Provably-Secure Traffic Analysis Defense Final Report

CSE 508

Chun Li (106459515), Raman Nanda (106171352)

Introduction

Throughout this project we hope accomplish the goal of developing a provable secure traffic analysis defense mechanism that will circumvent censorship by connecting to a proxy outside the control of their local authorities. For example countries like China have developed a system called the "Golden Shield Proxy", which serves as a firewall that analyzes and blocks traffic based on the content. Our initial approach to simulate this node-to-node communication was to take the already existing OpenSSH implementation and modify it to include certain new defense mechanisms against pattern based traffic analysis. The derived defense that we created is originally based on the BUFLO [(1)], however we had to make modifications to resolve some of the overhead issues that this system has. Ultimately, the system we created will be provable secure, that will allow us to maintain the same level of security regardless of any new attack strategies that will emerge in the future.

Development Environment

Virtual Environment

In order to keep the development environment uniform and convenient for everyone, we virtualized a Linux distribution known as Ubuntu in each of our machines. The Ubuntu community is widely popular and conveniently supports OpenSSH development. In order to communicate between each of the VMs we set our network preference to bridge our Host OS and the Visualized one to use the same physical device, but allow for multiple outgoing IP Addresses.

SSH Connection

We started our base code using the existing framework of OpenSSH 6.0, which conveniently was released on April 22, 2012. We were fortunately enough to find may resources that allowed us to set up the build environment and any additional libraries that were required in this process. In the end we ran the server with the command: "sshd -d" and the client with "ssh –D (PORT) -N -v ", these commands allowed us to view the current state of the applications as well as port forward the traffic for the client.

Web Browser Proxy

In order to test the usability of the proxy we were able to pipe the data arriving from the SSH connection to the browser using a built in SOCK5 Proxy. This is a standard preference in Firefox.

Traffic Analysis Tool

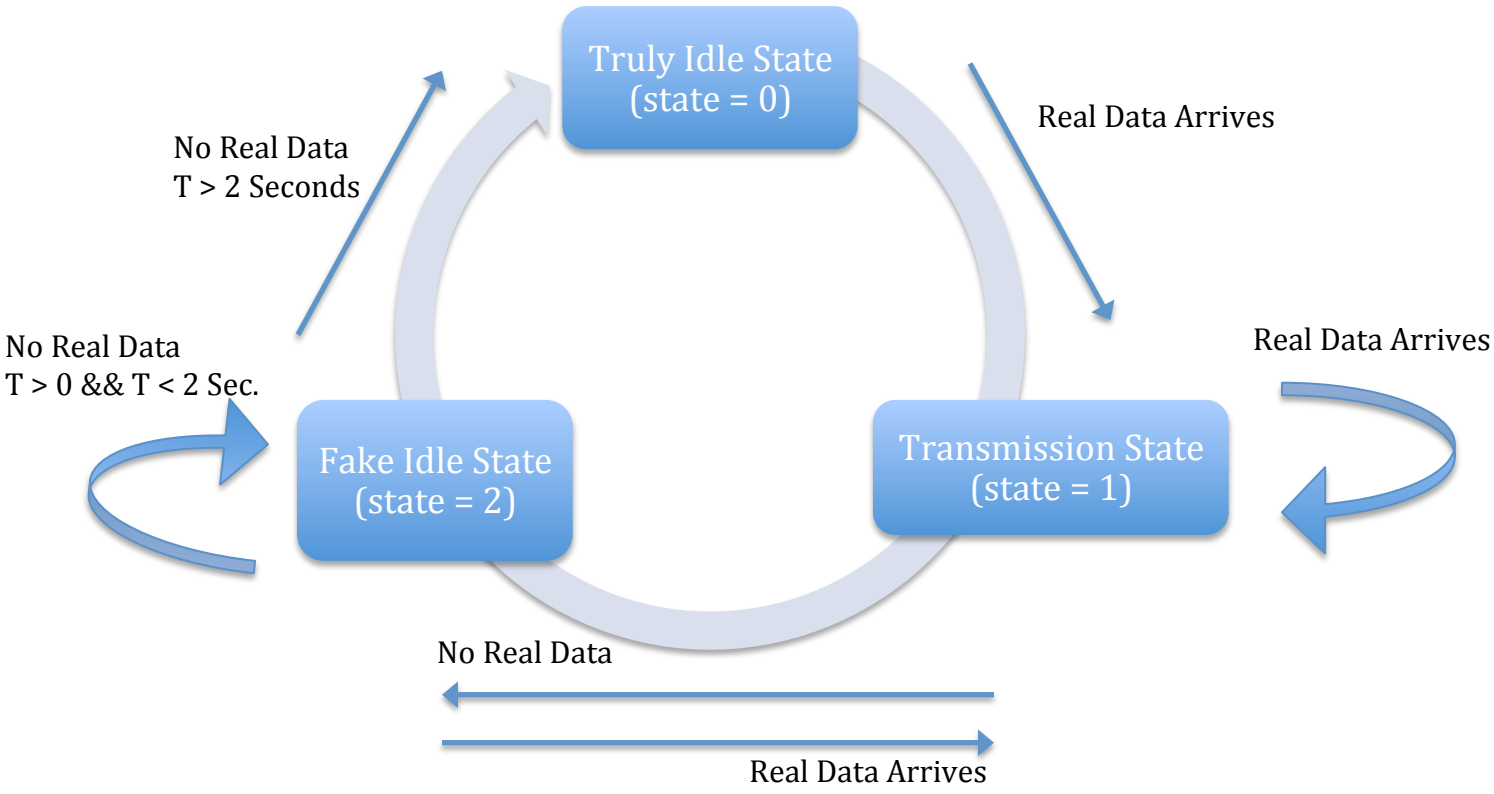
For the purpose of this project we were required to analysis the traffic size, latency and the randomness of our transmission times. In order to do this we used Wireshark, which proved to be a great tool due its ease of use and its configurable nature. Firebug, a Firefox extension, was also used as a tool to measure the latency the User experience, or in terms of our calculations: when the page's DOM loads.

