SmartAds: Bringing Contextual Ads to Mobile Apps

Suman Nath
Microsoft Research
sumann@microsoft.com

Felix Xiaozhu Lin*
Rice University
xzl@rice.edu

Lenin Ravindranath
Microsoft Research
lenin@csail.mit.edu

Jitendra Padhye
Microsoft Research
padhye@microsoft.com

ABSTRACT

A recent study showed that while US consumers spent 30% more time on mobile apps than on traditional web, advertisers spent 1600% less money on mobile ads. One key reason is that unlike most web ad providers, today's mobile ads are not *contextual*—they do not take into account the content of the page they are displayed on. Thus, most mobile ads are irrelevant to what the user is interested in. For example, it is not uncommon to see gambling ads being displayed in a Bible app. This irrelevance results in low clickthrough rates, and hence advertisers shy away from the mobile platform.

Using data from top 1200 apps in Windows Phone marketplace, and a one-week trace of ad keywords from Microsoft's ad network, we show that content displayed by mobile apps is a potential goldmine of keywords that advertisers are interested in.

However, unlike web pages, which can be crawled and indexed offline for contextual advertising, content shown on mobile apps is often either generated dynamically, or is embedded in the apps themselves; and hence cannot be crawled. The only solution is to scrape the content at runtime, extract keywords and fetch contextually relevant ads. The challenge is to do this without excessive overhead and without violating user privacy. In this paper, we describe a system called SmartAds to address this challenge.

We have built a prototype of SmartAds for Windows Phone apps. In a large user study with over 5000 ad impressions, we found that SmartAds nearly doubles the relevance score, while consuming minimal additional resources and preserving user privacy.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Communications Applications

*Work done while at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSys'13, June 25–28, 2013, Taipei, Taiwan
Copyright 2013 ACM 978-1-4503-1672-9/13/06 ...$15.00.

General Terms

Design, Experimentation, Performance

Keywords

Mobile, Apps, Contextual, Advertisement

1. INTRODUCTION

Recently, an interesting study pointed out that advertising market shares of dominant media such as TV, radio, and Web are proportional to the average number of minutes consumers spend on the media per day [13].

The mobile market is one prominent exception to this rule. In 2011, US consumers spent 1.3× more time within mobile apps alone than on the Web (through desktop or mobile browsers). In 2012, the gap was 1.8× and it is expected to grow in coming years. However, the mobile advertising market today is very small compared to TV, radio, and Web advertising. In 2011, advertisers spent less than 1% of their overall advertising budget for mobile advertising; in contrast, they spent 16% for Web advertising [13].

This striking gap between the opportunity and the reality of mobile advertising has been particularly damaging for companies such as Facebook and Google that derive bulk of their revenues from ads and whose services can be accessed from mobile apps as well. It also impacts amateur app developers, many of whom rely solely on advertising revenue from ads shown in their apps.

One key reason behind this gap is that unlike traditional web ads, *today's mobile ads are mostly irrelevant.* The screen capture in the left of Figure 1 starkly illustrates this. With such irrelevant ads being shown, today's mobile advertising is justly derided as taking a "spray and pray" [25] approach. It should come as no surprise that this approach does not yield expected revenue. In order to fully exploit the potential of mobile advertising, we must strive to show more relevant ads to the consumers.

For traditional web, the relevancy problem is addressed in part, by *contextual advertising*[1], wherein the ads displayed on the page are based on the content of the page. For example, when a user visits an astronomy blog, the contextual advertising network shows him an ad on a telescope. Contextual advertising was pioneered by Google AdSense [3].

[1]Contextual advertising is different from *context-aware advertising* where ads are shown depending on the user's physical context such as location. These two can be combined, but in this paper we focus on contextual advertising only.

The crucial fact that enables contextual advertising is that web pages can be crawled by a bot and indexed offline. We will describe this process in more detail in §2.

Can we use the same approach to deliver relevant ads within mobile apps? Unfortunately, this is not straightforward. The fundamental reason is that *unlike a web page, the content shown by a mobile app cannot be easily crawled and indexed.* For example, certain news outlets provide mobile-only content, which is not available via standard web interface. Other content is embedded within the mobile app itself (e.g. large collections of reference material). This makes it difficult, if not impossible, to show contextual ads in mobile apps. We will describe this in more detail in §2.

Some adhoc solutions for providing contextual mobile ads have been tried. For example, some ad systems try to glean contextual information from app metadata such as the name of the app and the category. We will discuss these solutions in more detail later in the paper. However, our study (§3) of over 1200 top apps from Windows Phone marketplace unambiguously shows that these adhoc measures are not sufficient.

What is needed is a system that can extract ad keywords from content displayed by mobile apps at runtime, and fetch relevant ads based on those keywords. Furthermore, this task must be accomplished with minimal overhead (memory, network bandwidth etc.) and without violating user's privacy.

To fulfill this need, we have built SmartAds. SmartAds dynamically scrapes page contents as a user uses an app, extracts relevant ad keywords and fetches and displays relevant ads. SmartAds architecture (§4) strikes the right balance between utility (how relevant ads are), efficiency (memory, network, battery and computation overhead), and user privacy. SmartAds achieves this with a novel client-server based keyword extraction technique, with various systems parameters carefully chosen based on a 1-week long trace of bidding keywords on Microsoft's ad network.

SmartAds consists of a client library that the developer includes in the app, and a server that the library communicates with. At runtime, the client and the server work together to extract keywords from the content being seen by the user. The server then fetches relevant ads from a third-party ad provider and sends it to the client library. The client library is responsible for displaying the ad.

SmartAds currently focuses on contextual advertising only. We note that there are various other techniques for delivering relevant ads. For example, physical context-aware advertising (such as AdNext [19] and CAMEO [18]) delivers ads based on users' locations, activities, and other physical contexts. Behavioral targeting [33] (such as Yahoo! Smart Ads [31]) delivers ads to targeted users based on information collected on each individual user's past web usage history. Contextual targeting of SmartAds is complimentary to, and can be combined with, these other types of targeting. Relative effectiveness of specific targeting techniques and their combinations within mobile apps are topics of future research.

We have built a prototype of the SmartAds system for Windows Phone apps. Using this prototype, and a large-scale user study, we show (§5) that SmartAds more than doubles the relevance score of mobile advertisements. We also show that the user privacy is not violated, and the resource overhead is minimal. An example of a *relevant* ad fetched by SmartAds is shown in Figure 1 (right).

Figure 1: Left: Irrelevant mobile ad. The mobile app shows a list of nearby restaurants, while the ad (the box at the bottom) is about real estate agents. Right: Relevant mobile ad about restaurant coupons delivered by SmartAds.

In summary, this paper makes the following two contributions. First, using data culled from 1200 top Windows Phone apps, we demonstrate the need for a online, contextual mobile advertising system. We believe that this is the first study of its kind. Second, we design, build and evaluate SmartAds, a contextual mobile advertising system that strikes the right balance between utility, efficiency and privacy. To best of our knowledge, this is the first such system.

The rest of the paper is organized as follows. We survey the state of the art in web contextual advertising in Section 2. In Section 3, we characterize contents of mobile apps based on data culled from 1200 top Windows Phone apps. We describe SmartAds design and implementation in Section 4 and evaluate it in Section 5. We discuss various extensions of SmartAds in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. BACKGROUND

In this section, we discuss the state of the art in web contextual advertising, and contrast it with the state of the art in mobile advertising.

2.1 Contextual Advertising

Contextual advertising is a form of targeted advertising for ads displayed on websites or other media, such as content displayed in mobile devices. The ads themselves are selected and served by automated systems based on the content displayed to the user. For example, if a user visits an astronomy blog and sees an ad on a telescope, that's contextual advertising.

We stress that the web advertising community uses the word `context` *in a narrow sense.* In the ad community, *context* typically implies only the content displayed on the page. The ad providers typically take many signals (e.g. location, prior history) besides the page content into account while selecting the ad, but the term *contextual advertising*

refers specifically to the idea of taking page content into account.

