usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard

Zhenhua Li, *Tsinghua University and Baidu Mobile Security;* Weiwei Wang, *Baidu Mobile Security;* Tianyin Xu, *University of California, San Diego;* Xin Zhong, *Tsinghua University and Baidu Mobile Security;* Xiang-Yang Li, *Tsinghua University, University of Science and Technology of China, and Illinois Institute of Technology;* Yunhao Liu, *Tsinghua University;* Christo Wilson, *Northeastern University;* Ben Y. Zhao, *University of California, Santa Barbara*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-zhenhua

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

Open access to the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) is sponsored by USENIX.

Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard

Zhenhua Li [1,2], Weiwei Wang [2], Tianyin Xu [3], Xin Zhong [1,2]
Xiang-Yang Li [1,4,5], Yunhao Liu [1], Christo Wilson [6], Ben Y. Zhao [7]

[1] *Tsinghua University* [2] *Baidu Mobile Security* [3] *University of California San Diego*
[4] *University of Science and Technology of China* [5] *Illinois Institute of Technology*
[6] *Northeastern University* [7] *University of California Santa Barbara*

Abstract

As mobile cellular devices and traffic continue their rapid growth, providers are taking larger steps to optimize traffic, with the hopes of improving user experiences while reducing congestion and bandwidth costs. This paper presents the design, deployment, and experiences with Baidu TrafficGuard, a cloud-based mobile proxy that reduces cellular traffic using a network-layer VPN. The VPN connects a client-side proxy to a centralized traffic processing cloud. TrafficGuard works transparently across heterogeneous applications, and effectively reduces cellular traffic by 36% and overage instances by 10.7 times for roughly 10 million Android users in China. We discuss a large-scale cellular traffic analysis effort, how the resulting insights guided the design of TrafficGuard, and our experiences with a variety of traffic optimization techniques over one year of deployment.

1 Introduction

Mobile cellular devices are changing today's Internet landscape. Growth in cellular devices today greatly outpaces that of traditional PCs, and global cellular traffic is growing by double digits annually, to an estimated 15.9 Exabytes in 2018 [4]. This growing traffic demand has led to significant congestion on today's cellular networks, resulting in bandwidth caps and throttling at major wireless providers. The challenges are more dramatic in developing countries, where low-capacity cellular networks often fail to deliver basic quality of service needed for simple applications [40, 41, 44].

While this is a well known problem, only recently have we seen efforts to address it at scale. Google took the unprecedented step of prioritizing mobile-friendly sites in its search algorithm [9]. This will likely spur further efforts to update popular websites for mobile devices. Recent reports estimate that most enterprise webpages are designed for PCs, and only 38% of webpages are mobile-friendly [24]. More recently, Google released details on their Flywheel proxy service for compressing content for the Chrome mobile browser [29].

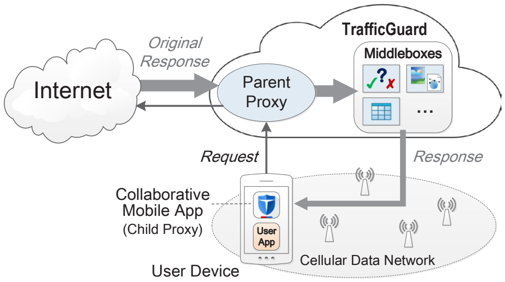

Figure 1: Architectural overview of TrafficGuard.

Competition in today's mobile platforms has led to numerous "walled-gardens," where developers build their own suites of applications that keep users within their ecosystem. The ongoing trend limits the benefits of application-specific proxies, even ones with user bases as large as Google Chrome [29, 18, 25, 21, 42, 43, 36]. In contrast, an alternative approach is to transparently intercept and optimize network traffic across all apps at the OS/network layer. Although some examples of this approach exist [16, 17, 6, 15], little is known about their design or impact on network performance.

This paper describes the design, deployment, and experiences with Baidu TrafficGuard, a third-party cellular traffic proxy widely deployed for Android devices in China [1]. As demonstrated in Figure 1, TrafficGuard is a cloud-based proxy that redirects traffic through a VPN to a client-side mobile app (*http://shoujiweishi.baidu.com*). It currently supports all Android 4.0+ devices, and does not require root privileges. Inside the cloud, a series of software middleboxes are utilized to monitor, filter, and reshape cellular traffic. TrafficGuard was first deployed in early 2014, and its Android app has been installed by

[1] Cellular data usage in Asia differs from that of US/European networks, in that HTTP traffic dominates 80.4% of cellular traffic in China and 74.6% in South Korea [62]. In comparison, HTTPS accounts for more than 50% of cellular traffic in the US [56, 47, 54].

roughly 10 million users. The average number of daily active users is around 0.2 million.

In designing a transparent mobile proxy for cellular traffic optimization, TrafficGuard targets four key goals:

- First, traffic optimization should not harm user experiences. For example, image compression through pixel scaling often distorts webpage and UI (user interface) rendering in user apps. Similarly, traffic processing should not introduce unacceptable delays.
- Second, our techniques must generalize to different apps, and thus proprietary APIs or data formats should be avoided. For example, Flywheel achieves significant traffic savings by transcoding images to the WebP format [27]. Though WebP offers high compression, not all apps support this format.
- Third, we wish to limit client-side resource consumption, in terms of memory, CPU, and battery. Note that the client needs to collaborate well with the cloud using a certain amount of resources.
- Finally, we wish to reduce system complexity, resource consumption, and monetary costs on the cloud side. In particular, the state information maintained for each client should be carefully determined.

In this paper, we document considerations in the design, implementation, and deployment of TrafficGuard. *First*, we analyze aggregate cellular traffic measurements over 110K users to understand the characteristics of cellular traffic in China. This gave us insights on the efficacy and impact of traditional data compression, as well as the role of useless content like broken images in cellular traffic. *Second*, we adopt a lightweight, adaptive approach to image compression, where more considerate compression schemes are constructed to achieve a sweet spot on the image-quality versus file-size tradeoff. This helps us achieve traffic savings comparable to Flywheel (27%) at roughly 10%–12% of the computation overhead. *Third*, we develop a customized VPN tunnel to efficiently filter users' unwanted traffic, including overnight, background, malicious, and advertisement traffic. *Finally*, we implement a cloud-client paired proxy system, and integrate best-of-breed caching techniques for duplicate content detection. The cloud-client paired design allows us to finely tune the tradeoff between traffic optimization and state maintenance.

TrafficGuard is the culmination of these efforts. For installed users, it reduces overall cellular traffic by an average of 36%, and instances of traffic overage (*i.e.,* going beyond the users' allotted data caps) by 10.7 times. Roughly 55% of users saw more than a quarter reduction in traffic, and 20% of users saw their traffic reduced by half. TrafficGuard introduces relatively small latency penalties (median of 53 ms, mean of 282 ms), and has little to no impact on the battery life of user devices.

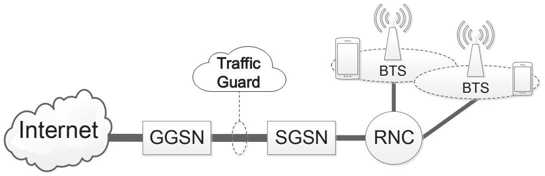

Figure 2: Potential integration of TrafficGuard into a 3G cellular carrier. Integration for 4G would be similar.

While already successful in its current deployment, TrafficGuard can achieve even higher efficiency if cellular carriers (are willing to) integrate it into their infrastructure. As demonstrated in Figure 2, carriers could deploy TrafficGuard between the GGSN (Gateway G-PRS Support Node) and SGSN (Serving GPRS Support Node). Then the optimized traffic is further transferred to the RNC (Radio Network Controller) and BTS (Base Transceiver Station). This would greatly simplify both the cloud-side and client-side components of TrafficGuard, and further reduce latency penalties for users.

Finally, we note that while Baidu does not have an internal IRB (institutional review board [13]) review process, all reasonable steps were taken at Baidu to protect user privacy during this study. All users who participated in the study opted-in as volunteers with informed consent, and full traffic traces were limited to one week of measurements (all other datasets are anonymized logs). Wherever possible, analysis was limited to anonymized metadata only. When necessary, content analysis was done on aggregate data, and fully decoupled from any user identifiers or personally identifiable information.

2 State-of-the-Art Systems

This section briefly surveys state-of-the-art mobile traffic proxy systems. As listed in Table 1, we compare seven systems with TrafficGuard. We focus on five of the most important and ubiquitous features supported by these systems: 1) image compression, 2) text compression, 3) content optimization, 4) traffic filtering, and 5) caching. In each case, we highlight the strengths of different approaches, as well as the shortcomings, which motivated our design of TrafficGuard.

Since most mobile traffic proxy systems are closed-source, we rely on a variety of methods to determine their features. The implementation of Google Flywheel is described in [29]. For Opera Turbo, UCBrowser (proxy), and QQBrowser (proxy), we are able to uncover most of their features through carefully controlled experiments. Specifically, we set up our own web server, used these proxies to browse our own content hosted by the server, and carefully compared the data sent by the server with what was received by our client device. Unfortunately, Opera Max, Microsoft Data Sense, and Onavo Extend

Table 1: Comparison of state-of-the-art mobile traffic proxy systems. "?" means unknown.

System	Image Compression	Text Compression	Content Optimization	Traffic Filtering	Caching
Google Flywheel	Transcoding to WebP	Yes	Lightweight error page	Safe Browsing	Server-side
Opera Turbo	Transcoding to WebP	Yes	Pre-executing JavaScript	Ad blocking	?
UCBrowser	Pixel Scaling	Yes	No	Ad blocking	?
QQBrowser	Transcoding to WebP	Yes	No	Ad blocking	?
Opera Max [17] (China's version)	Transcoding PNG to JPEG	Yes	No	Restricting overnight traffic	?
Microsoft Data Sense	?	Yes	No	Restricting background traffic, and ad blocking	?
Onavo Extend	Transcoding PNG and large GIF to JPEG	Yes	No	No	Client-side
TrafficGuard	Adaptive quality reduction	No	Attempting to discard useless content	Restricting overnight and background traffic, ad blocking, Safe Browsing	Server-side, and VBWC on both sides

use encrypted proxies, and thus we can only discover a subset of their implementation details.

