves the similarity between old and new files. Finally, it uses the HDiffPatch differencing algorithm since it performs consistently better than other algorithms such as xdelta3 and bsdiff.

Algorithm 1 and Algorithm 2 show the pseudocode for delta generation and new file reconstruction. In Algorithm 1, we first get a list of files which have been updated. We then perform fast deduplication to identify identical blocks between old and new files. Note that we cannot remove blocks within compressed files in the new APK. Otherwise, these files cannot be decompressed correctly. Afterwards, we use the perform difference computation within the sliding windows: we first try to decompress the files if it is possible; we then reuse the HDiffPatch algorithm to generate the difference within the sliding window. Finally, we write metadata to the delta and reuse zstd to further compress the delta file. Algorithm 2 reconstructs the new APK file using the old APK file and the delta.

We have also conducted experiments to compare our new algorithm with other algorithms we have studied in

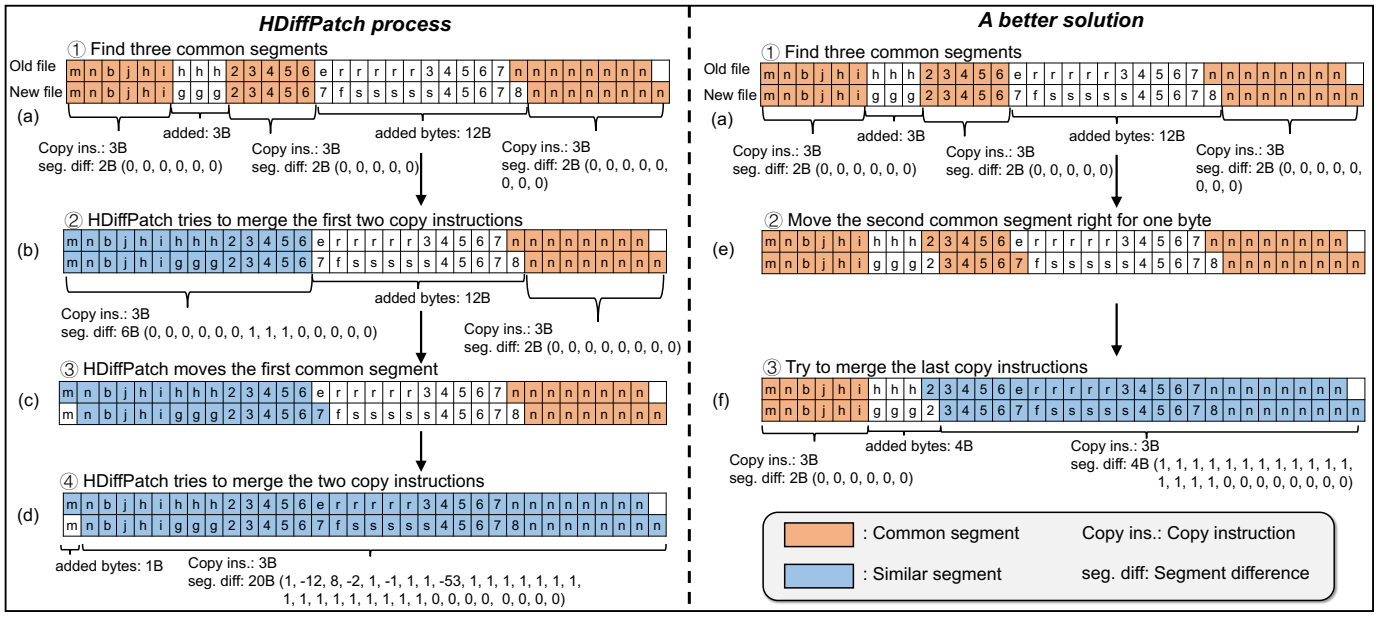


Fig. 10: The process of copy instruction merging. The left flow in the figure (a→b→c→d) illustrates the HDiffPatch process, while the right flow in the figure (a→e→f) represents a new approach that yields a better result.

