APPx: An Automated App Acceleration Framework for Low Latency Mobile App

Byungkwon Choi, Jeongmin Kim, Daeyang Cho, Seongmin Kim, Dongsu Han
KAIST

ABSTRACT

Minimizing response time of mobile applications is critical for user experience. Existing work predominantly focuses on reducing mobile Web latency, whereas users spend more time on native mobile apps than mobile Web. Similar to Web, mobile apps contain a chain of dependencies between successive requests. However, unlike Web acceleration where object dependencies can easily be identified by parsing Web documents, App acceleration is much more difficult because the dependency is encoded in the app binary.

Motivated by recent advances in program analysis, this paper presents a system that utilizes static program analysis to automatically generate acceleration proxies for mobile apps. Our framework takes Android app binary as input, performs program analysis to identify resource dependencies, and outputs an acceleration proxy that performs dynamic prefetching. Our evaluation using a user study from 30 participants and an in-depth analysis of popular commercial apps shows that an *acceleration proxy* reduces the median user-perceived latency by up to 64% (1,471 ms).

CCS CONCEPTS

• **Software and its engineering → Cloud computing**;

1 INTRODUCTION

Minimizing the response time of mobile apps is becoming increasingly critical as users expect mobile apps to respond quickly [14]. Response times impact the quality of user experience, which in turn affects the revenue of mobile services. Amazon reports one second of additional page load latency costs $1.6 billion in sales each year [10], and for Google, a 250 ms delay in search response can result in 8 million search losses per day [10].

At the same time, mobile apps are becoming much more popular than their mobile Web counterparts, dominating user attention by a factor of six times [15]. However, when it comes to reducing response times, mobile app acceleration is surprisingly underexplored in contrast to the large body of work in Web acceleration [39, 49, 52, 56–59, 61, 67, 68, 70, 73, 74].

Existing work on mobile app acceleration predominantly adopts a client-based approach [32, 46, 75] or focuses on computation offloading [36, 37, 42, 51, 71]. However, each approach has its own limitations. The client-based approach inherently suffers from a lack of resources on the mobile side, such as low computational power and a limited budget for cellular data usage. The computation offloading approach is mostly effective for compute-intensive apps, such as games and face recognition, but does not offer much benefit for network-intensive apps [42]. Many of these systems also require mobile OS or app modifications, which limits the deployment.

Unlike the existing approaches, this paper takes a proxy-based approach. Similar to the Web page preloading approaches [17, 20, 34, 48] that take advantage of the Web resource dependencies, our app-acceleration proxy exploits dependency relationships between network transactions (request-response pairs) and prefetches appropriate responses. This approach effectively reduces the response time of mobile app and does not require any code changes to mobile OS or client/server apps.

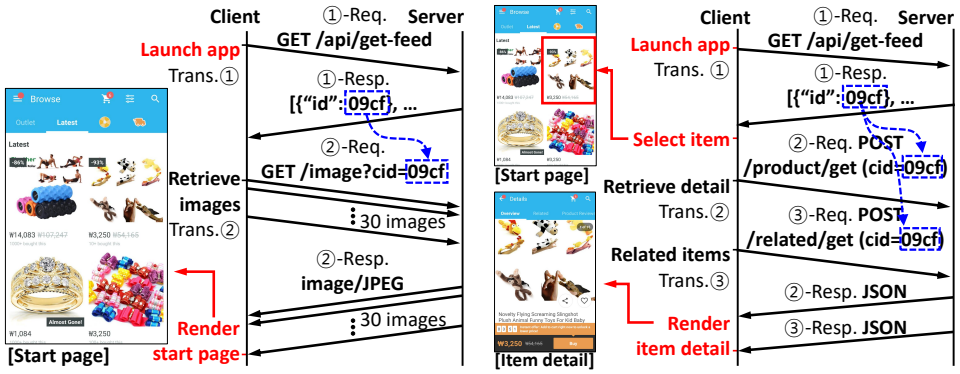
However, the core challenge is that unlike Web in which dependency information is embedded in Web documents, it is much more difficult to identify the inter-transaction dependencies in apps because the dependency information is embedded in the program itself. In addition, in contrast to Web documents that follow a standard format (e.g., HTML, CSS, Javascript) and whose source code is made available, mobile apps use a wide variety of APIs and often the source code is not available. Thus, identifying the dependencies and generating exact request messages ahead of time is very challenging.

This paper addresses the problem of automatically developing an mobile app acceleration framework by understanding the application message exchange behaviors. We present the first system, named APPx, that minimizes human effort in developing mobile app acceleration proxies. Our framework takes Android app binary as input and leverages existing protocol analysis framework to identify message formats of HTTP(S) request/response the app generates and infer dependencies between HTTP(S) transactions (request-response pairs). However, the static analysis lacks information determined at runtime. In contrast, prefetching HTTP(S) response requires the exact request message including URI, query string, header, and body. To overcome the limitation, we combine dynamic learning to fill in missing information at run-time by observing the actual traffic in the proxy. The framework then automatically finds out what and when to prefetch and generates a proxy that performs dynamic prefetching.

We present a full system implementation of the automated framework and the app-acceleration proxy. We evaluate our framework using a number of popular commercial apps on Google Play. We provide microbenchmark page load time improvement of key interactions for each app. To evaluate the system under real workload, we conduct a user study and capture the app usage of 30 participants. Our in-depth evaluation shows that the automatically generated proxy reduces the app response time by up to 62% and delivers 55% reduction of the response time on average.

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281416



(a) when app is launched (b) when choosing an item

Figure 1: Dependency found in #1 shopping app Wish [4]. Blue-dotted lines show dependency. "Trans." indicates a transaction.

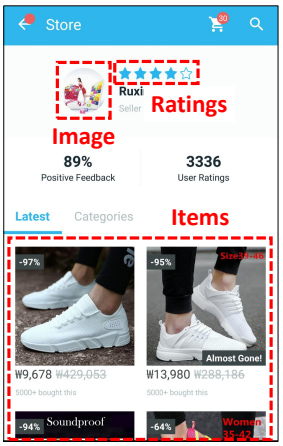


Figure 2: Merchant page.

In summary, we make the following contributions:

- We propose the first system to automatically generate an app acceleration proxy tailored to target apps.
- We present a design of the acceleration proxy. To produce accurate request messages ahead of time, we combine static analysis and dynamic learning.
- Finally, our evaluation with real apps and real user workload demonstrates the system improves the app response time up to 64%.

2 APP ACCELERATION SCENARIO

We envision an app-acceleration proxy located between user devices and remote servers and prefetches data based on dependency relationships between network messages. This approach is promising in a number of ways. First, the proxy effectively reduces the load time of static contents (e.g., image, video) as well as that of dynamically-generated or personalized contents. Second, the prefetching proxy can also hide latency even when the remote server itself is slow rather than the network [61]. Finally, it does not require modifications to client/server apps or mobile OS.

Note, in Web acceleration, such prefetching/preloading is widely used by proxies to make page load time faster [17, 20, 34, 48]. This paper demonstrates a similar technique can be used to accelerate mobile apps. We identify mobile apps contain many complex dependencies in the messages they generate. Using the #1 ranked app in the shopping category in Google Play US [4], called Wish, as an example we show message dependencies are commonly found in mobile apps. Then, we demonstrate how a proxy can accelerate them. Throughout the paper we use Wish as a working example.

Action 1: loading the start page. When the app starts, a list of recommended items along with their thumbnail images appears on the screen as shown in Figure 1(a). For this, the app first issues a `/api/get-feed` request to retrieve the item list to be displayed (Transaction ①), and the response contains a list of 30 items and their 'id's. Using the 'id's, subsequent requests retrieve a thumbnail for each item (Transaction ②). As shown in figure, the two transactions exhibit dependencies.

Action 2: selecting an item. When a user selects an item, a `/product/get` request is issued (Transaction ②) using the item 'id' from Transaction ① to retrieve item details, such as shipping information. At the same time, the app fetches related items using a `/related/get` request (Transaction ③) for the item 'id'. Finally, the item detail and related items appear on the screen. Transaction ② and ③ are dependent to Transaction ① as shown in Figure 1(b).