First, we examine the image compression techniques. Three systems transcode images to WebP, which effectively reduces network traffic [29]. However, this only works for user apps that support WebP (*e.g.,* Google Chrome). Similarly, Opera Max and Onavo Extend transcode PNGs to JPEGs, and Onavo Extend also transcodes large GIFs to JPEGs. Taking a different approach, UCBrowser rescales large images ($> 700 \times 700$ pixels) to small images ($< 150 \times 150$ pixels). Although rescaling reduces traffic, it could harm user experiences by significantly degrading image qualities. In contrast to these systems, TrafficGuard uses an adaptive quality reduction approach that is not CPU intensive, reduces traffic across apps, and generally does not harm user experiences (see § 5.1).

Second, we find that all the seven systems compress textual content, typically with gzip. However, our large-scale measurement findings (in § 3.2.2) reveal that the vast majority of textual content downloaded by smartphone users is very short, meaning that compression would be ineffective. Thus, TrafficGuard does not compress texts, since the CPU overhead of decompression is not worth the low (1.36%) HTTP traffic savings.

Third, we explore the *content optimization* strategies employed by mobile traffic proxies. We define *content optimization* as attempts to reduce network traffic by altering the semantics or functionality of content. For example, Flywheel replaces HTTP 404 error pages with a lightweight version. More aggressively, Opera Turbo executes JavaScript objects at the proxy, so that clients do not need to download and execute them. Although this can reduce traffic, it often breaks the original functionality of websites and user apps, *e.g.,* in the controlled experiments we often noticed that JavaScript functions like onscroll() and oninput() were not properly executed by Opera Turbo. Rather than adopt these approaches, TrafficGuard validates HTTP content and attempts to discard useless content like broken images (see § 5.2).

Fourth, we observe that many of the target systems implement traffic filtering. Four systems block advertisements, plus Flywheel using Google Safe Browsing [8] to block malicious content. Opera Max attempts to restrict apps' traffic usage during the night, when users are likely to be asleep. Microsoft Data Sense takes things a step further by also restricting traffic from background apps, under the assumption that apps which are not currently interactive should not be downloading lots of data. We discover that all these filtering techniques are beneficial to users (see § 3.2.4), and thus we incorporate all of them into TrafficGuard (see § 5.3).

Finally, we study the caching strategies of existing systems. Flywheel maintains a server-side cache of recently accessed objects, while Onavo Extend maintains a local cache (of 100 MB by default). In contrast, TrafficGuard adopts server-side strategies by maintaining a cache at the proxy (see § 4.2), as well as implementing Value-based Web Caching (VBWC) between the client and server (see § 5.4). Although we evaluated other sophisticated caching strategies (see Appendix A), we ultimately chose VBWC because it offers excellent performance and is straightforward to implement.

3 Measuring Cellular Traffic

In this section, we present a large-scale measurement study of cellular traffic usage by Android smartphone users. Unlike prior studies [37, 46, 52, 39, 34, 62], our analysis focuses on content and metadata. Using this dataset, we identify several key performance issues and tradeoffs that guide the design of TrafficGuard.

3.1 Dataset Collection

The ultimate goal of TrafficGuard is to improve smartphone users' experiences by decreasing network usage and filtering unwanted content. To achieve this goal, we decided to take a measurement-driven methodology, *i.e.,* we first observed the actual cellular traffic usage patterns of smartphone users, and then used the data to drive our design and implementation decisions.

Table 2: General statistics of our collected TGdataset.

Collection period	03/21 – 03/27, 2014
Unique users	111,910
Total requests	162M
Dataset size	1324 GB (100%)
Non-HTTP traffic (plus TCP/IP)	259 GB (19.6%)
HTTP traffic (plus TCP/IP)	1065 GB (80.4%)
HTTP header traffic	107 GB (8.1%)
HTTP body traffic	875 GB (66.1%)

Table 3: Statistics of HTTP content in TGdataset.

Type	**% of Requests**	**% of HTTP Traffic**	**Size (KB)**	
			Median	**Mean**
Image	**32%**	**71%**	**5.7**	**15.5**
Text	**49%**	**15.7%**	**0.2**	**2.2**
Octet-stream	10%	5.5%	0.4	3.8
Zip	8.1%	5.1%	0.5	4.3
Audio & Video	0.03%	2.6%	407	614
Other	0.87%	0.1%	0.3	0.7

When we first deployed TrafficGuard between Jan. 5–Mar. 31, 2014, the system only monitored users' cellular traffic; it did not filter or reshape traffic at all. We randomly invited users to test TrafficGuard from $\sim$100M existing mobile users of Baidu. We obtained informed consent from volunteers by prominently informing them that full traces of their cellular traffic would be collected and analyzed. We assigned a unique *ClientToken* to each user device that installed the mobile app of TrafficGuard.

We used two methods to collect packet traces from volunteers. For an HTTP request, the TrafficGuard app would insert the ClientToken into the HTTP header. The TrafficGuard cloud would then record the request, remove the injected header, complete the HTTP request, and store the server's response. However, for non-HTTP requests (most of which are HTTPS), it was not possible for the TrafficGuard cloud to read the injected *ClientToken* (we did not attack secure connections via man-in-the-middle). Thus, the TrafficGuard app locally recorded the non-HTTP traffic, and uploaded it to the cloud in a batch along with the ClientToken once per week. These uploads were restricted to WiFi [2], in order to avoid wasting volunteers' cellular data traffic. In both cases, we also recorded additional metadata like the specific app that initiated each request, and whether that app was working in the foreground or background.

We collected packet traces from volunteers for one week, between Mar. 21–27, 2014. In total, this dataset contains 320M requests from 0.65M unique ClientTokens. However, we observe that many user devices in the dataset only used their cellular connections for short periods of time. These *short-term* users might have good WiFi availability, or might be using their cellular connections but did not (remember to) run the mobile app of TrafficGuard. To avoid bias, we focus on the traces belonging to 111,910 *long-term* users who used their cellular connections in at least four days during the collection period. This final dataset is referred to as TGdataset, whose general statistics are listed in Table 2.

3.2 Content Analysis

Below, we analyze the content and metadata contained in TGdataset. In particular, we observe that today's cellular traffic can be effectively optimized in multiple ways.

3.2.1 General Characteristics

We begin by presenting some general characteristics of TGdataset. As listed in Table 2, 80.4% of TGdataset is HTTP traffic, most of which corresponds to the bodies of HTTP messages. This finding is positive for two reasons. First, it means content metadata (*e.g.,* Content-Length and Content-Type) is readily available for us to analyze. Second, it is clear that the TrafficGuard system will be able to analyze and modify the vast majority of cellular traffic, since it is in plaintext.

Table 3 presents information about the types of HTTP content in TGdataset. We observe that images are the second most frequent type of content, but consume 71% of the entire HTTP traffic. Textual content is the most frequent, while non-image binary content accounts for the remainder of HTTP traffic. We manually analyzed many of the octet-streams in our dataset and found that they mainly consist of software and video streams.

3.2.2 Size and Quality of Content

Next, we examine the size and quality of content in TGdataset, and relate these characteristics to the compressibility of content.

Images. Four image types dominate in our dataset: JPEG, WebP, PNG, and GIF. Certainly, all four types of images are already compressed. However, we observe that 40% of images are *large*, which we define as images of $w \times h$ pixels such that $w \times h \geq 250000 \wedge w \geq 150 \wedge h \geq 150$ (refer to § 5.1 for more details of image categorization). Some images even have over 4000×4000 pixels (exceeding 10 MB in size) in extreme cases.

More importantly, we observe that many JPEGs have high *quality factors* (QFs). QF determines the strength of JPEG's lossy-compression algorithm, with QF $= 100$ causing minimal loss but a larger file size. The median QF of JPEGs in TGdataset is 80 while the average is 74. Such high-quality images are unnecessary for most cellular users, considering their limited data plans and screen sizes. This presents us with an optimization opportunity that TrafficGuard takes advantage of (see § 5.1).

[2] Certainly TrafficGuard also has the *capability* of helping mobile users save WiFi traffic, just like what Google Flywheel does. However, at the moment TrafficGuard only targets at saving cellular traffic for two reasons. First, WiFi users generally do not care about the traffic usage since they do not pay for their Internet access in terms of traffic usage. Second, proxy-based traffic saving inevitably leads to latency penalty and thus would impact WiFi users' experiences.

Table 4: Validity and usefulness of images.

Type	% of	% of	Image Size (KB):	
	Requests	Image Traffic	Median	Mean
Correct	87%	95.9%	5.4	14.8
Broken	**10.6%**	**3.2%**	**0.13**	**3.2**
Blank	2.3%	0.57%	0	0
Incomplete	0.1%	0.21%	0.01	5.0
Inconsistent	0.04%	0.16%	4.8	33

Textual content. The six most common types of textual content in TGdataset are: JSON, HTML, PLAIN, JavaScript, XML, and CSS. Compared with images, textual content is much smaller: the median size is merely 0.2 KB. Compressing the short texts with the size less than 0.2 KB (*e.g.,* with gzip, bzip2, or 7-zip) cannot decrease their size; in fact, the additional compression metadata may even *increase* the size of such textual data.

