Abusing Cloud-based Browsers for Fun and Profit

Vasant Tendulkar
NC State University
vstendul@ncsu.edu

Joe Pletcher
University of Oregon
pletcher@cs.uoregon.edu

Ashwin Shashidharan
NC State University
ashashi3@ncsu.edu

Ryan Snyder
University of Oregon
rss@cs.uoregon.edu

Kevin Butler
University of Oregon
butler@cs.uoregon.edu

William Enck
NC State University
enck@cs.ncsu.edu

ABSTRACT

Cloud services have become a cheap and popular means of computing. They allow users to synchronize data between devices and relieve low-powered devices from heavy computations. In response to the surge of smartphones and mobile devices, several cloud-based Web browsers have become commercially available. These "cloud browsers" assemble and render Web pages within the cloud, executing JavaScript code for the mobile client. This paper explores how the computational abilities of cloud browsers may be exploited through a Browser MapReduce (BMR) architecture for executing large, parallel tasks. We explore the computation and memory limits of four cloud browsers, and demonstrate the viability of BMR by implementing a client based on a reverse engineering of the Puffin cloud browser. We implement and test three canonical MapReduce applications (word count, distributed grep, and distributed sort). While we perform experiments on relatively small amounts of data (100 MB) for ethical considerations, our results strongly suggest that current cloud browsers are a viable source of arbitrary free computing at large scale.

1. INTRODUCTION

Software and computation is increasingly moving into "the cloud." Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) have effectively commoditized computing resources, enabling pay-per-use computation. For example, in April 2012, Amazon's on-demand instances of EC2 cost as little as US$0.08 per hour [4]. This shift towards cloud computing provides many benefits to enterprises and developers. It consolidates hardware and maintenance, and it allows organizations to purchase only as much computing as they need. Equally importantly, the ubiquity of cloud providers and sophisticated interfaces make incorporating cloud functionality simple for virtually any piece of software.

Cloud computing has substantially benefited smartphones and mobile devices, relieving them of computation, storage, and energy constraints. Recently, several commercial ventures have deployed infrastructures for rendering Web pages in the cloud (e.g., Amazon Silk [5], Opera Mini [24], and Puffin [12]). The obvious benefit to this architecture is relieving the mobile device from the graphical rendering. However, this is less of a concern for newer, more powerful smartphones. Such devices benefit more from the cloud server downloading the many parts of a Web page using high-bandwidth links and only using the higher-latency, last-mile wireless network once. Proxy-based Web page rendering has existed in literature for more than a decade [17, 16, 20, 8, 9] and is of continued interest [10, 31]; however, it was not until the recent surge in smartphone popularity that commercial offerings became more widespread and well provisioned.

Cloud-based Web browsers (which we call "cloud browsers" for short) are often provisioned to exceed the computational power and functionality of a desktop browser. For example, CloudBrowse runs a modified version of the Firefox desktop browser [3]. Over the past decade, websites have evolved into full fledged applications executing nontrivial computations written in JavaScript. Cloud browsers must execute this JavaScript. Given this mix of powerful cloud-based computing ability and a substrate for general executions, we sought to investigate whether opportunities for exploiting unintended funcionality were now possible. Specifically, was it now possible to *perform arbitrary general-purpose computation within cloud-based browsers,* at no cost to the user?[1] A successful outcome would demonstrate the ability to perform *parasitic computing* [7] within the cloud environment, whereby the cloud is transformed into an unwitting computational resource merely through supplying browser requests.

In this paper, we explore the ability to use cloud browsers as *open computation centers*. To do this, we propose the Browser MapReduce (BMR—pronounced *beemer*) architecture, which is motivated by MapReduce [14], but contains distinct architectural differences. In our architecture, a master script running on a PC parameterizes and invokes *mapper* jobs in separate cloud browser rendering tasks. When complete, these workers save their state in free cloud storage facilities (e.g., provided by URL shortening services), and return a link to the storage location. The master script then spawns *reducer* jobs that retrieve the intermediate state and aggregate the mapper results.

To demonstrate the functionality of our cloud browser-based computational infrastructure, we implement three canon-

[1]Since JavaScript is a Turing-complete as it can be reduced to the lambda calculus [19], any comptuation is theoretically feasible.

ical MapReduce programs: *a*) word count, *b*) distributed grep, and *c*) distributed sorting. Compared to Amazon's Elastic MapReduce (EMR), BMR was faster for distributed grep, but several time slower for word count and distributed sort due to high communications costs. However, it does so at no monetary cost. Note that due to ethical considerations, we executed relatively small-scale computations in order to not overly tax the cloud browsers or the URL shortening services. As such our experiments only show a savings of only a few cents. However, larger jobs over longer periods of time can lead to substantial savings. Additionally, we explore the potential of BMR for performing parallelizable tasks that benefit from anonymity, e.g., cracking passwords. Attackers have already paid to use Amazon EC2 [2], and moving such activities to cloud browsers is likely.

This paper makes the following contributions:

- *We identify a source of free computation and characterize the limitations of four existing cloud browsers.* To our knowledge, we are the first to consider cloud-based Web browsers as a means of performing arbitrary computation.
- *We design and implement BMR, a MapReduce motivated architecture for performing large jobs within cloud browsers.* Cloud browser providers artificially limit computation to mitigate buggy Web pages. Using a MapReduce motivated architecture, we show how to coordinate resources in multiple cloud browser rendering tasks through the use of free storage made available by URL shortening services.
- *We port three existing sample MapReduce example applications to BMR and characterize their performance and monetary savings.* BMR has different limitations than traditional MapReduce (e.g., storage), and therefore must be optimized accordingly. We report on our experiences working within these limitations.

The remainder of this paper proceeds as follows. Section 2 overviews our architecture and lays out our design challenges. Section 3 characterizes the computation and memory limitations of several popular cloud browsers. Section 4 describes the BMR map and reduce primitives, scheduling, and the example applications. Section 5 discusses our implementation of BMR. Section 6 evaluates the performance of those examples in the Puffin browser. Section 7 discusses mitigations and optimizations. Section 8 overviews related work. Section 9 concludes.

2. APPROACH OVERVIEW

The goal of this paper is to explore methods of performing large computations within cloud-based Web browsers, ideally anonymously and at no monetary cost. While cloud browsers execute Javascript code, which is Turing-complete, we expect cloud browser providers to implement resource limits on Javascript execution. Therefore, we must divide our large job into smaller parts. MapReduce [14] has become a popular abstraction for executing large distributed computation. Therefore, we propose a MapReduce-motivated execution framework called *Browser MapReduce* (BMR).

