Binary Folding Compression for Efficient Software Distribution

Jinheng Li
City University of Hong Kong
Hong Kong, China
jinhengli2-c@my.cityu.edu.hk

Qiao Li
Xiamen University
Xiamen, China
liqiao@xmu.edu.cn

Hu Wan
City University of Hong Kong
Hong Kong, China
hu.wan@my.cityu.edu.hk

Chun Jason Xue
Mohamed bin Zayed University of Artificial Intelligence
Abu Dhabi, United Arab Emirates
jason.c.xue@gmail.com

ABSTRACT

This paper presents a simple yet effective approach for compressing binary files by detecting and folding similar patterns. Until now, methods for compressing these files were mostly limited to compiler optimization and traditional compression tools. However, our research suggests that there is an additional opportunity for compressing binary files and proposes an innovative way to allow multiple functions to share an extracted pattern code. The new method employs straightforward pattern recognition algorithms to spot recurring patterns in binary files and compresses them in order to reduce the file size. This results in a more efficient use of storage space and quicker data transfer speeds. Also, we introduce a novel feature that enables an extracted dynamic library to be shared among several binary files, further improving the method's effectiveness. We provide a detailed analysis comparing our method with the existing ones, showing that it performs better in a variety of practical implementations. This suggests that our humble attempt to improve binary file compression might pave the way for future developments in this field.

KEYWORDS

Compression, Binary Size, Software Distribution, Embedded Devices, Mobile Devices

ACM Reference Format:
Jinheng Li, Qiao Li, Hu Wan, and Chun Jason Xue. 2024. Binary Folding Compression for Efficient Software Distribution. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3636006

1 INTRODUCTION

Given the ongoing digitalization and widespread distribution of software solutions in various aspects of human life, the quest for efficient software transmission and storage has never been more crucial than it is today. This paper focuses on reducing binary program size without obtaining source code, primarily used to optimize software distribution specifically tailored for embedded devices or mobile smartphones, which are source-limited.

The most intuitive way to reduce the size of files is compression. Indeed, the smaller the binary size, the lower the data storage and transmission costs, which directly influence the speed and efficiency of software dissemination. However, despite current methods for binary compression, there is a critical gap. Many of these techniques require source code to apply compiler optimizations. Still, in the real world of distributing software, we may encounter a lot of scenarios in which we cannot access the source code.

Therefore, this paper negotiates the gap above by envisioning a tailored solution specifically optimized for binary programs, vouched by its growing prevalence in many computing contexts, including but not limited to mobile smartphones and embedded systems. The kernel of this approach lies in leveraging the intrinsic characteristics of fixed-length instruction binaries, specifically repeated patterns among binary files. By identifying these repeated patterns, we can perform binary folding. This process inherently mandates less valuable computing and storage resources, paving the way for a more efficient software distribution model.

However, it is worth noting that binary folding compression is not a stand-alone solution but an additional optimization method. Post binary folding, it is pivotal to implement existing mainstream compression techniques, such as gzip, to complement further and enhance the software reduction process. While binary folding capitalizes on binary contraction through pattern recognition in the software code, complementary tools such as gzip focus on eliminating redundancies at a Byte level, providing an extra layer of compression and efficiency to the software distribution process.

This paper's core objective is to reduce software distribution costs by reducing the software package, and reducing size of binary further by performing shared library creating. We achieve this goal using our proposed method, BinFolding, which can reduce size by folding repeat patterns and gain a better compression ratio than using singly other compression tools. Also, we propose a novel compression method for binary files that can reduce repeat parts inside binary files and among them, and we call it CrossFolding. Considering algorithms and basic computation among software may highly likely have everyday needs when implementing functions, we can find some repeat code patterns from different binary files; we could extract these codes and use binary rewriting or reverse engineer techniques to make them a dynamic library to free more space where they were. We also need to modify origin function calls into reference calls of new shared library.

Based on This paper's contributions are mainly as follows:

- We propose and evaluate a method flow called BinFolding (short for Binary Folding Compression), which can find repeat patterns and utilize their features to reduce binary size, and DeFolding to restore the origin binary files.
- We apply a trie as an efficient intermediate data structure to reduce the time and space cost of BinFolding, especially

for finding repeat patterns after scanning binary (after disassembled) files, so that BinFolding would be more acceptable on its usability.
- We bring a novel method, which was an untouched and challenging area that extracts similar or common parts of binary files to a dynamic library to reduce binary size further, which we call CrossFolding.

To better navigate this comprehensive study, the paper has been logically divided into various sections: In section 2, we provide background knowledge in related areas and where challenges are in the topic of this paper. Next, we will show examples and our observations in binary code where we can perform more optimizations to reduce size. And in Section 3, each of its part focusing on a specific aspect related to Binary Folding Compression. We'll also present how binary folding can benefit code size when applying CrossFolding. Subsequent sections will delve into the mechanism of binary folding, its integration with tools like gzip, and benchmark results, foregrounding the efficiency and benefits of implementing this method in practical scenarios.