Surprisingly, we find that compressing the other, larger half ($>$ 0.2 KB) of textual content with gzip brings limited benefits — it only reduced the HTTP traffic of texts by 8.7%, equal to 1.36% ($= 8.7\% \times 15.7\%$) of total HTTP traffic. Similarly, using bzip2 and 7-zip could not significantly increase the compression rate. However, decompressing texts on user devices does necessitate additional computation and thus causes battery overhead. Given the limited network efficiency gains and the toll on battery life, we opt to not compress texts in TrafficGuard, unlike all other systems as listed in Table 1.

Other content. For the remaining octet-stream, zip, audio & video content, we find that compression provides negligible benefits, since almost all of them are already compressed (*e.g.,* MP3 and VP9). Although it is possible to reduce network traffic by transcoding, scaling, or reducing the quality of multimedia content [42], we do not explore these potential optimizations in this work.

3.2.3 Content Validation

Delving deeper into the content downloaded by our volunteers, we discover a surprisingly high portion of useless content, particularly *broken* images. We define an image to be broken if it cannot be decoded by any of the three widely used image decoders: imghdr [12], Bitmap [23], and dwebp [7]. As shown in Table 4, 10.6% of images in TGdataset are broken, wasting 3.2% of all image traffic in our dataset (their average size is much smaller than that of correct images). Note that we also observe a small fraction of *blank* and *incomplete* images that we can decode, as well as a few *inconsistent* images that are actually not images, but we do not consider to obey our strict definition of correctness.

3.2.4 Traffic Filtering

As we note in § 2, existing mobile traffic proxies have adopted multiple strategies for traffic filtering. In this section, we investigate the potential of four particular filtering strategies by analyzing TGdataset.

Overnight traffic. Prior studies have observed that many smartphones generate data traffic late at night, even when users are not using the devices [52, 39, 34]. If we conservatively assume that our volunteers are asleep between 0–6 AM, then 11.4% of traffic in our dataset can potentially be filtered without noticeable impact on users. Based on this finding, we implemented a feature in TrafficGuard that allows users to specify a night time period during which cellular traffic is restricted (see § 5.3).

Background traffic. Users expect foreground apps to consume data since they are interactive, but background apps may also consume network resources. Although this is expected in some cases (*e.g.,* a user may stream music while also browsing the web), undesirable data consumption by background apps has become such a common complaint that numerous articles exist to help mitigate this problem [58, 3, 2, 10]. In TGdataset, we observe that 26.7% of cellular traffic is caused by background apps. To this end, we implemented dual filters in TrafficGuard specifically designed to reduce the network traffic of background apps (see § 5.3).

Malicious traffic. A recent measurement study of Google Play reveals that more than 25% of Android apps are malicious, including spammy, re-branded, and cloned apps [59]. We compare all the HTTP requests in TGdataset against a proprietary blacklist containing 29M links maintained by major Internet companies (including Baidu, Google, Microsoft, Symantec, Tencent, *etc.*), and find that 0.85% of requests were issued for malicious content. We addressed this issue in TrafficGuard by filtering out HTTP requests for blacklisted URLs.

Advertisement traffic. In addition to malicious content, we also find that 4.15% of HTTP requests in TGdataset were for ads. We determined this by comparing all the requested HTTP URLs in our dataset against a proprietary list of 102M known advertising URLs (similar to the well-known EasyList [1]). Ad blocking is a morally complicated practice, and thus we give TrafficGuard users the choice of whether to opt-in to ad filtering. Users' configuration data reveal that the majority (67%) of users have chosen to block ads. On the other hand, we did get pushback from a small number of advertisers; when this happened, usually we would remove the advertisers from our ad block list after verification.

3.2.5 Caching Strategies

Finally, we explore the feasibility of two common caching strategies. Unfortunately, we find neither technique offers satisfactory performance, which motives us to implement a more sophisticated caching strategy.

Name-based. Traditional web proxies like Squid [61] implement *name-based* caching of objects (*i.e.,* objects are indexed by their URLs). However, this approach is known to miss many opportunities for caching [38, 57, 51]. To make matters worse, we observe that over half of the content in TGdataset is not cacheable by Squid due to HTTP protocol issues. This situation is further exacerbated by the fact that many start-of-the-art HTTP libraries do not support caching at all [50]. Thus, although TrafficGuard uses Squid in the back-end cloud, we decided to augment it with an additional, object-level caching strategy (known as VBWC, see § 5.4).

HTTP ETag. The HTTP ETag [11] was introduced in HTTP/1.1 to mitigate the shortcomings of named-based caching. Unfortunately, the effectiveness of ETag is still limited by two constraints. First, as ETags are assigned arbitrarily by web servers, they do not allow clients to detect identical content served by multiple providers. This phenomenon is called content *aliasing* [55]. We observe that 14.16% of HTTP requests in TGdataset are for aliased content, corresponding to 7.28% of HTTP traffic. Second, we find that ETags are sparsely supported: only 5.76% of HTTP responses include ETags.

4 System Overview

Our measurement findings in § 3.2 provide useful guidelines for optimizing cellular traffic across apps. Additionally, we observe that some techniques used by prior systems (*e.g.,* text compression) are not useful in practice. These findings guide the design of TrafficGuard for optimizing users' cellular traffic.

This section presents an overview of TrafficGuard, which consists of a front-end mobile app on users' devices and a set of back-end services. Below, we present the basic components of each end, with an emphasis on how these components support various traffic optimization mechanisms. Additional details about specific traffic optimization mechanisms are explained in § 5.

4.1 Mobile App: The Client-side Support

The TrafficGuard mobile app is comprised of a user interface and a *child proxy*. The user interface is responsible for displaying cellular usage statistics, and allows users to configure TrafficGuard settings. The settings include enabling/disabling specific traffic optimization mechanisms, as well as options for specific mechanisms (the details are discussed in § 5). We also leverage the user interface to collect feedback from users, which help us continually improve the design of TrafficGuard.

The child proxy does the real work of traffic optimization on the client side. It intercepts incoming and outgoing HTTP requests at the cellular interface, performs computations on them, and forwards (some) requests to the back-end cloud via a customized VPN tunnel. As shown in Figure 3, the client-side VPN tunnel is implemented using the TUN virtual network-level device [26] that intercepts traffic from or injects traffic to the TCP/IP stack. HTTP GET requests [3] are captured by the child proxy, encapsulated, and then sent to the back-end cloud for further processing. Accordingly, the child proxy is responsible for receiving responses from the back-end.

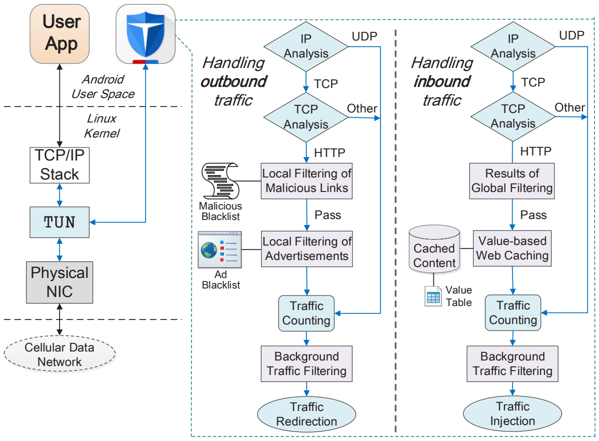

Figure 3: Basic design of the child proxy.

The mobile app provides client-side support for traffic optimization. First, it allows users to monitor and restrict cellular traffic at night and from background apps in a real-time manner. Users are given options to control how aggressively TrafficGuard filters these types of traffic. Second, it provides local filtering of malicious links and unwanted ads using two small blacklists of the most frequently visited malicious and advertising URLs. Requests for malicious URLs are dropped; users are given a choice of whether to enable ad blocking, in which case requests for ad-related URLs are also dropped.

Third, the child proxy acts as the client-side of a value-based web cache [55] (VBWC, see § 5.4 for details). At a high level, the child proxy maintains a key-value store that maps MD5 hashes to pieces of content. The back-end cloud may return "VBWC Hit" responses to the client that contain the MD5 hash of some content, rather than the content itself. In this case, the child proxy retrieves the content from the key-value store using the MD5 hash, and then locally constructs an HTTP response containing the cached content. The reconstructed HTTP response is then returned to the corresponding user app. This process is fully transparent to user apps.

4.2 Web Proxy: The Back-end Support

As shown in Figure 4, the cloud side of TrafficGuard consists of two components: a cluster of parent proxy servers that decapsulate users' HTTP GET requests and fetch content from the Internet; and a series of software middleboxes that process HTTP responses.

[3] Non-GET HTTP requests (*e.g.,* POST, HEAD, and PUT) and non-HTTP requests do not benefit from TrafficGuard's filtering and caching mechanisms, so the child proxy forwards them to the TCP/IP stack for regular processing. Furthermore, TrafficGuard makes no attempt to analyze SSL/TLS traffic for privacy reasons.

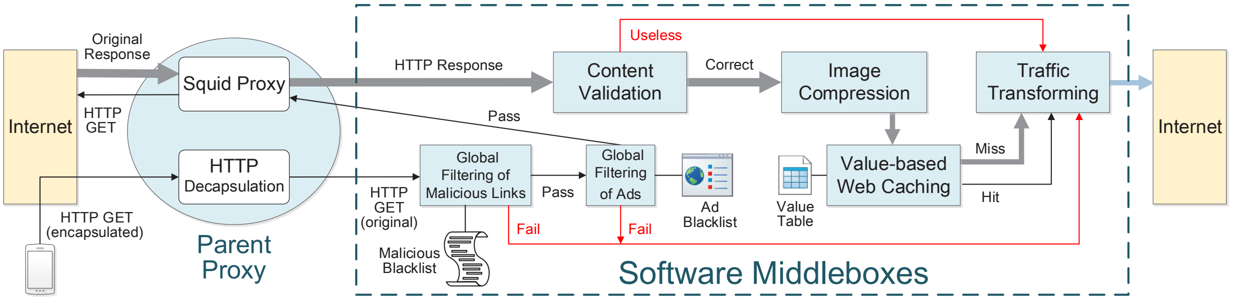

Figure 4: Cloud-side overview of TrafficGuard. HTTP requests are generally processed from left to right by a cluster of parent proxy servers and a series of software middleboxes implemented on top of Nginx.