Code Documentation

During development we understood that this project was broken into many different subtopics. We hope that each subsection here will give you some insight on how we solved the problems we encountered.

In the end we understood that the majority of our modification would reside in serverloop.c and clientloop.c. Since both sides needed to comply with the security goals we were trying to accomplish, we have to implement and mirror the same mechanisms. For the sake of clarity, we will only document the code in serverloop.c, it should be understood that the same concepts are applied to the client code.

State Transitions



Throughout the cycle of this application we realized that there are three main states: Truly Idle, Transmission and Fake Idle. The start of the application is always denoted as Truly Idle; this means that there is no new data arriving from the channel (Note: This is after the initial handshaking / key negotiation process). If we find there is real data arriving on the channel, the state changes to the Transmission State. Next when the data stops arriving, the state moves to Fake Idle. This state indicates a temporary silence in the transmissions between client and server. We used the time delimitation of 2 seconds to mark when the communication is silent, and the state completes its cycle back to Truly Idle.

In ordered to implement this process we had to monitor when the real data arrives in the channel. To achieve this, we modified packet_send() in packet.c so that it returns the size whenever an SSH packet is created. In turn we had to modify channel_out_poll() to return the real data size being polled.

At the moment we change from the transmission state to the fake idle state, we mark the current system time. Then, for every cycle in of the server loop, we check the marked time to see if we reached 2 seconds period. If during that time period, real data arrives; we go back to transmission state. If nothing arrives for 2 seconds, we then calculate the difference between total transmission size (including padding and junk data) and the next power of 2 and send that difference. If this process is completed and not interrupted by new data arriving, we transition back to the Truly Idle state.

Code: ServerLoop.c – void server_loop2(Authctxt *authctxt)

```
realdata = 0;
      realdata =channel_output_poll();
      totaldata += realdata;
      totaldata += junkdata;
      …
      if(realdata == 0) {
            if(state == 1) {
                  //transfer state
                  state = 2;
                  struct timeval currentTime;
                  gettimeofday(&currentTime, NULL);
                  fakeIdleStartTime = currentTime.tv_sec;

                  int next_power = pow(2,ceil(log(totaldata)/log(2)));
                  remaining_junk = next_power - totaldata;
            } else if (state == 2) {    //fake idle state
                  struct timeval currentTime;
                  gettimeofday(&currentTime, NULL);
                  time_t currentTimeSecs = currentTime.tv_sec;
                  double timeDiff = difftime(currentTimeSecs,
                        fakeIdleStartTime);
                  if(timeDiff < 2) {
                        // Continue Sending Padding Junk Packets
                  } else {
                        // Calculate Rounding Size Limit
                        int next_power =
                        pow(2,ceil(log(totaldata)/log(2)));
```

```
                        remaining_junk = next_power - totaldata;
                        if(state != 0)
                              debug("Remaining Junk Data: %d, Next
                              Power of 2: %d",remaining_junk,
                              next_power);
                        }
                  }
            else{
                  // truly idle state
                  remaining_junk =0;
                  fakeIdleStartTime =-1;
                  totaldata =0;
            }
      } else {
            state = 1;
            remaining_junk =0;
            fakeIdleStartTime =-1;
      }
```