To better understand BMR, we first describe MapReduce. A MapReduce job is implemented as two procedures: *Map* and *Reduce*. Execution always begins with the mapping

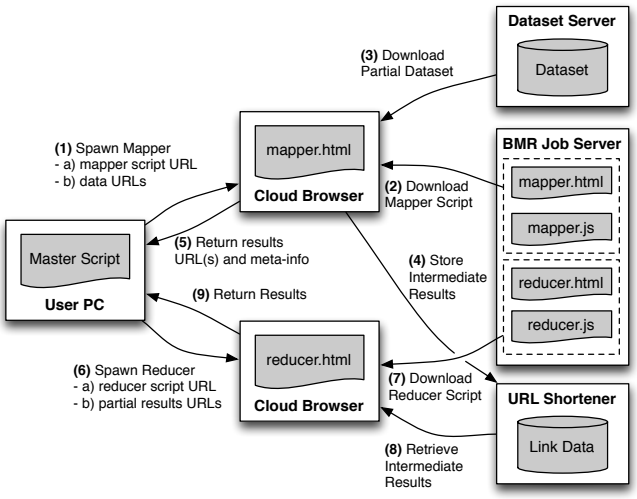

Figure 1: Browser MapReduce (BMR) Architecture

phase. A mapper extracts a set of key-value pairs of interest from each input record. For example, for a MapReduce job to count the number of words in a set of documents, the mapper determines the number of instances of each word in a small subset of the documents. Here, the word is the key and the number of instances is the value. The results of multiple mappers are then combined in the reducer phase. For word count, the reducer aggregates the word counts to produce an overall count for each word in the entire dataset.

In MapReduce, computational resources are abstracted as nodes within a cluster. Job coordination is performed by a *master* node. The master is responsible for handling communication synchronization, fault tolerance, and parallelization. Since a failure of the master node leads to a failed computation, the master node is often replicated. The remaining nodes in the cluster are worker nodes. A worker node can be a *mapper*, a *reducer*, or both. Note that a single MapReduce job consists of many mappers and reducers. To minimize communication overhead, the intermediate results generated by the mappers are stored locally and the locations are communicated to the master. By strategically partitioning the dataset, scheduling mapper and reducer jobs, and tracking these instances accordingly, the MapReduce framework can scale for both computationally intensive and large data processing applications.

Executing MapReduce operations within a cloud browser environment introduces several challenges:

- *Cloud browsers have artificial limitations.* Each cloud browser instance has artificial limitations placed on the amount of processing it can devote to a script, the size of memory allocated to that instance, and the time for which a script can execute on the browser safely without crashing the browser.
- *Job scheduling must account for the artificial limitations of the target cloud browser node* and *the expected compute time for the specific task.* Each MapReduce application requires different complexity for the mapper and reducer. The scheduler must partition the job based on this complexity and the limitations of the target cloud browser.

- *Mappers cannot use local storage to communicate intermediate results.* There is no guarantee that a reducer can be spawned in the same cloud browser instance as the mapper. Furthermore, cloud browser instances cannot communicate with one another. Therefore, we must identify an alternative (ideally free) storage location for intermediate results.

The BMR architecture, shown in Figure 1, is motivated by the MapReduce framework, but differs in certain aspects. First, the master node runs on the user's PC. Since the master node is running on the user's PC, we can assume it is reliable. However, we must also assume it has limited bandwidth. Both the Map and Reduce functions are written in Javascript and retrieved from the BMR job server hosting the application. To reduce the bandwidth requirements on the user PC, we assume that the dataset is statically served from a publicly accessible web server.[2] Due to the same-origin policy (SOP), if the data server is different than the script server, a CORS policy [30] must be set appropriately. Again, to reduce user PC bandwidth requirements, intermediate results are not returned directly to the master script. BMR uses a URL shortener service (e.g., `bit.ly`) for free cloud storage (see below). These URLs are then returned to the master script along with meta-information to aid reducer scheduling (discussed further in Section 4). Once the mappers complete, reducer scripts are similarly spawned and the final results returned. Note that the final results can be another set of URLs, or the data itself, depending on storage capacity limitations in the cloud browser instance.

As noted above, BMR requires cloud storage for intermediate results. We considered many alternatives for storing this data, and decided on a URL shortening service, as it is both free and semi-anonymous (an account is required, but doing so only requires a valid email address). Fundamentally, URL shortening services provide key-value storage, where a shortened URL returns a long data string, which in our experiments can be up to 2022 characters. Other options we considered included the simple strategy of returning the intermediate result data directly to the mapper script; however this requires user PC bandwidth. Another option is to store the intermediate results back to the dataset server. However, we wanted to decouple dataset storage from intermediate results storage for several reasons, e.g., read-only content is easier/cheaper to host. A third option was to pay for cloud storage (e.g., Amazon S3 [6]); however, computation would no longer be free.

3. BROWSER RESOURCE LIMITATIONS

BMR executes jobs (i.e., mappers and reducers) as rendering tasks for a cloud browser. In practice, cloud browser providers limit the resources provided to each rendering task to limit the consumption of buggy JavaScript. In order to optimally partition the input data and schedule workers, BMR must take into account the limitations of the target cloud browser. For each of the studied cloud browsers, we characterized the JavaScript capabilities; Flash and Java applets were not supported and therefore not characterized.

In the following discussion, we consider CPU cycles, elapsed execution time, and memory consumption. We also considered persistent storage provided by HTML5; however, we found it to be substantially lower than the RAM available to a cloud browser instance.

[2]If authentication is desired, authentication tokens can be passed to the mapper and reducer scripts.

```
function cpu_benchmark() {
  for(i=0; i<n; i++) {
    if(i%m == 0) {
      document.getElementById
          ("var").innerHTML = "Reached "+(i);
    }
  }
}
```

Figure 2: Computation benchmark

```
var time=0;
function time_benchmark() {
  document.getElementById
      ("var").innerHTML = "Time: "+time;
  time = time + n;
  var x = setTimeout(time_benchmark,n*1000);
}
```

Figure 3: Elapsed time benchmark

3.1 Benchmarks

We use simple JavaScript functions to benchmark the cloud browser capabilities. Each benchmark is designed to isolate a specific characteristic. Our benchmarking procedure has two stages. First, we use a small reporting interval to incrementally discover the limit. Then, we confirm that the interval reporting does not affect the results by specifying a report interval just below the determined limit. For example, the cloud browser might terminate a process if it is unresponsive, and updating the display indicates activity.

3.1.1 Computation

The CPU resource limitations configured by the cloud browser provider impacts how much data should be allocated to each worker. To measure CPU capacity, we perform a tight loop and report the maximum number of iterations reached. While this is not a direct measurement of CPU cycles, it provides an approximation for comparison and scheduling parameterization.