Once an HTTP GET request sent by the child proxy is received, the parent proxy decapsulates it and extracts the *original* HTTP GET request. Next, middleboxes compare the original HTTP GET request against large blacklists of known malicious and ads-related URLs. Note that this HTTP GET request has passed the client-side filtering with small blacklists. Together, this two-level filtering scheme prevents TrafficGuard users from wasting memory loading large blacklists on their own devices. If a URL hits either blacklist, it is reported back to the mobile app so the user can be notified.

An HTTP request that passes the blacklist filters is forwarded to a Squid proxy, which fetches the requested content from the original source. The Squid proxy implements name-based caching of objects using an LRU (Least Recently Used) scheme, which helps reduce latency for popular objects. Once the content has been retrieved by Squid, it is further processed by middleboxes that validate content (§ 5.2) and compress images (§ 5.1).

Lastly, before the content is returned to users, it is indexed by VBWC (§ 5.4). VBWC maintains a separate index of content for every active user, which contains the MD5 hash of each piece of content recently downloaded by that user. For a given user, if VBWC discovers that some content is already indexed, it returns that MD5 in a "VBWC Hit" response to the mobile app, instead of the actual content. As described above, the child proxy then constructs a valid HTTP response message containing the cached content. Otherwise, the MD5 is inserted into the table and the actual content is sent to the user.

5 Mechanisms

This section presents the details of specific traffic optimization mechanisms in TrafficGuard. Since many of the mechanisms include user-configurable parameters, we gathered users' configuration data between Jul. 4–Dec. 27, 2014. This dataset is referred to as TGconfig.

5.1 Image Compression

Overview. Image compression is the most important traffic-reduction mechanism implemented by TrafficGuard, since our TGdataset shows that cellular traffic is dominated by images. Based on our observation that the majority of JPEGs have quality factors (QFs) that are excessively high for display on smartphone screens, TrafficGuard adaptively compresses JPEGs by reducing their QFs to an acceptable level. Additionally, TrafficGuard transcodes PNGs and GIFs to JPEGs with an acceptable QF. Note that TrafficGuard does *not* transcode PNGs with transparency data or animated GIFs, to avoid image distortion. TrafficGuard ignores WebP images, since they are already highly compressed.

TrafficGuard's approach to image compression has three advantages over alternative strategies. First, as JPEG is the dominant image format supported by almost all (>99% to our knowledge) user apps, TrafficGuard does not need to transcode images back to their original formats on the client side. Second, our approach costs only 10%–12% as much CPU as Flywheel's WebP-based transcoding method (see § 6.3). Finally, our approach does not alter the pixel dimensions of images. This is important because many UI layout algorithms (*e.g.*, CSS) are sensitive to the pixel dimensions of images, so rescaling images may break webpage and app UIs.

Categorizing Images. The challenge of implementing our adaptive QF reduction strategy is deciding how much to reduce the QFs of images. Intuitively, the QFs of large images can be reduced more than small images, since the resulting visual artifacts will be less apparent in larger images. Thus, following the approach of Ziproxy [28] (an open-source HTTP proxy widely used to compress images), we classify images into four categories according to their width (w) and height (h) in pixels:

- *Tiny* images contain < 5000 pixels, *i.e.*, $w \times h < 5000$.
- *Small* images include images with less than 50000 pixels (*i.e.*, $5000 \leq w \times h < 50000$), as well as "slim" images with less than 150 width or height pixels (*i.e.*, $w \times h \geq 5000 \wedge (w < 150 \vee h < 150)$).
- *Mid-size* images contain less than 250000 pixels, that is $50000 \leq w \times h < 250000 \wedge w \geq 150 \wedge h \geq 150$.
- *Large* images contain no less than 250000 pixels, that is $w \times h \geq 250000 \wedge w \geq 150 \wedge h \geq 150$.

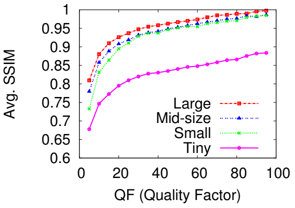

Figure 5: Average SSIM corresponding to consecutive QFs.

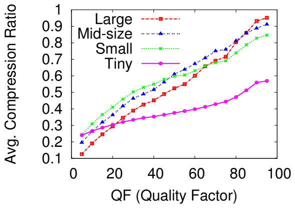

Figure 6: Average compression ratio corresponding to consecutive QFs.

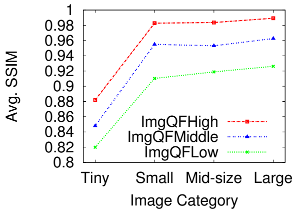

Figure 7: Average SSIM corresponding to the three QF schemes.

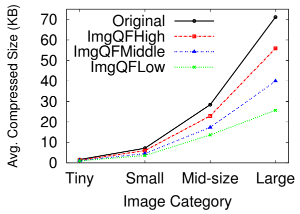

Figure 8: Average compressed size corresponding to the three QF schemes.

QF Reduction Scheme. After images are divided into the above four categories, we need to determine a proper *QF (reduction) scheme* for transcoding images in each category. Our goal is to maximize compression by reducing QF, while also minimizing the reductions of user-perceived image quality. To measure quality, we use Structural Similarity (SSIM) [60], which assesses the visual similarity between a compressed image and the original (1 means the two images are identical). Quantitatively, we calculate the SSIM and *compression ratio* $\left(= \frac{\text{Size of images after compression}}{\text{Size of images before compression}}\right)$ corresponding to consecutive QFs, based on all the correct images in TGdataset. The results are plotted in Figure 5 and Figure 6.

Specifically, we define a QF scheme *ImgQFScheme* $= \{T, S, M, L\}$ to mean that tiny, small, mid-size, and large images are compressed to QF $= T, S, M$, and L, respectively. In practice, we constructed three QF schemes that vary from high compression, less quality to low compression, high quality: *ImgQFLow* $= \{30, 25, 25, 20\}$, *ImgQFMiddle* $= \{60, 55, 50, 45\}$, and *ImgQFHigh* $= \{90, 90, 85, 80\}$. We then compressed all the correct images in TGdataset using each scheme to evaluate their impact on image quality and size.

Figure 7 examines the impact of each QF scheme on image quality. Prior work has shown that image compression with SSIM ≥ 0.85 is generally considered acceptable by users [29]. As shown in Figure 7, all three QF schemes manage to stay above the 0.85 quality threshold for small, mid-size, and large images. The two cases where image quality becomes questionable concern tiny images, which are the hardest case for any compression strategy. Overall, these results suggest that in most cases, even the aggressive *ImgQFLow* scheme will produce images with an acceptable level of fidelity.

Figure 8 examines the image size reduction enabled by each QF scheme, as compared to the original images. As expected, more aggressive QF schemes provide more size reduction, especially for large images.

User Behavior. The mobile app of TrafficGuard allows users to choose their desired QF scheme. Users must select a scheme after they install TrafficGuard. The data in TGconfig reveal that 95.4% of users selected the *ImgQFMiddle* scheme. Also, qualitative feedback from TrafficGuard users suggests that they are satisfied with the quality of images while using the system.

5.2 Content Validation

As mentioned in § 3.2.3, TrafficGuard users encounter a non-trivial amount of broken images when using apps. The back-end cloud of TrafficGuard naturally notices most broken images during the image analysis, transcoding, and compression process. In these cases, the cloud simply discards the broken image and sends a "Broken Warning" response to the client. From the requesting app's perspective, broken images appear to be missing due to a network error, and are handled as such.

5.3 Traffic Filtering

In this section, we present the implementation details of the four types of filters employed by TrafficGuard. Most traffic filtering in our system occurs on the client side (in the child proxy), including first-level filtering of malicious URLs and ads, and throttling of overnight and background traffic. Only second-level filtering of malicious URLs and ads occurs on the cloud side.

Restricting overnight traffic. The mobile app of TrafficGuard automatically turns the user's cellular data connection off between the hours of t_1 and t_2, which are configurable by the user. This feature is designed to halt device traffic during the night, when the user is likely to be asleep. TrafficGuard pops-up a notification just before t_1, alerting the user that her cellular connection will be turned off in ten seconds. Unless the user explicitly cancels the action, her cellular data connection will not be resumed until t_2. According to TGconfig, nearly 20% of users have enabled the overnight traffic filter, and 84% of them adopt the default night duration of 0–6 AM.

Throttling background traffic. To prevent malicious or buggy apps from draining users' limited data plans, TrafficGuard throttles traffic from background apps. Specifically, the TrafficGuard app has a configurable *warning bound* (B_1) and a *disconnection bound* (B_2), with $B_2 \gg B_1$. TrafficGuard also maintains a count c of the total bytes transferred by background apps. If c increases to B_1, TrafficGuard notifies the user that background apps are consuming a significant volume of traffic. If c reaches B_2, another notification is created to alert the user that her cellular data connection will be closed in ten seconds. Unless the user explicitly cancels this action or manually re-opens the cellular data connection, her cellular data connection will not be resumed. After the user responds to the B_2 notification, c is reset to zero.

According to TGconfig, 97.6% of users have enabled the background traffic filter, indicating that users actually care about background traffic usage. Initially, we set the default warning bound $B_1 = 1.0$ MB. However, we observed over 57% of users decreased B_1 to 0.5 MB, indicating that they wanted to be reminded of background traffic usage more frequently. Conversely, the initial disconnection bound was $B_2 = 5$ MB, but 69% of users raised B_2 to 20 MB, implying that the initial default setting was too aggressive. Based on this implicit feedback, we changed the default values of B_1 and B_2 to 0.5 MB and 20 MB. In comparison, Microsoft Data Sense only maintains a disconnection bound (B_2) to restrict background traffic, and there is no default value provided.