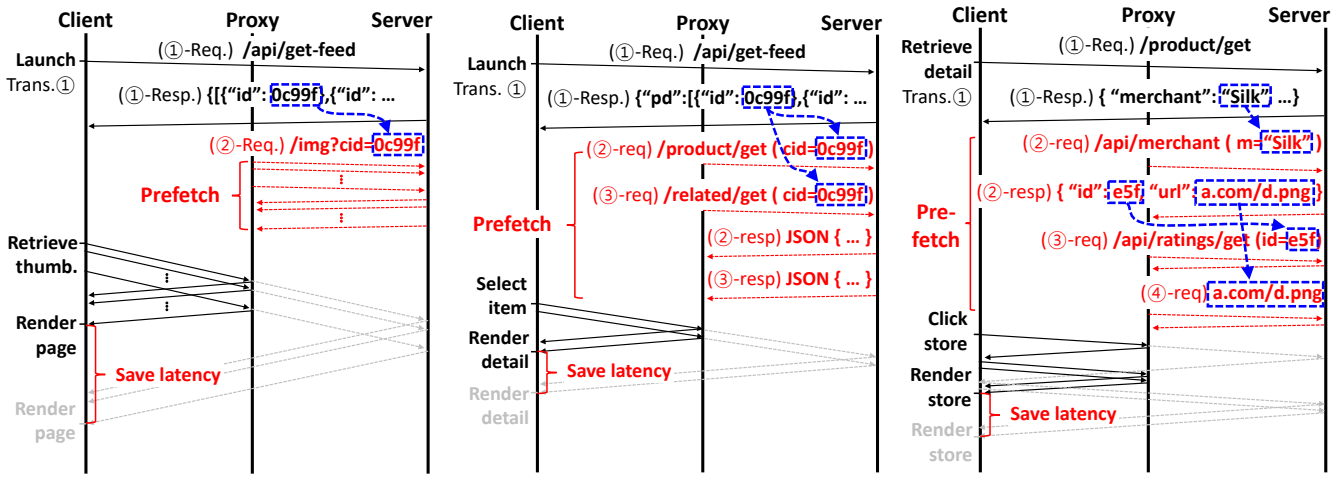
Action 3: merchant page. One can also visit a merchant page. When a user clicks on the merchant from the item detail page, the app issues a chain of transactions which takes multiple round trips. Using the merchant login name in item details (Transaction ② in Figure 1(b)), the app first issues a request to retrieve the merchant information, including its ID and items sold by the seller. The app then uses the ID to request the merchant ratings, its profile image, and other items carried by the seller as shown in Figure 2.

Note the examples are common interactions we find in typical apps, and such dependencies are common in many real-world mobile apps [42]. Through our in-depth evaluation of five top-ranked commercial Android apps, we identify that there are hundreds of dependencies whose maximum length of the successive dependency is 15 (§6.1).

Proxy-based app acceleration. We now show how our proxy reduces the page load time for each of the three cases.

Figure 3(a) shows accelerating app's start page. When a user launches Wish, the app issues a `/api/get-feed` request to retrieve the item list. When the list arrives at the proxy, it prefetches thumbnails by constructing multiple parallel requests using the 'id's from the item list, as shown in Figure 3(a). When the client requests for thumbnails, the proxy directly serves them, reducing the start page load time.

The proxy can accelerate other interactions. Figure 3(b) illustrates a timeline of app interaction when the proxy prefetches the item details and related items to accelerate the item details page (Figure 1(b)). Due to the prefetch, the app experiences faster page load. Finally, the proxy can also prefetch a chain of requests when loading the merchant as shown in Figure 3(c). In this case, the proxy first uses a merchant name in the item detail to prefetch the



(a) when app is launched (b) when choosing an item (c) when clicking on a store profile

Figure 3: Prefetch example for Wish. Blue lines and boxes indicate the dependencies. Red- and gray-dotted lines respectively indicate prefetched transactions and the original communication between client and server without the proxy. Gray-colored text indicates an original response without prefetching.

merchant information that contains the merchant ID, and issues subsequent requests to obtain the ratings and profile image of the merchant using its ID.

Deployment model. The app acceleration service model is similar to that of content distribution networks that accelerate dynamic Web content [6, 7]. We assume app service providers trust the proxy and the proxy can observe plain-text traffic between the client and server even when it is encrypted. This assumption is consistent with that of existing Web acceleration proxies [11, 56, 70, 74] and TLS/SSL proxies [11, 21]. Following this model, we assume the proxy provider is a third party, and the app service provider has contractual relationship and cooperates with the proxy provider (e.g., assists with proxy configuration) for app acceleration. Note, the proxy can accelerate multiple target apps. The proxy keeps track of user contexts (e.g., cookie) and manages prefetched response per use separately.

3 REQUIREMENTS AND CHALLENGES

Our goal is to automate the development of app acceleration proxies as much as possible and minimize the human effort. We would like user to be involved only in proxy configuration. We target mobile apps that use HTTP(S) as their primary protocol because most Android apps use HTTP(S) [38, 50, 66] because REST APIs are extremely popular [22, 27].

The system must satisfy three key requirements:

- **R1**: It must automatically identify dependencies between protocol messages.
- **R2**: The proxy should automatically reconstruct ahead of time a prefetch request identical to the original one.
- **R3**: The proxy must not alter the app behavior.

However, building a system that satisfies the requirements is challenging for a number of reasons.

C1: Complex dependencies between transactions. Identifying prefetch opportunities requires a deep understanding of complex relationships between message exchanges. They are app-specific, and a manual analysis requires significant human effort. An automated dependency analysis is promising, but it imposes a fundamental constraint: the dependency information should be rich enough to ensure high coverage and accurate enough to ensure correctness in prefetching. APPx leverages recent advances in *static program analysis* and re-purposes it for identifying prefetch opportunities. In particular, we extend Extractocol [50], a state-of-the-art static analysis tool that automatically identifies message formats and their dependencies from mobile apps (§4.1).

C2: Dynamically generated requests. To generate prefetch requests, the proxy needs to reconstruct exact request messages ahead of time. Although existing static analysis tools [30, 35, 50] provide a comprehensive characterization of mobile app behaviors including message formats, it cannot identify values that are determined at run-time. For example, the proxy cannot determine device-specific values (e.g., user-agent request header) or the host URI of HTTP requests that change dynamically until it receives exact values at run-time from a client or a server. To address the challenge, the proxy performs *dynamic learning* at run-time and adapts to its run-time behavior using the information acquired from static analysis as the baseline (§4.2).

C3: Controlling side-effect and ensuring correctness. Static analysis and dynamic learning allow us to construct prefetchable requests ahead of time. However, messages that have adverse side-effect (e.g., 1-click purchasing) must not be prefetched. In addition, prefetched messages might be stale when a response has been prefetched long ahead of time. It is difficult to resolve such issues using automated analysis because it is fundamentally linked to the app semantics. To address this issue, 1) we enable the service provider to have control on whether to prefetch a request and

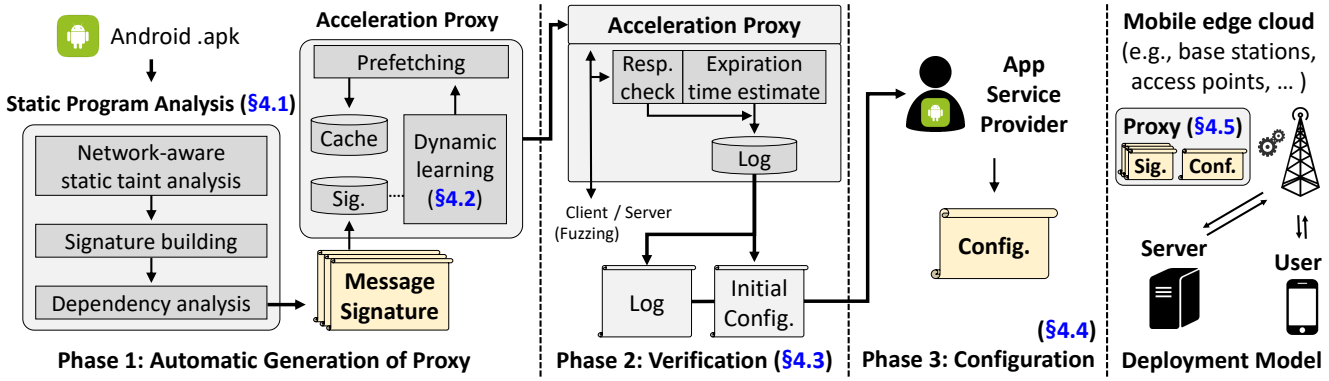


Figure 4: APPx framework overview and deployment model for acceleration proxy

set expiration times through a fine-grained proxy configuration (§4.4); and 2) the proxy performs *offline verification* on prefetch requests (§4.3). If the proxy generates incorrect requests, retrieves error message or fail to get response from the server during the verification, it excludes the signature from the prefetching target before run-time.

C4: Controlling the cost. A fundamental tradeoff exists between the response time and bandwidth usage. Unlike mobile prefetching approaches that use the resources of mobile devices [46], the proxy-based approach is much less resource constrained. Nevertheless, a careful policy design is required because cloud platforms [2, 9] charge the bandwidth usage. Naive prefetching might lead to significant bandwidth overhead. To address this issue, 1) we provide a mechanism to impose a constraint on data usage for prefetching (§4.4); and 2) our proxy tracks the cache hit-rate and the response time to prioritize frequently used interactions and a request that takes a longer time to complete (§5).

4 DESIGN

Figure 4 illustrates an overview of the APPx framework. It goes to three distinct phases before deployment. The first phase is automatic proxy generation that consists of a static program analysis module and a proxy that takes the analysis result as input. The static program analysis module takes an Android binary as input and extracts message formats for HTTP transactions (request-response pairs) and infers dependency relationships between the transactions. §4.1 presents how we extend existing static protocol analysis to overcome the challenges in identifying dependencies for app acceleration (**C1**). The proxy then uses the output to create data structures used for identifying information required for transactions from the actual network traffic the app generates. The proxy dynamically adapts to run-time conditions, learns run-time values from actual online interactions between the app and the server, reconstructs the prefetchable requests, and retrieves the corresponding response from the origin servers ahead of time. §4.2 presents the details of dynamic learning and message generation (**C2**).