Figure 2 shows our CPU cycle benchmark JavaScript function. The function assumes a global variables `n` and `m`. `n` is a positive integer (e.g., 1 billion) specifying the number of iterations to perform. `m` specifies the "printing" interval. As mentioned above, the interval reporting helps to identify the progress before the browser crashes. After a limit is determined, `m` is increased to the maximum reached value to ensure that the act of printing does not extend the computation limit. Finally, we note that printing a value takes CPU cycles itself. While changing `m` can affect the maximum number of iterations performed, our results are conservative.

3.1.2 Elapsed Time

While CPU cycles is a likely resource limit, a cloud browser provider might also place a limit on wall-clock time. Since the BMR architecture requires the worker to download and upload data to network servers, the scheduling algorithm must take into account limitations on elapsed execution time. To characterize execution time limits, we perform the JavaScript equivalent of `sleep()`.

Figure 3 shows our elapsed time benchmark JavaScript function. Here, we use `time_benchmark()` as a callback

```
function memory_benchmark() {
  var arr=new Array();
  for(i=0; i<n; i++) {
    arr.push(i);
    if (i%m == 0) {
      document.getElementById
          ("var").innerHTML="Reached " + i;
    }
  }
}
```

Figure 4: Memory consumption benchmark

function passed to JavaScript's built-in `setTimeout()` timer function. Note that `time_benchmark()` is not recursive. Rather, the function returns, and the JavaScript runtime will call it again after the specified delay. Finally, the benchmark is parameterized by a global variable `n` indicating the number of seconds to sleep on each iteration. The benchmark keeps track of the total time and reports it on each interval. For testing, we began with small `n` (e.g., 1 second) and repeated times with a much larger `n` (e.g., 1 hour). Note that since some cloud browsers continued for hours without termination (see Section 3.3), we did not confirm a final maximum execution time.

3.1.3 Memory

The last resource limitation we characterize is working memory (i.e., RAM). Since BMR jobs operate on partial datasets, the scheduling algorithm needs to account for resident memory capacity when partitioning the dataset. Note that we also considered persistent storage (e.g., HTML5's LocalStorage); however, the W3C specification[3] indicates an arbitrary limit of 5MB per origin, which is significantly less then the memory capacities we report in Section 3.3. We verified the small persistent storage capacity on each of our target cloud browsers using a publicly available benchmark.[4]

Figure 4 shows our memory benchmark JavaScript function. This benchmark simply continues to append values to a dynamically allocated array. Similar to the CPU benchmark, the memory benchmark is parameterized by global variables `n` and `m` that determine the maximum number of iterations, and the reporting interval, respectively. Note that `n` also defines the upperbound on memory allocation. Since `arr` is an integer array, the memory capacity in bytes is $\texttt{i} \times 4$.

3.2 Studied Cloud Browsers

We analyzed the resource limits of the following commercially available cloud browsers. When selecting cloud browsers for BMR, one must ensure that the framework is capable of executing JavaScript on the server. For example, even though Opera Mobile uses Opera Turbo, we found that Web pages were compressed and rendered locally. This was also the case for the Android version of UC Browser [29].

Amazon Silk: The Amazon Silk browser [5] is exclusively available for Amazon's Kindle Fire tablet. Every time the user loads a web page, Silk dynamically decides whether rendering should occur on the device or on Amazon Web Services. Due to the dynamic nature of Amazon Silk's rendering decision, we were unable to confirm without doubt that the JavaScript was executed on the server for our experiments. However, we did observe the device communicating with Amazon continually through each experiment.

Cloud Browse: The Cloud Browse browser [3] is developed by AlwaysOn Technologies. It hosts a Firefox browser session on cloud servers and relays the rendered page to the mobile device. Cloud Browse currently only exists for iOS, but the Android version is expected. Since Firefox runs on the server, and based on observing continuous communication with Amazon EC2 servers throughout each experiment, we conclude that JavaScript executes on the server.

Opera Mini: The Opera Mini browser [24] is designed specifically for mobile environments with limited memory and processing power. It uses the Opera Turbo technology for faster rendering and compression of web pages. Note that the Opera Mobile browser also allows the user to enable Opera Turbo; however, in our experiments, enabling Opera Turbo only added compression and did not appear to render the content on the server. In contrast, the Opera Mini experiments where highly indicative of JavaScript execution on the server. The browser communicated to the server throughout the experiment. Furthermore, when the computation limit was exceeded, Opera Mini navigated to a Web page with the URL "`b:D8EAD704Processing`." This page had the title "Internal server error" and indicated "Failed to transcode URL" within the main window.

Puffin: The Puffin browser [12], developed by CloudMosa Inc., is designed for mobile devices and is advertised as rendering web pages in the cloud. It is available for both Android and iOS. The developer indicates that the browser does not store users' private data (e.g., cookies and history) on the cloud servers. All communication between the client and cloud browser is encrypted via SSL. During our experiments, we observed continuous communication between the client and cloud browser servers. Server side JavaScript execution is further confirmed by our implementation of BMR using Puffin in the latter sections of this paper.

3.3 Benchmark Results

We experimentally determined a conservative lower bound on the computation, time, and memory limits for each of the four cloud browsers. Amazon Silk was tested using version 1.0.22.7_10013310 on a Kindle Fire tablet. Cloud Browse was tested using version 4.2.1 (33) on an iPhone 3G running iOS v. 5.1 (9B176). Opera Mini was tested using v. 7.0.3 on a Samsung Galaxy Nexus running Android v. 4.0.2. Finally, we tested Puffin v. 2.2.5065 on a Samsung Galaxy Nexus running Android v. 4.0.2. The experiments for the reported results were performed in late May 2012.

Each browser has different failure modes, which lead to different strategies for determining the reported limit. Recall that the computation and memory benchmarks are parameterized by global variables `n` and `m`. Ideally, when the browser reaches its limit, it crashes in such a way that the greatest multiple of `m` reached is displayed. For Amazon Silk, a dialog box is shown, but the values on the Web page are still viewable. For Cloud Browse and Puffin, the computation simply stalls when the limit is reached, and no error message is reported. Finally, as described above, Opera Mini redirect to a server error page. In this case, we resorted to using a binary search strategy to find the limit by adjusting `n` and keeping `m = n - 1`.