Contextual advertising for the web was pioneered by Google AdSense [3]. Many other ad providers such as Yahoo! Publisher Network [32], Microsoft AdCenter [21] and Advertising.com [4] followed in Google's footsteps. These advertising networks typically work by providing webmasters with JavaScript code that, when inserted into web pages, displays relevant ads from the ad network's ad inventory. The core task of matching ads to web pages consists of the following steps.

Offline ad labeling. Ads in ad network's inventory are labeled with keywords, called *bidding keywords*, which are provided by advertisers.

Offline keyword extraction. Ad network employs a bot (e.g., Google's MediaBot) that crawls web pages and uses machine learning algorithms (such as KEX [34]) to extract prominent keywords and concepts in them. Note that the bot can parse dynamically rendered web pages as well, as long as it does not require human input, such as solving Captchas.

Online web page to ad matching. When a user visits a website, the ad network selects an available ad whose bidding keywords best match the keywords/concepts in the web page.

Contextual advertising has been very successful on the web. Because the ads are more targeted, users are more likely to click on them, thus generating revenue for the owner of the website and the ad network.

2.2 Mobile Ads

Many mobile app developers use mobile advertisements within their apps as their only source of revenue. More than 50% of the apps in the major mobile app stores show ads [14].

To embed ads in an app, the app developer typically registers with a third-party mobile ad network such as AdMob [1], iAd [16], Microsoft Mobile Advertising [24] etc. The ad networks supply the developer with an ad control (i.e. library with some visual elements embedded within). The developer includes this ad control in his app, and assigns it some screen "real estate". When the app runs, the ad control is loaded, and it fetches ads from the ad network and displays it to the user.

Different ad networks use different signals to serve relevant ads. One of the main signals that mobile ad networks use today is the app metadata [24]. As part of the registration process, most ad networks ask the developer to provide metadata information about the app (for e.g. category of the app, link to the app store description etc.). This allows the ad network to serve ads related to the app metadata.

Ad networks also receive dynamic signals sent by the ad control every time it fetches a new ad. Depending on the privacy policies and the security architecture of the platform, these signals can include the location, user identity, etc. Note that unlike JavaScript embedded in the browsers, the ad controls are integral parts of the application, and have access to the all the APIs provided by the platform.

Our focus in this paper is contextual advertising, where the signal is the content of the app page that the user is viewing. We are not aware of any ad control that provides such contextual advertising for mobile apps.

This is because building contextual advertising systems for mobile apps is quite challenging. Unlike web pages, content displayed by mobile apps cannot be crawled and indexed by a bot in an offline manner, and hence existing contextual ad systems cannot perform the offline keyword extraction phase mentioned before. While some mobile apps fetch web content that is crawlable by bots (e.g. apps that fetch and display Wikipedia content), the apps often transform this content in a variety of ways, sometimes combining content from multiple sources.

For example, a news aggregator app, such as `News 360`, uses web API to pull news items from several web services and displays it. Many of these web services do not have any traditional web front-end that a web bot can discover and crawl. Furthermore, the content fetched from the web is modified before being shown to the user. Other apps have sizable amounts of content embedded in them (e.g. reference data or large religious texts).

Thus, the only surefire way of providing contextual ads to a mobile app is to parse the content in an online manner (i.e. as it is displayed), and to extract keywords. However, this is a challenging task for two reasons. First, the mobile platform has limited resources. Dynamically scraping the displayed content, extracting keywords and sending it through the network, without incurring unacceptable overhead is a non-trivial task. Second, sending the content of the page to the ad network can seriously compromise user privacy.

Note that our goal is related to recently proposed techniques for "just-in-time" contextual advertising in dynamic web pages [5, 20]. These techniques aim to deliver relevant ads to dynamic web pages based on useful signals extracted from them during runtime. In general, these techniques rely on various properties of web pages and are not directly applicable for mobile apps. For example, [5] uses page URL as well as referrer URL, which do not exist for mobile apps. It also sends short excerpts of the page to backend server, and hence violates privacy. [20] uses previously recorded traffic to the page that, again, does not generally exist for mobile apps. We therefore seek for alternative techniques that work for mobile apps.

We have built SmartAds to address the aforementioned challenges for mobile apps. Before we explain the design of SmartAds, we first characterize the content of existing mobile apps to motivate the need to deliver ads based on the app page content.

3. CHARACTERIZATION OF MOBILE APPS

Like websites, mobile apps typically display content in different screens (*App Pages*) that users can navigate between. In most platforms, only one app page is displayed to user at a time. Each of these app pages can contain different set of UI controls displaying different content. We will call the content displayed by an app page as *Page Data*. In Figure 1, page data consists of a list of restaurant names and their addresses.

An online system to extract keywords from page data is needed only if a significant fraction of mobile apps possess the following three characteristics.

First, the page data displayed by the app should be a rich source of ad keywords. If it turns out that most apps display content that do not contain keywords that advertisers bid on, there is no need for our system.

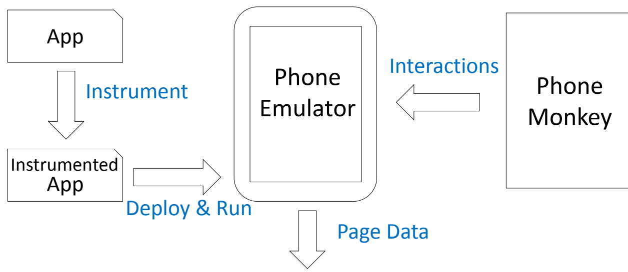

Figure 2: Schematics of page data logging

Second, the page data should provide significantly more keywords than the metadata about the app. As we described in §2.2, many ad providers today take metadata into account while selecting ads to display. Page data needs to do substantially better.

Third, the page data should change significantly over a period of time - i.e. between different invocations of the same app. Otherwise, one could build a system that runs the app in an emulator and extract the keywords (essentially, an offline system).

Consider, for example, the `Around Me` app that allows a user to search for local business around his location. If a user uses the app for finding local restaurants, he should be shown restaurant related ads (based on extracted keyword `restaurant`), but if he uses it for searching for nearby parking lots, he should be shown parking related ads. In other words, as the page data can change over time and location, offline extraction is not sufficient. Another example is a news aggregator app, where the content always changes with time.

To verify whether these conditions hold true for a significant fraction of mobile apps, we collected and analyzed page data from a large number of apps from the Windows Phone marketplace, as follows.

3.1 Methodology

To capture page data from a large number of apps in a scalable manner, we developed a UI automation tool called PhoneMonkey. The PhoneMonkey can emulate various user interactions (touch, swipe etc.) to navigate through various pages of the app. To capture the page contents, we leverage the binary instrumentation framework described in [28]. We instrument the app binary to insert custom logging code that logs the contents of each page as it is rendered. We then load the instrumented app in a phone emulator (e.g., Windows Phone Emulator for running Windows Phone apps) and use the PhoneMonkey to emulate various user interactions (e.g., by clicking a button, by selecting an item in a drop-down list box, etc.). As various app pages are rendered, their contents are logged, which we retrieve for later analysis. The overall architecture is shown in Figure 2.

Apps. Our binary instrumentation framework is designed for Silverlight [27] apps, which constitute a vast majority of the apps in the Windows Phone marketplace today. We focus on top 1200 Silverlight apps in the Windows Phone marketplace that do not require username and password for an external service (e.g. the Facebook app). We run each of these 1200 app 30 times with the PhoneMonkey. In each run, the PhoneMonkey starts the app, selects a random UI navigational control (such as a button) in the current page, acts on it to navigate to the next app page, and repeats the process until it reaches a page without any UI control or until it goes outside the app (e.g., a web page). We call each run an *app session*. Each session represents a random execution path through the app. For each session we vary the location and other sensor inputs fed to the emulator.

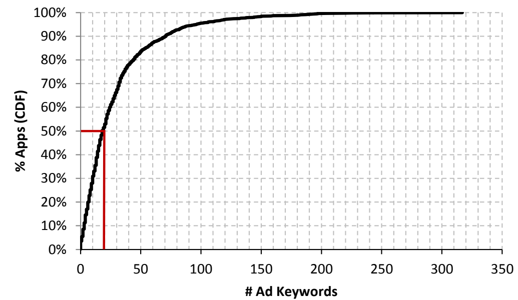

Figure 3: CDF of ad keyword counts extracted from page data. Page data of half the apps contain more than 20 ad keywords that could be targeted with contextual ads.