The second phase performs testing and verification to ensure the proxy behaves identical to the app and does not alter the app behavior in terms of the messages the app receives (**C3**). This is akin to app and server testing during development. Requests that fail to generate server response or resolve all run-time values are filtered out in this phase. §4.3 presents the details.

Finally, APPx provides a mechanism to specify the dynamic prefetching policy of the proxy. APPx configuration allows users to set the expiry time of prefetched content, forbid prefetching for specific request(s), or impose data usage limit. The proxy configuration is designed to minimize human effort in controlling the side-effect (**C3**) and cost of prefetching (**C4**). §4.4 describes the details.

4.1 Static Program Analysis

To automatically extract message formats and dependencies, the protocol analysis module needs to 1) track all the network-related objects and their data flow and 2) identify semantics of APIs that handle the objects. Extractocol [50] combines a network-aware static taint analysis on objects with a semantic analysis to effectively provide a characterization of protocol behaviors. Using an Android app binary (apk) as an input, it automatically constructs signatures that specify app-specific HTTP transactions and inter-transaction dependencies. Figure 5 describes signature examples and their dependency relationships Extractocol identifies. Through the program analysis, Extractocol figures out dependencies between the URI/-header/body of request messages (e.g., 'cid' in Transaction ②) and other messages (e.g., 'id' in Transaction ①).

Although existing static analysis tools provide a starting point for protocol analysis, they are not designed for app acceleration. The key challenge is to ensure high coverage in identifying prefetch requests and generate accurate prefetch request signatures same as the original requests for correctness. For example, it should not miss any sub-fields in the network messages. Otherwise, it cannot be used for prefetching. To make protocol analysis more accurate, we extend Extractocol in three ways:

Supporting Android intent. It is well-known that Android has many implicit call flows [33, 60]. Extractocol covers some implicit call flows, but does not support Intents that are often used to deliver objects across different components in Android. For example, an activity may pass a message to a service using a put method with a key (e.g., `PutExtra('key', 'msg')`), and a get method (e.g., `GetIntent('key')`) is used by the service to retrieve the message.

To support Intents, our protocol analysis module constructs an "Intent map". An Intent map consists of pairs that have a key name and the corresponding value. It first finds every put method in

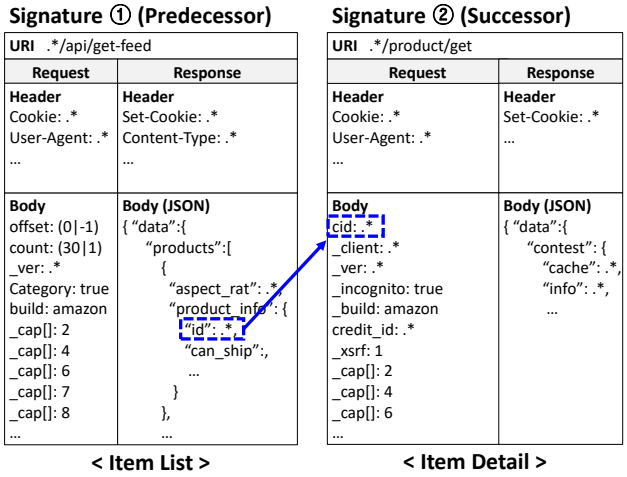


Figure 5: Signature examples of Wish. The blue-colored line and boxes indicate the dependency relationship between Signature ① and ②.

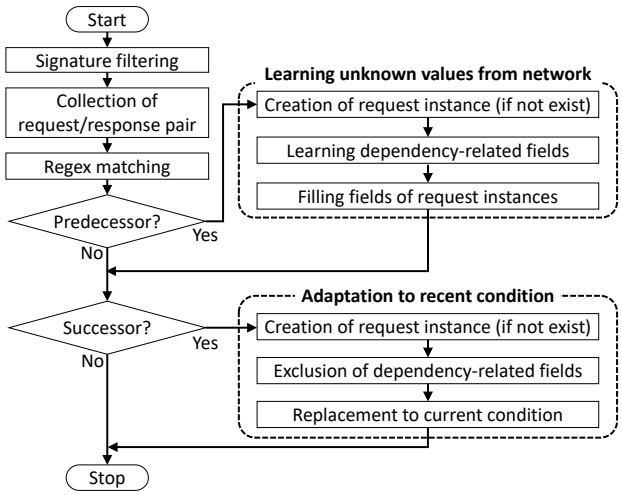


Figure 6: Flowchart diagram of dynamic learning.

the program code and performs backward slicing [50] to track the arguments of each put method. Then, the module tracks every object in the slice, reconstructs the signature of each object and repeats them for every put method. This enables identification of the signature for all the arguments of put methods. We extend Extractocol to use the Intent map to identify the value of an object passed through Intents when building HTTP message formats. This enables us to follow information delivered through and track message dependencies originating from Intents.

Supporting new programming models. Recent advances in mobile app development ecosystem introduce new programming abstractions. One example is RxAndroid [18], an Android version of ReactiveX, that forms a recent trend [25, 26]. It provides programming APIs to asynchronously handle event-based programs whose objects are constantly requested/updated, using observable sequences. Extractocol cannot handle observable sequences properly because it is unaware of the semantic of RxAndroid's APIs (e.g., `flatMap()`, `map()`, and `defer()`). In contrast, APPx's program analysis fully supports RxAndroid.

Precise alias and complex heap object analysis. When constructing HTTP(S) requests, apps typically use heap objects (HTTP Request) derived from other heap objects (e.g., URL of request message). In commercial apps, the relationship between these heap objects forms a long chain because an object contains multiple fields. For example, HTTP Request object may contain a json object that are derived from a key-value map whose value may come from an array of options. In addition, heap objects have multiple aliases that are difficult to track [30]. FlowDroid [30] in particular has an on-demand backward alias analysis module to resolve all aliases. However, Extractocol fails to track dependencies when multiple aliases form a complex relationship. To address this, we leverage the on-demand backward taint analysis used in FlowDroid. Note, Extractocol performs backward (forward) taint analysis to identify program slices that contain request (response) messages from network I/O methods. To track aliases, we apply on-demand backward analysis when identifying request slices. Similarly, we also apply on-demand forward taint analysis when identifying response slides. During the bidirectional analysis, we keep track of the relationship among heap objects in a graph (i.e., which object maps to which field in a larger object). The on-demand bidirectional taint analysis combined with heap object analysis identifies more fields and provides a more complete message signature that allows us to produce accurate prefetch messages.

4.2 Dynamic Learning

The information static analysis outputs is insufficient and thus cannot be used directly for prefetching for two major reasons: first, the regex signature identified by static analysis encompasses all possible cases. However, the actual manifestation depends on dynamic conditions unknown to the proxy a priori. For example, some request headers or some json fields in the request body may not be present depending on dynamic conditions due to branch condition in the code. To tackle this challenge, we design the proxy to adapt to the most recent condition observed through network messages. Second, the regex signatures contain wildcard values that must be resolved at run-time. The key challenge here is to automatically construct a complete prefetch request by learning the unknown values from network messages. For this, the proxy constantly learns these values from network messages and automatically updates its state.

Figure 6 shows the workflow of how the proxy performs dynamic learning to address the issues above. First, the proxy filters out signatures that do not exhibit dependencies because it is only interested in prefetchable requests. Next, for each HTTP request-response pair (HTTP transaction) it receives, it identifies a learning target which is the signature whose network message format corresponds to the request-response pair. To identify the learning target, the proxy performs regular expression matching on the URI of the incoming transaction with that of the signatures. By design, when a match is found the transaction is part of a dependency chain. When a field in signature A is derived from (or dependant on) a field in signature B, we call the former a *successor* and the latter a *predecessor*. For example, as shown in Figure 5, the 'cid' field of signature ② is directly derived from the 'id' field of signature ①. Thus,

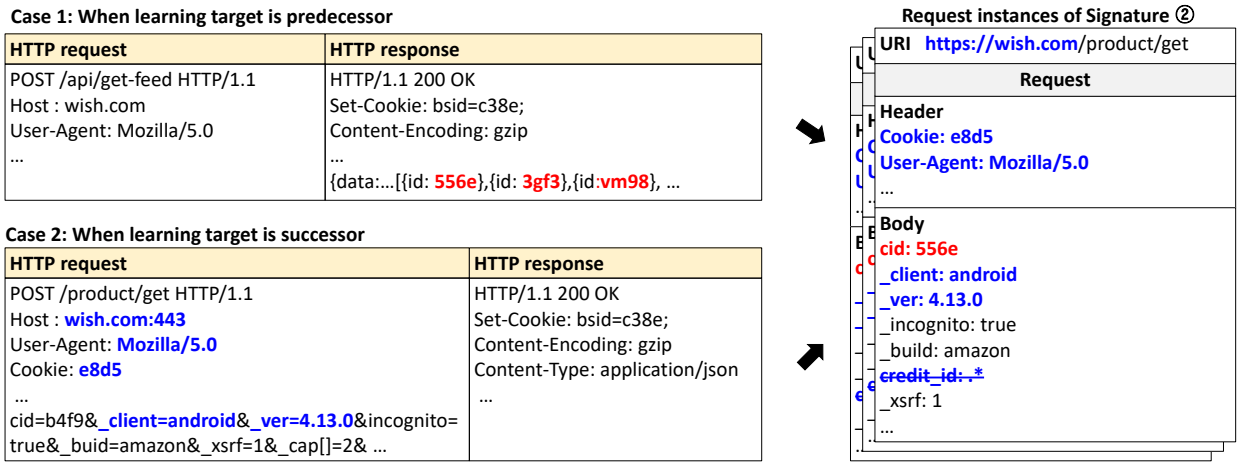


Figure 7: Dynamic learning example.

signature ① is a predecessor of signature ②, and ② is a successor of ①. The proxy handles the two cases differently.