[3] `http://dev.w3.org/html5/webstorage/#disk-space`
[4] `http://arty.name/localstorage.html`

Table 1: Cloud browser benchmark results

Browser	Computation		Elapsed	Memory	
	Iterations	≈ Time	Time	Array Size	Data Size
Amazon Silk	140,000,000	30 secs	24 hrs*	16,000,000	61 MB
Opera Mini	50,000,000	7 secs	6 secs	33,525,000	128 MB
Cloud Browse	40,000,000,000	1 hr	24 hrs*	121,000,000	462 MB
Puffin	200,000,000,000	2 hrs	24 hrs*	58,850,000	224 MB

* The benchmark was terminated after 24 hours.

Computation: Table 1 shows both the number of iterations of the `for` loop that can be computed before the cloud browser fails. Our tests increase loops in increments of 1,000,000. We confirm the calculated value by setting `n` to that value and re-running the experiment 20 times. Table 1 also reports an approximate time for each experiment. The time varied per execution, but not significantly with respect to the listed duration.

Elapsed Time: The next column in Table 1 shows elapsed execution time when performing negligible computation (i.e., using `setTimeout()`). We let Amazon Silk, Cloud Browse, and Puffin run for 24 hours before terminating the experiment. Clearly, these browsers do not have a limit purely based on wall-clock time. However, Opera Mini consistently terminated after exactly six seconds. This is notable since computation benchmark experiments consistently exceed six seconds, indicating that Opera Mini terminates JavaScript execution if it is not performing computation.

Memory: The final two columns of Table 1 show the results of the memory benchmark. Using the appropriate search strategy, we discovered the reported value in increments of 500,000. Similar to the computation benchmark, we validated the value by re-executing the experiment a total of 20 times. For Opera Mini, we initially determined 34,000,000 array elements; however, after a few re-executions, this value failed. Reducing the limit to 33,500,000 for the subsequent runs succeeded. The listed value is the average of 20 trials. The memory for Puffin varied more greatly. When the memory limit of Puffin is exceeded, the display shows the last `m` value reached. Since updating the page does not affect Puffin runtime, we executed the experiments with `m` set to 500,000. The array size ranged from 50,000,000 to 66,500,000. We report the average in Table 1. This ranges indicates a dependence on either load on the server, or individual servers configured with different memory resource limits. Finally, the last column in the table simply converts the array size to bytes, assuming each array element occupies four bytes, and rounds to the nearest MB.

Summary: Based on our experiments, Cloud Browse and Puffin provide significantly more computational ability than Amazon Silk and Opera Mini; each provide at least an hour of computation time. While Cloud Browse provides more RAM, the difference turns out to be inconsequential. As discussed in Section 2, BMR uses a URL shortening service for communication between workers. For this paper, we use the popular `bit.ly` service. We experimentally determined that `bit.ly` can encode long URLs up to 2022 characters in length but rate-limits requests to 99 per IP address per minute. These limitations on communication size limitation makes BMR best suited for CPU-intensive tasks, as opposed to storage heavy tasks. As Puffin provides a higher computation limit than Cloud Browse, we chose it for our proof-of-concept BMR implementation.

4. DESIGNING AND SCHEDULING JOBS

In the previous section, we characterized the resource limitations of four cloud browsers and found that two of the cloud browsers provide a significant amount of processing capability. In this section, we show how one can break up larger jobs to run within executions of JavaScript (i.e., requests for Web pages) on cloud browsers. Our experiments led to two guiding principles: 1) optimize jobs for higher worker computation loads; and 2) limit worker communication data size where possible. This leads to different optimizations than one might find in MapReduce. For example, a BMR mapper can be more complex if it reduces the data communication size.

To understand BMR's ability to provide MapReduce functionality, we explore three canonical applications:

- *Word count:* Counts the number of instances of each word in a set of input files. The output has an entry for each word and the number of times it appears.
- *Distributed grep:* Finds instances of an regular expression pattern in a set of input files. The output is the matching lines. In the case of BMR, we output the file and line number of each match.
- *Distributed sort:* TeraSort [23] is a popular benchmark for MapReduce frameworks that implements a simplified bucket sort in a distributed fashion. We use input from the `teragen` application, which provides 98 character records containing 10 characters for the key and 88 characters for the value. Our BMR bucket sort application outputs the sorted keys and the file number in which the records originated.

By first looking at how map and reduce abstractions are designed and jobs scheduled, we will better demonstrate how these applications are implemented within BMR.

4.1 Map and Reduce Abstraction

Both the mapper and reducer jobs are implemented as Web pages that include JavaScript to perform the desired functionality. Each BMR application consists of `mapper.html` and `reducer.html` Web pages that include `mapper.js` and `reducer.js`, respectively.

Figure 5 depicts the BMR mapper abstraction. The mapper execution begins with the master script using the cloud browser to visit the URL of the mapper Web page (`http://bmr_server/mapper.html`). BMR assumes that the input for the overall MapReduce application is separated into many small text files. When each mapper job is started, the mapper URL includes HTTP GET parameters specifying private data `pdat` (e.g., range parameters for dividing data into URLs) and an array of the URLs of the input

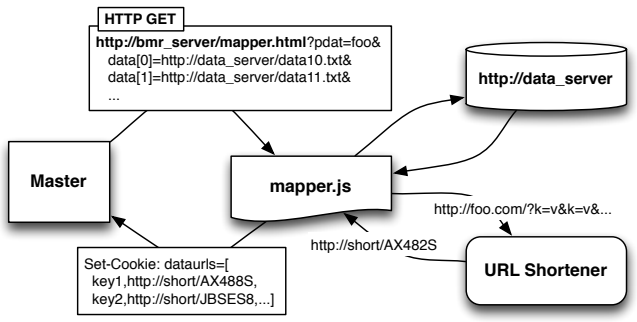

Figure 5: BMR mapper abstraction

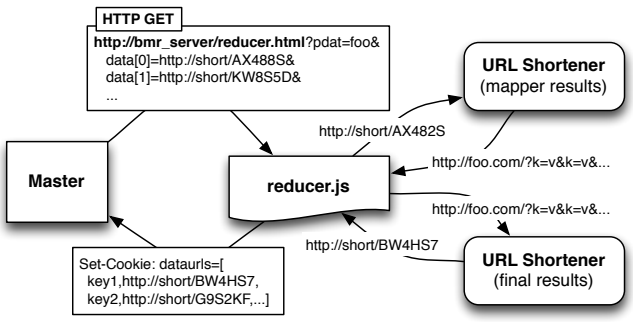

Figure 6: BMR reducer abstraction

data for that worker. The mapper logic parses these HTTP GET parameters, downloads the corresponding files using `XMLHttpRequest`, and performs the desired map operation.

