Exhibit A.2

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
TYLER DIVISION

BEDROCK COMPUTER TECHNOLOGIES LLC,	§	
	§	
Plaintiff,	§	
	§	**CASE NO. 6:09-cv-269**
v.	§	
	§	**Jury Trial Demanded**
SOFTLAYER TECHNOLOGIES, INC., et al.	§	
	§	
Defendants.	§	

OPENING EXPERT REPORT OF DR. MARK JONES

or utilizing software based on Linux version 2.6.25 and onwards practices all claims of the ’120 patent and therefore embodies Bedrock’s patented invention. My review of the Defendants’ disclosures and testimony in this case reveals that the Defendants have made—and continue to make, and have used—and continue to use computer equipment configured with or utilizing software based on Linux version 2.4.22 and onwards as well as computer equipment configured with or utilizing software based on Linux version 2.6.25 and onwards. Since the ’120 patent was filed after June 8, 1995, it expires 20 years after its filing date, see 35 U.S.C. § 154. Under this calculation, the end of the term of the ’120 patent is on January 2, 2017. I am not aware of any authority that the Defendants have to make or use Bedrock’s patented invention. Therefore, it is my opinion that the Defendants directly infringe the claims of the ’120 patent.

9. Ultimate Opinions on Infringement

It is my opinion that the Defendants, in their making and using of the Linux Systems I discussed in § 5 literally and directly infringe claims 1-4 and 7-8 of the ’120 patent, and the Defendants, in their making and using of the Linux Systems I discussed in § 6 literally and directly infringe all claims of the ’120 patent. Appendices A-G to this report take into account the effect, if any, of the Defendant’s respective network architectures or modifications to the Linux kernel on the infringement or role of the claims of the patent in the Defendants’ networks.

10. The Performance Advantage of the ’120 Patent

The ’120 invention[5] improves the efficiency of a system that employs a hash table with chaining where the records in the hash table are automatically expiring.[6] This efficiency can

[5] Note that this section is a general discussion of the advantages of the ‘120 invention and, therefore, the patent is, at times, broadly characterized for ease of discussion. The precise analysis of infringement is found in §§ 5 and 6 of this report.

[6] Where that improvement in efficiency is compared against a system that uses on-demand garbage collection to remove records that are no longer desired.

come in the form of reduced CPU time for removing records that are no longer desired, fewer workload spikes to dedicated garbage collection operations, and more quickly returning memory to a free pool.

The improved efficiency arises because the ’120 invention “piggybacks” garbage collection on top of searching the hash data structure. In other words, when the system is already walking through a chain in the hash table, the ’120 invention checks to see if records in that chain should be removed. Thus, the additional cost for walking through the chain at a later time to remove records is avoided. By examining records that have already been brought into the hardware memory cache (and the registers of the processor), the time to access these records is reduced when compared to a dedicated garbage collector and the system is able to avoid displacing other data from the hardware memory cache during a later run. Further, by performing these operations during normal access to chains in the hash table, the garbage collection is not only less costly, but that cost is more evenly spread over time, which will avoid spikes in CPU usage that may delay the real-time response to events. Additionally, by removing records from the linked list in a timely fashion, the next search of that linked list will not have to make an unnecessary traversal of that no longer desired record.

The average cost of searching a hash table depends on the average number of records in a chain; thus, deleting records from chains in a timely fashion will improve that efficiency. In some systems, not only is the average cost important, but the maximum cost is also important. The maximum cost of searching a hash table depends on the length of the single longest chain in the hash table; thus, deleting records from chains in a timely fashion can be even more important in such a system.

