Abusing Cloud-based Browsers for Fun and Profit

Joe Pletcher, Ryan Snyder, Dr Kevin Butler – Computer and Information Science Department, University of Oregon

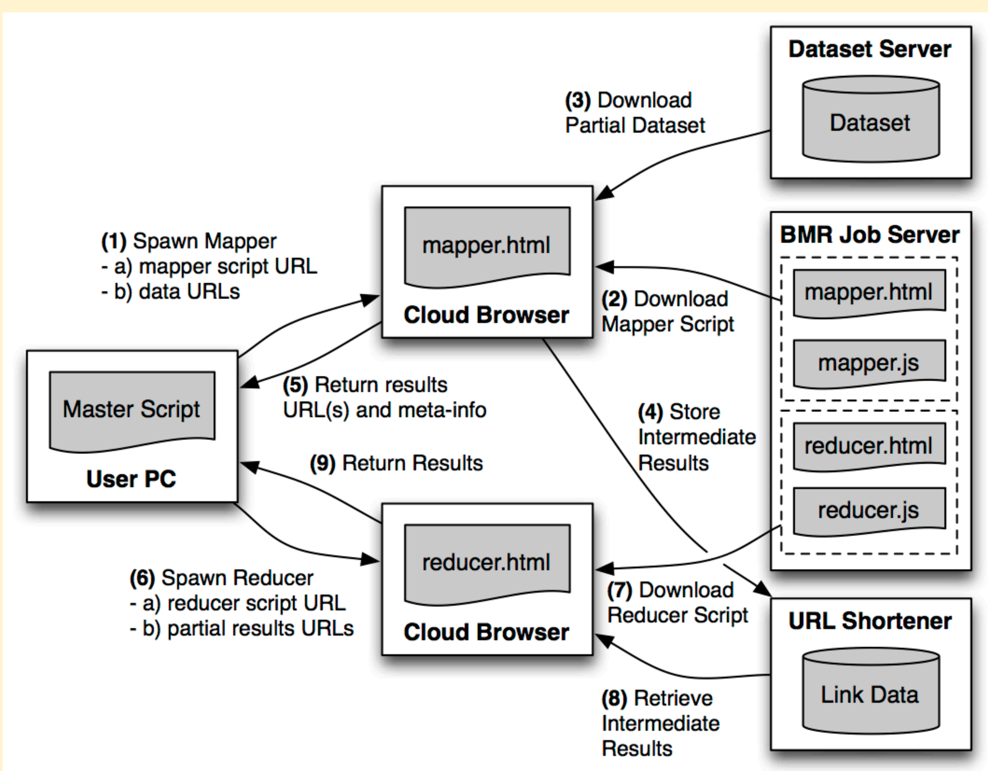
with Vasant Tendulkar, Ashwin Shashidharan, Will Enck (North Carolina State University)

Increasingly mobile devices are making use of off-device computation for performance improvements or savings in battery life. For example, mobile browsers can make use of off-device website rendering to achieve both these benefits. However, in this model a malicious client could exploit these remote resources for what is in effect free computation. However, many of these services have no serious forms of authentication and furthermore don't even identify individual clients. This combination makes it nearly impossible to filter malicious clients.

In this work, we reverse engineer a popular, cross platform cloud-based web browser (Puffin), and implement our own version, Lundi. Using Lundi we are able to implement Google's MapReduce algorithm on Puffin's servers. Using MapReduce we can compute arbitrary sized jobs on Puffin's servers, in some cases faster than commercial MapReduce offerings. We call our system Browser Map Reduce (BMR).

BMR Architecture

To implement MapReduce in the browser, we create Javascript programs which do the mapping and reducing respectively. These are served off a script host which we control. We launch browser instances



which fetch the data a user wants to process from a remote server. From there, the browser instances go to work, pirating computation on someone else's dime. When the mapper has finished its dataset, it stores the intermediate results in the parameters of a URL which we shorten, and pass back to us. Next, we fire up a reducer which fetches these intermediate values and reduces them using the same process as the mapper, returning the final values to the user.

Lundi Design

To reverse engineering Puffin, we needed the traffic between the Puffin client and Puffin's servers in the clear. Puffin uses TLS for end-to-end encryption, and the limited debugging capabilities of the Android platform made this a difficult task. We started by decompiling the Dalvik bytecode, and quickly realized it made all important calls through an included C library. By disassembling this library (libpuffin) we were able to begin to understand the workings of Puffin.

Patching Libpuffin

We found the SSL verification function inside Puffin and patched it to always verify the presented certificate.

```
SecureSocketStream::OnSslClientVerifyCallback
   EXPORT _ZN9cloudmosa18SecureSocketStream25
18SecureSocketStream25OnSslClientVerifyCallba
                              ; DATA XREF: cloud
                              ; cloudmosa::Secur
   PUSH    {R4-R6,LR}
   MOVS    R0, #1             ; Force failure co
   CMP     R0, #0
   BNE     skip_error_report
```

This allowed us to man-in-the-middle all Puffin traffic, and get the data in the clear. However, Puffin uses a binary protocol, and it is not immediately clear which messages do what.