The mapper job communicates the intermediate map results to the master script using a URL shortening service (e.g., `bit.ly`). The map results are a set of key-value pairs. The BMR mapper encodes the key-value pairs as a "long URL", e.g., `http://foo.com/?k1=v1&k2=v2`. The long URL is sent to the URL shortening service in exchange for a short URL, e.g., `http://short/AX482S`. Because shorteners limit the number of characters in a long URL, the BMR mapper stores results across multiple URLs. Furthermore, the mapper might designate URLs as belonging to different partitions or buckets to help the master schedule reducers. Finally, the set of short URLs is returned to the master script along with a key value for each URL. The means of communicating the short URLs back to the master script varies per cloud browser platform. For the Puffin platform, we use the browser cookie field. More details of this design choice are discussed in Section 5.

Figure 6 depicts the BMR reducer abstraction. The BMR reducer is very similar to the BMR mapper. However, instead of retrieving its input from a data server, the inputs to the reducer are shortened URLs that are expanded to obtain the input data. In Figure 6, the final results are encoded as another set of long URLs. If the final results are small enough, they could easily be returned as cookie data.

Cross-Origin Requests: Both the mapper and reducer scripts perform `XMLHttpRequest` operations to retrieve and store data. JavaScript executing in a Web browser is subject to the *same-origin policy* (SOP), which prevents the script from making network requests to any origin (i.e., domain) except for the origin from which the JavaScript was downloaded. For example, if `mapper.js` is downloaded from `foo.com`, it cannot retrieve data from `bar.com`. This restriction can be overcome using cross-origin resource sharing (CORS) [30]. To allow `mapper.js` to request data from another origin, the Web server hosting the data must modify its HTTP headers to either explicitly allow scripts from the domain hosting `mapper.js`, or allow any domain (i.e., `*`). Note that `bit.ly` uses CORS to allow JavaScript running on any domain to perform network requests to its API.

4.2 Scheduling Jobs

Just as in MapReduce, BMR executes a mapper phase followed by a reducer phase. To effectively use cloud browser and URL shortening service resources, the master script must carefully partition the job. There are an arbitrary number of complex heuristics that one can use to tweak and optimize the master script scheduling. However, the goal of this paper is to explore how to perform computations within cloud browsers. Therefore, we assume: a) the input is divided into a large number of equally sized files that are accessible to the mapper and reducer JavaScript; and b) the following constants are specified by the BMR user:

f_s:	size of each input file (in bytes)
f_n:	number of input files
b_s:	maximum data size for cloud browser (in bytes)
u_s:	size of data in a shortened URL (in characters)
u_n:	number of shortened URLs a worker can create
α_m :	mapper compression factor for the BRM application ($\alpha_m > 0$)

Note that b_s must be empirically derived for the target cloud browser and the target BMR application; u_s is specific to both the BMR application and the URL shortening service; and α_m defines the compression factor from input file size to the output of the mapper (discussed below).

Mapper Scheduling: In the mapping phase, the master determines 1) the number of mappers to spawn, M_n, and 2) the number of input files to pass to each mapper, M_f.

Cloud browsers are limited in the amount of memory allotted to the worker. Scheduling must account for both the memory required to load the input data and the internal data structures to perform the processing. Because the total amount of required memory required by the worker is dependent on the specific BMR application, the BMR user must empirically determine b_s. We assume the input files are several times smaller than b_s (e.g., f_s is 2-5 MB).

As previously discussed, the key limiting factor is the number of shortened URLs that must be created. If the number of shortened URLs did not matter, the mapper scheduling is straightforward:

$$M_f = \left\lfloor \frac{b_s}{f_s} \right\rfloor \quad (1) \qquad M_n = \left\lceil \frac{f_n}{M_f} \right\rceil \quad (2)$$

However, as described above, URL shortening services such as `bit.ly` use rate limiting. It is therefore to our advantage to minimize the number of shortened URLs, since they are long-lived and and take up part of the URL shortening service namespace. Recall that `bit.ly` can store 2022 characters per URL ($u_s = 2022$) and is limited to 99 URLs per minute. To avoid unnecessary delay, the BMR user can set $u_n = 99$; however, if multiple minutes wait in the mapper can be tolerated, u_n can be set higher.

Note that when choosing u_n, the BMR user must ensure that the mapper can transmit all of the shortened URLs

back to the master. Our proof-of-concept implementation of BMR using Puffin (Section 5) returns the shortened URLs in a cookie value. Puffin allows a maximum of 4053 characters in the cookie, and the identifying portion of `bit.ly` links is 10 characters, therefore u_n should be less than 400. Depending on the size of the key, even less links should be used. In the event that more links per worker are required, a tree structure of shortened links can be created; however, this results in extra processing.

Each BMR application has a different mapper compression factor α_m specified by the user. Typically, $\alpha_m > 1$, indicating that the output data is smaller than the input data. For example, our mapper for word count (described in Section 4.3) consolidates all instances of a word in the input text. We use $\alpha_m = 4.26$ for word count in Section 6. Using α_m, we redefine M_f as follows:

$$M_f = min\left(\left\lfloor \frac{b_s}{f_s} \right\rfloor, \left\lfloor \frac{\alpha_m \cdot u_n \cdot u_s}{f_s} \right\rfloor\right) \quad (3)$$

This modified equation accounts for the URL shortening service storage limitation.

Reducer Scheduling: The scheduling for the reducer phase is application specific. As a generic abstraction, we assume that the mapper stores key-value pairs into shortened URLs based on some partitioning strategy. For example, in word count, a partition is a range of characters, and in distributed sort, it is a bucket used in the bucket sort algorithm. When the mapper returns the set of URLs, it specifies a key for each URL. The master script schedules a reducer for each key, passing it all URLs corresponding to that key. Note that the number of keys or partition definition is passed as the private data (`pdat`) to the mapper and also affects the number of URLs used by the mapper (i.e., using partitions can lead to internal fragmentation).

4.3 Example Applications

To demonstrate the functional capability of BMR, we implement three canonical MapReduce applications: word count, distributed grep, and distributed sort.

Word Count: Word count determines how many times each word appears is a large set of text. This task lends itself well to the map and reduce abstraction. Traditionally, the word count map function parses part of the dataset, and for each word in the file, it prints, "`word: 1`". The reduce function tallies a count for each word in multiple files. Our BMR mapper behavior differs slightly. Since the BMR mapper must maintain the words in memory before "writing" them to the URL shortening service, it maintains a count for each word in the input files. We also include this reducer functionality within the BMR mapper to reduce the storage overhead of the intermediate results. To encode these results the BMR mapper creates a long URL similar to the following: `http://foo.com/?word1=5&word2=7&...`. As discussed in Section 4.2, the mapper partitions results into URLs to aid reducer scheduling. For simplicity, we use ranges of letters. For example, if three partitions are used, words are starting with a-h are in partition 1, i-p in partition 2, and q-z in partition 3. Note that multiple URLs will correspond to each partition.