When the learning target is a predecessor, the proxy learns fields that are used to construct successor requests. The proxy first creates a successor instance identical to the corresponding successor signature, and copies the missing information from the incoming transactions. Each step of learning makes the request more specific. Figure 7 illustrates an actual example of how the prefetch instance evolves over time. The proxy creates a prefetch request instance from the successor's signature, learns the 'id' field from a predecessor response, and fills the 'cid' field to the request instance. In this case, there are multiple 'id' fields in the response and the proxy replicates the request instance as many as the number of the 'id' fields. Each instance has a different 'cid' field according to the learned value. When the prefetch request is complete and has no more missing values to learn, the proxy issues the prefetch request.

When the learning target is a successor, the proxy has a chance to learn from an actual example because it is a (prefetchable) request instance in itself. In this case, the proxy learns dynamic conditions from the actual message and mimics the behavior. The learning target signature usually contains more fields than run-time messages because it enumerates all possible cases, but inclusion of some fields are determined by run-time conditions. Figure 8 shows such an example where a common request body signature results in multiple possible instance classes. The proxy remembers all possible instance classes and selects the most recent one. In addition, it also learns missing values, such as HTTP header fields (e.g., User-Agent value) from the instances derived from the same signature. For example, in Figure 7 (case 2), the proxy adapts the request instance to the most recent condition observed and replaces the wildcard part in the URI of the request instance to the host field of the latest instance of the same signature ('https://wish.com'). The 'Cookie' and 'User-Agent' fields in header and '_client' and '_ver' in the request body are also updated. The 'credit_id' field is removed because the incoming request does not have the field. The proxy completes the request instance which becomes ready for prefetching.

In summary, using dynamic learning through predecessor and successor, the proxy builds complete request instances identical to

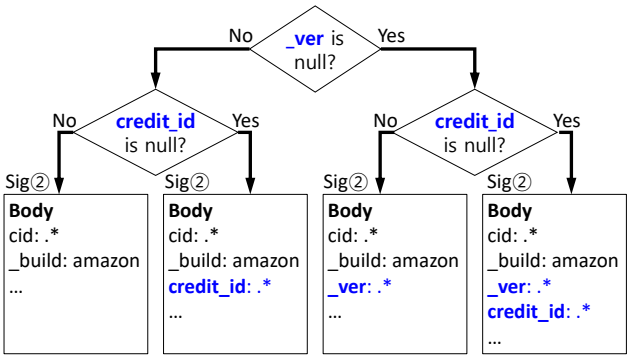


Figure 8: Example request body signatures based on branch conditions in Wish.

the request that the app originally generates. Finally, some signatures can be a predecessor as well as a successor. For those signatures, the proxy executes both predecessor and successor routines as illustrated in Figure 6.

4.3 Testing and Verification

APPx performs testing and verification to ensure correctness before its deployment. This phase uses UI-fuzzing tools [23, 45] to generate random streams of user events, at the client side, such as touches or gestures as well as number of system-level events. Using the tools, the app generates and transmits actual requests to remote server through the proxy. The proxy performs the reconstruction of requests and prefetching. If the proxy retrieves an error message or fails to get any responses from remote servers, this phase finally logs the response state and the corresponding dependency relationship and updates the configurations to disable prefetching for the particular transactions. Note, this phase also collects and logs an estimate of the expiration time for each prefetch request. For this, the proxy periodically prefetches and checks the difference between the new one and old one. The prefetch period is getting increased until the new one is different with the old one. The proxy logs the period.

This phase ensures the proxy does not reconstruct the abnormal requests that cause an error message or timeout. However, this

```
Probability 0.5

Signature {
        hash: ar93ba;
        uri: .*/api/get-feed;
        expiration_time: none;
        prefetch: false;
}

(continue on the right)
```

```
Signature {
        hash: 3853be;
        uri: .*/product/get;
        expiration_time: 1 day;
        prefetch: true;
        probability: 0.8;
        add_header: "proxy","prefetch";
        condition: "price" gt "1000";
}
(the rest omitted)
```

Figure 9: Example configuration

phase cannot identify an exact expiry date that an app service provider wants or which requests must not be prefetched. To ensure the proxy delivers fresh data and is consistent with the app usage semantics, a careful configuration is required.

4.4 Configuration

Our framework provides a mechanism for enforcing fine-grained policies on proxy's prefetching behavior. Unlike automatic analysis and prefetching, implementing a policy requires understanding of the app usage and service semantics. This may require assistance from the app service provider, similar to Web configuration (e.g., expiration time, server push, and proxy preload configurations). Specifically, APPx provides three types of configuration: (1) setting an expiry time of prefetched responses; (2) probabilistic prefetching; and (3) setting a field-specific prefetching policy.

When a response has been prefetched long ahead of time, it may become stale. For example, the number of comments and purchasers in a shopping app change over time. Our configuration supports expiration time for each response. Because setting an expiration time for each request might be tedious, we provide a default expiration time by estimating it from the logs in the testing and verification phase.

We also support probabilistic prefetching to control proxy behavior and manage data usage. Some prefetchable requests are always generated without user intervention, and prefetching them does not incur additional bandwidth overhead. However, some require user action (e.g., click) and prefetching them does not provide benefit unless the user actually performs the action. Some even have adverse side-effects (e.g., clicking on a "like" button or purchasing an item). The configuration allows one to selectively disable prefetching or limit prefetching unpopular actions/items.

Finally, the configuration allows one to specify field-specific policies, which can be used in multiple contexts. It can be used to attach a custom indicator when building a request message. For example, one can add a prefetch indicator in the HTTP header that enables the server to distinguish whether the request is from an actual client or from the proxy. This can be used to precisely handle statistics (e.g., view counts on an item) at the server side. Note, FireFox similarly adds a 'X-moz:prefetch' header when performing link prefetching [12]. In addition, the configuration allows one to specify field-specific conditions for prefetching, which enables fine-grained proxy control. This, for example, can be used to deliver better service (i.e. aggressive prefetching) to premium customers as many shopping apps have tiered customer programs.

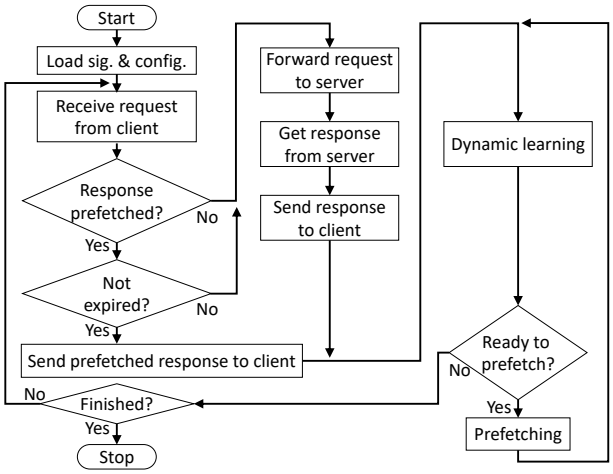


Figure 10: Flowchart diagram of the prefetching proxy.

Example. Figure 9 shows an example configuration. A default initial configuration is automatically generated in the testing and verification phase, which can be later customized. We currently support seven fields: 'hash', 'uri', 'expiration_time', 'prefetch', 'probability', 'add_header', and 'condition'. The 'hash' is a hash of the signature, internally used by APPx to distinguish each signature. The 'uri' field provides readability for service providers who modify the configuration. The 'expiration_time' specifies when the prefetched response expires. The proxy only prefetches when 'prefetch' field is 'true'. The proxy prefetches the response according to the 'probability' specified. A service provider can also define the probability globally. The proxy adds HTTP header fields specified by 'add_header' to prefetching requests. Each policy can have multiple 'add_header' fields. Finally, the proxy performs prefetching only when a predecessor satisfies the 'condition' field. In this example, prefetching for `.*/product/get` is triggered only when the "price" field of the predecessor is greater than "1000" dollars.

4.5 Proxy in Operation

Figure 10 summarizes the workflow of the prefetching proxy. The proxy loads its configuration and the signatures generated from the program analysis. When a client request arrives at the proxy, it first checks whether the corresponding response has already been prefetched. If the request is identical to that of prefetched transactions, including URI, query string, header, and body of request, the proxy sends the prefetched response message to the client on behalf of the origin server unless the response has already been expired. Otherwise, the proxy forwards the request to the origin server. Note the proxy does not break the app behavior even if a prefetched request is different from the original client request, because the proxy sends the response only when the prefetch request is identical to the client's request.