Algorithm 1: sdiff (differencing phase)

Input : old source APK A_{old}, new target APK A_{new}, source window size S_{old}, target window size S_{new}
Output: delta file Δ

// Get a list of pairs of files in APK that have changed.
$L \leftarrow \{(f_{\text{old}}(i), f_{\text{new}}(i))\}_{i=1}^{n}$, $f_{\text{new}}(i) \in A_{\text{new}}$ and $f_{\text{old}}(i)$ is the corresponding file in A_{old}, $f_{\text{new}}(i) \neq f_{\text{old}}(i)$
Perform fast deduplication on A_{old} and A_{new} except for bytes in $f_{\text{new}}(i)$ that is compressed, write copy instructions to Δ.
$W_{\text{new}} \leftarrow [0, S_{\text{new}}]$
while *W_{new} does not reach the end of A_{new}* **do**
  Find corresponding W_{old}
  $B_{\text{new}} \leftarrow$ bytes within W_{new} in A_{new}
  $B_{\text{old}} \leftarrow$ bytes within W_{old} in A_{old}
  for *each file f_{new} whose start offset is in W_{new}* **do**
    if *$f_{new} \in L$ and f_{new} is compressed* **then**
      try to decompress f_{new} and update B_{new} with decompressed f_{new}
      if *f_{old} is compressed* **then**
        decompress f_{old} and update B_{old} with decompressed f_{old}
  Perform HDiffPatch (without zstd) between B_{old} and B_{new} and write instructions to Δ
  W_{new} moves right for S_{new} bytes
Add metadata L_D to Δ where L_D denotes the set of files in A_{old} that require decompression during reconstruction.
Add metadata L_C to Δ where L_C denotes the set of files that require re-compression during reconstruction.
Perform `zstd` compression on Δ

Algorithm 2: spatch (reconstruction phase)

Input : old source APK A_{old}, delta file Δ
Output: new target APK A_{new}

Perform `zstd` decompression on Δ
From Δ, get file list L_D in A_{old} in which the files require decompression
From Δ, get file list L_C in which the files require re-compression
for *each file f_{old} in A_{old}* **do**
  if *f_{old} in L_D* **then**
    update A_{old} with decompressed f_{old}
for *each instruction i in Δ* **do**
  reconstruct A_{new} according to i
for *each f_{new} in L_C* **do**
  update A_{new} with compressed f_{new}

this paper. We evaluate sdiff with 200 app updates using the same methodology (see details in Section 4). We have used a sufficiently large window size of 500MB so as to mitigate its negative impact on the compression ratio. Table 4 shows the evaluation results for 200 app updates with respect to five metrics described in Section 4. We can see that: *(i)* sdiff achieves the best compression ratio, i.e., 7.8% reduction compared with the best state of the art, archive-patcher. *(ii)* sdiff uses the smallest memory during delta generation since it utilizes the sliding window mechanism and fast deduplication techniques. *(iii)* sdiff has a smaller reconstruction time compared with archive-patcher due to its use of the C version of zlib for fast decompression. *(iv)* Compared with archive-patcher, sdiff significantly reduces the differencing time and memory overhead as well as the

TABLE 4: Comparison of sdiff with four other algorithms for 200 app updates. M1: Compression ratio. M2: Differencing time. M3: Differencing memory usage. M4: Reconstructing time. M5: Reconstructing memory usage.

Algorithm	Metric				
	M1	M2	M3	M4	M5
xdelta3	43.28%	**10.02 sec**	139.51 MB	0.99 sec	64.91 MB
bsdiff	42.57%	170.12 sec	1414.40 MB	5.46 sec	131.29 MB
archive-patcher	31.58%	123.20 sec	1106.22 MB	6.56 sec	61.35 MB
HDiffPatch	42.00%	27.17 sec	440.14 MB	**0.46 sec**	**28.48 MB**
sdiff (Ours)	**28.99%**	32.23 sec	**129.54 MB**	2.40 sec	47.44 MB

TABLE 5: Cost of client-side processing power for updating the Weibo application from version 12.10.2 to 12.11.0. The delta size generated by sdiff is 67.0MB, and the new version size is 207.2MB. The battery capacity of our phone is 4000mAh.