Distributed Grep: Distributed grep performs a pattern match across many input files. In MapReduce, all of the work occurs in the mapper, and the reducer is simply the identity function. As such, our BMR implementation only requires a mapper; executing the reducer in a cloud browser provides negligible advantage. For distributed grep, the BMR mapper performs a pattern match on the input file. When a line i matches the pattern in file f, the mapper adds "`&f=i`" to the long URL. For example, if the mapper works on `bar1.txt` and `bar2.txt`, the resulting long URL will be encoded similar to the following: `http://foo.com/?bar1.txt=45&bar1.txt=48&bar2.txt=34`.

Distributed Sort: The popular TeraSort framework implements a distributed bucket sort. The keyspace is divided into n buckets (where n is provided as the `pdat` private data passed to the BMR mapper). The mapper sorts the input into the n buckets, but does not care about the order within the bucket. In the reducer phase, each reducer is given a bucket to sort. Since the buckets are ordered, the total ordering is obtained. For our experiments, we use input data generated by the `teragen` program included in the Hadoop framework (version 0.19.0). `teragen` produces 98 character records. Each record consists of a 10 character key and an 88 character value. Our BMR mapper only encodes the keys of the records due to the limited storage in shortened URLs. We store both the key and the file number in which the key originated, e.g., `http://foo.com/?key1=file1&key2=file2key3=file3`. Including the file number allows the master script to easily recombine the key with the value in post processing. Note that we assume files are sequentially numbered, therefore we only need to store the file number and not the entire filename.

Of the three applications, distributed sort has the greatest storage requirements. For each record in the input data, we store a 10 character key and several digits for the file number. Furthermore, the keys created by `teragen` include several non-alphanumeric characters that must be URL encoded, thereby occupying additional space. As such, the BMR user must define the compression factor α_m and number of buckets n accordingly. In Section 6, we use $\alpha_m = 2.513$ and $n = \left\lceil \frac{\text{No. of records}}{5000} \right\rceil$. By ensuring that reducer only shortens 5000 keys, we prevent issues with the `bit.ly` service, which is rate-limited to generating 99 URLs per minute per IP address, corresponding to slightly over 5000 keys.

5. IMPLEMENTATION

To demonstrate our ability to leverage computational resources from cloud-based Web browsers, we needed to have the ability to send these browsers to the URLs we desired, such that our jobs would be properly executed. Based on its overall high performance as shown in Section 3, we focused our efforts on the Puffin browser. Below, we describe how we adapted Puffin to work within the BMR framework.

In order to use Puffin within BMR, we required an understanding of how Puffin sends and receives messages, necessitating knowledge of how messages are generated and their format. Puffin is an Android application, so our first goal was to examine its operation to attempt to determine its message format. We used the *ded* decompiler [15] to convert the `.dex` file within the Android package into readable code. This allowed us to reconstruct the program flow, which we followed into the `libpuffin.so` library. From there, we used IDA Pro to dissassembled the ARM binary.

Puffin transmits its messages using SSL, thus simply intercepting messages using tools such as *wireshark* would not

be successful in allowing us to understand their format. To intercept traffic in the clear, we modified `libpuffin` to disable SSL certificate checking by inverting the logic for error check at the time of certificate validation. This allowed us to man-in-the-middle the connection with ease.

We wrote a parser after decompressing the cleartext in order to reverse the framing protocols. Individual messages form channels, which add flow semantics that make differing use of packed serialized objects and data dictionaries depending on the type of channel created. As an example, browsing a website leads to creation of a channel with a packed name-value pair object with name `service_id` and value *view*. Accessing cookie data registers a channel with `service_id` storing a value of *cookie_store*. Other channels are used for activities such as video streaming. Data directories are used with view channels to store additional information such as URLs and binary objects.

Puffin used an unusual encoding scheme with internal functionals called *Q_encode* and *Q_decode*, though they have no relation to the standard Q-encoding scheme. The encoding appears to be an obfuscation measure which creates data larger than its corresponding plaintext. Characters are rotated deterministically and a counter added to their value before converting them into their hex representation, which is stored in ASCII. A checksum is included in a footer, likely to detect request tampering. Data is returned in a prerendered format from the Puffin servers.

Using this information that we gleaned through our analysis of the Puffin client, we wrote our own client that implemented the functionality required for connecting to the service. Our *Lundi* client creates channels for devices to connect to, a cookie store, and views for any URLs present. Because data is returned rendered as an image, we cannot scrape the page, but we do set cookies for operations that we perform and use those received cookies as intermediate data stores which can be stored as `bit.ly` links. The *Lundi* client is compact, written in under 900 lines of Python.

6. EVALUATION

In this section, we empirically evaluate the performance of the BMR system presented in the previous sections. We begin by describing our experimental setup. We then present a profile of the BMR results and a comparison to Amazon's Elastic MapReduce (EMR) and Hadoop on Amazon EC2.

6.1 Experimental Setup

To evaluate BMR with the word count, distributed grep, and distributed sort applications described in Section 4.3, we implemented a `mapper.js` and `reducer.js` library for each applications. The libraries consisted over over 1,000 lines of JavaScript. Both the JavaScript libraries and input data were hosted on the same Apache server.

For each application, we performed tests on input data of sizes 1 MB, 10 MB and 100 MB. The input data was partitioned in multiple files, which varied per application. For word count, we downloaded the top 100 most downloaded books from `www.gutenberg.com/ebooks/`. To obtain 100 MB input, we downloaded additional books from the Top 100 most downloaded authors. For the experiments, the book text was concatenated and split into files of 1 MB in size. For distributed grep, we downloaded 140 MB of IRC logs for the *#debian* channel on freenode.net, splitting the input into files of size 10 MB. We grepped for a specific user entering and exiting the channel. Finally, for distributed sort, we used the Hadoop `teragen` program to create random records. Due to BMR's limitation on `bit.ly` URL creation, we split the input data into 0.5 MB files.

Table 2: Profile of BMR on Example Applications

Experiment*	# M	U/M	# R	U/R	Time
W.C. 1 MB	1	64	8	9.625	164.633s
W.C. 10 MB	10	24.7	8	17	178.859s
W.C. 100 MB	100	17.66	8	39	899.003
D.G. 1 MB	1	1	-	-	17.701s
D.G. 10 MB	1	1	-	-	18.680s
D.G. 100 MB	8	1	-	-	26.596s
D.S. 1 MB	2	56	2	43	61.040s
D.S. 10 MB	20	58.8	20	42.05	279.390s

* W.C. = Word Count; D.G. = Distributed Grep; D.S. = Distributed Sort; M = Mappers; R = Reducers; U = URLs; U/M and U/R are averages.