Further control over the costs incurred by on-the-fly garbage collection can be achieved by limiting the number of records to be deleted during on-the-fly garbage collection. This can, for example, further smooth out the cost of a normal operation on the hash table by limiting the amount of time spent on any single operation. As another example, by cutting short the examination of records for deletion, the system can avoid the cost of pulling new records into the hardware memory cache as well as perturbing the contents of the hardware memory cache.[7]

In contrast, the old way of removing records from the chains in the hash table is to run a dedicated garbage collection routine that is called either periodically or when the system detects a problem. This dedicated garbage collection routine walks through all (or some) of the linked lists in the hash table and examines the records in those lists to determine which ones to remove and then removes them. This has the disadvantage of requiring that the linked lists be walked for the sole purpose of garbage collection, incurring the cost of walking the linked lists and of perturbing the contents of the hardware memory cache. This dedicated garbage collection must be scheduled frequently enough to scan the hash table (either in whole or in part) to remove entries that are no longer desired in a timely fashion. This additional work can result in "bursty" demand on the CPU that can affect the response time of the CPU to other tasks. Further, dedicated garbage collection is often run as an additional task in the system, requiring coordination with (and potentially locking out or delaying) other tasks accessing the hash table and, particularly in modern multi-core systems, can be result in delaying those other tasks.

10.1 The Role of the '120 Invention in the Linux Route Cache

It is a common practice in computing to use a cache to store recently computed or accessed data for fast access rather than recomputed or repeat the retrieval process. Using a

[7] Note: I may show the jury with traditional illustrations of the operation of the hardware memory cache and the operation of the invention with respect to the hardware memory cache.

cache can provide a performance advantage when the same data is retrieved multiple times from the cache rather than being recomputed or retrieved from other storage. The costs of using a cache in software (e.g., Linux) come from the additional storage required for the cache as well as the time required to store and retrieve to/from the cache. In addition, when the nature of the entries in the cache are such that they can become no longer desired, the costs of using a cache include removing entries from the cache that are no longer desired.

A hash table with chaining can be an efficient data structure for a cache in software. A hash table can provide an average access time for storage and retrieval of records that is proportional to the average number of records in a chain even when the potential "key space" is very large. This is particularly attractive for the Linux route cache where the key space is a combination of source IP address, destination IP address, as well as other values. When working efficiently, the route cache can provide a performance advantage over having to use more expensive means to determine a route.

But, without the '120 invention, the use of a hash table can lead to performance issues when the records in the hash table are such they can become no longer desired. The Linux route cache can become a liability when it, for example, consumes too much memory. To guard against this, it is necessary to remove entries that are no longer desired, perhaps because they are no longer valid or because the cache is consuming too much memory. As another example, the route cache can become a liability when one or more hash chains become so long that retrieving a record results in the consumption of too much CPU time and/or unacceptable response times. In an attempt to maintain the route cache, early designs of the source code relied solely upon dedicated garbage collection that, as described above, has significant disadvantages.

Like any cache, the benefits of the route cache vary with the current workload placed upon the computer where it is running. For example, if the computer has zero network traffic, then the route cache will not be in use. Or, alternatively, if data stored in the cache is never used again, then the route cache will not have any performance advantage. Further, the cache should be sized and maintained to allow it to store the "working set" of entries so that when a query to the cache is repeated, the requested record is still in the cache and has not been removed. The route cache in Linux can, in some versions, be disabled by setting a variable via the sysctl mechanism in Linux.[8]

The '120 invention was added to the Linux route cache[9] to maintain the efficiency of the route cache. It does so through the advantages described above. Note that the '120 invention allows the route cache to be efficiently maintained under a wide range of operating conditions without intervention by a system administrator. As my experimental results below will show, these advantages may be large in some operating conditions and small in other conditions. Generally, the Linux route cache with the '120 invention allows a server with network traffic to operate more efficiently than with the route cache disabled.[10] Further, the '120 invention allows the Linux route cache to operate more efficiently across a range of conditions and helps it avoid severe deteriorations in performance under certain conditions.

10.2 The Importance of Server Efficiency in a Datacenter

[8] Note that mechanism does not remove the functionality from the system and it can be re-enabled using the same mechanism without rebooting the operating system.

[9] The different versions of the Linux source code are described elsewhere in the report.

[10] With the caveats previously discussed regarding the extremes of workload or an inappropriately sized cache.