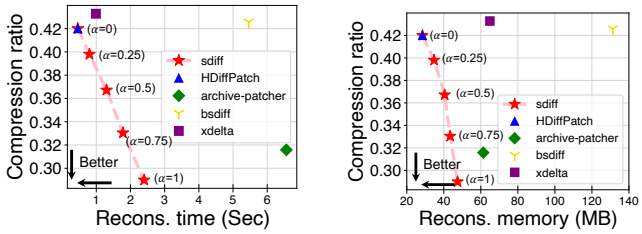
Differencing algorithm	Network type	Energy consumption			% of Battery consumption
		Transmission	Reconstruction	Total	
sdiff	5G	67MB*5.85*10^{-8}mAh/bit = 31.356 mAh	0.0416mAh	31.40mAh	0.79
	WiFi	67MB*2.12*10^{-8}mAh/bit = 11.363mAh	0.0416mAh	11.30mAh	0.28
None	5G	207.2MB*5.85*10^{-8}mAh/bit = 96.970mAh	0	96.97mAh	2.42
	WiFi	207.2MB*2.12*10^{-8}mAh/bit = 35.141mAh	0	35.14mAh	0.88

reconstruction time and memory overhead.

The overall cost of sdiff can be assessed through two aspects: latency and processing power. We first analyze the end-to-end latency. The end-to-end latency of sdiff consists of downloading time and reconstruction time. The downloading time depends on the network bandwidth. For example, the delta size of two Weibo application versions (v12.10.2→v12.11.0) which is generated by sdiff is 67.0MB, and the size of the new version (v12.11.0) is 207.2MB. The reconstruction time of this update on the mobile is 1.1 seconds. Thus, in this case, if the downloading speed is less than 129 MB/s, the end-to-end latency of sdiff is less than transmitting the whole APK. In practice, most users' network download speeds cannot reach it.

The overall cost of processing power has two aspects: the service provider and the client. For the service processing power, the cost of the server performing a differencing is low. This is because the server only needs to generate the delta file once for an old version, and all users who hold that old version can update it to the latest version. However, the CDN cost can be significantly reduced by distributing delta files. The client-side processing power consists of network transmitting power and reconstruction power. For users, the cost of charging phones energy is negligible. They care more about mobile phone battery life. Thus, we are concerned about the battery consumption of phones. We utilize the Android `batterystats` tool to measure the energy used during the Weibo app's update from version 12.10.2 to 12.11.0. The reconstruction battery cost of the spatch is 0.0416mAh. The battery capability of our phone is 4000mAh, thus the reconstruction battery consumption of sdiff is approximately 0.001%. We then calculate the network transmitting power. The average energy efficiency of our phone's 5G transmission is about 0.8 uJ/bit [42], while WiFi's efficiency is typically 0.29 uJ/bit [43]. Thus, we calculate the total cost of client-side processing power, as shown in Table 5. We can observe that: *(i)* Compared to without differencing, using sdiff can save battery consumption at 67.4% and 68.2% with 5G and WiFi, respectively. *(ii)* During the application update process, the main energy consumption is network transmission, while the sdiff reconstruction process only accounts for less than 0.1% of the energy consumption in the entire update process.

There is a trade-off between the user's QoE (Quality of Experience) and the server's CDN cost. The provider hopes that after performing the differencing, the user's QoE will not be reduced, but the CDN cost can be saved significantly. To better tradeoff that, we extend our sdiff with a tunable hyperparameter α. We call it a flexible sdiff. The $\alpha \in [0, 1]$ is an additional input parameter in the differencing stage to selectively decompress files before differencing. We evaluate the flexible sdiff with 200 app updates using the same methodology. Specifically, We set different α parameters, respectively 0, 0.25, 0.5, 0.75, and 1. The results are shown in Figure 11. We can observe that: *(i)* as α increases, the reconstruction time and reconstruction memory of sdiff also increase. *(ii)* When $\alpha = 0$, the sdiff algorithm can achieve the same reconstruction performance as the HDiffPatch algorithm, i.e., the sdiff algorithm achieves the same reconstruction time and memory reconstruction as HDiffPatch, making it more suitable for low-end mobile devices. *(iii)* When $\alpha = 1$, the sdiff algorithm can achieve the lowest compression ratio, thereby producing the smallest delta size, which is more suitable for powerful mobile devices.