As discussed in Section 4.2, the BMR user must specify a cloud browser data size, b_s, and a compression factor α_m. For all applications, we found $b_s = 1MB$ to be sufficient. For the word count α_m, we selected 10 books at random and performed word count to determine the reduction in text size. We used the average reduction: $\alpha_m = 4.2673$. For distributed grep, α_m is intuitively very large, since the user entering and exiting events are only a small portion of the IRC logs. Furthermore, BMR need only store the input file and line number. Therefore, the α_m parameter in Equation 3 will not be used. We conservatively set $\alpha_m = 1000$. Finally, for distributed sort, we calculated a conservative upper bound on data compression. For each 98 character record in the input, the distributed sort mapper must store 39 characters: *a*) a 10 character key, which may be expanded to as large as 30 characters due to URI encoding; *b*) three characters to URI encode the "="; *c*) three characters to store the file number; and *d*) three characters to URI encode the "&" character that separates key-value pairs. Therefore, we used $\alpha_m = 2.513$.

We execute each mapper and reducer in a separate worker nodes (i.e., cloud browser server), up to 20 simultaneous nodes, after which mappers wait for an open worker node. To prevent rate limiting from the back-end Puffin render servers, we sleep for 5 seconds between launching map instances on new nodes. Finally, to address the rate limitations on `bit.ly` URLs during our experiments, mappers and reducers randomly choose from a pool of 72 `bit.ly` accounts.

6.2 Results

BMR Results: Table 2 enumerates the results of the BMR experiments for the three applications, listing the number of mappers and reducers needed, as well as the average number of URLs needed per mapper and reducer. Note that we only performed each experiment once as to limit the total number of shortened URLs created; running the experiments multiple times does not change the distribution of data into mappers and reducers. Due to the compression factor, distributed grep passed the smallest amount of data, and therefore required both the smallest number of mappers and reducers and completed the quickest. As one might expect, the URL based communication between workers is a significant bottleneck.

Finally, the distributed sort 100 MB experiment tested the limits of our data passing methods. To meet the `bit.ly`

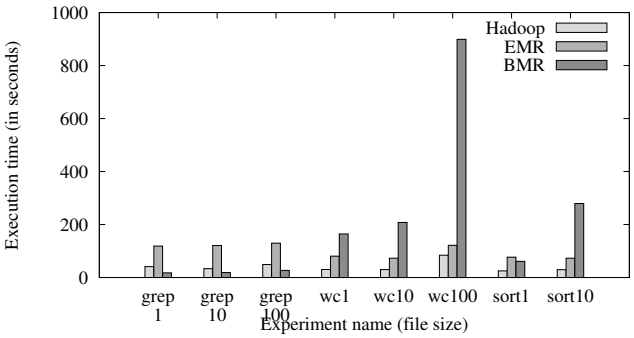

Figure 7: Comparison to Hadoop and EMR

rate limitation for reducers, we configured 200 buckets ($n = 200$). Returning 200 `bit.ly` links from the mapper exceeded the maximum cookie size. Therefore, alternative techniques such as a tree of `bit.ly` links are needed, as described in Section 4.2.

Comparison to MapReduce: To provide an intuition of the performance of BMR, we executed the same jobs on both Amazon's Elastic MapReduce (EMR) and Hadoop running on Amazon EC2. These experiments were performed on a cluster of 11 32-bit m1.small instances, allocating 1 instance for the master and using the other 10 as workers. Each m1.small instance was allocated 1.7 GB RAM, 1 EC2 Compute Unit, and 160GB storage. For EMR, data was stored in an Amazon S3 bucket, and for Hadoop, it was stored using Hadoop DFS. We used the Hadoop 0.19.0 Java example applications. Finally, for EC2, each m1.small instance cost US$0.08 per hour, and for EMR it cost US$0.095 per hour.

Figure 7 shows the comparison between BMR, Hadoop, and EMR for our test data. For distributed grep, BMR surprisingly performed better than both Hadoop and EMR. This is likely due to the relatively small amount of communication that was required; only one mapper was needed for 1 MB and 10 MB. However, in the word count and distributed sort experiments where substantially more data was communicated between the map and reduce phase, BMR's performance suffered. However, one should note that BMR was not designed to outperform existing MapReduce frameworks. Given BMR's limitations, it performed rather well.

Finally, given our small experiments, the cost savings were small. Hadoop experiments individually cost less than US$0.03, and the EMR experiments peaked at just under US$0.04. However, when performed at much larger scale and over a long period of time, BMR can amount to significant savings.

7. DISCUSSION

Recommendations for Cloud Browser Providers: By rendering Web pages in the cloud, the providers of cloud browsers can become *open computation centers*, much in the same way that poorly configured mail servers become open relays. The example applications shown in this paper were an academic exercise targeted at demonstrating the capabilities of cloud browsers. There is great potential to abuse these services for other purposes. We ran a series of hashing operations on the BMR insfrastructure to determine how a password cracking implementation may be deployed and found with Puffin, 24,096 hashes could be generated per second, or 200 million per job. The infrastructure could be used for far more sinister purposes as well such as DoS and other amplification attacks, and pose ethical and possibly legal concerns. When deploying a cloud browser platform, providers should take care to place resource limitations on rendering tasks. As discussed in Section 3, two of the tested cloud browsers were capable of rendering for at least an hour. However, stricter resource limitations is not enough. A framework such as BMR can be used to link together rendering tasks into a larger computation. To mitigate such parallel jobs, providers should rate limit connections from mobile clients. The most primitive form of rate limiting is by IP address. However, NAT is used by some cellular providers, thereby making rate limitation by IP address impractical. As an alternative, users of cloud browsers should be required to create accounts, and rate limits should be placed on authenticated users. In our investigation of different cloub browsers, we observed that the Amazon Kindle Fire's Silk browser requires registration and sends a device-specific private key as part of its handshake protocol with the cloud-based renderers. Such a strategy is particularly helpful in mitigating the ability to clone instances. Additionally, existing techniques such as CAPTCHAs can limit the rate of creating new accounts.