For prefetching, the proxy performs dynamic learning as described in §4.2. It learns the most recent condition of network messages and run-time fields, continuously updating its state. A prefetch request becomes ready for prefetching when all dynamic values have been resolved. Next, the proxy sends the prefetch requests to the origin server following the prefetching frequency specified in the configuration and caches the response.

App	Category	Main Interaction
Wish	Shopping	Loads an item detail
Geek	Shopping	Loads an item detail
DoorDash	Food delivery	Loads a restaurant info.
Purple Ocean	Psychic reading	Loads an advisor page
Postmates	Food delivery	Loads a restaurant info.

Table 1: Description of apps and main interactions.

App	Transactions of Main Interaction	RTT to Origin Server
Wish	Product detail	165 ms
	Product image	16 ms
Geek	Product detail	165ms
	Product image	6 ms
DoorDash	Menu	145 ms
	Restaurant schedule	145 ms
Purple Ocean	Advisor information	230 ms
	Profile image	15 ms
	Video still image	15 ms
Postmates	Restaurant menu & info	5 ms

Table 2: Transactions of main interaction and RTT to origin servers.

5 IMPLEMENTATION

We develop our proxy based on `mitmproxy` [13], an open source man-in-the-middle proxy. We modify the program slicing module, dependency analysis module, and the semantic model of Extractocol to implement our static analysis module. We extend 2.4K lines of code over `mitmproxy` and 8.4K lines of code over Extractocol to implement APPx. The proxy uses multi-threading. We assign different worker threads to handle dynamic learning and prefetching for efficiency. The request instances reconstructed through dynamic learning are stored in a queue, and a prefetching thread de-queues each instance to issue the request. The prefetching thread determines whether to issue a request according to the frequency specified in the configuration. The proxy stores the responses in a hashmap with the corresponding request as the key. Finally, prefetched responses are not shared across users, and the prototype distinguishes users by IP address.

Prefetching priority. Multiple prefetch request can be outstanding at any moment. To minimize the overall response time, our proxy uses priority scheduling. We prioritize request that takes longer to complete and signatures that generate higher hit rates. Because prefetched responses are not always used and some are used more frequently than others, the hit-rate based weight assignment results in more efficient resource use. For this, the proxy maintains a running average of response time between the proxy and the server for each signature. Then, we use the linear combination of the two as the priority.

6 EVALUATION

Methodology: To evaluate our framework, we use five popular commercial apps available in Google Play, each of which ranked within the top five in shopping, food delivery, and psychic reading app categories [3]. Table 1 shows the description of each app. We exclude apps that use public-key pinning because we cannot decrypt network messages they generate using the man-in-the-middle. For actual deployment, APPx requires app provider's support and visibility over plain-text traffic, as described in §2.

For each app, we select a representative user interaction, shown in Table 1 that reflects the main usage of the app (e.g., browsing items on Wish) as the prefetching target and configure the proxy as such. Throughout the evaluation, we focus on the main interaction because it reflects the key functionality of the app to accelerate. In particular, by using Frida [8] we measure the time between the user input that triggers the main interaction and when the app displays the final output. We use this "response time" as our key metric and refer to it as the user-perceived latency.

We conduct an IRB-approved user study with 30 participants to evaluate the prefetching proxy under a more realistic workload that reflects how users use the app. We record the user event traces (e.g., click and scrolling) using Appetizer [5] while each user freely uses each app for three minutes. The trace is 450 minutes long in total. To drive our proxy against the workload, we replay the event traces on a Google Nexus 6 smartphone, while all traffic passes through the proxy and goes to our own server that acts identical to the original one based on pre-recorded traces from the user study.

We evaluate APPx by answering three key questions:

- Is the framework effective in identifying prefetch opportunities for mobile apps?
- How much does the prefetching proxy reduce response time of mobile app?
- Does APPx effectively control the trade-off between latency reduction and data usage overhead?

6.1 Message Dependency Analysis

To demonstrate the effectiveness of APPx in identifying prefetch opportunities, we compare the result of APPx with that of automatic fuzzing and our user study trace. APPx takes 41 hours to extract the signatures for the all apps. The time depends on the app code size; larger apps take more time (up to 9 hours) than others. For automatic fuzzing, we use a UI automation tool, called Monkey [23], provided by Android Studio. We use Monkey to generate an arbitrary stream of user events, such as click or scrolling, at a 500 msec interval for a duration of an hour. We then collect the network trace that the apps generate. Finally, we identify the unique signatures of automatic fuzzing and our user study trace by regex-matching the URI of the signatures identified by APPx with that of the traces.

Table 3 shows the result. APPx identifies much more unique signatures than those of auto fuzzing and user study trace. However, UI-fuzzing is fundamentally lacking in providing a wide coverage [50] because some requests are not triggered by user events (e.g., push notification). It is difficult to navigate through all the cases without forcing a server to trigger those requests. We also compare prefetchable signatures of APPx, UI-fuzzing and user study

App	# of Unique Signatures Identified		# of Dependency Relationships	
	Total	Prefetchable	Total	Max len.
	(**APPx** / Auto UI fuzzing / User study)			
Wish	**120** / 47 / 16	**33** / 8 / 7	**794** / 78 / 49	**12** / 5 / 5
Geek	**118** / 51 / 31	**45** / 11 / 13	**388** / 39 / 31	**10** / 4 / 4
DoorDash	**63** / 29 / 21	**31** / 10 / 10	**160** / 30 / 36	**7** / 3 / 5
Purple Ocean	**109** / 25 / 10	**37** / 4 / 4	**72** / 4 / 6	**4** / 2 / 2
Postmates	**83** / 18 / 14	**35** / 6 / 8	**272** / 10 / 16	**15** / 2 / 3

Table 3: Signatures and dependency relationships identified for commercial apps.

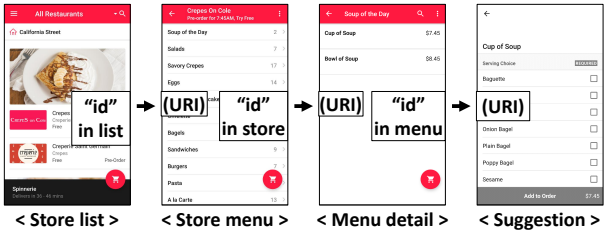


Figure 11: Dependency case study in DoorDash (Successive dependency)

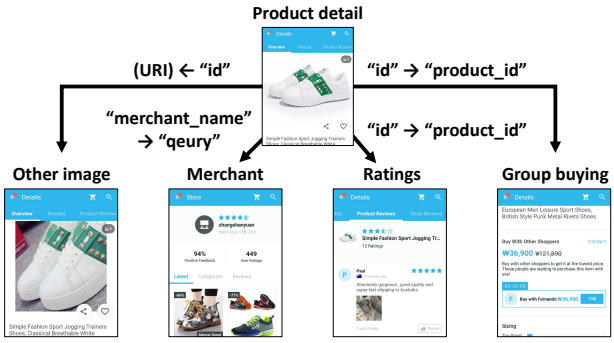


Figure 12: Dependency case study in Wish (Multiple relationships on a single transaction)

trace. A prefetchable signature is a successor, which means that some fields of the signature is originated from its predecessor requests. The static analysis of APPx collects the unique signatures up to 4x compared to UI-fuzzing and up to 9x more prefetchable signatures, respectively.

APPx identifies many opportunities for prefetching. APPx discovers the dependency relationships up to 794 and the maximum length of a successive dependency chain up to 16, which outperforms those of UI-fuzzing. This implies it is very difficult for human to manually recognize the relationships in mobile apps. We describe a few of examples. Figure 11 shows dependencies from DoorDash. A restaurant id from the "Store list" is used to look up the "Store menu". The `id` field of "Store menu" is used to get "Menu detail", whose `id` is, in turn, used in the loading the "Suggestion" page. APPx recognize such a successive dependency chain. Figure 11 shows another example from Wish. In this case, a single transaction is a predecessor of multiple transactions. The `id` field of "Product details" response is used to load three different pages, and the `merchant_name` field from the same response is used to load the

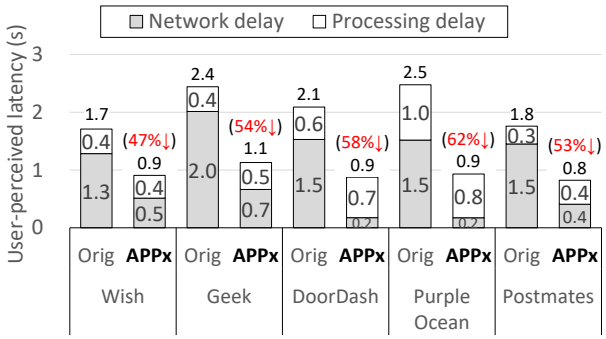


Figure 13: User-perceived latency of main interactions when communicating with origin server. "Orig" and "APPx" indicate the case that does not prefetch and the case that prefetches, respectively.