Keyword Extraction. Our logging instrumentation captures the complete contents of all pages rendered during each session. For advertising purpose, we need to extract a small number of keywords that are prominent in these pages. Ads for these pages will be chosen based on these keywords. Keyword extraction from a webpage is a well-studied area. We use a modified version of the well-known KEX [34] keyword extractor[2] that was originally designed for webpages. KEX uses document features (such as where in the page a word appears, whether it is bolded) and some global information (how many advertisers are interested in the word) to assign each word in the document a weight. We will discuss KEX in more detail in Section 4. For the purpose of this section we treat KEX as a blackbox. The blackbox takes in the page data and a one week trace of bidding keywords from Microsoft's advertising network collected during the first week of July 2012, and assigns each word on the page a weight between 0 and 1 that indicates how "useful" the word is as keyword to base the ad upon. For example, the words "is" and "zebra" may both get a low score: the former because it is too common, while the latter because no one is bidding on it. On the other hand, the word "pipe" may get higher weight because a number of plumbing business have bid upon it.

For the results described below, we consider all words with score higher than 0.1. Using other thresholds changes the absolute numbers, but our conclusions will still hold.

3.2 Results

Page data is a good source of ad keywords. Figure 3 shows the CDF of ad keywords extracted from page data of various apps after running each app 30 times. As shown, most apps contain a good number (> 20) of ad keywords that ad networks can exploit to select contextual ads. This

[2]For convenience, we use the term KEX to refer to [34]; authors did not use any specific name of their system.

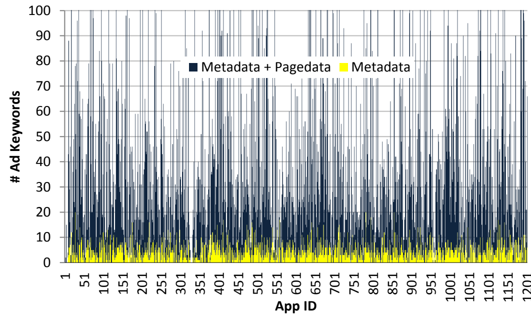

Figure 4: Number of ad keywords extracted from page data and metadata. (The y-axis, with a max value of 317, is clipped at 100 for clarity.) For most apps, page data contains more ad keywords than metadata.

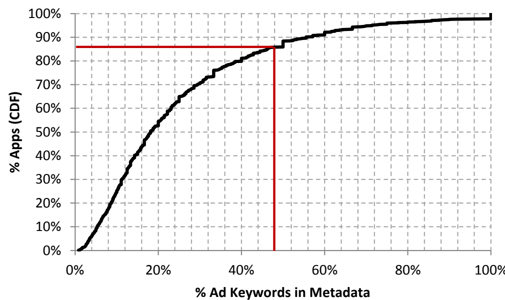

Figure 5: CDF of fraction of ad keywords extracted from app metadata alone. For $>85\%$ apps, metadata contains $<50\%$ ad keywords.

demonstrates the unique opportunity of delivering contextual ads based on page data. We stress that this result is a conservative estimate and the true potential for contextual ads based on page data is likely to be bigger. This is because the PhoneMonkey fails to explore certain execution paths of an app if they require textual inputs (such as a search query) or are behind app-specific UI control that the PhoneMonkey does not know how to interact with (such as an image acting as a button). Thus, in hands of human users, the app is likely to generate more keywords.

Page data yields more keywords than metadata. While page data may be a rich source of ad keywords, we need to show that it yields substantially more keywords than app metadata (such as the name, category and description) – today's mobile ad networks already take metadata into account.

To this end, we separately extract keywords from app metadata and from page data. Figure 4 shows the number of keywords we learn from page data and from app metadata of all 1200 apps. We see that for most apps, page data contains more ad keywords than app metadata. In Figure 5 we show CDF of what fraction of total ad keywords extracted from an app indeed come from metadata alone (i.e., if an app's metadata contains 5 keywords and its page data contains 15 additional ad keywords *that do not appear in metadata*, 25% of its ad keywords come from metadata). The graph shows that more than 85% of the apps have more ad keywords in page data than in metadata. These numbers clearly show the value of extracting ad keywords from page data.

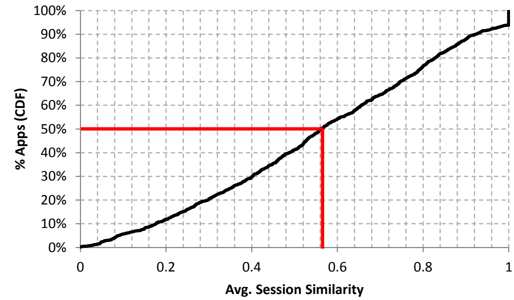

Figure 6: CDF of session similarities of apps. Half the apps have session similarity <0.55.

Page data is dynamic, and requires online keyword extraction. Our results so far demonstrate that extracting ad keywords from page data can be helpful in selecting contextual ads for apps. But we need to decide whether collecting this data at runtime (i.e. online) is necessary.

To understand how important online extraction is, we measure *average session similarity* of various apps. To compute session similarity of an app, we generate $n=30$ app sessions of an app, where each session is a random execution path from the start page of the app. For each session x, we extract the set K_x of ad keywords in all pages in the session. For each pair of sessions x and y, we compute their Jackard similarity $S_{xy}=|K_x\cap K_y|/|K_x\cup K_y|$, which denotes the fraction of keywords common between x and y. Finally, we compute session similarity of the app as the average similarities of all session pairs. Note that the value of session similarity lies within the range $[0,1]$. Intuitively, a small value of average session similarity implies that two random sessions of the app have diverse set of ad keywords and hence they should be shown different ads.

Figure 6 shows the CDF of session similarities of various apps. Median session similarity is 0.55. This means that half the apps have average session similarity less than 0.55. For these apps, each session looks significantly different from other sessions of the same app (with almost half new keywords compared to the other session). This highlights the fact that an offline keyword extraction process is unlikely to work well for all user sessions.

The above result is a conservative one. Recall that for each app we use $n=30$ sessions, which are supposed to be random samples of execution paths in the app. For some apps, however, PhoneMonkey explores only a small number of execution paths. For those apps, many of our sampled sessions look exactly same or very similar, boosting their average session similarity values. With a better PhoneMonkey that can explore more execution paths, app sessions will exhibit even less similarities, further strengthening our argument for online keyword extraction.

Note that another important argument behind the need of online keyword extraction is that a PhoneMonkey may not be able to explore many important pages inside an app. This can happen for many reasons: e.g., if a page can be reached only after some human inputs (e.g., username/password)

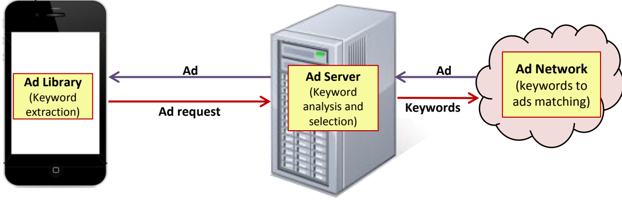

Figure 7: SmartAds Architecture

that the PhoneMonkey cannot emulate, if a page is behind a custom UI control that the PhoneMonkey does not know how to interact with, if the app contains a large number of pages and the PhoneMonkey explores only a fraction of them due to time constraints, etc. With online extraction, we can leave navigation within an app to its user, making sure that we extract keywords from the page the user is currently consuming.

3.3 Summary

The results in this section show that page data provides a rich set of ad keywords, and it is a substantially richer source of ad keywords than the app metadata. Finally, we also showed that page content of the app varies significantly between sessions, and thus, online extraction of keywords from page data is necessary.

4. SmartAds ARCHITECTURE

SmartAds consists of a client-side ad library and an ad server (Figure 7).