Padding the Output Buffer

The project requires that the data we transmit should be independent from the data arriving from the user. For this reason, we have to make sure that the outgoing buffer will always have some data ready to be transmitted. The amount of data that pad at each cycle is the difference between 1330 and the current buffer size. The reason we chose the static number of 1330 was because when optimizing our code based on the Wireshark Output, we notice the majority of the packet are completed filled and don't overflow to smaller packets. By doing this, we can further hide the pattern of individual packet sizes.

*Code: ServerLoop.c – void server_loop2(Authctxt *authctxt)*

```
/**
*  Padding Modification
*/
if(state !=0)   //padding
{
      int sleeptime = rand() % randomPacketTime;
      /*sleep some random time before we write to sockets*/
      int junkdata =0;
      debug("---------------------- Sleeping For %d", sleeptime);
      usleep(sleeptime);
      total_sleep_time += (double)sleeptime;

      int buffer_size = packet_buffer_size();
      int buffer_limit=0;

      if(state== 2){
            // Fake Idle State - No Real Data to Pad
```

```
}
else{
      buffer_limit = 1330;
      if(buffer_limit-buffer_size>=0 && state!= 0){
            packet_send_ignore(buffer_limit-buffer_size);
            junkdata = packet_send();

            totaldata+=junkdata;
            t_totaldata+=junkdata;

      }
}
debug("------------- Total data Transmitted:
%d",totaldata);
} else{
      //trule idle no padding
}
```

Transmission Time

We want to randomize the transmission time so that there is no pattern of transmission of packets. In order to do this we keep a variable called "randomPacketTime" which is configurable based on the current throughput. The randomPacketTime is chosen from a set of times that range from 2^1 to 2^5 ms. During every loop cycle we generate a random sleep time based on the previous value.

In order to achieve better latency and low overhead, we choose to implement a dynamic configurable random packet time, using an alarm system call every 5 minutes. We used the pseudo code given to us in the project specifications to calculate the adjusting sleep time, based on the throughput of the last alarm. The implementation of this procedure is similar to the methods of TCP congestion control (Multiplicative Increase / Decrease). The reason we use this mechanism is to further hide the throughput information of the website since pattern matching can determine what website you are accessing based on the intervals. By changing the random time gradually, there will be weaker connection between the interval (randomPacketTime) and actual throughput since the user will finish loading the page long time before he/she reaches the true random T.

*Code: ServerLoop.c – void server_loop2(Authctxt *authctxt)*

```
int sleeptime = rand() % randomPacketTime;
/*sleep some random time before we write to sockets*/
int junkdata =0;
debug("---------------------- Sleeping For %d", sleeptime);
usleep(sleeptime);
total_sleep_time += (double)sleeptime;
```

Code: ServerLoop.c - static void handler(int sig)

```
signal(SIGALRM, SIG_IGN);

/* an estimation of throughput */
double alive_time = alarm_time-(total_sleep_time/1000.0);
throughput =(double)(t_totaldata)/alive_time;

if(throughput > 2*randomPacketTime){
      randomPacketTime = randomPacketTime*2;
      if(randomPacketTime> 32000){
            randomPacketTime = 32000;
            //cap the max sleep interval to be 64ms
      }
      }else{
            randomPacketTime = randomPacketTime/2;
            if(randomPacketTime< 2000){
                  randomPacketTime = 2000;
                  //cap the min sleep interval to be 2ms
            }
      }

      throughput= 0;
      total_sleep_time =0;
      t_totaldata =0;

      signal(SIGALRM, handler);
      alarm(alarm_time);
```

Other Issues / Interesting Facts

SSH Server Alive Check Message

Since we are only checking the “idleness” based on the real data that is being polled from the channel. The server alive check message will not affect the measurement and will not cause the state transition.