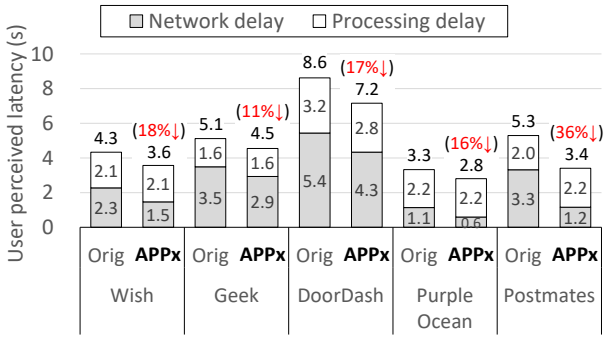


Figure 14: User-perceived latency of app launch when communicating with origin server. "Orig" and "APPx" indicate the case that does not prefetch and the case that prefetches, respectively.

merchant page. We find that there exists a combination of above two types of the relationships in mobile apps and APPx discovers such complex relationships, which presents many prefetching opportunities. APPx allows the proxy provider and app providers to choose from these options which ones to prefetch based on their service preference and business perspective.

6.2 Proxy Evaluation

Proxy evaluation with origin servers: To evaluate the performance of the prefetching proxy, we measure the user-perceived latency of the main interaction and app launch with and without prefetching. For the app launch delay, we measure the difference between the time to execute an app and the time to display all contents on the screen. Each app is running on a Google Nexus 6 smartphone and communicates with their origin servers through the proxy. The phone is wired to the local network to eliminate the latency and bandwidth variance that might be caused from a wireless connection. We set the round-trip-time of 55 ms and the bandwidth of 25 Mbps between the client and proxy, which reflect the average 4G latency and bandwidth [1]. This represents the case where the proxy is located in close proximity to a mobile client. Table 2 shows the latency between the proxy and the origin servers

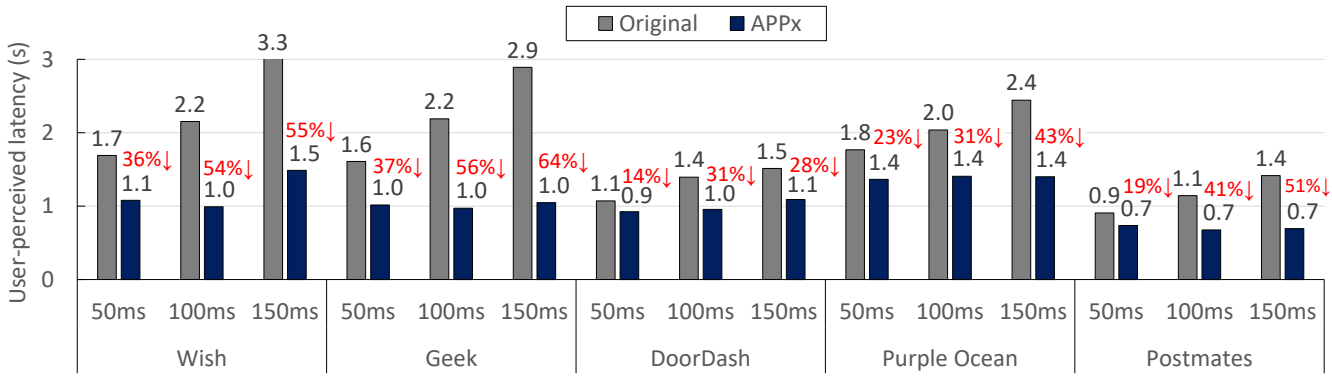


Figure 15: 90%-tile latency for app's main interaction using user traces while varying RTT between proxy and server

for each app. Finally, the proxy prefetches content in advance for the main interaction.

Figure 13 and Figure 14 show the user-perceived latency of the main interaction and app launch for each app, respectively. We measure the average latency of 10 runs. The results show APPx reduces the user-perceived latency of the main interaction by 47-62% and the app launch by 11-36% across apps. The figures further break down the user-perceived latency into the network and processing delay. The processing delay includes the time to perform some initial processing, such as sensor data acquisition and radio wake-up, and the time to render responses from remote servers to the screen [66]. We calculate the processing delay by subtracting the network delay from the user-perceived latency. APPx reduces the network delay and the speedup factor ranges from 2.5 to 8.7x for the main interaction and from 1.2 to 2.9x for the app launch. Purple Ocean benefits the most in terms of network delay because their servers are located far away (see Table 2). Wish and Geek have a longer network delay than the others even with prefetching because the size of their product images is much greater (~ 315KB) than the other's transactions (~ 14KB). For the app launch, the benefits are less than that of the main interaction because most of the apps do multiple requests in serial when launching and the requests usually arrive at the proxy while the proxy prefetches the corresponding responses.

Evaluation based on workload from user study: We measure the performance of APPx using user-event traces (e.g., click and scrolling) collected from the user study. This reflects the actual usage. We replay the trace in real time to reflect the user "think time". We measure the user-perceived latency of the main interaction. Each app communicates with our own server that acts identical to the original ones. To emulate a realistic experimental environment, we set the RTT of 55 ms and the bandwidth of 25 Mbps between the client and proxy and vary the RTT between the proxy and server from 50 to 150 msec. This effectively varies the proxy location between the client and server. We also measure data usage of the proxy to quantify the overhead of APPx. To this end, we measure the size of responses transmitted between the proxy and server and normalize it to the size of the environment that does not prefetch. Finally, we set the bandwidth of 25 Mbps between the proxy and server.

Figure 15 shows the 90%-tile user-perceived latency for the main interaction. APPx significantly reduces the latency across all applications. The latency reduction ranges from 14 to 64%. Figure 16 shows the cumulative distribution function of latency for the app's main interaction and the normalized data usage. Across all applications and environments, the prefetching proxy reduces the median latency between 17% (252 ms) to 64% (1,471 ms). The improvement is more dramatic when the proxy is located closer to the client.

APPx dramatically reduces latency for Wish and Geek because the size of their product images is much greater than the others. For Purple Ocean, the proxy effectively reduces the latency in absolute terms (252 to 906 ms), but because its processing delay is large (≈ 0.8 sec) the relative reduction appears small. The proxy uses 1.08 to 4.17x more data to perform prefetching across the apps. Note, the mobile device itself does not use any more data. For the shopping app Wish and Geek, the proxy prefetches a relatively large amount of data because participants in the user study usually glance over many items and the proxy prefetches their product image whose size is large. In Postmates, the data usage overhead is small (8%) because the size of restaurant image is much greater (168 KB on average) than that of restaurant menu and info (7 KB on average) that the proxy prefetches. The ratio of data actually used by app among all prefetched transactions ranges from 1 to 5% across the apps. It is attributed to the fact that the apps load a list of contents and the proxy prefetches all the contents in the list and users typically consume part of them. The traces from the user study include all interactions without any modification. Users typically have to carry out some interactions before reaching the main interaction. For example, Postmates loads multiple restaurant images whose traffic volume is sizable upon the app launch. User then selects a few of them from the list of restaurants, which represents the app's main interaction. As a result, the data usage of the proxy is between 1.08x and 4.17x compared to the original data consumption without prefetching.

6.3 Time and Data Usage Trade-off

APPx allows the proxy operator to configure the prefetch probability to control the tradeoff between latency reduction and data usage. As we configure the proxy to prefetch less aggressively, the proxy and server use less bandwidth at the cost of less improvement in average response time. Our fine-grained configuration allows