The developer includes the ad client within his app. When a user uses the app, the ad client periodically extracts prominent keywords from the current app page and requests an ad from the ad server. The ad server analyzes the keywords sent by the client and ranks them. It then requests an ad network for ads matching these keywords. The ad network is a third-party entity that accepts bids and ads from a variety of sources (e.g. a plumber may create an ad for his business and bid on the word "pipe"). The ad network returns any ads that it may have for the keywords sent by the ad server. The ad server selects the ad matching the highest-ranked keyword, and returns it to the client for displaying.

In this section, we focus on how the ad client and the ad server selects the best keyword to request ads from the ad network. The process that the ad network uses to pick ads that match the keywords is outside the scope of this paper. Our ad server can use any ad network that can return an ad for a given keyword.

The design of SmartAds is guided by the following concerns:

Utility. SmartAds should be able to extract keywords that are prominent on the current app page, while relevant to available mobile ads.

Efficiency. The SmartAds client should have minimal impact on memory consumption of the app, as well as on its network, CPU and energy footprint.

Privacy. SmartAds should not violate user's privacy, e.g., by sending sensitive words such as user's bank account number from an app page to the ad server.

The core functionality of SmartAds is keyword extraction. Given an app's page data, SmartAds extracts prominent keywords that describe the theme of the app page and can be matched with available ads. One might consider using existing keyword extractors, such as the well-known KEX [34], designed for extracting ad keywords from web pages. They can offer good utility, but they pose a tradeoff between efficiency and privacy depending on where the extraction is done.

Extracting keywords entirely on the client is infeasible, because good keyword extractors uses some *global* knowledge that can be too big to fit on the client's memory. For example, an important component of SmartAds's keyword extractor is a dictionary of bidding keywords and their global popularity among ads. However, the database of all keywords that advertisers are bidding on can be several hundred MBs (§4.2.2). This database needs to be in the RAM for fast lookup. Most mobile platforms limit the amount of RAM the app can consume to avoid memory pressure. For example, the Windows Phone limits apps to consume only 90MB of RAM at runtime. Android and iOS also impose a similar restriction.

Running KEX at the server is also problematic. This is because the client would have to upload the entire content and layout information of the page, to allow KEX to extract all the necessary features. This not only wastes communication bandwidth (average page size, including their layout information, is several KBs in the 1200 apps we studied), but can also compromise user privacy, since sensitive information such as a user's name or bank account number, could be sent to the server.

To address these concerns, SmartAds uses a novel keyword extraction architecture. In the rest of the section, we describe various components of the architecture and how they achieve good utility, efficiency, and privacy.

4.1 Achieving Good Utility

To extract prominent keywords from an app page, we start with KEX that has been found effective for web pages and modify it to address efficiency and privacy concern. Given a page, it produces a ranked list of keywords with scores between 0 and 1, indicating how useful each keywords is, to base the ad on.

The core of KEX is a classifier. Given a feature vector of a word W in document D, it determines the likelihood score of W being an advertising keyword. More formally, it predicts an output variable Y given a set of input features $\overline{X}$ associated with a word W. Y is 1 if W is a relevant keyword, and 0 otherwise. The classifier returns the estimated probability, $P(Y = 1|\overline{X} = \overline{x})$:

$$P(Y = 1|\overline{X} = \overline{x}) = \frac{\exp(\overline{x} \cdot \overline{w})}{1 + \exp(\overline{x} \cdot \overline{w})} \quad (1)$$

where the vector of weights is $\overline{w}$, w_i is the weight of input feature x_i.

The original KEX implementation was designed to extract keywords from web pages. We now describe how we modified it for SmartAds.

4.1.1 Local features

KEX uses many document features that are specific to web pages. We exclude features that do not apply to app pages. For example, KEX assigns higher weight to a word

that appears in the HTML header. This does not apply for app pages. We retained the following KEX local features that are applicable to app pages.

- `AnywhereCount:` The total number of times the word appears in the page
- `NearBeginningCount:` The total number of times the word appears in the beginning of the page. Beginning is defined as the top 33% of the screen.
- `SentenceBeginningCount:` The number of times the word starts a sentence.
- `PhraseLengthInWord:` Number of words in the phrase.
- `PhraseLengthInChar:` Number of characters in the phrase.
- `MessageLength:` The length of the line, in characters, containing the word.
- `Capitalization:` Number of times the word is capitalized in the page. It indicates whether the word is a proper noun or an important word.
- `Font size:` Font size of the word.

In addition, app pages have features that are not found in HTML pages. For example, based on the data collected in §3, we found that the UI element (e.g. TextBox) containing user input is a good indicator of the word's importance. Thus, we needed to include the UI element containing a word to the list of document features that KEX considers in its ranking function. However, we used a somewhat ad hoc approach to determine the weights of various UI controls to be used in KEX's model: we mapped various UI controls to existing KEX features. For example, if a word appeared in a user input `TextBox`, it is considered to be `Capitalized` and `NearBeginning`.

In principle, one should train KEX's machine learning model with a large corpus of labeled page data to determine the weight of various UI elements. However, innovating on core machine learning techniques for keyword extraction is outside the scope of this paper; our aim in this paper is to build an end-to-end system for delivering contextual ads to mobile apps while striking a right balance between relevance, efficiency and privacy. We therefore leave such training as future work. Once such weights are learned from training data, they can be readily incorporated into SmartAds to further improve its relevance.

4.1.2 Global knowledge

KEX is designed to use a large English dictionary that ascribes relative importance to words, based on how common they are in the English language. For example, words such as "the" are assigned a lower weight.

In our setting, we instead use the knowledge about how often advertisers bid on a keyword. To learn this, we use a 1-week bidding keyword trace collected from Microsoft's Bing ad network during the first week of July 2012. As we show in Section 4.2.2, using a longer trace has negligible impact. Having this trace, we assign each word a weight equal to $\log(1 + frequency)$, where $frequency$ is how many times the word appears in the bidding keyword trace. This reflects the distribution of the keywords advertisers are most interested in.

Using the above local features and global knowledge, our modified KEX produces a good set of ad keywords from an app page (as is evident in our evaluation results in §5). However, an ad network can have millions of bidding keywords, making the global knowledge too large to fit in a mobile device's memory. Therefore, KEX cannot be run entirely on the client. On the other hand, running KEX entirely on the server poses network overhead and privacy concerns. We now discuss how SmartAds addresses these concerns.

4.2 Achieving Efficiency

4.2.1 Addressing memory overhead

The large memory overhead of keyword extraction comes from the large global knowledge. To avoid this overhead at the client side, we split the KEX functionality between the client and the server such that the global knowledge (and associated computation) is maintained at the server and the client does only as much as is possible without the global knowledge. Observe that the scoring function as shown in Equation 1, is based on dot products of the feature vector $\overline{x}$ and the weight vector $\overline{w}$. Dot product is *partitionable*: it can be computed partially at the client and partially at the server. Thus, we partition $\overline{x}$ into a vector of local features $\overline{x_l}$ (e.g., `AnywhereCount`) and a vector of global features $\overline{x_g}$ (e.g, global knowledge of a keyword), with weight vectors $\overline{w_l}$ and $\overline{w_g}$ respectively. For each word w in a given page, the client computes its local score $L_w = \overline{x_l} \cdot \overline{w_l}$ and sends (w, L_w) to the server.[3] The server uses the global knowledge to compute w's global score $G_w = \overline{x_g} \cdot \overline{w_g}$. Finally, the server computes the overall score of the word w as $S_w = L_w + G_w$. The score S_w is then used in Equation 1 to compute w's probability of being an advertising keyword.

4.2.2 Addressing communication overhead

Note however, that uploading all words on the page to the server is both wasteful, and can potentially violate privacy. We now describe how we tackle these problems.

Intuitively, the client does not need to send a word if it does not have any chance of being selected as one of the extracted keywords at the server. *How can the ad client locally prune unnecessary keywords?*

We exploit knowledge about the keywords that advertisers bid on to achieve such pruning. The client keeps a list of all bidding keywords and sends a word to the server only if it is one of the bidding keywords. However, there can be too many bidding keywords (typically hundreds of millions) to fit in client's memory. Moreover, merely checking bidding keywords is not sufficient; we also need to consider words that are *related* to the bidding keywords, further increasing the memory overhead. (See §6.1 on a discussion on how SmartAds gets related words).