Two-level filtering of malicious links and ads. To avoid wasting cellular traffic on unwanted content, TrafficGuard always prevents users from accessing malicious links, while giving users the choice of whether to opt-in to ad blocking. In § 4.1 and § 4.2, we have presented high-level design of the two-level filtering. Here we talk about two more nuanced implementation issues.

The first issue is about the sizes of the local, small blacklists. Both lists have to be loaded in memory by the child proxy for quick searching, so they must be much shorter than the cloud-side large blacklists (which contain 29M malicious URLs and 102M ads-related URLs). To balance memory overhead with effective local traffic filtering, we limit the maximum size of the local blacklists to 40 MB. Consequently, the local blacklists usually contain around 1M links in total, which we observe are able to identify 72%–78% of malicious and ads links.

The second issue concerns updates to blacklists. As mentioned in § 3.2.4, the large blacklists are maintained by an industrial union that typically updates them once per month. Accordingly, the TrafficGuard cloud automatically creates updated small blacklists and pushes them to mobile users.

5.4 Value-based Web Caching (VBWC)

Early in 2003, Rhea *et al.* proposed VBWC to overcome the shortcomings of traditional HTTP caching [55]. The key idea of VBWC is to index objects by their *hash values* rather than their URLs, since an object may have many *aliases*. VBWC has a much better hit rate than HTTP caching because it handles aliased content. However, prior to TrafficGuard, VBWC has not been widely deployed in practice due to two problems: 1) the *complexity* of segmenting an object into KB-sized blocks and choosing proper block boundaries; 2) its *incompatibility* with the HTTP protocol, since VBWC requires that the proxy and the client maintain significant state information, *i.e.,* a mapping from hash values to cached content.

Reducing complexity. To determine whether TrafficGuard's VBWC implementation should segment content into blocks (and if so, at what granularity), we conduct trace-driven simulations using the content in TGdataset. Specifically, we played back each user's log of requests, and inserted the content into VBWC using 8 KB, 32 KB, 128 KB, and full content segmentation strategies. To determine segment boundaries, we ran experiments with simple fixed-size segments [55] and variable-sized, Rabin-fingerprinting based segments [53]. We also examined the handprinting-based approach that combines Rabin-fingerprinting and deterministic sampling [48].

Through these simulations, we discovered that 13% of HTTP requests would hit the VBWC cache if we stored content whole, *i.e.,* with no segmentation. Surprisingly, even if we segmented content into 8 KB blocks using the Rabin-fingerprinting (the most aggressive caching strategy we evaluated), the hit rate only increased to 15%. The handprinting-based approach exhibited similar performance to Rabin-fingerprinting when a typical number ($k = 4$) of handprint samples are selected, while incurring a bit lower computation overhead. By carefully analyzing the cache-hit results, we find that the whole-content hashing is good enough for two reasons: 1) images dominate the size of cache-hit objects in TGdataset; 2) there are almost no partial matches among images. Thus, we conclude that a simple implementation of content-level VBWC is sufficient to achieve high hit rates.

Addressing incompatibility. As discussed above, VBWC is incompatible with standard HTTP clients and proxies. Fortunately, we have complete control over the TrafficGuard system, particulary the cloud-client paired proxies, which enabled us to implement VBWC. The front-end child proxy takes care of encapsulating HTTP requests from user apps and decapsulating responses from the back-end cloud, meaning that VBWC is transparent to user apps. In practice, the mobile app of TrafficGuard maintains a 50-MB content cache on the client's file system, along with an in-memory table mapping content hashes to filenames that is a few KB large.

Ideally, every change to the cloud-side mapping table triggers a change to the client-side mapping table accordingly. But in practice, for various reasons (*e.g.,* network packet loss) this pair of tables may be different at some time, so we need to synchronize them with proper overhead. In TrafficGuard, the client-side mapping table is loosely synchronized with the cloud-side mapping table on an hourly basis, making the synchronization traffic negligible and VBWC mostly effective.

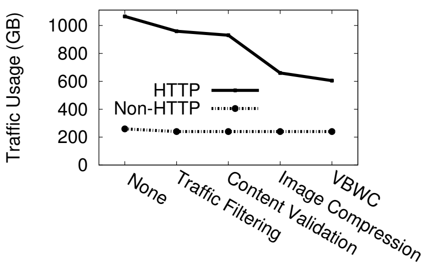

Figure 9: Total cellular traffic usage optimized by each mechanism.

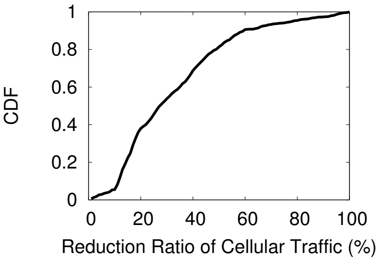

Figure 10: Distribution of users' cellular traffic reduction ratios.

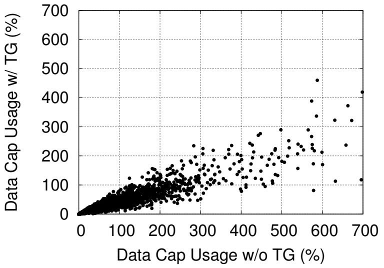

Figure 11: Users' cellular traffic usage relative to their data caps.

6 Evaluation

In this section, we evaluate the traffic reduction, system overhead, and latency penalty brought by TrafficGuard.

6.1 Data Collection and Methodology

We evaluate the performance of TrafficGuard using both real-system logs and trace-driven simulations. We collect working logs from TrafficGuard's back-end cloud servers between Dec. 21–27, 2014, which include traces of 350M HTTP requests issued from 0.6M users, as well as records of CPU and memory utilization over time on the cloud servers. We refer to this dataset as TGworklog.

On the other hand, as the client-side traffic optimization mechanisms mainly help users reduce traffic by suppressing unwanted requests, it is not possible to accurately record the corresponding saved traffic (which never occurred in reality). Instead, we rely on trace-driven simulations using TGdataset to estimate the client-side and overall traffic savings. In addition, we report real-world traffic reduction results using TGworklog in Appendix B, which mainly record the cloud-side traffic savings.

6.2 Traffic Reduction

Client-side. First, we examine the effectiveness of TrafficGuard's client-side mechanisms at reducing traffic. In TGdataset, 11.4% of cellular traffic is transferred at night, and according to TGconfig, 20% of users have enabled overnight traffic filtering. Thus, we estimate that users eliminate 2.3% (= 11.4% × 20%) of cellular traffic using the overnight traffic filter.

Moreover, we observe that 1% of users in TGdataset regularly exceed the disconnection bound $B_2 = 20$ MB per day of background traffic. The resulting overage traffic amounts to 5.33% of cellular traffic. In TGconfig, 97.6% of users have enabled background traffic filtering. Therefore, we estimate that the background traffic filter reduces cellular traffic by 5.2% (= 5.33% × 97.6%). Note that this background traffic saving is an *underestimation*, since we do not take the potential effect of B_1 (= 0.5 MB, the warning bound) into account.

Additionally, in TGdataset malicious content accounts for 0.8% of HTTP traffic while ads account for 4%. According to TGconfig, 67% of users have chosen to drop ads. Consequently, after all malicious content and unwanted ads are filtered, 3.48% $(= 0.8\% + 4\% \times 67\%)$ of HTTP traffic can be saved. This is equal to 2.8% $(= 3.48\% \times 80.4\%)$ of total cellular traffic.

Overall. Next, we evaluate how much traffic TrafficGuard is able to reduce overall through trace-driven simulations. Specifically, we play back all the requests in TGdataset, and record how many bytes are saved by each mechanism: traffic filtering, content validation, image compression, and VBWC. As shown in Figure 9, TrafficGuard is able to reduce HTTP traffic by 43% and non-HTTP traffic by 7.4% when all four mechanisms are combined. In summary, the overall cellular traffic usage is reduced by 36%, from 1324 GB to 845 GB.

As expected, image compression is the most important mechanism when used in isolation. 38% of the image traffic is reduced by our implemented adaptive quality reduction approach. In other words, our approach saves a comparable portion $(27\% = 38\% \times 71\%)$ of HTTP traffic as compared to Flywheel's WebP-based transcoding method, at a small fraction of the CPU cost (see § 6.3).

To understand how traffic savings are spread across users, we plot the distribution of cellular traffic reduction ratios for our users in Figure 10. We observe that 55% of users saved over a quarter of cellular traffic, and 20% users saved over a half (most of whom benefit a lot from traffic filtering and VBWC). These results demonstrate that most users received significant traffic savings.

Using TrafficGuard's built-in user-feedback facility, we asked users to report their cellular data caps. 95% of the long-term volunteers in TGdataset reported their caps to us. Using this information, we plot Figure 11, which shows the percentage of each user's data cap that would be used with and without TrafficGuard (again, based on trace-driven simulations). We observe that 58.2% of users exceed their usage caps under normal circumstances, and that TrafficGuard grants significant practical benefits for these users, *e.g.,* users who would normally be using 200%–300% of their allocation (and thus pay overage fees) are able to stay below 100% usage with TrafficGuard. Overall, TrafficGuard reduces the number of users who exceed their data caps by 10.7 times.

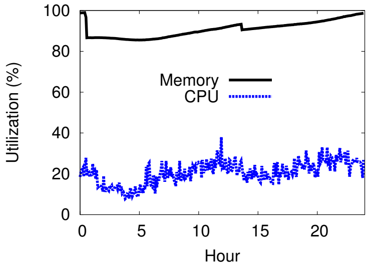

Figure 12: CPU and memory overhead of the back-end servers.