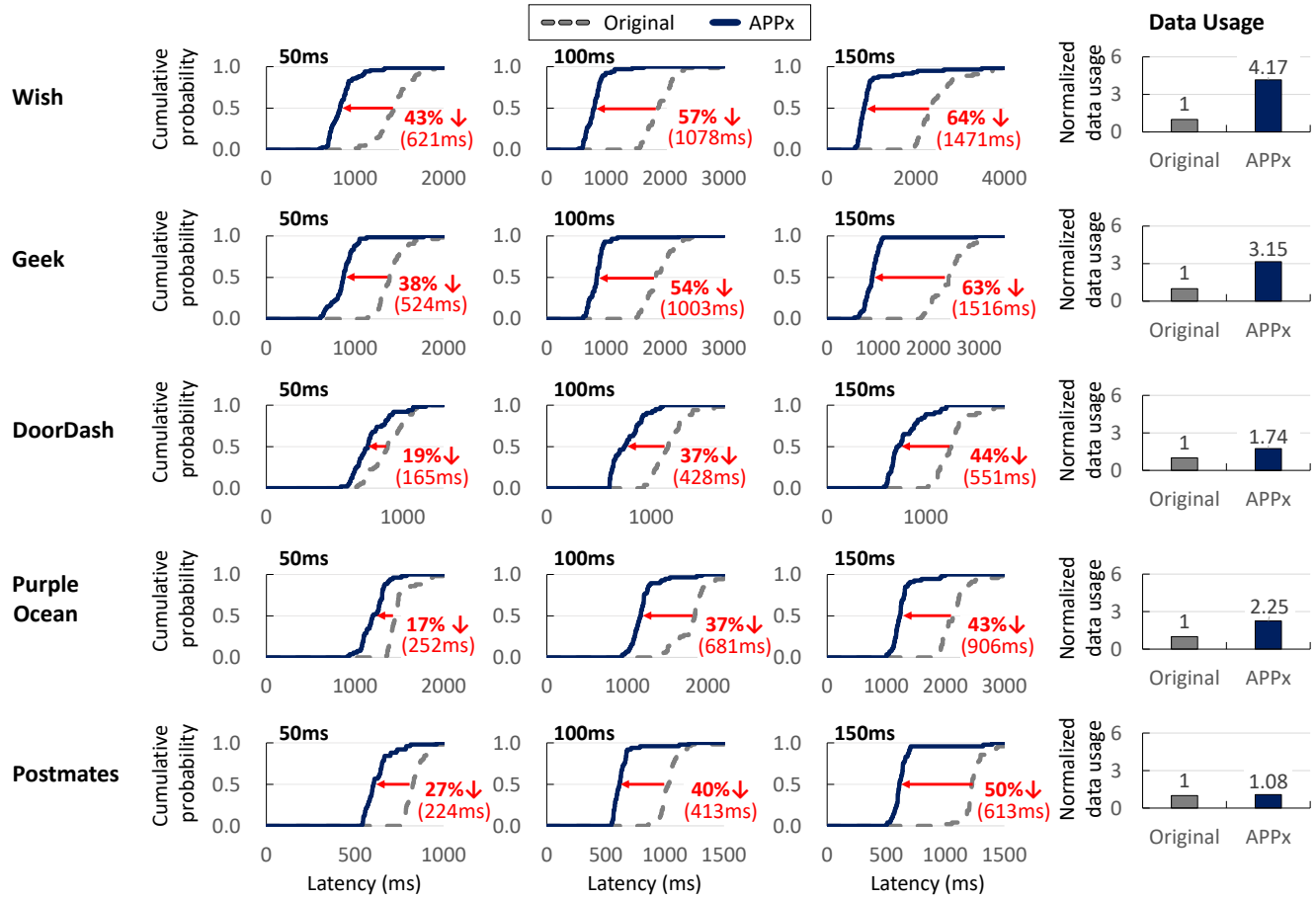


Figure 16: Cumulative distribution function of the user-perceived latency and data usage.

the operator to set prefetching probabilities for individual requests. We believe that APPx can perform prefetching more effectively by making the proxy to collect and use fine-grained popularity of each request or item. This can also be used to enable service differentiation across users (e.g., based on customer tiers in a shopping app).

Figure 17 shows the trade-off between the median latency and the data usage overhead of Wish, as we change the probability of prefetching. The result shows that the proxy provides a knob to adjust the tradeoff between latency and bandwidth. As we prefetch more aggressively, the latency decreases half (from 1,881 to 947 ms) when the data usage increases 2.1 times. We also observe that the median latency dramatically decreases when we prefetch the majority of transactions.

7 RELATED WORK

Mobile app acceleration: Several studies [42, 44, 46, 75] address the problem of reducing response time of mobile apps. PALOMA [75] is a concurrent work to ours that shares similar ideas. Similar to APPx, PALOMA uses static program analysis to identify prefetching opportunities. Unlike APPx, it instruments the mobile app to interact with a proxy-agent on the mobile device and communicate with remote servers. However, PALOMA does not handle the case when the exact format of request message is determined at run-time. It requires that an exact request message be identified during static analysis, which limits the applicability. We show that many requests are determined dynamically depending on the dynamic control flow of the program (e.g., 'credit_id' field in Figure 7). APPx overcomes the limitation of PALOMA by combining static analysis and dynamic learning and observing the actual traffic at runtime. Looxy [44] performs caching and prefetching in a local proxy that communicates with a client device through a WiFi connection. Looxy prefetches using only the full URLs of HTTP request contained in the response. In our observations, however, many of dependency relationships between network transactions of the mobile app are found in the parts of HTTP request (e.g., 'cid' field in Figure 7). Looxy does not handle that dependency relationships. By using static program analysis and dynamic learning, APPx identifies the dependency relationships and reconstructs request messages to prefetch. IMP [46] allows prefetching on mobile side and provides APIs that hide the complexity of the prefetching decision. However, app developer must manually instruct what to prefetch by modifying the app. Tango [42] replicates app execution on a cloud server that has greater computational power and broader network bandwidth than mobile device. Replicating the entire execution without understanding the app behavior requires running a Dalvik virtual

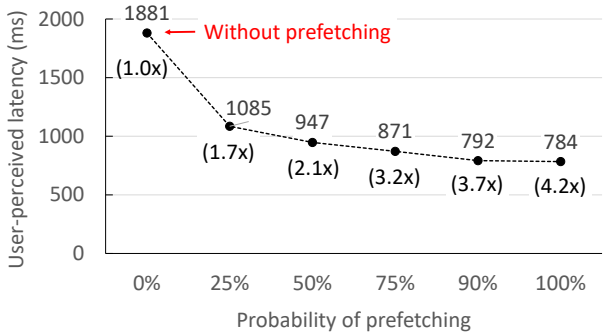


Figure 17: Trade-off between latency and data usage of Wish. The numbers in parentheses indicate the normalized data usage.

machine, which consumes more resource than running a proxy. APPx replicates only the app's protocol behavior in the proxy. This is much lighter-weight than running the entire app on top of a VM in the cloud. In fact, our approach can support multiple apps using the same proxy. EBC [32] reduces app launch delays by scheduling app prefetches upon screen unlock. EBC determines when and for which apps prefetches should be triggered through estimating the app usage probabilities and their traffic volume. We believe that one can effectively reduce the response time of mobile app by combining EBC and APPx.

EdgeReduce [62] and Procrastinator [65] focus on reducing data usage of mobile apps, at the cost of increased response time. In contrast, APPx reduces the response time at the expense of using more bandwidth at the proxy. According to our own private conversations with several mobile service providers, they are willing to make this tradeoff because latency reduction is far more important to their bottom line.

Mobile code offloading: Many studies [36, 37, 40, 41, 43, 51, 54, 71] have proposed different code-offloading strategies to improve the responsiveness of mobile apps and save the energy consumption of mobile devices. Maui [37] accelerates mobile apps and saves energy by allowing developers to selectively offload methods to cloud servers by adding annotations on the code. Thinkair [51] and Cloudlets [71] make automated offloading decisions by performing static and dynamic analysis of mobile apps. Note that these approaches often require modifications on client or server application. They are mainly beneficial to compute-intensive mobile apps. In contrast, we focus on accelerating message exchanges and it mainly benefits network-intensive interactive, latency-critical apps.

Server push: HTTP/2.0 and SPDY allow a server to push embedded objects before they are requested. This reduces the page load time for Web content. This requires both server and client support. The benefit is also limited to Web content and Web apps, and they do not benefit mobile apps.

Web acceleration: Web acceleration is used to reduce the latency of web pages and server load, and data usage [16, 19, 47, 49, 56, 74]. SPDY [19] combines multiple techniques, such as compression, multiplexing, and prioritization, to reduce the Web latency. mod_pagespeed is an open-source Apache module that rewrites web pages to load faster [16]. It performs image optimization, compression, resizing, static web file minification, and caching. WProf [72] is a Web profiling tool that analyzes the dependency between Web objects and their load times. It shows that SPDY and mod_pagespeed significantly reduce the size of downloads, but they are not always effective in reducing page load time because they do not always affect the critical path. Klotski [56] prioritizes the content most relevant to a user's preferences. But it must analyze dependencies between web objects within a page, before they establish a prioritization plan. Unlike Klotski, Shadian [74] performs speeding up web page loads without any knowledge of web objects. Finally, NutShell [70] tackles scalability challenges in JavaScript execution for proxy-based Web acceleration. NutShell's proxy only executes the part of JavaScript code necessary to identify and fetch Web objects rather than executing the entire code. This is similar in spirit to APPx that only mimics app's network behavior instead of replicating the entire app execution [42].

TCP-level acceleration: TCP acceleration is a considerably well explored area. There has been a number of studies [29, 31, 53, 55, 63, 64, 69] to reduce service access times by performing TCP acceleration. They split long connections into multiple shorter connections [53, 55], optimize the establishment of TCP connection and slow start phases [29, 64], and/or adopt proxy-based approach [31, 63, 69] to improve TCP performance. Another approach [16, 24, 28] performs caching and/or compression to reduce transmission time and network bandwidth. We believe that our approach can be combined with the general transport-level approaches to accelerate mobile apps.

8 CONCLUSION

This paper presents a novel approach for mobile app acceleration. Leveraging recent advances in static program analysis, it automatically discovers opportunities for prefetching. In particular, the framework takes the app binary as input and combines static and dynamic analysis to generate prefetch requests that look identical to the original request. It allows service providers to easily generate app specific proxies and configure them to fit their policy. Our evaluation results show that app-specific acceleration proxies generated from our framework dramatically reduce the response times of apps, enhancing the quality of user experience for mobile apps and services. Furthermore, we show that the framework allows us to balance the tradeoff between the latency reduction and proxy bandwidth usage through policy specification. We believe that our framework will be particularly useful in accelerating various mobile apps in lightly multiplexed environments, such as the mobile edge cloud.