To address this challenge, we compress the list of bidding keywords and related keywords using a *Bloom filter* [6]. A Bloom filter is a space-efficient probabilistic data structure, that can be used to test whether an element is a member of a set. False positive retrieval results are possible, but false negatives are not. The Bloom filter is constructed by the server, from its knowledge of bidding keywords, and sent to the ad client. The ad client uses the Bloom filter to check whether a candidate word is included in the list of bidding keywords or not. The client sends a word to the ad server only if it passes the Bloom filter.

[3] If the page does not contain enough ad keywords (e.g. a page with just images), we fall back to ad keywords in previous app pages in the same session or to ad keywords in app metadata. We discuss this in detail in (§6.1).

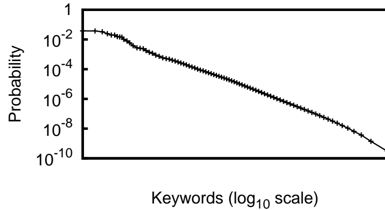

Figure 8: PDF of bidding keyword frequencies in a 1-week long Bing ad trace

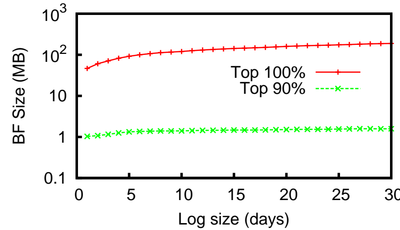

Figure 9: Bloom filter size for top x% and related keywords, for various trace length

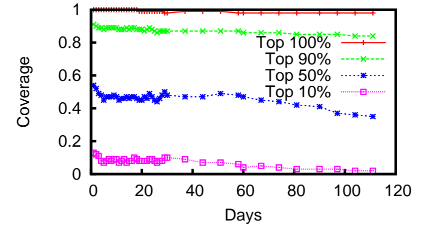

Figure 10: Coverage of top x% and related keywords from 1-week trace over time

There can be tens of millions of bidding keywords in an ad network and hence a Bloom filter can be very large if we include all bidding keywords. Therefore, we use the second optimization of including only a relatively small number of bidding keywords that cover most of ads in the ad network.

To see whether the above techniques are feasible, we need to answer two important questions: (1) what is the memory footprint of the Bloom filter? (2) how often does the Bloom filter change? We answer these questions with a 1-week trace of bidding keywords collected from Microsoft's ad network within the first week of July 2012.

Bloom filter size at the client. The size of a Bloom filter depends on the number of items and false positive rates of lookups one is willing to tolerate. Simple mathematical analysis shows that for n items and a false positive rate of p, the optimal size of a Bloom filter is $-\frac{n \ln p}{(\ln 2)^2}$ bits [8].

Ideally, we would like to use all bidding keywords. Our bidding keyword trace shows that this size can be prohibitively large to fit in a smartphone. Fortunately, there are many popular bidding keywords each of which appears in labels of a large number of ads. In particular, frequencies of bidding keywords follow a power law distribution, as shown by the pdf of bidding keywords in our one week trace in Figure 8. (The units on the x-axis are omitted as it is sensitive to Microsoft's business.) This implies that a small number of bidding keywords appear in most of the ads. More precisely, in the 1-week trace, 2.2% most frequent bidding keywords can fit in a smartphone's memory and yet cover 90% of the ads. We can therefore use this small fraction of bidding keywords and yet achieve a high coverage of ads.[4]

Figure 9 shows the size of the Bloom filter maintained at the client, for various lengths of Bing traces. The size of the Bloom filter is chosen to be optimal to bound the false positive rate to $< 1\%$. We show numbers for two cases when we consider (a) all unique keywords in the trace and (b) only the most frequent unique keywords covering 90% of the ads. As shown, the Bloom filter size is small (around 1MB) when we consider keywords covering 90% of the ads. The graph also shows that Bloom filter size is not much sensitive to how long a trace we use.

Dynamics of bidding keywords. A Bloom filter is not incrementally updatable: even though new items can be added dynamically, items cannot be deleted[5]. Therefore, if the set of bidding keywords that we use for local pruning changes dramatically, the client needs to download the entire Bloom filter from the server. For practical reasons, this should happen rarely.

To see how frequently bidding keywords change, we do the following experiments with our trace. We first build a Bloom filter with a base keyword set S_x, which consists of top keywords covering x% of the ads in a 1-week long trace. Then, on each subsequent day, we compute what fraction of ads available on that day on Microsoft's ad network are labeled with the keywords in S_x. If there are too much churns of bidding keywords, the coverage should drop down quickly. Figure 10 shows the results for various value of x. It shows that coverage is relatively stable for higher values of x. For example, starting with keywords covering $x = 90\%$ of the ads, coverage remains higher than 85% even after three months. This suggests that the Bloom filter at the client does not need to be updated very often. Bloom filter's small size and infrequent update rate make it practical to be used in our application.

4.3 Achieving Privacy

Privacy and contextual ads are at odds with each other since for the ad server to select an ad relevant to an app page, it needs to know the page content. The solution we have presented so far already provides some form of privacy: *the ad server knows only the ad keywords in the page and nothing else.* Since ad keywords are essentially popular keywords bid by advertisers, they are likely to be non-sensitive keywords. This also raises the bar of an adversarial advertiser to exploit the system: since we pick only popular bidding keywords, an adversary is unlikely to make a sensitive word into our list of popular keywords without making a large number of bids for the same keyword. Finally, the ad server makes the list of popular keywords public so that a third party can audit if the list contains any sensitive keywords.

Note that SmartAds does not guarantee absolute privacy, where no information about the client is disclosed to the server. In fact, as shown in [15], *it is impossible to guarantee such absolute privacy* in a client-server contextual ad system without sacrificing ad quality or system efficiency. The only guarantee SmartAds provides is that the ad server does not know the existence of any word in a user's app page unless the word is an ad keyword.

Even achieving this guarantee is not straightforward while using Bloom filter as it can have false positives. Therefore, the ad client may occasionally send to the ad server sensitive words (such as his SSN or a name of a disease) that appear in an app page but are not ad keywords. This can violate user's privacy.

To avoid potential privacy breach, the ad client and the

[4]We discuss in Section 6.2 how the remaining 10% ads actually get served to clients.

[5]Deletion is supported in counting Bloom filter that has more memory footprint and therefore we do not use this.

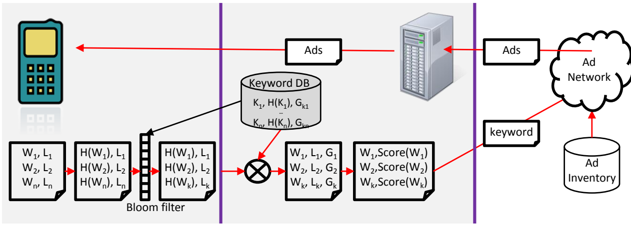

Figure 11: Keyword extraction in SmartAds

ad server use a one-way hash function and operate on hash values of all keywords instead of their plaintexts. The server builds the Bloom filter of hashvalues of all ad keywords. The client hashes all candidate keywords on the current page and sends only the hashvalues if they exist in the Bloom filter. The server maintains a dictionary of all ad keywords and their hashvalues; hence it can map a hashvalue to its plaintext only if it is an ad keyword. The server ignores all hashvalues that do not appear in its dictionary, without knowing their plaintexts. In this way, SmartAds achieves our target privacy goal that the ad server knows plaintexts of only the words that are popular ad keywords.

4.4 End-to-end workflow in SmartAds

Figure 11 shows the overall operation of the SmartAds system. It works as follows.