Avoiding Getting Blocked by Select

If we find that we are in the “Fake Idle” state, we will queue Junk data onto the outgoing buffer before the select is called, so we can guarantee that process will wake up every cycle before we transition to the truly idle state.

Determining Buffer Padding Size

The reason we chose the static number of 1330 was because when optimizing our code based on the Wireshark Output, we notice the majority of the

packet are completed filled and don't overflow to smaller packets. By doing this, we can further hide the pattern of individual packet sizes.

Data Analysis

In order to test the result of our modification we had Wireshark running outside our virtualized environment which would sniff the SSH packets between the client and the server after the handshaking process. After we finished the measure of each website we would look on the statistic summary generated by Wireshark to see the total number of packets sent, total transmission size, throughput and average packet size. Alongside with this application we would run Firebug to note the time it took to load the DOM of the page. It is important to note that Wireshark only returns the Ethernet Frame Size, so we had to manually calculate the SSH packet size based on the formula: Ethernet Frame Size – 66 (Ethernet Frame Header Size based on our measurements).

We decided to choose a set of ten websites, each would be recorded five times with both our modified SSH as well as the original application. The website should represent popular traffic that will most like be pattern matched against and blocked by foreign firewalls. During each trial we were able to record the calculated rounding size and its difference between the SSH total transmission. Size Prediction Error is the percent difference between what was actually transmitted and the size that was calculated.

Below is our analysis and report:

Data

YouTube Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error
1	4478726	1416	8.59	1.55	3163	4194304	4269968	1.803970337
2	6735956	1420	7.9	1.75	4742	6291456	6422984	2.090581258
3	2283874	1405	4.5	1.45	1628	2099200	2176426	3.67883003
4	8940642	1418	10.2	2.1	6304	8388608	8524578	1.62088871
5	4547014	1419	6.8	1.8	3203	4194304	4335616	3.369140625

YouTube Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	58984	983	0.2	1.77	60	55024	N/A
2	60522	992	0.24	1.48	61	56496	N/A
3	57534	913	1.9	1.66	63	53376	N/A

4	51086	929	0.22	1.51	55	47456	N/A
5	56518	958	0.22	1.56	59	52624	N/A

Amazon Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error
1	4584062	1404	1.7	1.47	3263	4196352	4368704	4.107186432
2	13457048	1418	8.2	1.56	9491	12582912	12830642	1.968781153
3	17881566	1419	10.1	2.42	12601	16777216	17049900	1.625323296
4	8979620	1421	8.7	1.39	6318	8388608	8562632	2.07452774
5	4579574	1412	4.9	1.78	3243	4200400	4365536	3.931435101

Amazon Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	142668	860	0.23	1.3	166	131712	N/A
2	151094	854	0.36	1.39	177	139412	N/A
3	140284	888	0.29	1.5	158	129856	N/A
4	139094	897	0.34	1.3	155	128864	N/A
5	154838	870	0.25	1.63	178	143090	N/A

Hulu Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error (%)
1	17902916	1420	10.1	4.36	12606	16777216	17070920	1.750612259
2	13666990	1422	8.5	3.73	9612	12582912	13032598	3.573783239
3	17895530	1420	10.2	4.74	12608	16777216	17063402	1.705801487
4	17943364	1422	9.7	5.88	14396	16777216	16993228	1.287531853
5	18042150	1418	7.7	4.95	12717	16777216	17202828	2.536845207

Hulu Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	170392	906	0.22	3.26	188	157984	N/A
2	178066	791	0.14	4.15	225	163216	N/A
3	169778	959	0.29	3.06	177	158096	N/A
4	171504	964	0.23	3.38	178	159756	N/A
5	171450	907	0.22	3.93	189	158976	N/A

Stack Overflow Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error (%)
1	2290370	1400	3.3	1.57	1635	2101248	2182460	3.864941216
2	4516054	1413	5.4	1.18	3194	4198400	4305250	2.545017149
3	2296048	1397	1.13	1.21	2359	2099200	2140354	1.960461128
4	2279282	1396	1.2	1.14	1633	2099200	2171504	3.444359756
5	4577626	1412	2.2	1.13	3240	4196352	4363786	3.9899894

Stack Overflow Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	61666	743	0.25	1.41	83	56188	N/A
2	58550	760	0.32	1.13	77	53468	N/A
3	60736	759	0.26	1.46	80	55456	N/A
4	55288	813	0.33	1.04	68	50800	N/A
5	65234	673	0.27	1.02	97	58832	N/A