ACKNOWLEDGEMENT

We thank our shepherd Matteo Varvello and anonymous reviewers for their valuable feedback. This work is supported by Institute for Information & communications Technology Promotion (IITP) funded by the Korea government (MSIT) [2018-0-00693]; and Institute of Civil Military Technology Cooperation Center (ICMTC) funded by the Korea government (MOTIE & DAPA) [18-CM-SW-09].

REFERENCES

[1] 4G and 3G mobile broadband speeds research. https://www.ofcom.org.uk/about-ofcom/latest/media/media-releases/2014/3g-4g-bb-speeds.

[2] Amazon CloudFront Pricing. https://aws.amazon.com/ko/cloudfront/pricing/.

[3] App Annie App Store Statistics. https://www.appannie.com/apps/google-play/matrix/?country=US. [accessed 13-Mar-2018].

[4] App Annie Google Play Statistics. https://www.appannie.com/apps/google-play/top-chart/?country=US&category=30&device=&date=2018-06-09&feed=All&rank_sorting_type=rank&page_number=0&page_size=100&table_selections=. [accessed 09-Jun-2018].

[5] AppetizerIO - Mobile DevOps Platform. https://www.appetizer.io/en/index.html.

[6] Dynamic site acceleration via Azure CDN. https://docs.microsoft.com/en-us/azure/cdn/cdn-dynamic-site-acceleration.

[7] Dynamic Site Accelerator | Akamai. https://www.akamai.com/us/en/products/web-performance/dynamic-site-accelerator.jsp.

[8] Frida - A Dynamic Instrumentation Framework. https://www.frida.re/docs/home/.

[9] Google Cloud Platform Pricing. https://cloud.google.com/cdn/pricing.

[10] How One Second Could Cost Amazon $1.6 Billion In Sales. http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[11] Keyless SSL | Cloudflare. https://www.cloudflare.com/ssl/keyless-ssl/.

[12] Link prefetching FAQ - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Link_prefetching_FAQ.

[13] Mitmproxy. https://mitmproxy.org.

[14] Mobile App Acceleration - A Neumob White Paper. https://www.neumob.com/wp-content/uploads/2016/05/Neumob-Understanding-and-Improving-Mobile-App-Acceleration.pdf.

[15] Mobile App Usage Increases In 2014, As Mobile Web Surfing Declines. https://techcrunch.com/2014/04/01/mobile-app-usage-increases-in-2014-as-mobile-web-surfing-declines/.

[16] Modepage - Aphache Module for Rewriting Web Pages to Reduce Latency and Bandwidth. http://modpagespeed.com.

[17] Preloading Web Search Top Hits in Safari. https://support.apple.com/kb/PH21448.

[18] RxAndroid: Reactive Extensions for Android. https://github.com/ReactiveX/RxAndroid.

[19] SPDY - An Experiment with Protocols for the Web. Its Goal is to Reduce the Latency of Web Pages. http://dev.chromium.org/spdy.

[20] Speed Up Google Chrome. https://support.google.com/chrome/answer/1385029.

[21] SSL Proxy Overview – Juniper Networks. https://www.juniper.net/documentation/en_US/junos/topics/concept/ssl-proxy-overview.html.

[22] The Rise of REST API. https://blog.restcase.com/the-rise-of-rest-api/.

[23] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.

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

[25] Why is RxJava gaining so much popularity in the Android world? https://www.quora.com/Why-is-RxJava-gaining-so-much-popularity-in-the-Android-world.

[26] Why is RxJava so popular with Android Developers? https://www.youtube.com/watch?v=TjaLKduVM6w.

[27] Why REST is So Popular. https://www.serviceobjects.com/resources/articles-whitepapers/why-rest-popular.

[28] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 367–380, 2015.

[29] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky. Overclocking the Yahoo!: CDN for Faster Web Page Loads. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC)*, pages 569–584. ACM, 2011.

[30] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2014.

[31] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st annual international conference on Mobile computing and networking (MobiCom)*, pages 2–11. ACM, 1995.

[32] P. Baumann and S. Santini. Every Byte Counts: Selective Prefetching for Mobile Applications. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 1(2):6, 2017.

[33] Y. Cao, Y. Fratantonio, M. Egele, A. Bianchi, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[34] X. Chen and X. Zhang. A Popularity-based Prediction Model for Web Prefetching. *Computer*, 36(3):63–70, 2003.

[35] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, pages 239–252. ACM, 2011.

[36] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 301–314. ACM, 2011.

[37] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys)*, pages 49–62. ACM, 2010.

[38] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, volume 13, pages 809–817, 2013.

[39] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, pages 97–108, 2002.

[40] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile Code Offloading: from Concept to Practice and Beyond. *IEEE Communications Magazine*, 53(3):80–88, 2015.

[41] H. Flores and S. Srirama. Adaptive Code Offloading for Mobile Cloud Applications: Exploiting Fuzzy Sets and Evidence-based Learning. In *Proceeding of the fourth ACM workshop on Mobile cloud computing and services (MCS)*, pages 9–16. ACM, 2013.

[42] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating Mobile Applications through Flip-flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 137–150. ACM, 2015.

[43] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–106, 2012.

[44] Y. Guo, M. Liu, and X. Chen. Looxy: Web Access Optimization for Mobile Applications with a Local Proxy. In *Proceedings of Vehicular Technology Conference (VTC) Spring*, pages 1–5. IEEE, 2017.

[45] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys)*, pages 204–217. ACM, 2014.

[46] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed Mobile Prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, pages 155–168. ACM, 2012.

[47] M. B. Jamshed Vesuna, Colin Scott and M. Piatek. Caching Doesn't Improve Mobile Web Performance (Much). In *Proceedings of the USENIX Annual Technical Conference (ATC)*, April 2016.

[48] Z. Jiang and L. Kleinrock. Web Prefetching in a Mobile Environment. *IEEE Personal Communications*, 5(5):25–34, 1998.

[49] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving User Perceived Page Load Times Using Gaze. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 545–559, 2017.

[50] J. Kim, H. Choi, H. Namkung, W. Choi, B. Choi, H. Hong, Y. Kim, J. Lee, and D. Han. Enabling Automatic Protocol Behavior Analysis for Android Applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 281–295. ACM, 2016.

[51] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 945–953. IEEE, 2012.

[52] T. M. Kroeger, D. D. Long, J. C. Mogul, et al. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 13–22, 1997.

[53] S. Ladiwala, R. Ramaswamy, and T. Wolf. Transparent TCP Acceleration. *Computer Communications*, 32(4):691–702, 2009.

[54] C.-K. Lin and H. Kung. Mobile App Acceleration via Fine-Grain Offloading to the Cloud. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[55] Y. Liu, Y. Gu, H. Zhang, W. Gong, and D. Towsley. Application Level Relay for High-bandwidth Data Transport. *GridNets*, 2004.

[56] Z. w. Michael Butkiewicz, Daimeng Wang, H. V.Madhyastha, and V. Sekar. KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, May 2015.

[57] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2010.

[58] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of USENIX Symposium on Networked*

Systems Design and Implementation (NSDI), 2018.

[59] R. Netravali and J. Mickens. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *Proceedings of International Workshop on Mobile Computing Systems & Applications (HotMobile)*, pages 63–68, 2018.

[60] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of USENIX Security Symposium*, 2013.

[61] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Computer Communication Review (CCR)*, 26(3):22–36, July 1996.

[62] A. Pamboris and P. Pietzuch. Edge Reduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies. In *Proceedings of the ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 72–82. IEEE, 2015.

[63] A. Pathak, Y. A. Wang, C. Huang, A. Greenberg, Y. C. Hu, R. Kern, J. Li, and K. W. Ross. Measuring and Evaluating TCP Splitting for Cloud Services. In *Proceedings of International Conference on Passive and Active Network Measurement (PAM)*, pages 41–50. Springer, 2010.

[64] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies (CoNEXT)*, page 21. ACM, 2011.

[65] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: pacing mobile apps' usage of the network. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys)*, pages 232–244. ACM, 2014.

[66] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100. ACM, 2013.

[67] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 390–403, 2017.

[68] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 604–616, 2015.

[69] G. Siracusano, R. Bifulco, S. Kuenzer, S. Salsano, N. B. Melazzi, and F. Huici. On-the-Fly TCP Acceleration with Miniproxy. *arXiv preprint arXiv:1605.06285*, 2016.

[70] A. Sivakumar, C. Jiang, Y. S. Nam, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. G. Rao, S. Sen, M. Thottethodi, and T. N. Vijaykumar. NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 448–461, 2017.

[71] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. Cloudlets: Bringing the Cloud to the Mobile User. In *Proceedings of the third ACM workshop on Mobile cloud computing and services (MCS)*, pages 29–36. ACM, 2012.

[72] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 473–485, 2013.

[73] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web (WWW)*, pages 31–40, 2012.

[74] A. K. Xiao Sophia Wang and D. Wetherall. Speeding up Web Page Loads with Shandian. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, March 2016.

[75] Y. Zhao, M. S. Laser, Y. Lyu, and N. Medvidovic. Leveraging Program Analysis to Reduce User-Perceived Latency in Mobile Applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2018.