Offline Processing. The ad server maintains a database containing all ad keywords. For each keyword k, the database maintains k, a hashvalue $H(k)$ of k, and a global feature value G_k of K. The value G_k is used by the keyword extraction algorithm for computing overall ranking of a keyword. For our KEX-based keyword extractor, G_k is computed as $\log(1 + freq_k)$, where $freq_k$ is the number of times K is used to label any ad in the ad inventory. The database is updated as the ad inventory is updated.

Periodically (e.g., once in every three months) the server computes a Bloom filter from all the $H(k)$ values in the keyword database and sends it to mobile phone clients. The size of the Bloom filter is optimally selected based on the number of keywords in the keyword database and a target false positive rate.

Online Processing. The ad client works as follows. After an app page is loaded, it "scrapes" the current page content to generate a list of candidate keywords. Typical app pages are organized as a hierarchy of UI controls (e.g., text box, images, list box); scraping is done by traversing the hierarchy and extracting texts in all UI controls. For each scraped word W, the client module computes its hash $H(W)$ (using the same hash function the server uses to generate the keyword database) and its local feature vector L_w. If $H(W)$ passes the Bloom filter, the pair $(H(W), L_w)$ is sent to the server. If a $H(W)$ value does not appear in the server's keyword database (i.e., a false positive in the client's Bloom filter), it discards the value, without knowing the corresponding word W. Otherwise, it retrieves G_w from the keyword database and combines it with L_w to compute the overall score of the word W. Keywords with scores above a threshold are selected as extracted keywords.

Our Implementation. We have implemented SmartAds system for Windows Phone apps. The ad client is a DLL that developers can include in an app page programmatically or by dragging and dropping from Microsoft Visual Studio control toolbox. We have also implemented a tool that can insert the ad client into existing apps with binary rewriting techniques [28]. The ad server runs in Windows Azure. We use Microsoft Bing's ad network for retrieving suitable ads matching a given keyword.

5. RESULTS

In this section, we evaluate the performance of SmartAds. The most important metric for evaluating the performance of any ad selection system is the relevance of the ads shown. As shown in Figure 1, SmartAds can display a more contextually relevant ads, compared to other ad controls. Since relevance is subjective, we quantify relevance of ads shown by SmartAds via a large user study described in §5.1. The next important metric is the delay incurred in fetching and showing the ad. Users of mobile apps typically have short attention spans [12] and the best time to draw a user's attention is during or as soon as he moves from one page to another [17]. So, ads must be fetched and displayed as soon as possible after the page loads. Our results (§5.2) show that the SmartAds system is responsive and is able to fetch relevant ads quickly. Finally, in §5.3, we evaluate the impact of the ad control on the mobile phone, in terms of CPU, memory, network and battery consumption. If this overhead is excessive, app developers may not use the ad control. Our results show that the overhead is minimal.

5.1 Relevance

SmartAds dynamically retrieves ads based on the keywords extracted from the content of the app page and hence delivers ads that are contextual and more relevant. We evaluate the increase in relevance via a user study.

As a baseline for our comparison, we consider the relevance of the ads delivered by a major ad control for Windows Phone. We refer to the ad control as the *baseline ad control* and the ads shown by it as *baseline ads*. The baseline ad control uses a number of useful signals such as app metadata (e.g., app description and category in the marketplace) and user's location to choose relevant ads.Based on official documentation, other ad controls use the same or a subset of signals as the baseline ad control uses: some use location only, some use location plus metadata, etc. (No ad control for Windows Phone uses page data for choosing ads.) Therefore, we believe ads from our baseline ad control is a reasonable comparison point for our study.

The user study was based on top 353 apps from Windows Phone marketplace that used the baseline ad control. We ran these 353 apps using the PhoneMonkey (§3.1). Each app was run for 10 minutes. As the PhoneMonkey visited various pages in the application, it captured and logged the XML used to render the page. Various heuristics were used to ensure that the XML was captured only after the page was fully loaded. The contents included the ad displayed by the baseline ad control. We captured no more than 10 unique pages per app. In total, we captured a total of 2500 unique pages from the 353 apps.

Next, we removed the existing ad (provided by the baseline ad control) from these pages and ran SmartAds offline on the page content. SmartAds extracted keywords from the page and fetched suitable ads for the given keywords from the Microsoft Bing's ad network. At this point, for each

page we had an ad served by the baseline ad control and by the SmartAds ad control.

Then, using this data we generated two different screenshots for each page, an original one showing the ad from the baseline ad control and the other showing the ad obtained by SmartAds. Hence, in total, we had a database of 5000 app screenshots showing ad either from baseline ad control or from SmartAds.

The user study was based on these 5000 screenshots. We created a website where users could indicate how relevant the ad in the screenshot is to the content of the screenshot. The user study website is shown in Figure 12. We provided users with three score levels[6] (i) Very relevant (ii) Somewhat relevant and (iii) Not relevant. Users were not told the source of the ad in the image. After a user labeled an image, he/she was shown another randomly selected image.

We asked users in a mailing group consisting of 400 people in our organization to visit the website and label the images. The responses were completely voluntary and anonymous. The website was accessible only from our corporate intranet.

In total, we obtained 6230 labels. Each screenshot got at least one label and a random subset of them got more than one. There were 3105 labels for baseline ads and 3125 labels for SmartAds.

The results from the user study are shown in Figure 13. When the ad was generated by SmartAds, over 75% of the ads were labeled either "Very relevant" or "Somewhat relevant". In contrast, only 41% of the ads generated by the baseline ad control were labeled "Very relevant" or "Somewhat relevant". This result clearly demonstrates the importance of using the content of the page while generating the ad.

From the results we also see that about 25% of the ads generated by SmartAds were labeled non-relevant. We investigated these pages further and found that the non-relevancy stems from two major factors. First, there are apps which display minimal text content in the screen and mostly show images or controls (for e.g. games). Second, in certain apps, the keywords on the screen does not match any bidding keyword i.e. there are no ad providers bidding for content (keyword) related to the app. In both these cases, SmartAds falls back to retrieving keywords from the metadata of the app (See §6.1) which in many cases are not relevant to the content on the screen.

Note that in practice Windows Phone ads are likely to be more relevant than our study shows. This is because Microsoft Ad Control, which is used by $> 80\%$ ad supported apps, uses behavioral and location-based targeting as well. With behavioral targeting, a user is shown ads relevant to his search history in Bing (if he is signed in to Bing by using the same Microsoft ID he uses in his Windows Phone). With location-based targeting, a user is shown ads on near-by business. We disabled these targeting in our study (i.e., we disabled phone's GPS and the PhoneMonkey did not have any Bing history) as we wanted to demonstrate benefits of using page data over metadata in choosing ads. Behavioral and location-based targeting are orthogonal to contextual targeting, and they can be combined. We leave studying the effectiveness of such combined targeting as part of our future research.

[6]We experimented with the number of levels in smaller settings, and discovered that three levels struck the right balance between useful granularity and user irritation.