2 BACKGROUND

2.1 Compaction of Programs

To reduce the size of programs, people today can apply a series of optimizations provided by compilers like dead code elimination [19], dead loop deleting and function merging [27]. Researchers and engineers have been trying to add more size optimization inside compiler chains, from front-ends to back-ends, at several levels [2, 31]. Also, there are some optimization target link time (LTO) for garbage collections on dead function symbols [15, 18]. But all the above methods mentioned involve source code obtaining. Otherwise all these optimizations have no chance to be applied to programs. There is also some other work trying to optimize binary, from optimizing function layout [14], debloat dynamic libraries themselves [1, 30] to loop rerolling [11]. Even though these methods have been trying to reduce size at the binary level as well, they mainly focus on the semantics of binary rather than the structure of binary files. There is still redundancy that exists after such optimization.

2.2 Distribution of Software

The delivery of software to users or devices, known as software distribution, can occur through various mediums like the Internet or physical items like DVDs. This ensures everyone access to the most recent software versions. Most of the time, software distribution, whether through physical media or the Internet, rarely comes with the source code (except for the open-source community), which means it may be difficult for users or administrators to further compress these executable files to save space after getting them.

Challenges are inevitable, with hurdles such as slow processors, physical transportation, and bandwidth constraints. The exponential rise in internet-reliant devices necessitates the Internet as the primary distribution channel. The software is uploaded to servers for further distribution, causing an increase in network pressure due to significant data consumption during uploads and downloads.

Nowadays, software gets distributed via application marketplace on mobile platforms, like Google Play and App Store, respectively dominate on Android [16, 32] and iOS. Both Google and Apple have been limiting developers on the size of their products [4, 10].

For embedded scenarios, currently, the primary method by which embedded IoT devices receive software downloads and updates is through the Cloud IoT Device SDK and docker [7]. This software development kit consists of client libraries created in Embedded C, designed to allow developers safe and efficient connection, provisioning, and management of devices through the Cloud IoT Core. The SDK is geared towards applications with constraints on size and energy, such as asset trackers functioning as battery-powered cellular devices or Wi-Fi-enabled smart home devices with limited flash ROM (read-only memory). Despite the compact nature of these applications, there remains a significant potential to further compress valuable space, thereby reducing distribution costs. Given the vast quantity of IoT devices, currently numbering around 20 billion, even minor optimization in the distribution channel could lead to substantial benefits [13].

And in today's cloud platforms which provide cloud machines for users, there are plenty of similar binary files, including system images, exist on the same physical machine. As cloud platforms host a myriad of applications, many of which contain repetitive patterns across binaries, the opportunity for deduplication is vast. By sharing patterns across multiple binary files using a dynamic library, there's potential for considerable space saving. In cloud environments where storage cost is a significant concern, this space optimization could translate to tangible cost savings.

2.3 Fixed Length Instructions

Unlike variable length instructions (CISC), fixed length instructions (RISC) [23] may involve large size of binary due to its protocol limitation. Advanced RISC Machine (ARM) instructions, characterized by their simplicity and uniformity, contain fewer and simpler addressing modes. They operate in a load/store model where only load/store instructions can access memory, thereby keeping computational instructions purely arithmetic. The uniformity of ARM instructions architecture results in most instructions taking up a uniform length of 32 bits. ARM's set of "conditional" instructions, which only execute based on the state of specific flags, is also unique [5].

Most of smartphones and embedded devices, which resource are limited, today choose ARM processors as their processors. So, reducing size for these devices is important and critical. Compared to x86 architecture, ARM is generally considered easier to analyze [25]. This is largely due to the fact that x86 instruction lengths vary and can be anywhere between 1 and 15 bytes, which means that disassembly at a random byte in the code can lead to false instructions. The fixed instruction length in ARM's instruction set avoids this problem, making instruction decoding more straightforward.

Moreover, ARM's load/store architecture simplifies the control flow analysis as only a few instructions change control flow, unlike x86 [6]. This simplicity and predictability result in better instruction alignment and allow for more efficient folding and compression. Because of these reasons, ARM instructions are easier to handle when writing binary analysis tools, thus making studies and analysis much more feasible.

3 MOTIVATION

In the age of mobile and embedded devices, optimization has become more critical than ever. The digital landscape is continuously