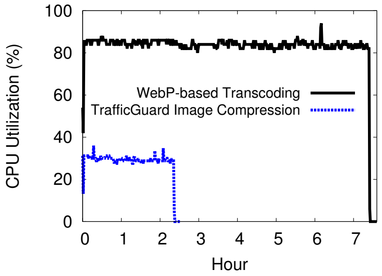

Figure 13: CPU overhead of different image compression strategies.

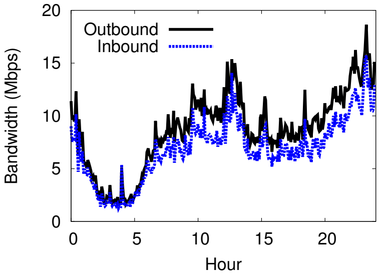

Figure 14: Inbound and outbound bandwidth for back-end servers.

Table 5: Top-10 applications served by TrafficGuard, ordered by popularity and by greatest traffic reduction.

By User Ratio (UR)			By Traffic Saving Ratio (TSR)		
App Name	**UR**	**TSR**	**App Name**	**UR**	**TSR**
WeChat	74%	22%	Android Browser	0.11%	84%
QQ	66%	22%	Zhihu Q&A	0.15%	81%
Baidu Search	29%	21%	iAround	0.03%	63%
Taobao	23%	42%	No.1 Store	0.26%	61%
QQBrowser	22%	27%	Baidu News	0.45%	57%
Sogou Pinyin	20%	12%	Tiexue Military	0.01%	56%
Baidu Browser	16%	30%	WoChaCha	0.34%	54%
Toutiao News	14%	22%	Mogujie Store	0.91%	53%
Sohu News	10%	30%	Koudai Store	0.26%	53%
QQ Zone	10%	33%	Papa Photo	0.02%	52%

At last, we wonder how TrafficGuard's traffic reduction gains are spread across user apps. Table 5 lists the top-10 apps ordered by popularity (the fraction of users with the app) as well as by the fraction of traffic eliminated. We observe that TrafficGuard is able to eliminate 12%–42% of traffic for popular apps, but that the apps with the greatest traffic savings (52%–84%) tend to be unpopular. This indicates that the developers of popular apps may already be taking steps to optimize their network traffic, while most unpopular apps can hardly become mobile-friendly in the near future.

6.3 System Overhead

Cloud-side overhead. The major cost of operating TrafficGuard lies in provisioning back-end cloud servers and supplying them with bandwidth. TrafficGuard has been able to support ~0.2M users who send ~90M requests per day using only 23 commodity servers (HP ProLiant DL380). The configuration of each server is: 2*4-core Xeon CPU E5-2609 @2.50GHz, 4*8-GB memory, and 6*300-GB 10K-RPM SAS disk (RAID-6).

Figure 12 illustrates the CPU/memory utilization of cloud servers on a typical day. Mainly thanks to our lightweight image compression strategy, the CPU utilization stays below 40%. Further, to compare the computation overhead of our image compression strategy with Flywheel's WebP-based transcoding (based on the cwebp [5] encoder), we conduct offline experiments on two identical server machines using 1M correct images randomly picked from TGdataset as the workload. Images are compressed one by one without intermission. The results in Figure 13 confirm that the computation overhead (= average CPU utilization × total running time) of TrafficGuard image compression is only a small portion (10%–12%) of that of WebP-based transcoding.

Memory utilization is typically >90% since content is in-memory cached whenever possible. Using a higher memory capacity, say 1 TB per server, can accelerate the back-end processing and thus decrease the corresponding latency penalty. Nonetheless, as shown in Figure 16, the back-end processing latency constitutes only a minor portion of the total latency penalty, so we do not consider extending the memory capacity in the short term.

Figure 14 reveals the inbound/outbound bandwidth for back-end servers. Interestingly, we observe that a back-end server uses more outbound bandwidth than inbound, though inbound traffic has been optimized. This happens because the back-end has a 38% cache hit rate (with 4 TB of disk cache), so many objects are downloaded from the Internet once but then downloaded by many clients.

Client-side overhead. The client-side overhead of TrafficGuard comes from three sources: memory, computation, and battery usage. The memory usage is modest, requiring 40 MB for local blacklists, and 10–20 MB for the VBWC mapping table. Similarly, while running on a typical (8-core ARM CPU @1.7GHz) Android smartphone, TrafficGuard's single-core CPU usage is generally below 20% when the cellular modem is active, and almost zero when the network is inactive.

To understand the impact of TrafficGuard on battery life, we record the battery power consumption of its mobile app when the child proxy is processing data packets. As shown in Figure 15, its working-state battery power is 93 mW on average, given that the battery capacity of today's smartphones lies between 5–20 Wh and their working-state battery power lies between 500 mW and a few watts [35, 45]. We also conduct micro-benchmarks to examine specific facets of TrafficGuard's battery consumption (see Appendix C for details). Micro-benchmark results illustrate that TrafficGuard can effectively reduce the battery consumption of user apps by optimizing their traffic.

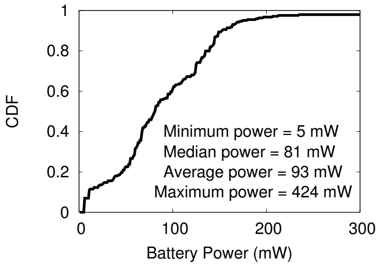

Figure 15: Distribution of client-side battery power consumption.

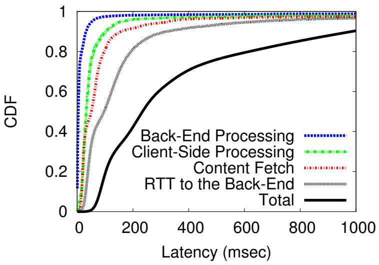

Figure 16: Latency for each phase content processing and retrieval.

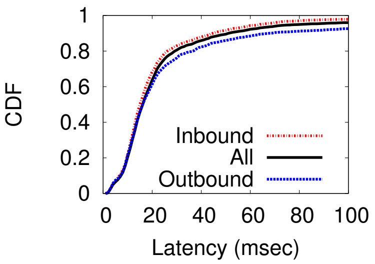

Figure 17: Latency for clients' processing data packets.

6.4 Latency Penalty

As TrafficGuard forwards HTTP GET requests to a back-end proxy rather than directly to the source, it may add response latency to clients' requests. In addition, client-side packet processing by the child proxy also brings extra latency. To put the latency penalty into perspective, first, we note three mitigating factors that effectively reduce latency: 1) TrafficGuard filters out ∼10.3% of requests locally, which eliminates all latency except for client-side processing; 2) the ∼21.2% of traffic that is not owing to HTTP GET requests is delivered over the Internet normally, thus only incurring the latency penalty for client-side processing; and 3) 38% of HTTP GET requests hit the back-end Squid cache, thus eliminating the time needed to fetch the content from the Internet.

Next, to understand TrafficGuard's latency penalty in the worst-case scenario (unfiltered HTTP GETs that do not hit the Squid cache), we examine latency data from TGworklog. Figure 16 plots the total latency of requests that go through the TrafficGuard back-end and miss the cache, as well as the individual latency costs of four aspects of the system: 1) processing time on the client side, 2) processing time in the back-end, 3) time for the back-end to fetch the desired content, and 4) the RTT from the client to the back-end. Figure 16 shows that both client-side processing and back-end processing add little delay to requests. Instead, the majority of delay comes from fetching content, and the RTT from clients to the back-end cloud. Interestingly, Figure 17 reveals that the average processing time of an outbound packet is longer than that of an inbound packet, although outbound packets are usually smaller than inbound packets. This is because the client-side filtering of malicious links and ads is the major source of client-side latency penalty.

In the worst-case scenario, we see that TrafficGuard does add significant latency to user requests. If we conservatively assume that clients can fetch content with the same latency distribution as Baidu's servers, then TrafficGuard adds 131 ms of latency in the median case and 474 ms of latency in the average case. However, if we take into account the three mitigating factors listed at the beginning of this section (which all reduce latency), the median latency penalty across all traffic is reduced to merely 53 ms, and the average is reduced to 282 ms.

7 Conclusion

Traffic optimization is a common desire of today's cellular users, carriers, and service developers. Although several existing systems can optimize the cellular traffic for specific apps (typically web browsers), cross-app systems are much rarer, and have not been comprehensively studied. In this paper, we share our design approach and implementation experiences in building and maintaining TrafficGuard, a real-world cross-app cellular traffic optimization system used by 10 million users.

To design TrafficGuard, we took a measurement-driven methodology to select optimization strategies that are not only high-impact (*i.e.,* they significantly reduce traffic) but also efficient, easy to implement, and compatible with heterogenous apps. This methodology led to some surprising findings, including the relative ineffectiveness of text compression. Real-world performance together with trace-driven experiments indicate that our system meets its stated goal of reducing traffic (by 36% on average), while also being efficient (23 commodity servers are able to handle the entire workload). In the future, we plan to approach cellular carriers about integrating TrafficGuard into their networks, since this will substantially decrease latency penalties for users and simplify the overall design of the system.

Acknowledgments. We wish to thank the following people for their contributions to the system or the paper. Changqing Han, Junyi Shao, Xuefeng Luo, Min Guo, Cheng Peng, Tianwei Wen, and Zhefu Jiang helped develop the system. Yuxuan Yan and Jian Chen helped evaluate the client-side latency penalty and battery consumption. Our shepherd Matt Welsh guided the preparation of the camera-ready paper.

This work is supported by the High-Tech R&D ("863 – China Cloud") Program of China under grant 2015AA01A201, China NSF under grants 61432002 and 61471217, NSF under grants IIS-1321083, ECCS-1247944, CNS-1526638 and CNS-1319019, and CCF-Tencent Open Fund under grant IAGR20150101.