Facebook Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error (%)
1	2120048	1424	5	1.05	1488	2099200	2021840	3.685213415
2	2256696	1421	4.5	1.15	1588	2101248	2151888	2.409996345
3	1136470	1415	4.3	1.2	803	1052624	1083472	2.9305811
4	2248310	1420	3.1	1.02	1583	2099200	2143832	2.126143293
5	2258990	1420	5.7	1	1591	2101248	2153984	2.509746589

Facebook Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	22014	957	0.26	1.14	23	20496	N/A
2	22112	921	0.29	1.14	24	20528	N/A
3	22180	853	0.29	1.05	26	20464	N/A
4	22098	884	0.23	1.23	25	20448	N/A
5	22080	920	0.28	0.997	24	20496	N/A

NY Times Home Page (Modified SSH)

#	Total Transmission	Avg. Packet	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction

	(Bytes)	(Bytes)						Error (%)
1	8936456	1420	5.3	3.98	6292	8388608	8521184	1.580429077
2	8946956	1412	5.6	3.75	6338	8388608	8528648	1.669406891
3	9005500	1415	8.7	3.4	6364	8388608	8585476	2.346849442
4	17873052	1421	8.7	5.12	12578	16777216	17042904	1.583623886
5	13403234	1420	8.1	4.01	9439	12582912	12780260	1.568380992

NY Times Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	208824	967	0.56	2.76	216	194568	N/A
2	208292	964	0.5	3.18	216	194036	N/A
3	208174	968	0.46	3.08	215	193984	N/A
4	230070	902	0.52	3.04	255	213240	N/A
5	219238	966	0.48	2.78	227	204256	N/A

Stony Brook Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error (%)
1	4473612	1420	9.3	1.75	3150	4194304	4265712	1.70249939
2	6761210	1422	3.5	1.97	4755	6291456	6447380	2.478345235
3	4475826	1420	9.7	1.66	3153	4194304	4267728	1.750564575
4	6717880	1418	3.4	2.42	4738	6291456	6405172	1.807467143
5	4502118	1420	9.1	1.97	3171	4194304	4292832	2.349090576

Stony Brook Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	97894	851	0.17	1.13	115	90304	N/A
2	95214	924	0.48	1.86	103	88416	N/A
3	96238	934	0.47	1.37	103	89440	N/A
4	96404	918	0.69	0.888	105	89474	N/A
5	95354	944	0.3	1.05	101	88688	N/A

Yahoo Home Page (Modified SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	Rounding Size	SSH Total Size	Size Prediction Error (%)
1	4502946	1417	2.8	0.469	3178	4196352	4293198	2.307861686
2	4649668	1417	2.1	0.303	3282	4196352	4433056	5.640708882
3	4668380	1415	5.2	0.449	3393	4196352	4444442	5.912039791
4	2266064	1416	3.5	0.376	1600	2099200	2160464	2.918445122
5	4503832	1422	5.8	0.408	3168	4196352	4294744	2.344703209

Yahoo Home Page (Stock SSH)

#	Total Transmission (Bytes)	Avg. Packet (Bytes)	Throughput (Mbit/s)	Latency (Sec.)	Packet Count	SSH Total Size	Size Prediction Error
1	344628	1123	0.51	2.77	307	324366	N/A
2	215194	1010	0.34	3.13	213	201136	N/A
3	249950	1046	0.39	2.51	239	234176	N/A
4	169458	906	0.17	2.58	187	157116	N/A
5	141778	978	0.22	2.43	104	134914	N/A

Evaluation

Basic Defense

By padding the outgoing buffer and the transmission time of the packet on the buffer we were able to hide the transmission time and the size for each packet. In Figure 1, we have the plot of the average packets size of our modified SSH application verses the stock SSH. We found a more consistent line for our modified version. This means the packet size do not vary that much amongst all the measurements we tested. By successfully normalizing each individual packet size we were able to hide the packet size pattern of each website. In Figure 2, we can see that by randomizing the transmission time for each packet, we get the random transmission size because the application wakes up randomly and transmits the padded buffer size.