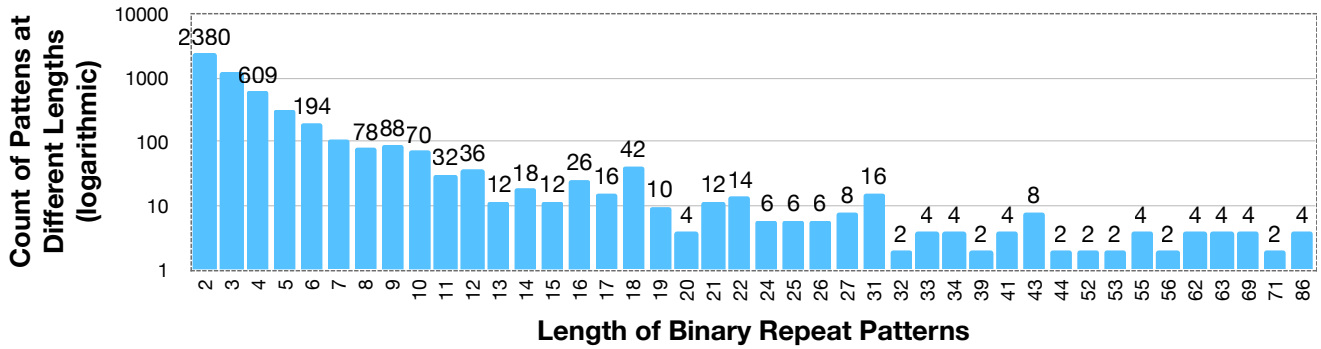

Figure 1: Distribution of Patterns with Different Lengths

evolving, leading to an incessant demand for efficient storage, transmission, and execution of binary files. While software and hardware have indeed progressed in leaps and bounds, they also bring forth challenges, especially when the binary size of applications becomes a limiting factor for storage and distribution. As we dive deeper into the intricate assembly code of these binary files, we encounter a promising avenue of optimization - the repetition of sequential code snippets with only variable constants.

Figure 2 provided aptly illustrates this phenomenon, showcasing multiple instances of similar sequential code patterns that differ solely in terms of their constants. Such repetition, while potentially stemming from recurrent functionality or programming paradigms, results in redundancy. If the binary file of a software application is imagined as its DNA, these repeating patterns can be likened to genetic motifs that appear time and again in various contexts. However, unlike biological systems where such repetition can have functional significance, in software binaries, predominantly consume unnecessary space. This observation, thus, brings forth an intriguing question - can we harness these redundancies to economize the binary size? We investigate and test different programs and draw Figure 1, which presents the statistic (count) data of repeat patterns with different lengths in these programs. It shows us that we can surly find a lot of redundant information in these binary files, and it's also a good opportunity for us to utilize and optimize the size of binary files.

Reducing binary size has multifaceted advantages. Firstly, it directly impacts the storage requirements. In scenarios with limited storage capacities - think Internet of Things (IoT) devices or embedded systems - even a minor reduction can be of great significance. Secondly, a smaller binary can lead to faster transmission times, which is especially crucial for applications that need to be frequently updated or transmitted over networks with bandwidth limitations. Moreover, optimizing binary size can potentially lead to improvements in cache performance, resulting in quicker software execution times.

Furthermore, consider cloud-based services and applications. Here, binaries are often transmitted across multiple nodes, replicated for redundancy, or stored for backup. Optimizing binary size can lead to significant bandwidth and storage savings at this scale. For mobile applications, where the initial download size can be a

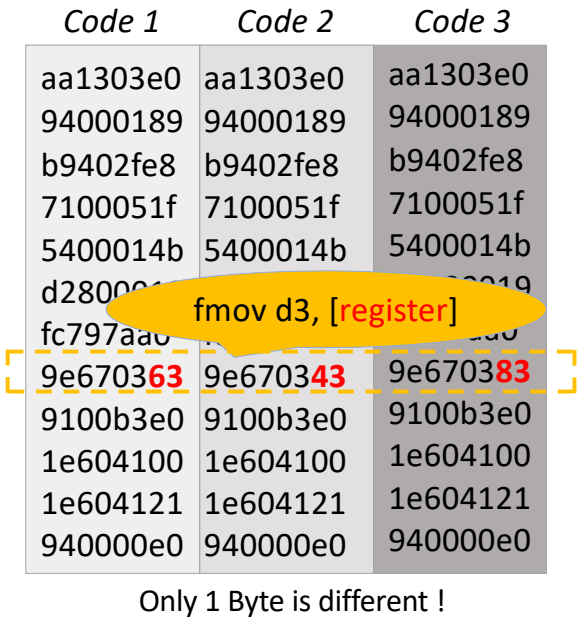

Figure 2: Sequential Similar Code Snippets

deciding factor for potential users, reducing the binary size can enhance user acquisition rates.

Upon a cursory glance, one might wonder why modern compilers and linkers haven't already addressed this apparent redundancy. The reality is nuanced. Compilers are primarily designed to optimize performance, ensuring that the generated machine code runs as fast as possible. This emphasis on performance sometimes comes at the cost of increased binary size. Additionally, specific repetitive patterns might arise from libraries, third-party modules, or even deliberate coding practices, which are not always in the purview of standard compiler optimizations. The opportunity here, then, is twofold. On the one hand, there is a potential to develop advanced algorithms and tools that can identify these repeated patterns, abstract the shared parts, and save space. On the other, there's scope for creating awareness among developers and programmers about these redundancies, fostering coding practices that inherently minimize such repetitions.