(a) (b)

Fig. 11: Comparison of flexible sdiff with four other algorithms for 200 app updates. (a) compression ratio w.r.t. reconstruction time. (b) compression ratio w.r.t. reconstruction memory.

8 Discussion

In this section, we summarize the important observations and implications as well as discussing possible future directions.

Observation ①: Our results show that both bsdiff and archive-patcher fail in generating delta files for large APKs. In addition, archive-patcher fails for some cases during reconstruction at the mobile side.

Implication ①: While some failures can be addressed by either relaxing the threshold or modifying the temporal directory, it requires a more systematic approach to addressing failures caused by excessive memory consumption during delta generation. We believe that a sliding window mechanism (like the one used in xdelta3) is necessary to limit the maximum memory usage during delta generation.

Observation ②: A non-negligible portion of APK files is compressed. The archive-patcher algorithm utilizes the technique of decompressing-before-differencing, yielding a smaller compression ratio since it preserves a higher similarity. Our results also show that an average of 24% of bytes in the APK cannot be decompressed with the current zlib [23]

tool used in archive-patcher. Otherwise, they cannot be recompressed in the same manner at the mobile side.

Implication ②: This result indicates that the potential of decompressing-before-differencing is not fully exploited. For example, it is possible to further improve the performance by using more sophisticated algorithms such as 7zip.

Observation ③: HDiffPatch performs consistently better than bsdiff, especially for large files.

Implication ③: It is possible to combine HDiffPatch with other techniques such as sliding window and decompressing-before-differencing to devise a better algorithm. For illustrative purposes, we have proposed sdiff in Section 7. Our algorithm exhibited a notable 7.8% performance enhancement compared with the state-of-the-art algorithm.

We have also made other observations that could lead to multiple future directions that are worthy of pursuing. *(i)* We observe that decompressing-before-differencing exhibits negative benefits for some files in the APK. Considering that decompressing will cause additional overhead at both the server side and the mobile side, it is worth studying a model to quantify the benefits and decompressing a selected set of files for performance improvement. *(ii)* We find that the current copy instruction merging algorithm used in HDiffPatch is still not optimal. It is thus valuable to study the optimality problem and to design a better algorithm in the future. *(iii)* We find that current compression tools have not added compression parameters to their compressed files because they are only concerned about compression and decompression instead of differencing. The tools are forced to guess at the compression parameters of archives to find a reversible recompression. Interestingly, we find that only minor modifications can make differencing algorithms avoid guessing parameters. For example, the APK includes an extra field starting from the 30th byte plus 'n' bytes, where 'n' denotes the file name's length, in each local file header. This field presents an opportunity for adding compression information. By adjusting existing compression tools to inscribe compression parameters (e.g., compression level) into this field, App Markets could implement a reversible transformation process more easily. This approach will slightly increase APK size due to encoding compression parameters into APK. It can help differencing algorithms utilizing the decompressing-before-differencing technique to potentially reduce delta file size because all files' compression parameters are known.

9 Conclusion

This paper conducts a systematic study to understand the performance of four widely used differencing algorithms for mobile application updates. We evaluate these algorithms, i.e., xdelta3, bsdiff, archive-patcher, and HDiffPatch with respect to five key metrics including compression ratio, differencing/reconstruction time and memory overhead for 200 application updates. Our findings provide important insights for developers to further optimize for performance improvement. Guided by these insights, we have also provided a new algorithm sdiff, with the smallest compression ratio compared with existing algorithms.

References

[1] 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.

[2] X. Teng, D. Guo, Y. Guo, X. Zhao, and Z. Liu, "SISE: Self-Updating of Indoor Semantic Floorplans for General Entities," *IEEE Transactions on Mobile Computing*, vol. 17, no. 11, pp. 2646–2659, 2018.

[3] X. Zheng, Z. Cai, J. Li, and H. Gao, "A Study on Application-Aware Scheduling in Wireless Networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 7, pp. 1787–1801, 2016.