Enhancing BMR: Our implementation of BMR was provided as a proof-of-concept. There are several aspects in which it could be improved. First, the use of `bit.ly` became an unintended bottleneck in the computation. We chose `bit.ly` due to its popularity as a URL shortening service as well as its easy to use APIs for creating short URLs. There are several alternative URL shortening services that could be substituted; however, these services likely have similar rate limits. Therefore, using a combination of URL shortening services may be more ideal. Furthermore, the use of shortened URLs may not be the best choice for applications that have a low compression factor (α_m). BMR simply needs some form of free key-value, cloud-based storage for communicating mapper results to the reducers. Services such as Pastbin (`pastbin.com`) can store significantly more than 2022 characters. However, account creation and rate limitation are still a concern. A second way BMR could be improved is scheduling. Our scheduling algorithms are relatively simple, and much more advanced scheduling strategies have been proposed for MapReduce [1]. Finally, BMR could be made to use multiple cloud browsers. Different cloud browsers have different benefits. For example, Puffin has more computational abilities than Cloud Browse, but Cloud Browse has more available memory. By using multiple cloud browsers, BMR could schedule mappers and reducers based on expected workloads.

8. RELATED WORK

Cloud computing creates a powerful new computing model but carries inherent threats and risks. Numerous studies [18, 11, 13, 26] have surveyed and examined security and privacy vulnerabilties in the cloud computing model from architectural, technical, and legal standpoints.

Ristenpart et al. [25] demonstrate that it is possible to map the internal cloud infrastructure and thus identify a particular VM of interest and spawn another VM as the co-

resident of the target to mount cross-VM-side-channel attacks. Somorovsky et al. [28] perform security analysis pertaining to the control interfaces of both Amazon and Eucalyptus clouds, determining they can be compromised using signature wrapping and XSS techniques. There have been multiple attempts to exploit these vulnerabilities. For example, Mulazzani et al. [22] successfully demonstrate an exploit of the popular Dropbox cloud service, analyzing the client and protocol to successfully test if a given file is present within Dropbox and consequently breaking confidentiality.

The large computation platform provided by cloud services such as Amazon's EC2 allows for large-scale password hashing and cracking. Moxie Marlinspike's CloudCracker service uses cloud services for cracking WPA passwords and other encryption types [21], while Amazon's GPU clusters have been used to crack 6-character passwords in under one hour [27]. Both of these services require payment to the cloud provider; BMR is the first service to show how free computation can be exploited through cloud browsing clients to peform arbitrary operations.

9. CONCLUSION

This paper investigated the ability to use cloud based Web browsers for computation. We designed and implemented a Browser MapReduce (BMR) architecture to tie together individual rendering tasks, then designed and executed three canonical MapReduce applications within our architecture. However, these example applications were simply an academic exercise to demonstrate the capabilities of cloud browsers, and form a preliminary investigation into a new way of performing parasitic computing. Based on our findings, we observe that the computational ability made freely available by cloud browsers allows for an *open compute center* that is valuable and warrants substantially more careful protection.

Acknowledgements

This work is supported in part by the National Science Foundation under award CNS-1118046.

10. REFERENCES

[1] A. Aboulnaga, Z. Wang, and Z. Y. Zhang. Packing the most onto your cloud. In *CloudDB*, Hong Kong, China, Nov. 2009.

[2] P. Alpeyev, J. Galante, and M. Yasu. Amazon.com Server Said to Have Been Used in Sony Attack. Bloomberg, May 2011.

[3] AlwaysOn Technoligies. Cloud Browse. `http://www.alwaysontechnologies.com/cloudbrowse/`.

[4] Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing/`. Accessed April 2012.

[5] Amazon Inc. Amazon Silk FAQ's. `http://www.amazon.com/gp/help/customer/display.html/?nodeId=200775440`.

[6] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[7] A.-L. Barabasi, V. W. Freeh, H. Jeong, and J. B. Brockman. Parasitic computing. *Nature*, 412:894–897, 30 August 2001.

[8] S. Bjork, L. E. Holmquist, J. Redstrom, et al. WEST: A Web Browser for Small Terminals. In *ACM Symp. User Interface Software and Technology (UIST)*, 1999.

[9] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd. Power Browser: Efficient Web Browsing for PDAs. In *ACM SIGCHI*, pages 430–437, 2000.

[10] Y. Chen, X. Xie, W.-Y. Ma, and H.-J. Zhang. Adapting Web Pages for Small-Screen Devices. *IEEE Internet Computing*, 9(1):50–56, 2005.

[11] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proc. ACM CCSW'09*, Chicago, IL„ 2009.

[12] CloudMosa. Introducing Puffin. http://www.cloudmosa.com.

[13] K. Dahbur, B. Mohammad, and A. B. Tarakji. A survey of risks, threats and vulnerabilities in cloud computing. In *Intl. Conf. Intelligent Semantic Web-Services and Applications*, pages 12:1–12:6, 2011.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symp.*, San Francisco, CA, USA, Aug. 2011.

[16] R. Floyd, B. Housel, and C. Tait. Mobile Web Access Using eNetwork Web Express. *IEEE Personal Communications*, 5(6), 1998.

[17] A. Fox, I. Goldberg, S. D. Gribble, et al. Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the 3Com PalmPilot. In *IFIP Middleware Conference*, 1998.

[18] B. Grobauer, T. Walloschek, and E. Stocker. Understanding cloud computing vulnerabilities. *IEEE Security and Privacy*, 9(2):50–57, Mar. 2011.

[19] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP '10: Proceedings of the European Conference on Object-Oriented Programming*, 2010.

[20] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. *IEEE Personal Communications*, 5(6), 1998.

[21] M. Marlinspike. Cloudcracker. `https://www.wpacracker.com`, 2012.

[22] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space. In *Proc. 20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[23] O. O'Malley. Terabyte sort on apache hadoop. `http://sortbenchmark.org/Yahoo-Hadoop.pdf`, May 2008.

[24] Opera mini & opera mobile browsers. `http://www.opera.com/mobile`.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[26] J. C. Roberts II and W. Al-Hamdani. Who can you trust in the cloud?: A review of security issues within cloud computing. In *Info. Sec. Curriculum Development Conf.*, Kennesaw, GA, 2011.

[27] T. Roth. Cracking Passwords In The Cloud: Amazon's New EC2 GPU Instances. `http://stacksmashing.net/2010/11/15/cracking-in-the-cloud-amazons-new-ec2-gpu-instances/`, 2010.

[28] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *ACM CCSW'11*, 2011.

[29] UCWeb. UC Browser. `http://www.ucweb.com/English/UCbrowser/patent.html`.

[30] W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, Apr. 2012. WD 3.

[31] X. Xiao, Q. Luo, D. Hong, H. Fu, X. Xie, and W.-Y. Ma. Browsing on Small Displays by Transforming Web Pages into Hierarchically Structured Sub-PAGES. *ACM Trans. Web*, 3(1):4:1–4:36, Jan. 2009.