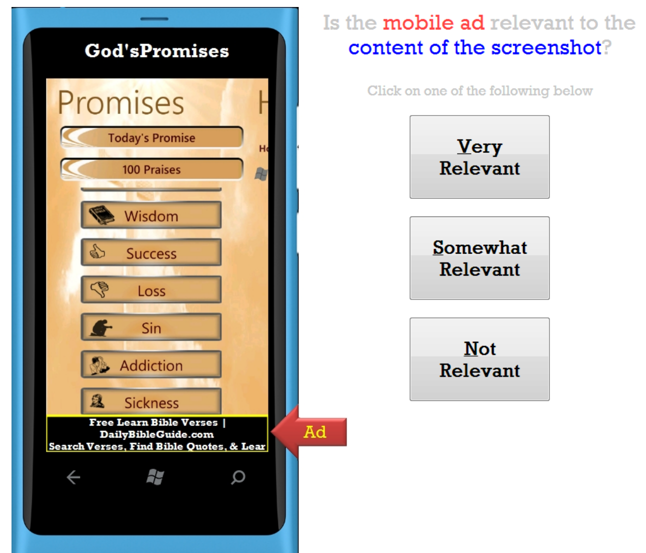

Figure 12: User Study Website

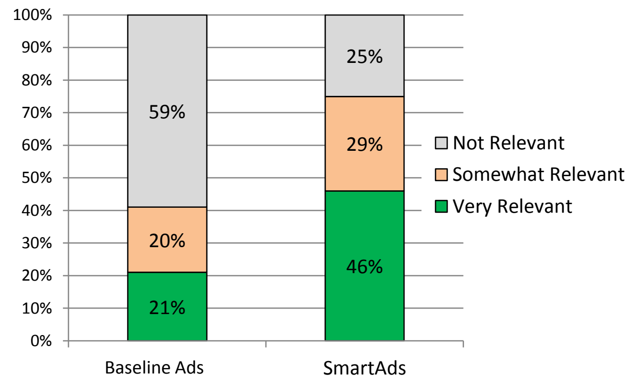

Figure 13: Ad relevance results from the user study

5.2 End-to-end performance

Mobile app users are impatient, and in many cases spend only tens of seconds on each app [12]. Moreover, the best time to draw a user's attention is during or as soon as he moves from one page to another [17]. is not fetched and displayed sufficiently quickly after a page is loaded, the user may never notice it. To characterize this end-to-end performance, we measure the time to deliver an ad starting from Page Load. This is the time that elapses between the instance the user navigates to a page, and before he sees the ad.

To measure the end-to-end performance and overhead, we inserted SmartAds client into a set of existing apps by rewriting app binaries [28]. These apps already had the baseline ad control and binary rewriting simply replaced the the ad control library with ours. We asked 5 volunteers to run these apps over a period of 5 days. All users had a Samsung Focus phone running Windows Phone 7.1. We instrumented both the ad client and the ad server to measure the end-to-end performance and the performance of different components.

When a page loads, SmartAds scrapes the page, does key-

Component	Average Runtime (ms)
Page Scraping	30
KEX Local	20
KEX Server	<1
Ad network query	500

Table 1: Average runtime of various components in the SmartAds implementation.

word extraction, runs them through the bloom filter and sends the filtered keywords to the ad server. The ad server in turn queries the ad provider for a relevant ad and sends the ad back to the app.

In our deployment, the average end-to-end time to deliver an ad after the page loads is 650ms. SmartAds performance is comparable to the performance of other ad controls.

The breakdown of the time consumed by the different components of the SmartAds system is shown in Table 1. We see that the keyword extraction at the client and the server are implemented efficiently and takes only 20ms on average. Page scraping takes only 30ms on average. Our biggest bottleneck is the time to query the ad network from the ad server. This delay is not under our control, although if we were to deploy SmartAds in production, we could work with the ad network to reduce this delay.

5.3 Overheads

Our distributed keyword extraction technique significantly reduces the amount of resources consumed at the mobile device. In this section, we show the end-to-end performance of SmartAds and the overhead across various resources including compute, memory, network and battery.

CPU Overhead. SmartAds runs asynchronously and does not affect the performance of the app. Even so, the CPU overhead is minimal. The combined runtime of page scraping and local keyword extraction represents the computation overhead at the mobile device. As seen from Table 1, this overhead is minimal.

Memory Overhead. As mentioned is Section 4, SmartAds uses a compact bloom filter to check for ad keywords instead of storing all ad keywords in memory. The size of our bloom filter is 1MB. To measure the memory overhead of our ad control, we run apps with and without the ad control and measure the average peak memory load. Based on our measurements, SmartAds control consumes around 2.8MB of memory on an average. This overhead is low compared to the typical amount of memory consumed by apps today.

Network Overhead. In comparison to existing mobile ad controls, the extra data that SmartAds control sends are the keywords and their weights to the ad server. We send 12 bytes of data per keyword (8 bytes for keyword hash and 4 bytes for weight). From our analysis of 1200 apps, we find that the average number of keywords extracted per page is 7.6. Hence, the average extra bytes sent is 91.

We also measured how much existing ad controls upload and download each time they fetch ads. The number varies across ad controls; but on average various ad controls upload 1.5KB and download 5KB of data. Thus, even if SmartAds functionality are incorporated into existing ad controls, the additional overhead (< 100 bytes) would be negligible.

Battery Overhead. To measure the battery overhead, we use a software power meter [22]. We run apps with and without the ad control through the power meter and measure the average power consumed. The increase in power consumed was less than 1% and well within experimental noise.

5.4 Summary

The results in this section show that the SmartAds system delivers relevant ads. Compared to the baseline ad control, which takes into account only the application description, SmartAds significantly improves relevance score. The results also show that SmartAds is efficient and consumes minimal resources.

6. DISCUSSION

6.1 Optimizations

In §4.2, we briefly mentioned two optimizations, without discussing them in detail. Specifically, we mentioned that we rely on metadata if the page data does not contain any keywords. We also mentioned that we augment ad keywords with related words. We now describe these two optimizations in more detail.

Addressing lack of text. One problem of extracting ad keywords from app pages is that some pages do not contain enough texts and hence keyword extractor do not produce any advertising keywords. To show relevant ads even on those pages, we use three levels of keywords. Level 1 keywords are the ones dynamically learned from the current page. Level 2 keywords are the ones dynamically learned from all the pages the user has viewed in the current session. Additionally, the ad server maintains Level 3 keywords for each app, learned offline from that app's metadata. If Level 1 keywords are empty, we fall back to Level 2 keywords. If both Level 1 and Level 2 keywords are empty, we fall back to Level 3 keywords to select relevant ads. Intuitively this means that we always give preference to the current page to show ads. If current page does not contain any advertising keywords, we consider all the pages the user has visited in the current session. Finally we consider app descriptions and content of all app pages (including the ones the user has not visited in this session) to extract keywords.

Handling related keywords. Suppose, the set of bidding keywords contains only one keyword {`HDTV`} and the current app page contains the words `LED TVs are cool`. Clearly, after filtering based on bidding keywords, the ad client will not extract any keywords even though it could show an ad for `HDTV` on this page because `LED TVs` and `HDTV` are related. Typical keyword extraction tools (including KEX) ignore such related words. However, we would like to capture such relations because a typical app page contains small amounts of texts, and hence capturing related words would give us opportunities to show more relevant ads. We therefore extend the set of original bidding keywords with their related words: {`HDTV`, `LED TV`, `LCD TV`}. We call this extended set of bidding keywords and their related words as *ad keywords*.

We use two data sources to find related words. The first source is a database of related keywords automatically extracted by analyzing Bing web queries and click logs. The degree of relationship between two keywords is computed based on how often Bing users searching for those two key-

words click on the same URL. The second source is a web service provided by `http://veryrelated.com` that, given a keyword, returns a list of related keywords. Based on a WWW snapshot from Yahoo, it automatically discovers related words and concepts. The degree of relationship between two keywords is computed based on how often they appear in the same web page and how popular they are on the Internet.

Our experiments show that such keyword expansion is useful. For example, starting with a list of 1.3 million most frequent bidding keywords, the first and the second source above expanded the list to 1.96 million and 2.06 keywords respectively. We also found that around 25% of the keywords we extracted from app pages are these related keywords, which means we would have lost the opportunities of delivering relevant ads had we not considered these related keywords.

6.2 Dealing with Tail Bidding Keywords

Recall that we have configured SmartAds's client-side Bloom filter to include a small number ($\approx 2\%$) of popular bidding keywords that cover around 90% of the ads. The reader might worry that ads with bidding keywords in the tail of the distribution (i.e., remaining 10% of the ads) may never get served to users.

In practice, however, contextual signal (i.e., ad keywords extracted from current page) is only one of many signals used by a real ad system; in many occasions the ad system would use other signals such as a user's location (for context-aware targeting), his web or app usage history (for behavioral targeting), etc. The remaining 10% of the ads can be served in these situations, by matching their keywords with a user's search history for example. Even if the ad server decides to use the contextual signal to serve ad, it can occasionally serve these 10% tail ads in multiple ways: (1) by serving them when their keywords are semantically related to any keyword in the Bloom filter, (2) by prioritizing them when the current page does not contain any important ad keywords, or (3) by occasionally serving them even if the contextual signals do not match them.