[4] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression," *J. ACM*, vol. 49, no. 3, p. 318–367, May 2002.

[5] M. J. May, "Donag: Generating efficient patches and diffs for compressed archives," *ACM Transactions on Storage*, vol. 18, no. 3, pp. 1–41, 2022.

[6] R. Burns, L. Stockmeyer, and D. D. Long, "In-place reconstruction of version differences," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, pp. 973–984, 2003.

[7] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proc. of IEEE/ACM ASE*, 2004.

[8] A. Langiu, C. A. Boano, M. Schuß, and K. Römer, "Upkit: An open-source, portable, and lightweight update framework for constrained iot devices," in *Proc. of IEEE ICDCS*, 2019.

[9] N. Samteladze and K. Christensen, "Delta++: Reducing the size of android application updates," *IEEE Internet Computing*, vol. 18, no. 2, pp. 50–57, 2013.

[10] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, "R2: incremental reprogramming using relocatable code in networked embedded systems," *IEEE Trans. Computers*, vol. 62, no. 9, pp. 1837–1849, 2013.

[11] W. Dong, C. Chen, J. Bu, and W. Liu, "Optimizing relocatable code for efficient software update in networked embedded systems," *ACM Transactions on Sensor Networks*, vol. 11, no. 2, pp. 1–34, 2014.

[12] B. Mo, W. Dong, C. Chen, J. Bu, and Q. Wang, "An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks," in *Proc. of IEEE ICC*, 2012.

[13] B. Li, C. Tong, Y. Gao, and W. Dong, "S2: a small delta and small memory differencing algorithm for reprogramming resource-constrained iot devices," in *Proc. of IEEE INFOCOM (WKSHPS)*, 2021.

[14] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for iot networks-a survey," *ACM Computing Surveys*, vol. 54, no. 9, pp. 1–36, 2021.

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

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

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

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

[19] ——, "Google Play," https://play.google.com/store/games, 2023.

[20] OPPO, "OPPO APP Market," https://developers.oppomobile.com/newservice/capability?pagename=app_store, 2023.

[21] Xiaomi, "Xiaomi app store," https://app.mi.com/, 2023.

[22] N. Li, "Tencent myapp (Yingyong Bao): Android app stores and the appification of everything," in *Appified: Culture in the age of apps*. University of Michigan Press, 2018, pp. 42–50.

[23] R. Greg, G. Jeanloup, and M. Adler, "zlib Technical details," https://www.zlib.net/zlib_tech.html, 2022.

[24] A. Tridgell, P. Mackerras *et al.*, "The rsync algorithm," 1996.

[25] I. Pavlov, "Lempel–Ziv–Markov chain algorithm." https://7-zip.org/sdk.html, 2023.

[26] J. Seward, "bzip2 and libbzip2," *avaliable at http://www. bzip. org*, 1996.

[27] Tencent, "Tinker," https://github.com/Tencent/tinker, 2023.

[28] A. Sharma and R. Nasre, "QADroid: Regression Event Selection for Android Applications," in *Proc. of ACM ISSTA*, 2019.

[29] Z. Hu, X. Zou, W. Xia, S. Jin, D. Tao, Y. Liu, W. Zhang, and Z. Zhang, "Delta-dnn: Efficiently compressing deep neural networks via exploiting floats similarity," in *Proc. of ACM ICPP*, 2020.

[30] S. Zhang, D. Wu, H. Jin, X. Zou, W. Xia, and X. Huang, "QD-Compressor: A Quantization-based Delta Compression Framework for Deep Neural Networks," in *Proc. of IEEE ICCD*, 2021.