In conclusion, the motivation behind exploring this avenue of optimization is rooted in the evident need and the immense potential benefits it promises. As software continues to shape the world in unprecedented ways, efficiency in its very foundation, the binary code, can pave the way for a future where software is not only functional but also incredibly optimized.

4 BINARY FOLDING COMPRESSION

4.1 Overview

The BinFolding design constitutes four stages. Initially, an equivalent transformation between binary and assembly code is completed for preprocessing and analysis. In this step, we will scan each sub-strings of instruction sequences into a trie, repetitive patterns are identified from the instruction sequences. These patterns are assessed for their *profitability* using a unique model. Only those deemed profitable are extracted. Finally, we will operate binary files based the assembly code and repeat patterns we found for final extraction. Decompression (Defolding) follows an inverse route, with placeholders integrated for original instructions. DeFolding restores the original binary information, providing a counter-part to BinFolding. Lastly, CrossFolding targets every binary file on the same physical machine; it identifies repetitive patterns as shared functions and creates shared dynamic libraries to save space. And then rewrite origin binary function calls into library references. CrossFolding also accommodates constants' variations, making them function parameters.

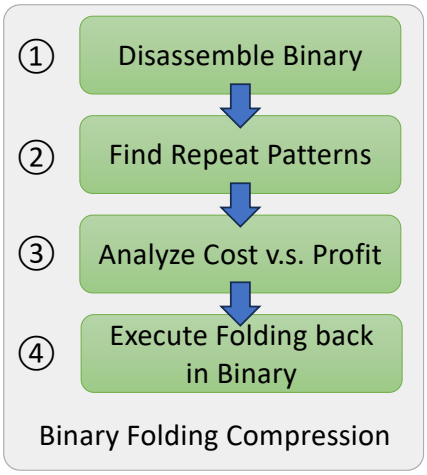

Figure 3: Workflow of BinFolding

4.2 Design of BinFolding

The methodology of BinFolding is characterized by four principal stages, as depicted in Figure 3.

In the initial stage, an equivalent transformation is implemented between binary code and assembly code. This is an essential step as it provides a convenient platform for further analysis and processing. The transformation necessitates the use of a sophisticated reverse tool like IDA Pro, which successfully converts the binary files into the assembly format.

Subsequently, the primary procedure of BinFolding commences, which entails scanning to locate repetitive patterns. The system initially scans the symbol table to acquire information about the function definitions encapsulated within the individual binary files. BinFolding proceeds to read the functions line by line, strategically extracting instructions as a sequence so as to adequately represent the functions. This specific approach is adopted due to the fixed instruction length found in ARM. Post this phase, a trove of instruction sequences specific to each function is generated.

The following stage (Step 2) employs a Trie to process the intermediate data consolidated in the preceding stage and derive repetitive patterns. Each instruction, designed as a TrieNode, is inserted into the Trie, as shown in Figure 4. The subsequent progressions in this stage draw out the anticipated repetitive patterns.

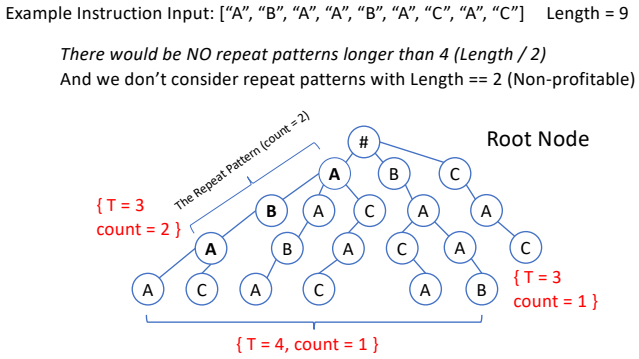

Figure 4: Use a Trie to Find Repeat Patterns

The third stage, being a prominent part of the methodology, corresponds to the examination of repetitive patterns of instructions. Each pattern is scrutinized based on their individual details of constants and registers (addresses) following the instructions. The associated profit model, as proposed in Equation 1, is referred to during this inquiry. The decision to extract a pattern is contingent upon whether it is profitable or not, a profitability measure that is calculated as per the model. As per Figure 5, if the resultant value is not less than 0.5, the pattern is designated as non-profitable; consequently, it will not be extracted. Yet, an exception is associated where constants in patterns may vary but may follow a constant step rule—a phenomenon depicted in Figure 6. This effectively implies that storing every constant is not mandatory, particularly when there exists a pattern within the constants.

In the final stage, modifications are made to the original assembly code. This not only includes extracting patterns but also incorporates the inclusion of placeholders and markers. The extracted patterns are streamlined into a separate file.