References

[1] Adblock Plus EasyList for ad blocking. http://easylist.adblockplus.org.

[2] Android OS background data increase since 4.4.4 update. http://forums.androidcentral.com/moto-g-2013/422075-android-os-background-data-increase-since-4-4-4-update-please-help.html.

[3] Android OS is continuously downloading something in the background. http://android.stackexchange.com/questions/28100/android-os-is-continuously-downloading-something-in-the-background-how-can-i.

[4] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2014-2019 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[5] cwebp – Compress an image file to a WebP file. http://developers.google.com/speed/webp/docs/cwebp.

[6] Data Sense for Windows Phone apps. http://www.windowsphone.com/en-us/how-to/wp8/connectivity/use-data-sense-to-manage-data-usage.

[7] dwebp – Decompress a WebP file to an image file. http://developers.google.com/speed/webp/docs/dwebp.

[8] Google Safe Browsing. http://developers.google.com/safe-browsing.

[9] Google to websites: Be mobile-friendly or get buried in search results. http://mashable.com/2015/04/21/google-mobile-search-2/#UbuurRKFaPqU.

[10] How to Minimize Your Android Data Usage and Avoid Overage Charges. http://www.howtogeek.com/140261/how-to-minimize-your-android-data-usage-and-avoid-overage-charges.

[11] HTTP ETag. http://en.wikipedia.org/wiki/HTTP_ETag.

[12] imghdr – Determine the type of an image. http://docs.python.org/2/library/imghdr.html.

[13] Institutional review board (IRB). https://en.wikipedia.org/wiki/Institutional_review_board.

[14] Netequalizer Bandwidth Shaper. http://www.netequalizer.com.

[15] Onavo Extend for Android. http://www.onavo.com/apps/android_extend.

[16] Opera Max. http://www.operasoftware.com/products/opera-max.

[17] Opera Max, China's version. http://www.oupeng.com/max.

[18] Opera Turbo mobile web proxy. http://www.opera.com/turbo.

[19] Packeteer WAN Optimization Solutions. http://www.bluecoat.com/packeteer.

[20] Peribit WAN Optimization. http://www.juniper.net.

[21] QQBrowser. http://browser.qq.com.

[22] Riverbed Networks. http://www.riverbed.com.

[23] System.Drawing.Bitmap class in the .NET Framework. http://msdn.microsoft.com/library/system.drawing.bitmap(v=vs.110).aspx.

[24] The State Of Digital Experience Delivery, 2015. https://www.forrester.com/The+State+Of+Digital+Experience+Delivery+2015/fulltext/-/E-RES120070.

[25] UCBrowser. http://www.ucweb.com.

[26] Universal TUN/TAP device driver. http://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[27] WebP: A new image format for the Web. http://developers.google.com/speed/webp.

[28] Ziproxy: the HTTP traffic compressor. http://ziproxy.sourceforge.net.

[29] AGABABOV, V., BUETTNER, M., CHUDNOVSKY, V., COGAN, M., GREENSTEIN, B., MCDANIEL, S., PIATEK, M., SCOTT, C., WELSH, M., AND YIN, B. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proc. of NSDI* (2015), USENIX, pp. 367–380.

[30] AGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proc. of NSDI* (2010), USENIX, pp. 419–432.

[31] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. of SIGCOMM* (2008), ACM, pp. 219–230.

[32] ANAND, A., MUTHUKRISHNAN, C., AKELLA, A., AND RAMJEE, R. Redundancy in Network Traffic: Findings and Implications. In *Proc. of SIGMETRICS* (2009), ACM, pp. 37–48.

[33] ANAND, A., SEKAR, V., AND AKELLA, A. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In *Proc. of SIGCOMM* (2009), ACM, pp. 87–98.

[34] AUCINAS, A., VALLINA-RODRIGUEZ, N., GRUNENBERGER, Y., ERRAMILLI, V., PAPAGIANNAKI, K., CROWCROFT, J., AND WETHERALL, D. Staying Online While Mobile: The Hidden Costs. In *Proc. of CoNEXT* (2013), ACM, pp. 315–320.

[35] CARROLL, A., AND HEISER, G. An Analysis of Power Consumption in a Smartphone. In *Proc. of USENIX ATC* (2010).

[36] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proc. of MobiCom* (2015), ACM, pp. 592–603.

[37] FALAKI, H., LYMBEROPOULOS, D., MAHAJAN, R., KANDULA, S., AND ESTRIN, D. A First Look at Traffic on Smartphones. In *Proc. of IMC* (2010), ACM, pp. 281–287.

[38] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext transfer protocol – HTTP/1.1, 1999.

[39] HUANG, J., QIAN, F., MAO, Z., SEN, S., AND SPATSCHECK, O. Screen-Off Traffic Characterization and Optimization in 3G/4G Networks. In *Proc. of IMC* (2012), ACM, pp. 357–364.

[40] ISAACMAN, S., AND MARTONOSI, M. Low-Infrastructure Methods to Improve Internet Access for Mobile Users in Emerging Regions. In *Proc. of WWW* (2011), ACM, pp. 473–482.

[41] JOHNSON, D., PEJOVIC, V., BELDING, E., AND VAN STAM, G. Traffic Characterization and Internet Usage in Rural Africa. In *Proc. of WWW* (2011), ACM, pp. 493–502.

[42] Li, Z., Huang, Y., Liu, G., Wang, F., Zhang, Z.-L., and Dai, Y. Cloud Transcoder: Bridging the Format and Resolution Gap between Internet Videos and Mobile Devices. In *Proc. of NOSSDAV* (2012), ACM, pp. 33–38.

[43] Li, Z., Jin, C., Xu, T., Wilson, C., Liu, Y., Cheng, L., Liu, Y., Dai, Y., and Zhang, Z.-L. Towards Network-level Efficiency for Cloud Storage Services. In *Proc. of IMC* (2014), ACM, pp. 115–128.

[44] Li, Z., Wilson, C., Xu, T., Liu, Y., Lu, Z., and Wang, Y. Offline Downloading in China: A Comparative Study. In *Proc. of IMC* (2015), ACM, pp. 473–486.

[45] Liu, Y., Xiao, M., Zhang, M., Li, X., Dong, M., Ma, Z., Li, Z., and Chen, S. GoCAD: GPU-assisted Online Content Adaptive Display Power Saving for Mobile Devices in Internet Streaming. In *Proc. of WWW* (2016), ACM.

[46] Lumezanu, C., Guo, K., Spring, N., and Bhattacharjee, B. The Effect of Packet Loss on Redundancy Elimination in Cellular Wireless Networks. In *Proc. of IMC* (2010), ACM, pp. 294–300.

[47] Naylor, D., Schomp, K., Varvello, M., Leontiadis, I., Blackburn, J., Lopez, D., Papagiannaki, K., Rodriguez, P., and Steenkiste, P. multi-context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proc. of SIGCOMM* (2015), ACM, pp. 199–212.

[48] Pucha, H., Andersen, D., and Kaminsky, M. Exploiting Similarity for Multi-Source Downloads Using File Handprints. In *Prof. of NSDI* (2007), USENIX.

[49] Qian, F., Huang, J., Erman, J., Mao, Z., Sen, S., and Spatscheck, O. How to Reduce Smartphone Traffic Volume by 30%? In *Proc. of PAM* (2013), Springer, pp. 42–52.

[50] Qian, F., Quah, K., Huang, J., Erman, J., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Web Caching on Smartphones: Ideal vs. Reality. In *Proc. of MobiSys* (2012), ACM, pp. 127–140.

[51] Qian, F., Sen, S., and Spatscheck, O. Characterizing Resource Usage for Mobile Web Browsing. In *Proc. of MobiSys* (2014), ACM, pp. 218–231.

[52] Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization. In *Proc. of WWW* (2012), ACM, pp. 51–60.

[53] Rabin, M. *Fingerprinting by Random Polynomials*. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[54] Rao, A., Kakhki, A., Razaghpanah, A., Tang, A., Wang, S., Sherry, J., Gill, P., Krishnamurthy, A., Legout, A., Mislove, A., and Choffnes, D. Using the Middle to Meddle with Mobile. *Tech. Report NEU-CCS-2013-12-10, CCIS, Northeastern University*.

[55] Rhea, S., Liang, K., and Brewer, E. Value-based Web Caching. In *Proc. of WWW* (2003), ACM, pp. 619–628.

[56] Sherry, J., Lan, C., Popa, R., and Ratnasamy, S. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proc. of SIGCOMM* (2015), ACM, pp. 213–226.

[57] Spring, N., and Wetherall, D. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proc. of SIGCOMM* (2000), ACM, pp. 87–95.

[58] Vergara, E., Sanjuan, J., and Nadjm-Tehrani, S. Kernel Level Energy-efficient 3G Background Traffic Shaper for Android Smartphones. In *Prof. of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC)* (2013), IEEE, pp. 443–449.

[59] Viennot, N., Garcia, E., and Nieh, J. A Measurement Study of Google Play. In *Proc. of SIGMETRICS* (2014), ACM, pp. 221–233.

[60] Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[61] Wessels, D. *Squid: The Definitive Guide*. O'Reilly Media, Inc., 2004.

[62] Woo, S., Jeong, E., Park, S., Lee, J., Ihm, S., and Park, K. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proc. of MobiSys* (2013), ACM, pp. 319–332.

[63] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. An Untold Story of Redundant Clouds: Making Your Service Deployment Truly Reliable. In *Proc. of HotDep* (2013), ACM.

[64] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. Heading Off Correlated Failures through Independence-as-a-Service. In *Proc. of OSDI* (2014), USENIX, pp. 317–334.