[31] X. Zhou, C. R. Qi, Y. Zhou, and D. Anguelov, "Riddle: Lidar data compression with range image deep delta encoding," in *Proc. of IEEE/CVF CVPR*, 2022.
[32] C. Deng, Q. Chen, X. Zou, E. Xu, B. Tang, and W. Xia, "imDedup: A Lossless Deduplication Scheme to Eliminate Fine-grained Redundancy among Images," in *Proc. of IEEE ICDE*, 2022.
[33] Y. Collet and M. Kucherawy, "Zstandard compression and the application/zstd media type," Tech. Rep., 2018.
[34] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The vcdiff generic differencing and compression data format," Tech. Rep., 2002.
[35] N. J. Larsson and K. Sadakane, "Faster suffix sorting," *Theoretical Computer Science (TCS)*, vol. 387, no. 3, pp. 258–272, 2007.
[36] S. Golomb, "Run-length encodings (corresp.)," *IEEE Transactions on Information Theory (TIT)*, vol. 12, no. 3, pp. 399–401, 1966.
[37] J. Fischer and F. Kurpicz, "Dismantling divsufsort," *arXiv preprint arXiv:1710.01896*, 2017.
[38] P. Deutsch, "Rfc1951: Deflate compressed data format specification version 1.3," 1996.
[39] A. Datta, K. F. Ng, D. Balakrishnan, M. Ding, S. W. Chee, Y. Ban, J. Shi, and N. D. Loh, "A data reduction and compression description for high throughput time-resolved electron microscopy," *Nature Communications*, vol. 12, no. 1, p. 664, 2021.
[40] T. Lu, W. Xia, X. Zou, and Q. Xia, "Adaptively compressing iot data on the resource-constrained edge." in *HotEdge*, 2020.
[41] Oracle, "java.io.File Java Docs." https://docs.oracle.com/javase/7/docs/api/java/io/File.html, 2023.
[42] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding Operational 5G: A First Measurement Study on its Coverage, Performance and Energy Consumption," in *Proc. of ACM SIGCOMM*, 2020.
[43] O. B. Yetim and M. Martonosi, "Adaptive delay-tolerant scheduling for efficient cellular and wifi usage," in *Proc. of IEEE WoWMoM*, 2014.

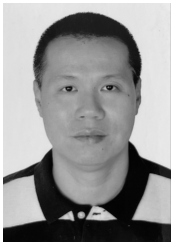
Tong Sun received the B.S. degree in Electronic and Information Engineering from Hangzhou Dianzi University, Hangzhou, in 2022. He is currently a Ph.D. candidate in the College of Computer Science at Zhejiang University, Hangzhou. His research interests include the Internet of Things, Edge Computing, and Security.

Bowen Jiang received the B.S. degree in Software Engineering from Dalian University of Technology, Dalian, in 2022. He is currently working toward the master's degree in the School of Software Technology at Zhejiang University, Hangzhou. His research interests include the Internet of Things and Edge Computing.

Lewei Jin received the B.S. degree in Electronic and Information Engineering from Hangzhou Dianzi University, Hangzhou, in 2022. He is currently working toward the master's degree in the College of Computer Science at Zhejiang University, Hangzhou. His research interests include the Internet of Things and Edge Computing.

Wenzhao Zhang received the BS degree from the college of Computer Science and Engineering, Northeastern University of China in 2018. He received the PhD degree from the College of Computer Science and Technology, Zhejiang University in 2023. His research interests include Internet of Things and edge computing.

Yi Gao received the BS and PhD degrees in Zhejiang University in 2009 and 2014, respectively. He is currently a full professor in Zhejiang University, China. From 2015 to 2016, he visited McGill University as a visiting scholar. His research interests include Internet of Things, edge computing, wireless and mobile computing. He is a member of the IEEE and the ACM.



Zhendong Li has been working in Huawei service lab since 2020, focusing on database, AI4DB and AIOps, solving reliability and performance problems in the O&M field.

Wei Dong received the BS and PhD degrees from the College of Computer Science, Zhejiang University, China, in 2005 and 2011, respectively. He is currently a full professor in the College of Computer Science, Zhejiang University, China. He leads the EmNets Lab in Zhejiang University, China. He has published more than 170 papers in prestigious conferences and journals including MobiCom, NSDI, ATC, MobiSys, ICNP, Ubicomp, INFOCOM, IEEE/ACM Transactions on Networking, IEEE Transactions on Mobile Computing, etc. His research interests include Internet of Things and sensor networks, wireless and mobile computing, and network measurement. He is a member of the ACM.