Our design of SmartAds exposes a tradeoff between ad coverage and memory/network overhead, and the ad system can operate at a desirable sweet spot by tuning the size of the Bloom filter. The smaller the Bloom filter, the smaller the memory and network overhead and the larger the fraction of uncovered tail ads. SmartAds could also disable the Bloom filter at the client, in which case the client would send local features of *all* words in the current page to the server. This would increase network overhead compared to our current Bloom-filter-based design, but would still be cheaper than sending the raw text and metadata of the whole page.

6.3 Developer provided keywords

The reader may have wondered whether app developers could supply the right keywords to ad controls at run time, obviating the need for SmartAds. A few mobile ad controls do have this option [2]. For example, the developer of a recipe app can specify appropriate keywords, allowing the ad server to deliver ads on food ingredients or groceries. One might argue that an app developer could hard-code static ad keywords for every page of his app. He could also implement some logic to dynamically generate them during runtime. However, such logic is hard to implement in practice because the quality of an ad keyword depends on information the developer might not have access to (e.g., how popular a keyword is among advertisers). SmartAds automates the whole process *with zero developer effort.* Note also that for certain apps such as news aggregator, the developer simply does not know what content may be displayed at runtime. Finally, allowing app developers to specify keywords opens up the possibility of *keyword pollution*, where a developer intentionally provides popular keywords that do not appear in the page, in an attempt to increase the chance of getting matching ads from the ad network. Therefore, some ad networks simply ignore developer-provided keywords.

7. RELATED WORK

There is very little research literature specifically related to improving the relevance of mobile advertisements, and most of that has focused on using location to increase the relevancy of the advertisements shown. In the AdNext system [19], authors propose to use mobility patterns to predict which store the user is likely to visit next, and show him advertisements related to that store. In [15], authors consider using various physical contexts such as location and user activities to serve ads. These works are orthogonal to SmartAds; SmartAds could incorporate such additional signals to further increase ad relevance.

The CAMEO framework [18] and the study in [23] consider pre-fetching advertisements and displaying relevant ads at the right time. The focus of these works is on saving bandwidth by pre-fetching ad — not on increasing relevance. CAMEO considers location and app name as the only relevance signal. We have not considered the possibility of prefetching the ads in SmartAds design, since we want to adapt the ads dynamically to the content being displayed to the user.

Many researchers have focused on power consumption and privacy issues raised by mobile apps, and by advertisement controls. While we are indeed concerned about resource utilization and privacy, these issues are not the primary focus of our paper. However, we will discuss representative papers for sake of completeness. In [26], authors have shown that much of the energy consumed by mobile apps is used by third-party ad controls. The SmartAds control is designed to consume as few resources as possible. A number of researchers have looked at privacy violations by mobile apps [11, 10, 15] and especially those of third-party ad controls [14]. SmartAds protects app user's privacy by leaking only which ad keywords occur on the screen.

Delivering relevant ads to web pages has been well studied. In achieving the goal, one of the most popular approaches is keyword extraction. Riberio-Neto *et al.* [29] propose a series of strategies to match ads with web pages, based on keywords extracted from both the ad body and the web page content. Centering on keyword extraction, many methods have been proposed to better describe both advertisements and web pages, in order to match them more accurately. For instance, eBay uses their custom features (such as merchandise categories) to depict their advertisements, in matching their ads with numerous 3rd-party web pages [30]. KEX [34] uses additional web-specific features (e.g., HTML tags, URL, etc.) to identify keywords that carry more weights from any given web page. Inspired by them, SmartAds further employs features that are specific to mobile application UI in order to improve the quality of extracted keywords.

"Just-in-time" contextual advertising dynamically extracts keywords and other signals from dynamic web pages for selecting relevant ads [5, 20]; however, as mentioned in Section 2, they use web page-related features and are not directly applicable to mobile apps. Beyond keywords, several 'deeper' features have been employed for delivering relevant ads, including semantics mined from web pages (e.g., topics) [7] and user interactions in browsing [9]. However, such in-depth understanding requires to run intensive algorithms (usually in the cloud) over full page contents or user inputs, making them infeasible in the case of mobile Ads due to privacy concerns.

8. CONCLUSION

Our focus in this paper was to bring contextual advertising to mobile ads. Using data crawled from 1200 top Windows Phone apps, we demonstrated the need for a system that delivers contextual ads. We designed and built SmartAds, a contextual mobile advertising system that strikes a right balance between ad relevance, efficiency and privacy.

Through a large user study, we showed that SmartAds significantly improves the relevance of mobile ads compared to systems that rely only on the metadata information of the app. We also analyzed the performance of SmartAds end-to-end and showed that SmartAds is efficient, consumes minimal resources and incurs low overhead.

9. REFERENCES

[1] Admob. http://www.google.com/ads/admob.

[2] Admob. keyword-targeted ads. http://support.google.com/admob/bin/answer.py?hl=en&answer=1307264.

[3] Google adsense. http://www.google.com/adsense.

[4] Advertising.com. http://www.advertising.com/.

[5] A. Anagnostopoulos, A. Z. Broder, E. Gabrilovich, V. Josifovski, and L. Riedel. Web page summarization for just-in-time contextual advertising. *ACM Trans. Intell. Syst. Technol.*, 3(1):14:1–14:32, 2011.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] A. Broder, M. Fontoura, V. Josifovski, and L. Riedel. A semantic approach to contextual advertising. In *SIGIR*, 2007.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2002.

[9] K. S. Dave. Computational advertising: leveraging user interaction and contextual factors for improved ad retrieval and ranking. In *WWW*, 2011.

[10] D. Davidson and B. Livshits. Morepriv: Mobile os support for application personalization and privacy. Technical report, Microsoft Research, 2012.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Seth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[12] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *MobiSys*, 2010.

[13] Flurry Blog. The great mobile ad spending gap. http://blog.flurry.com, February 2012.

[14] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *WiSec*, 2012.

[15] M. Hardt and S. Nath. Privacy-aware personalization for mobile advertising. In *ACM CCS*, 2012.

[16] iAd. http://advertising.apple.com/.

[17] S. Iqbal and B. Bailey. Effects of intelligent notification management on users and their tasks. In *ACM CHI*, 2008.

[18] A. J. Khan, K. Jayarajah, D. Han, A. Misra, R. Balan, and S. Seshan. CAMEO: A middleware for mobile advertisement delivery. In *ACM MobiSys*, 2013.

[19] B. Kim, S. Kang, J.-Y. Ha, Y. Rhee, and J. Song. AdNext: A Visit-Pattern-Aware Mobile Advertising System for Urban Commercial Complexes. In *HotMobile*, 2012.

[20] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, 2010.

[21] Microsoft advertising. http://advertising.microsoft.com/.

[22] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *MobiCom*, 2012.

[23] P. Mohan, S. Nath, , and O. Riva. Prefetching mobile ads: Can advertising systems afford it? In *EuroSys*, 2013.

[24] Microsoft mobile ad control. http://advertising.microsoft.com/mobile-apps.

[25] S. Ovide and G. Bensinger. Mobile ads: Here's what works and what doesn't. http://online.wsj.com/article/SB10000872396390444083304578016373342878556.html?mod=WSJ_hp_EditorsPicks, September 2012.

[26] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys*, 2012.

[27] C. Perzold. *Microsoft Silverlight Edition: Programming Windows Phone 7*. Microsoft Press, 2010.

[28] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI*, 2012.

[29] B. Ribeiro-Neto, M. Cristo, P. Golgher, and E. de Moura. Impedance coupling in content-targeted advertising. In *SIGIR*, 2005.

[30] X. Wu and A. Bolivar. Keyword extraction for contextual advertisement. In *WWW*, 2008.

[31] Yahoo! Smart Ads. ttp://advertising.yaoo.com/marketing/smartads/.

[32] Yahoo publisher network. http://advertisingcentral.yahoo.com/publisher/index.

[33] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen. How much can behavioral targeting help online advertising? In *WWW*, 2009.

[34] W.-t. Yih, J. Goodman, and V. R. Carvalho. Finding advertising keywords on web pages. In *WWW*, 2006.