Reversing Traffic

After decompressing the traffic and staring at it for hours, we were able to extract structure from the messages. Given enough time, messages like this make sense

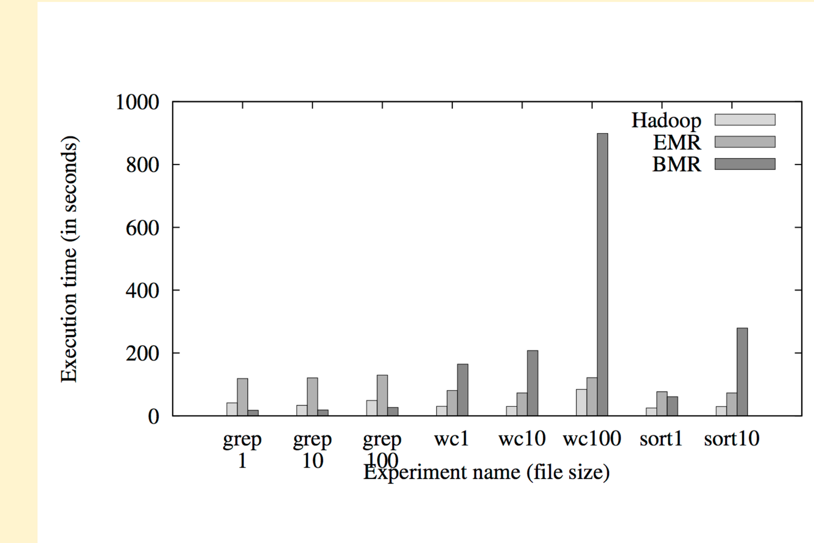
```
00000000  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
00000010  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
00000020  01 00 00 80 03 10 00 00  00 00 ff ff ff ff 02 e9  |................|
00000030  03 00 00 43 43 43 43 43  43 43 43 43 43 43 43 43  |...CCCCCCCCCCCCC|
00000040  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
00000050  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  |BBBBBBBBBBBBBBBB|
00000060  01 00 00 00 02 08 0b 06  35 33 34 2e 33 35 0b 0b  |........534.35..|
00000070  31 31 2e 30 2e 36 39 36  2e 36 35 0b 08 32 2e 30  |11.0.696.65..2.0|
00000080  2e 35 32 33 36 43 43 43  43 43 43 43 43 43 43 43  |.5236CCCCCCCCCCC|
00000090  03 00 00 00 02 00 01 00  00 00 22 56 00 00 01 00  |.........."V....|
000000a0  00 00 dc 01 00 00 28 00  00 00 43 43 43 43 43 43  |......(...CCCCCC|
000000b0  03 00 00 00 02 03 43 43  43 43 43 43 43 43 43 43  |......CCCCCCCCCC|
000000c0  02 00 00 80 03 10 00 00  00 00 ff ff ff ff 02 ea  |................|
000000d0  03 00 00 00 02 00 00 00  02 00 00 00 00 43 43 43  |.............CCC|
000000e0  03 00 00 80 03 10 00 00  00 00 ff ff ff ff 0b 05  |................|
000000f0  66 72 61 6d 65 00 02 00  00 00 02 04 00 00 00 00  |frame...........|
00000100  04 00 00 00 0e 10 00 00  00 00 00 00 00 00 e0 01  |................|
00000110  00 00 fa 02 00 00 00 05  00 00 00 0e 08 e0 01 00  |................|
00000120  00 fa 02 00 00 00 06 00  00 00 0e 08 00 00 00 00  |................|
00000130  00 00 00 00 00 08 00 00  00 02 02 00 00 80 43 43  |..............CC|
```

Using Cookies

Finally, Puffin's servers send down a video stream containing the requested page. Rather than perform OCR on the stream, we found that cookies are sent in plain text. Using a combination of traffic in the clear and cookies, we can roll our a functional Lundi.

EVALUATION

We tested our systems against two commerical MapReduce offerings, Amazon's Elastic Map Reduce (EMR) and Hadoop running on EC2. We see that in general our numbers are similar. We find that



problems which require high data transfer we perform worse, and problems with low data transfer we perform very well. Intuitively, this makes sense as data storage and retrieval adds full HTTP round trips to the system, and proportionally these make up a large fraction of the processing time. By removing these RTT's, our performance improves dramatically.

This indicates that MapReduce might not be the best fit given our constraints. On the other hand, it allows us to run jobs of arbitrary size, and provides easy job sharding. Alternately, we need to find faster persistent storage for browser instances. This is left as future work.

Mitigations

While fully preventing such attacks is impossible there are a few steps browser creators could take to mitigate the potential for attack. First, clients need to be uniquely identified, and second, the resources a single client can use should be capped. With these two changes in place, such as attack would likely not be worth pursuing.

If each client went through a registration process and were identified with a public-private keypair, then used this pair to sign requests, this would go a long way in preventing this attack. While this could still be spoofed, it would raise the barrier to entry and make the attack less desirable.

For further information contact Joe Pletcher (*pletcher@cs.uoregon.edu*)