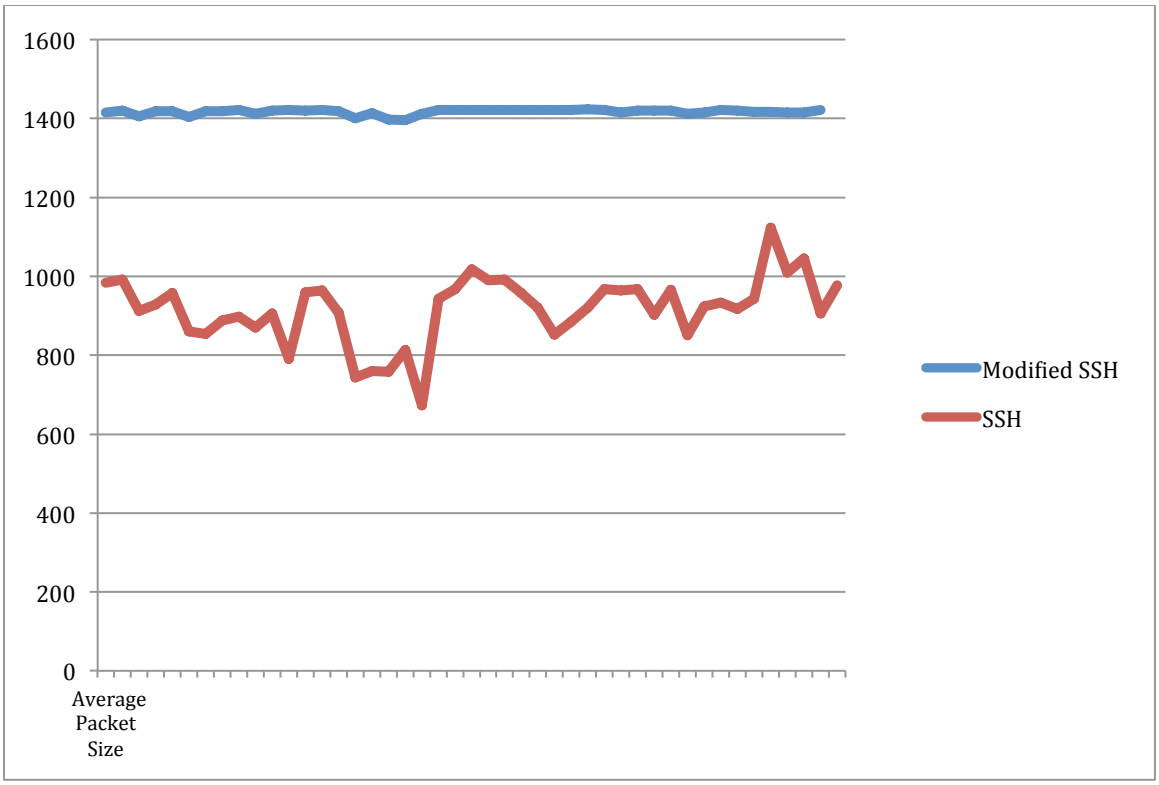


Figure 1

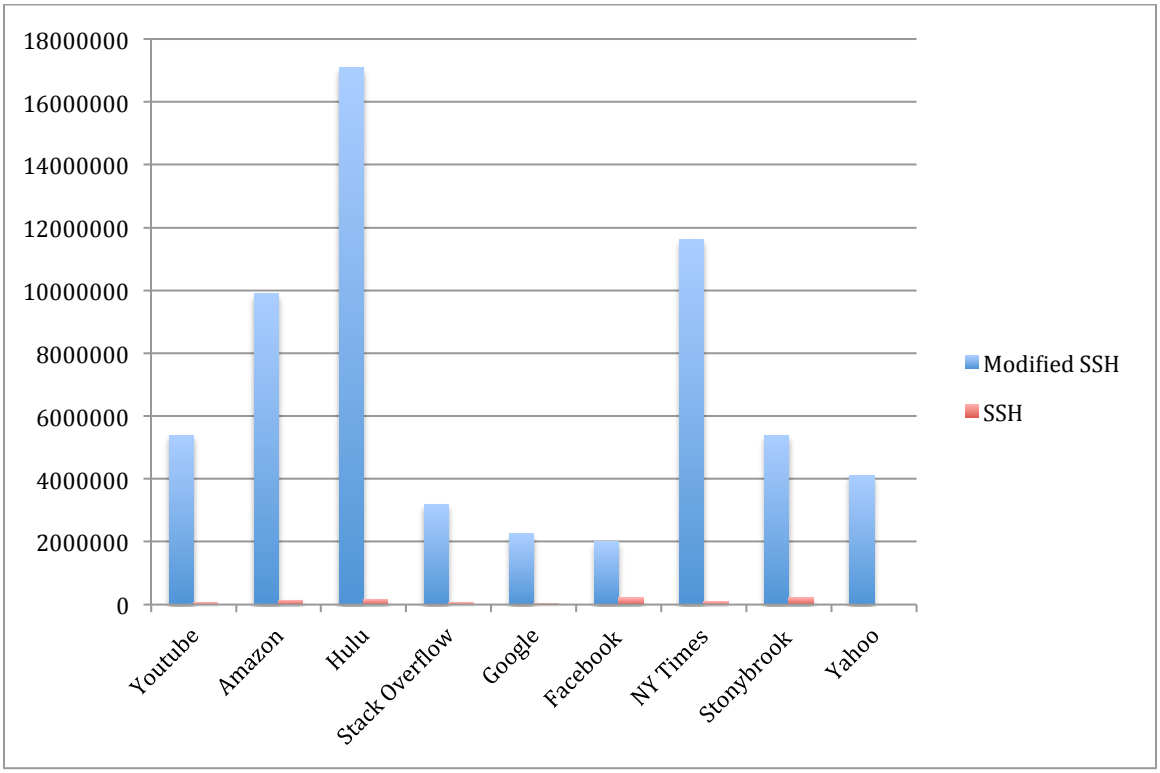


Figure 2

Hiding of Size Information

In Figure 2, we can see the total transmission size in our modified SSH is independent of the real transmission data recorded by the stock SSH application. From the data tables recorded above, we were able to calculate the predicted size error. The average error amongst all measurements is 2.56%, which indicates the preciseness of our rounding implementation. The reason there is a small amount of error between our predicted and actual rounding size is due packet retransmissions and SSH window-probing messages sent between the server and client, which is not recorded by our code.

Speed Tuning Option to Increase Efficiency

In Figure 3, we plotted the latency recorded in seconds using Firebug in both the modified SSH application and the stock version, as you can the differences are really small and understandable due to the larger transmissions.