A Performance Analysis of Cross-App RE Techniques and VBWC

In this appendix, we discuss alternative caching strategies to VBWC, and motivate our ultimate selection of VBWC for TrafficGuard. A common approach to optimizing cross-app cellular traffic is called redundancy elimination (RE) that removes repeated data transfer [57, 31, 33, 32, 30, 46, 62, 49, 63, 64]. It can be deployed in ISP middleboxes [14, 19, 20, 22], on Internet routers [31], or in an end-to-end manner (*i.e.,* EndRE [30]).

RE relies on a pair of synchronized packet caches deployed at each end of a network path [57]. At one end, the sender (*e.g.,* the parent proxy in TrafficGuard) compresses data packets by replacing sequences of bytes that have appeared in previous packets with fixed-size pointers. At the other end, the receiver (*e.g.,* the child proxy in TrafficGuard) decodes data packets by following the pointers and replacing compressed data with the cached original data.

Informed Marking RE. Lumezanu *et al.* point out that TCP/IP packet loss (also including the cases of packet disorder and retransmission) can considerably degrade the performance of RE in cellular networks, and thus propose the enhanced *Informed Marking* RE algorithm [46]. To quantitatively understand the effect of Informed Marking RE, we conduct trace-driven simulations based on TGdataset. The simulation results indicate that merely 4.6% of HTTP traffic and 1.5% of non-HTTP traffic can be saved.

EndRE vs. VBWC. EndRE (*i.e.,* end-to-end RE) usually runs above the transport layer, so it is immune to TCP/IP packet loss. Following the design principle in [30], we simulate EndRE on TGdataset, and observe that as high as 10.2% of HTTP traffic can be saved — even better than the savings of VBWC (9%).

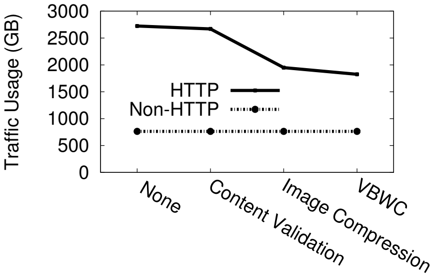

Figure 18: Real-world cellular traffic usage optimized by each mechanism.

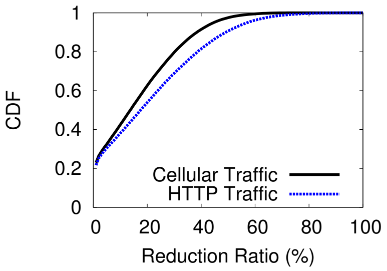

Figure 19: Distribution of real-world traffic reduction ratios across users.

Unfortunately, EndRE incurs much higher complexity in terms of implementation, computation, and cache maintenance. First, it is fairly straightforward to implement VBWC (refer to § 5.4), but implementing EndRE is not simple. Second, a poorly-provisioned EndRE client needs 60 MB of memory [30], which is even larger than the total client-side memory overhead (< 60 MB) of TrafficGuard. Even worse, the server-side memory overhead of EndRE can be hundreds of times higher than that of VBWC. Third, EndRE needs to maintain simultaneous TCP connections to guarantee cache consistency [30], while VBWC uses soft-state and is robust to temporary cache inconsistency.

Collaborative Caching. By conducting a week-long measurement of 3G traffic at a large cellular ISP in South Korea, Woo *et al.* observe that simple TCP-level RE can save 27%–42% of traffic with a *collaborative cache* of 512 GB [62]. However, such saving ratios can only be acquired at a centralized vantage point in the cellular backhaul networks, rather than an end point from a cellular user's perspective. In fact, the dataset collected by Woo *et al.* does not contain the identifiers (*e.g.,* IMEI or IMSI) of user devices, thus making the per-user analysis of traffic saving impossible.

B Real-world Traffic Reduction Results

As mentioned in § 6.1, the real-world working logs of TrafficGuard (*i.e.,* TGworklog) do not include the detailed information of filtered traffic (since they never occurred in reality). In other words, TGworklog mainly records the known traffic reduction results on the cloud side, through the traffic optimization mechanisms of content validation, image compression, and VBWC.

Meanwhile, as we note in § 4.1, the mobile app of TrafficGuard provides the user with an interface for displaying various cellular usage statistics, particularly the traffic saving statistics. Here the traffic saving statistics are also extracted from the real-world working logs of TrafficGuard, so they are less than the overall traffic savings studied in § 6.2.

As shown in Figure 18, HTTP traffic is reduced by 33%, while non-HTTP traffic cannot be reduced since non-HTTP traffic is not forwarded and processed by the back-end servers of TrafficGuard. In total, 26% of cellular traffic is reduced according to TGworklog.

In detail, we plot the distribution of real-world cellular/HTTP traffic reduction ratios across users in Figure 19. We observe that 38% of users saved over a quarter of HTTP traffic, and 10% of users saved over a half. In comparison, 29% of users saved over a quarter of cellular traffic, and merely 2.5% users saved over a half.

C Micro-Benchmark Results of TrafficGuard's Battery Consumption

To understand specific facets of TrafficGuard's battery consumption, we conduct micro-benchmarks on the client side with three popular, diverse user apps: the stock Android Browser, WeChat (the most popular app in China, similar to WhatsApp), and Youku (China's equivalent of YouTube). In each case, we drove the app for five minutes with and without TrafficGuard enabled while connected to a 4G network.

Figures 20, 21, and 22 show the battery usage in each experiment. Meanwhile, Figures 23, 24, and 25 depict the corresponding CPU usage; Figures 26, 27, and 28 plot the corresponding memory usage. All these results reveal that in cases where TrafficGuard can effectively reduce network traffic (*e.g.,* while browsing the web), it also saves battery life or has little impact on battery life, because the user app needs to process less traffic; accordingly, TrafficGuard does not increase CPU/memory usage on the whole. However, in cases where TrafficGuard can hardly reduce any traffic (*e.g.,* Youku video streaming), it reduces battery life and increases CPU/memory usage. Thus, we are planning to improve the design of TrafficGuard, in order that it can recognize and bypass the traffic from audio/video streams.

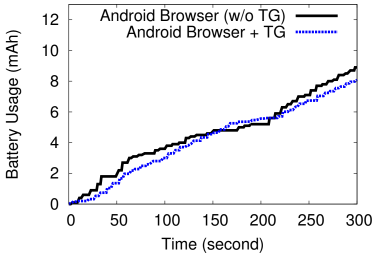

Figure 20: Battery usage of Android Browser with and without TrafficGuard (abbreviated as TG).

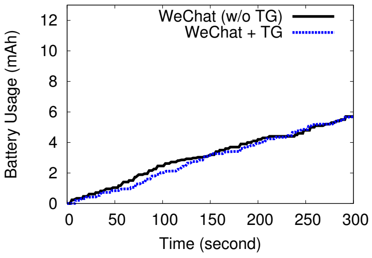

Figure 21: Battery usage of WeChat with and without TrafficGuard.

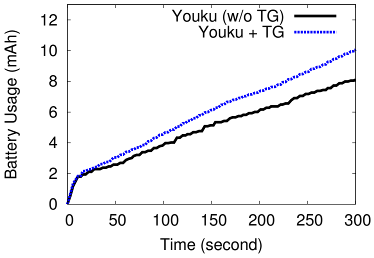

Figure 22: Battery usage of Youku with and without TrafficGuard.

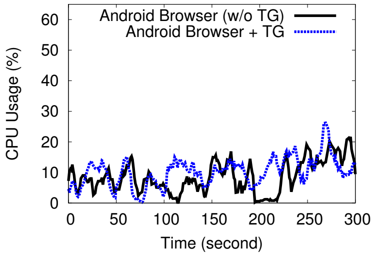

Figure 23: CPU usage of Android Browser w/ and w/o TrafficGuard.

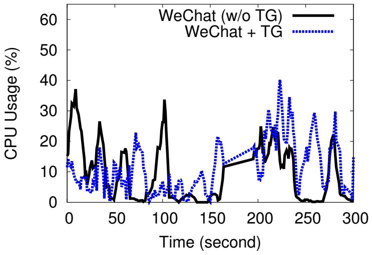

Figure 24: CPU usage of WeChat with and without TrafficGuard.

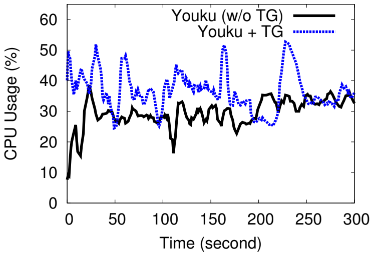

Figure 25: CPU usage of Youku with and without TrafficGuard.

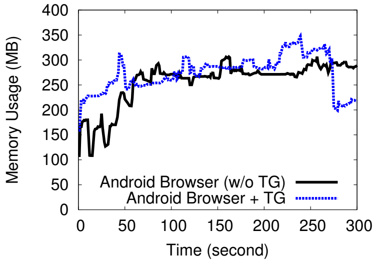

Figure 26: Memory usage of Android Browser with and without TrafficGuard.

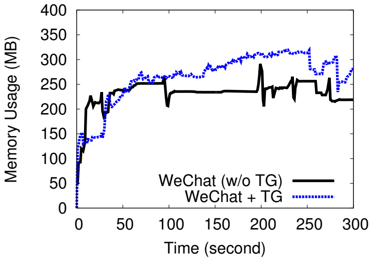

Figure 27: Memory usage of WeChat with and without TrafficGuard.

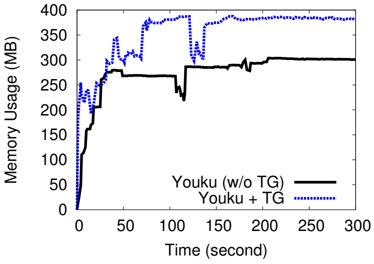

Figure 28: Memory usage of Youku with and without TrafficGuard.