There is a critical question in Step 3, how does the cost model improve the final compression ratios? Our method can make sure origin files are reduced obviously, but final output reduction requires a combined analysis with the following compress tools. We studied the algorithm of gzip tool, which is LZ77 method, using the Huffman Coding idea. Our cost model and design of BinFolding

```
1000039b0 :    add    x0, x0, #3780
1000039b0 :    mov    w1, #15
1000039b0 :    mov    w2, #1

100003a60 :    add    x1, x1, #3760
100003a64 :    mov    x0, x20
100003a68 :    mov    w2, #800

100003d5c :    add    x0, x0, #3864
100003d60 :    mov    w1, #40
100003d64 :    mov    w2, #1
```

Profitable = 18 / 30 = 0.6 > 0.5 => Non Profitable!

Figure 5: Example of Non-Profitable

```
100003ee8: adrp    x16, 0x100004000 <__stubs+0x4>
100003eec: ldr x16, [x16, #8]
100003ef0: br  x16
100003ef4: adrp    x16, 0x100004000 <__stubs+0x10>
100003ef8: ldr x16, [x16, #32]
100003efc: br  x16
100003f00: adrp    x16, 0x100004000 <__stubs+0x1c>
100003f04: ldr x16, [x16, #40]
100003f08: br  x16
100003f0c: adrp    x16, 0x100004000 <__stubs+0x28>
100003f10: ldr x16, [x16, #48]
100003f14: br  x16
100003f18: adrp    x16, 0x100004000 <__stubs+0x34>
100003f1c: ldr x16, [x16, #56]
100003f20: br  x16
100003f24: adrp    x16, 0x100004000 <__stubs+0x40>
100003f28: ldr x16, [x16, #64]
100003f2c: br  x16
100003f30: adrp    x16, 0x100004000 <__stubs+0x4c>
100003f34: ldr x16, [x16, #72]
100003f38: br  x16
```

Figure 6: Constant Step among Constants in Repeat Patterns

make files easier to be compressed by the same family of compress tools like gzip into a smaller size.

$$Profitable_i = \frac{count(DiffCons_i) + count(DiffReg_i)}{len(Pattern_i) + count(AllCons_i) + count(AllReg_i)} \quad (1)$$

Decompression within the context of BinFolding swiftly adopts an inverse path. It involves strategic placement of placeholders for original instructions that are suitably filled during the process of decompression. It is crucial to note that the design of the decompression process also significantly impacts the performance of BinFolding due to the space occupied by placeholders.

In the prototype of BinFolding, the placeholder design strategy is quite straightforward and highly intuitive; placeholders are inserted specifically for recognition during decompression. Therefore, it is worth mentioning that the incorporation and management of placeholders substantially influence both the compression and decompression processes in BinFolding's design, betraying the tool's bidirectional nature amidst the operational dynamics.

4.3 DeFolding

In this section, we introduce how we design the part of DeFolding. BinFolding directly operates on binary files, which is a challenging

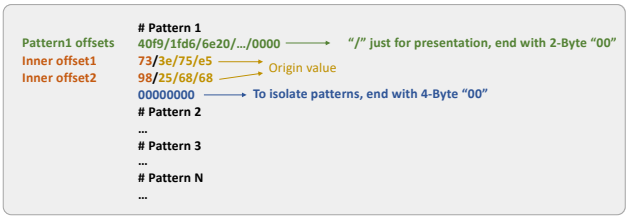

Figure 7: How Patterns Being Stored for DeFolding

and dangerous method. So, the reverse method that can restore the origin binary information is critical for BinFolding, and decides the usability of BinFolding. As Figure 7 shows, the main idea here is that we use relative offset address to mark repeat patterns and use a sequence of data to contain the different parts of each pattern. After the scan of BinFolding, we will get the repeat pattern sequences in assembly code, then we'll map them into pure binary code and extract the different part. And then BinFolding will delete each copy of patterns, and record the current relative offset address. When in the process of DeFolding, we will restore information reversely because the relative offset of each pattern depends on previous ones.

4.4 Design of CrossFolding

As illustrated in Figure 8, the concept of CrossFolding is introduced. In essence, the operations conducted by CrossFolding bear resemblance to machine outlining as referenced in [2, 21]. However, the ambition of CrossFolding stems beyond that, targeting every binary file situated on the same physical machine.

Post BinFolding, each binary file possesses its distinct set of repetitive patterns, poised for extraction. This multitude of pattern sets subsequently undergoes an intersection operation. The resultant production is a final cross set comprising of repetitive patterns that appear in more than one binary file.

It is crucial to highlight that during the CrossFolding process, there is no compulsion to detect and identify patterns that manifest across every binary file; patterns found in a fraction of the binary files suffice. This is attributable to the fact that the process involves the creation of a shared dynamic library; comprehensively, not every program necessitates the utilization of every function enclosed within a library.

The application of CrossFolding promises profitable returns, one of which is the saving of space. Specifically, the space saved is equivalent to the size identified in the BinFolding process but replicated amongst other binary files. A salient feature of CrossFolding is its ability to accommodate constants with variations, transforming them into parameters of functions within the library. Consequently, CrossFolding serves as an advanced tool that streamlines pattern recognition, space optimization, and function parameterization.