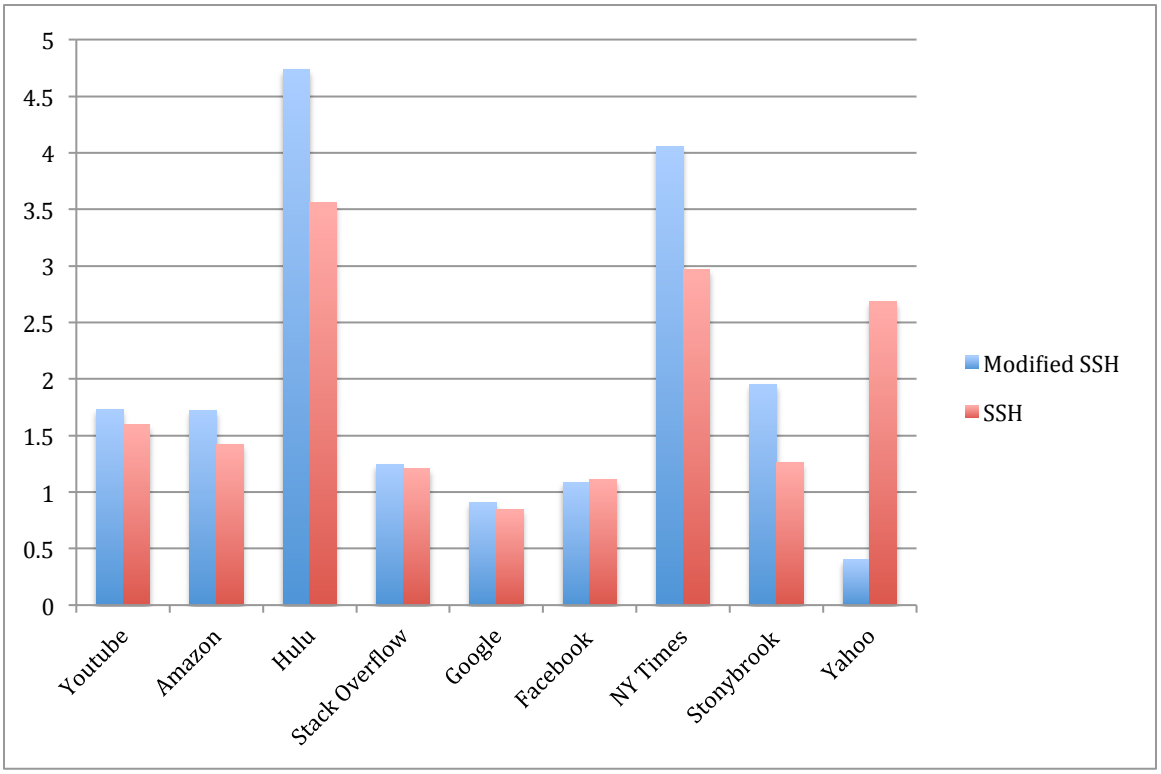


Figure 3

Overhead Measurement

In Figure 4, we were able to plot the throughput of both the modified and the stock SSH during the transmissions. From this graph we can see, the extra bandwidth used in our modified version is at least ten fold. This is our trade off when bypassing pattern detection.

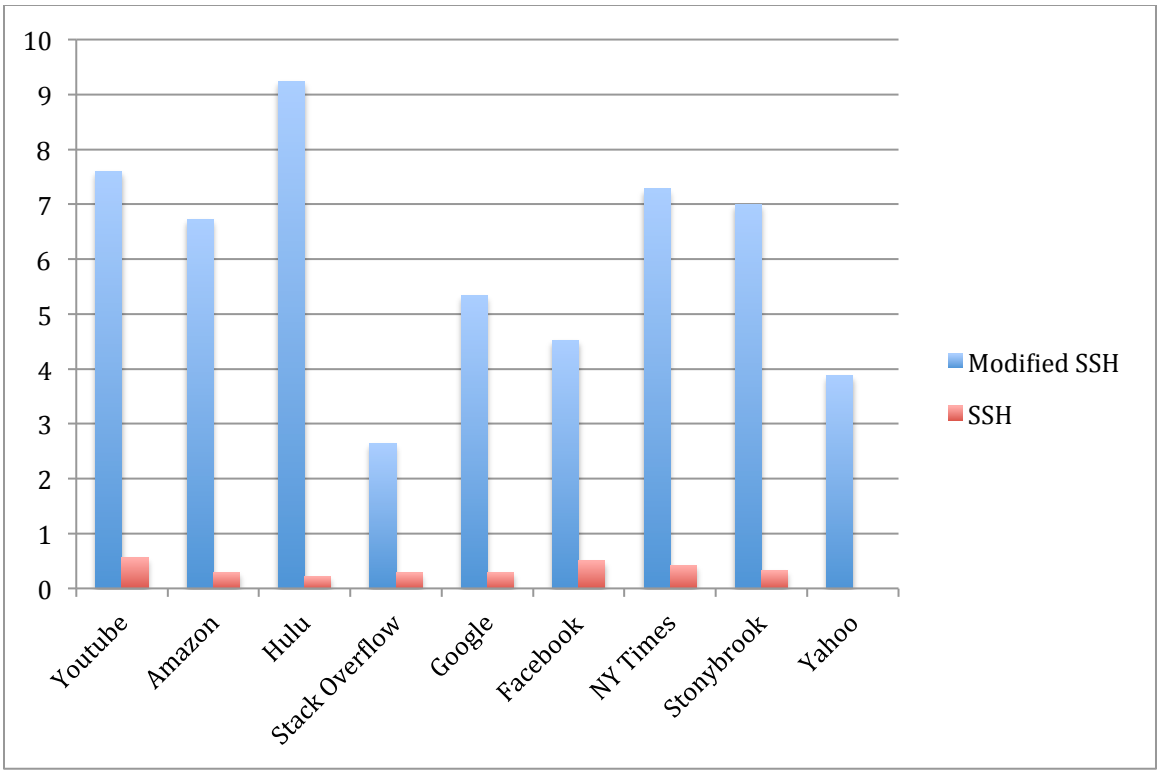


Figure 4

Conclusion

After the completion of the project, we feel we have achieved our goal of implementing a provable secure traffic analysis defense and optimized it to reduce overhead. We implemented this system as a real world application to prove the usability for user who wants to keep their web traffic anonymous. The evaluation provided is strong support of its security.

Citations

1. Dyer, Kevin P., Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail."*Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail*. Web. <http://web.cecs.pdx.edu/~teshrim/tmAnotherLook.pdf>.