5 EVALUATION

5.1 Experiments Setup

In order to evaluate the performance of BinFolding on various programs, a testing environment was established. As BinFolding specifically operates around the ARM architecture, the Apple Silicon

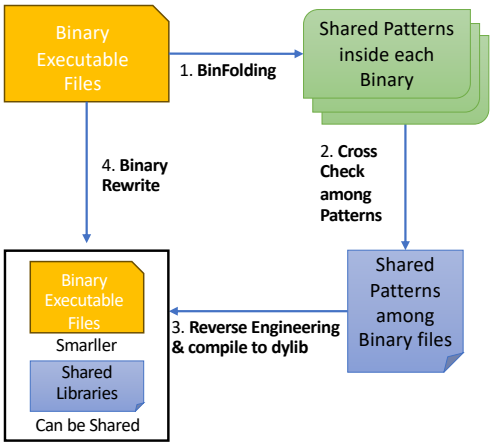

Figure 8: CrossFolding

Platform, with its ARM cores, was elected for this purpose. The specific hardware specifications are elaborated in Table 1.

The benchmark chosen for this process was MiBench [12]. This decision rested on its widespread use in the realm of code size reduction alongside a comprehensive compilation of documents. And we also add some self-implemented programs as examples to evaluate.

To assess the temporal cost during experimentation, the *time* module embedded within Python was employed. The entire BinFolding pipeline BinFolding's five times in order to estimate the average data. The outcomes from these tests are revealed in the following sections. This approach offers a practical exploration of BinFolding's efficacy within a controlled testing environment.

5.2 Performance on Compression

BinFolding is not conceived to supersede the mainstream compression tools such as gzip [24], which is the most popular [33] zip tools nowadays. Rather, it is designed to complement such tools, enhancing their performance on the final compression ratio when applied post-BinFolding. This assessment result is showcased in Table 2.

In the table, the programs for evaluation are listed along with the ratio improved by BinFolding when using zip to compress packages. This ratio represents the optimization in size after extraction of redundant data and then being compressed by compress tools (zip). BinFolding demonstrates an impressive improvement of approximately 6.05% on average when compared to the original assembly code, showcasing its effective size reduction capability.

The concept of CrossFolding, as previously discussed in the design portion, was also tested. The results were promising; over forty patterns were identified that could potentially be extracted as functions in a shared dynamic library. We acknowledge that modifying original binary function calls and transitioning them into calls of dynamic libraries remBinFolding'senging task due to the inherent complexities. Nonetheless, such findings hint towards significant room for further optimizations, thus marking a positive trajectory for BinFolding's implementation and progress.

Table 1: Environment Setup of Experiments

Environment	Model	Details
CPU	Apple M1 Max	10 Cores, 2.06 - 3.22 GHz
Memory	32 GB	LPDDR5-6400
Benchmark	MiBench	Version 1.0
Compress Tools	gzip	Version 3.0

And the CrossFolding, which based on the BinFolding's output, is designed for reducing binary size even after decompression. We can find same sequences when checking repeat patterns of each binary file crossly. As Figure 9 presents, we can find many same sequences from binary files. Most of them are short sequences with nonfunctional code (a lot of *mov* operations), so we only need to focus on sequences with larger size and compile them into dynamic library. One more time, CrossFolding is still a conceptual method and requires familiarity with binary files' content. And it would be used in the runtime, so every function we extracted by CrossFolding, would decrease the size of original one.

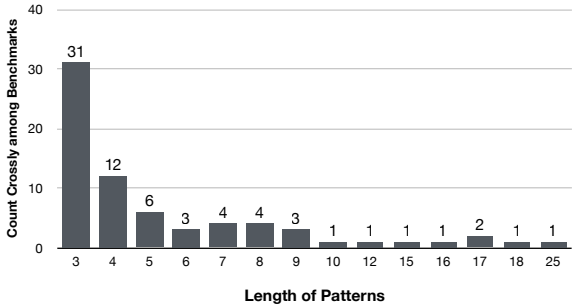

Figure 9: Redundancy Found in Binary Files Crossly

5.3 Cost Analysis

Pattern detection within the BinFolding methodology is a resource-intensive task that necessitates a substantial investment of time. While this process is conducted offline, limiting any severe impacts on application scenarios, it still poses a significant time-cost element, as shown in Figure 10. Coupled with the data given in 2, it's clear that the time cost increases linearly with the growth of binary size.

Within BinFolding, space cost is influenced by two key factors. The first cost is associated with the trie used for temporarily storing patterns. The second revolves around the CrossFolding operation, which requires each pattern to be loaded and cross-checked to determine those eligible for extraction as shared dynamic libraries. A set is constructed during this process to aid in filtering and matching tasks. Notably, both of these costs are directly linked to the size of the programs and their quantity.

6 RELATED WORK

Research in both academia and the industry has made significant strides towards reducing code size. This paper provides a brief yet in-depth examination of these advancements and differentiates our suggested method from the current strategies.

Table 2: Performance of BinFolding (Comparison between w/ or w/o BinFolding when using gzip to compress)

Benchmark	Origin Size (Byte)	Ratio Improved	Improved After gzip
dijkstra_large	17461	10.35%	4.98%
cjpeg	714682	0.73%	0.47%
jpegtran	108904	0.71%	0.48%
bank	15521	20.66%	9.61%
crc	6349	10.68%	2.47%
basicmath_large	36338	34.54%	23.96%
djpeg	756623	0.94%	0.62%
patricia	22549	18.52%	13.34%
qsort_large	7110	13.91%	3.50%
wrjpgcom	34048	15.29%	9.22%
fft	29354	10.56%	2.30%
search_large	47530	11.67%	7.93%
rijndael	121952	2.48%	2.05%
bitcnts	43087	8.05%	3.76%
rdjpgcom	23696	9.55%	4.64%

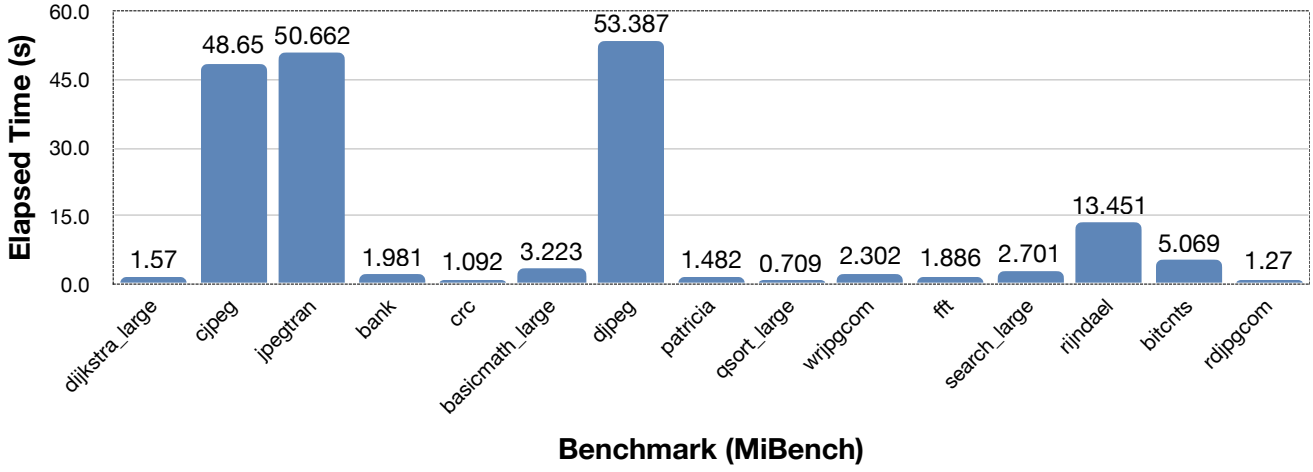

Figure 10: Time Cost of BinFolding

Beginning with source code level, work by Park et al. involves identifying similarities in code via code abstraction, aiding the detection of likeness within large-scale software products [22]. In an attempt to solve the issue of duplication detection across different programming languages and dialects, Ducasse et al. employed a string-matching approach to find code clones within large codebases [8].

Shifting focus to the compiler level, where most optimization endeavors are focused, Evans et al. demonstrated the use of structural abstraction in analyzing abstract syntax trees at the Intermediate Representation (IR) layer [9]. Furthermore, Rocha et al. devised a unique method for matching similar functions, resulting in a striking compression ratio but at a significant expenditure of time and space [26, 28, 29].

At the binary level, the primary focus of research has been on binary rewriting. Research by Sutter et al. highlighted possible application of binary rewriting for code compaction during Link-Time [3]. More recently, a proprietary compression tool named Superpack was developed by Meta Platforms, Inc., who have claimed remarkable binary code compression outcomes [20]. However, our method has not been directly compared with Superpack due to its proprietary nature.

When considering binary files as pure data rather than programs, there are also other similar research areas such as audio analysis and protein analysis, which also trying to find repeat patterns and use folding strategies [17, 34].

7 FUTURE WORK

Enhancing binary file compression remains complex, but the introduction of BinFolding sets a promising precedent. This paper has evaluated its potential to reduce binary file size, assessing its impact on alleviating storage pressure and streamlining software distribution. However, BinFolding currently operates offline, necessitating file decompression prior to loading for execution. Our

future research will address this shortcoming by enabling offloaded parts of programs to return to storage in compressed form. This approach aims to enhance memory handling by keeping only the active code operational, thereby decreasing distribution burdens and saving operating costs. This approach is particularly relevant within the context of Internet of Things (IoT) scenarios, where bandwidth and storage resources are limited. Success in this area could greatly reduce the memory overhead, filling a critical need in IoT. Future research will aim to refine BinFolding's capabilities, to achieve these goals and pioneer advancements in efficient software distribution. Also, we can combine computer vision algorithms with binary compression by folding segments into one image to reduce the dimension of space for further analysis.

8 CONCLUSIONS

In this paper, we introduce a novel approach called BinFolding, devised to decrease binary size, specifically for ARM devices, in an effort to ease storage and distribution burdens. BinFolding operates on the principle of identifying recurring patterns that appear multiple times, extracting these as shared segments to conserve space. We detail and analyze these patterns within the paper, evaluating the profitability of implementing BinFolding post-application.

There might be minor variations within each pattern, but these typically constitute a small fraction. This entails that we need to retain only a single copy of shared parts and original differing parts for compression, restoring them during the process of decompression. Within the BinFolding framework, an efficient strategy for identifying repeat patterns and avoiding a global $O(n^2)$ time complexity has been employed. We use a Trie to decrease search time and intermediate memory costs for storing instruction sequences.

Furthermore, both time and space costs exist and are directly linked to the quantity of the programs in BinFolding. Yet, these costs are acceptable, considering the significant size reduction capabilities demonstrated and the costs were offline and don't need to perform repeatly. In conclusion, BinFolding showcases promising potential as a complementary tool to existing compression methods, offering valuable contributions to the field of binary compression.

ACKNOWLEDGEMENT

This work was supported in part by the National Natural Science Foundation of China under Grant No. 62202396. The corresponding author is Qiao Li (liqiao@xmu.edu.cn).

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[2] Milind Chabbi, Jin Lin, and Raj Barik. An experience with code-size optimization for production ios mobile applications. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 363–377. IEEE, 2021.

[3] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 2005.

[4] Apple Developer. Doing advanced optimization to further reduce your app's size. https://developer.apple.com/documentation/xcode/doing-advanced-optimization-to-further-reduce-your-app-s-size, 2022.

[5] ARM Developer. Arm developer suite assembler guide. https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference.

[6] ARM Developer. Arm instruction set reference guide. https://developer.arm.com/documentation/100076/0100?lang=en.

[7] Inc. Docker. What is a container? https://www.docker.com/resources/what-container/.

[8] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[9] William S Evans, Christopher W Fraser, and Fei Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

[10] Documentation Android for Developers. Compressed download size restriction. https://developer.android.com/guide/app-bundle#size_restrictions, 2021.

[11] Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. Rollbin: reducing code-size via loop rerolling at binary level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 99–110, 2022.

[12] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[13] Mohammad Hasan. State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally. https://iot-analytics.com/number-connected-iot-devices/, 2022.

[14] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. Optimizing function layout for mobile applications. *arXiv preprint arXiv:2211.09285*, 2022.

[15] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

[16] Purnima Kochikar. Celebrating 10 years of google play. together. https://android-developers.googleblog.com/2022/03/celebrating-10-years-of-google-play.html, 2022.

[17] Zong Lin and Hays S Rye. Expansion and compression of a protein folding intermediate by groel. *Molecular cell*, 16(1):23–34, 2004.

[18] Gai Liu, Umar Farooq, Chengyan Zhao, Xia Liu, and Nian Sun. Linker code size optimization for native mobile applications. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 168–179, 2023.

[19] LLVM. Code to perform dead code elimination. https://llvm.org/doxygen/DCE_8cpp_source.html.

[20] Meta. Superpack: Pushing the limits of compression in facebook's mobile apps. https://engineering.fb.com/2021/09/13/core-data/superpack/, 2021.

[21] Jessica Paquette. Reducing code size using outlining. https://llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf, 2016.

[22] Seongsoo Park, Seungcheol Ko, Jungsik Choi, Hwansoo Han, Seong-Je Cho, and Jongmoo Choi. Detecting source code similarity using code abstraction. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, pages 1–9, 2013.

[23] David A Patterson and David R Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[24] GNU Project. Gnu gzip. https://www.gnu.org/software/gzip/, 2020.

[25] Redhat. Arm vs x86: What's the difference? https://www.redhat.com/en/topics/linux/ARM-vs-x86, 2022.

[26] Rodrigo Rocha. *Reducing code size with function merging*. PhD thesis, University of Edinburgh, 2021.

[27] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. Hyfm: Function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 110–121, 2021.

[28] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163. IEEE, 2019.

[29] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 854–868, 2020.

[30] Benjamin Shteinfeld. Libfilter: Debloating dynamically-linked libraries through binary recompilation. *Undergraduate Honors Thesis. Brown University*, 2019.

[31] Anderson Faustino da Silva, Bernardo NB de Lima, and Fernando Magno Quintão Pereira. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 47–58, 2021.

[32] Statista. Number of apps available in leading app stores as of 1st quarter 2022. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2022.

[33] w3techs.com. Usage statistics of gzip compression for websites. https://w3techs.com/technologies/details/ce-gzipcompression#:~:text=Gzip%20Compression%20is%20used%20by%2052.2%25%20of%20all%20the%20websites.

[34] Suresh Yerva, Smita Nair, and Krishnan Kutty. Lossless image compression based on data folding. In *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*, pages 999–1004. IEEE, 2011.