The design of a computer system to meet the requirements of a particular use involves making choices to satisfy multiple interrelated criteria, including processor capabilities, networking capabilities, cost considerations, power considerations (direct energy costs as well as cooling costs), and size considerations. This is particularly true in the design of a datacenter, such as those used by the defendants.[11] When designing the system (or updating a design) for a datacenter, the system must be designed to handle peak loads that may greatly exceed the average load.[12] [13] Further, even though the utilization of a server may rarely reach 90%, datacenter operators keep reserve capacity for unexpected load spikes as well as for taking on the load of a failed cluster at another location.[14] Note that the system is not designed for the "average daily peak load" but for the maximum peak load.

In addition to day-to-day (and within day) fluctuations in workload, the characteristics of the workload placed on servers in a datacenter operating on the Internet are likely evolve rapidly over time.[15] Such workload changes can arise due to, for example, changes in the mix of services provided by the data center (e.g., rising popularity of a service over time or introduction of a new service with different operational characteristics), changes in the efficiency or design of software applications, and, in the case of centers providing hosting services, changes in the clientele and the clientele's services.[16] Thus, selection of a server tailored for a specific, current workload is very likely to result in a server that is suboptimal in a short time span. Further, it is

[11] *See* BTEX0752258, BTEX0752263, BTEX0752383, and BTEX0752440.

[12] See, for example, Kravchenko Transcript at 19:8-22:11.

[13] *See* BTEX0752263.

[14] *See* BTEX0752263.

[15] *See* BTEX0752263.

[16] *See* http://news.netcraft.com/busiest-sites-switching-analysis/

desirable within a datacenter to deploy a small number of server configurations at any particular time because it simplifies load-balancing and maintenance costs.[17]

To summarize, a software feature that improves the efficiency of a server, provides at least the following benefits:

- Fewer servers need to be used to meet the reserve capacity requirements for which datacenter operators plan and design their systems.
- Fewer servers need to be used to meet the requirement that the datacenter be able to support an evolving workload while maintaining a small number of server configurations.
- Fewer servers need to be used to meet the requirement that the datacenter be able to support a heterogeneous mix of services each with different workload characteristics while maintaining a small number of server configurations.

If a datacenter is required to meet a performance goal of C=N*X and it is designed using N servers each with capability X, then if the capability of servers is reduced to X*(1-Y), then at least N/(1-Y) servers are now required.[18] For example, if Y=0.1 (or 10%), then the number of servers increases by a factor of (1/0.9) or approximately 1.11. The percentages of performance degradation that I calculate in this report correspond to the "Y" of this formula.

10.3 Discussion of Experimental Results

To demonstrate the advantages of the claims of the '120 patent, I have performed a series of experiments with differing workloads, including different applications. I have compared three configurations of the route cache in these experiments: (i) the route cache is enabled and the '120 invention is present (the default condition); (ii) the route cache is enabled and the '120 invention is removed (modified version); and (iii) the route cache is disabled (via the sysctl mechanism).

[17] *See* BTEX0752263.

[18] Note that other factors, including increased communication or unacceptable latencies, may lead to the requirement of additional servers beyond this number or other changes in architecture.

Before going into the details of the experimental, it is important to note that the purpose of these experiments is not to duplicate operating conditions for any specific server. Instead, the purpose is to show the differences in performance for a server for the three configurations of the route cache. As can be seen in Figure 1 through Figure 3,[19] the advantages of the ’120 invention over the other two configurations vary with the current workload. Under certain conditions in these figures, the current performance advantage is negligible while in others it is close to 20%. The performance advantage on a particular server at a particular time is dependent on the workload associated with that time and the configuration of the server (including the hardware, the operating system, and the applications). I note that under a wide range of operating conditions, the advantage achieved by the ’120 invention over the route cache disabled generally exceeds 10%.

[19] Note that these three figures are generated directly from the values in the table given in Appendix H. At trial, I may present similar figures from the data in the Appendices for the Squid Results.

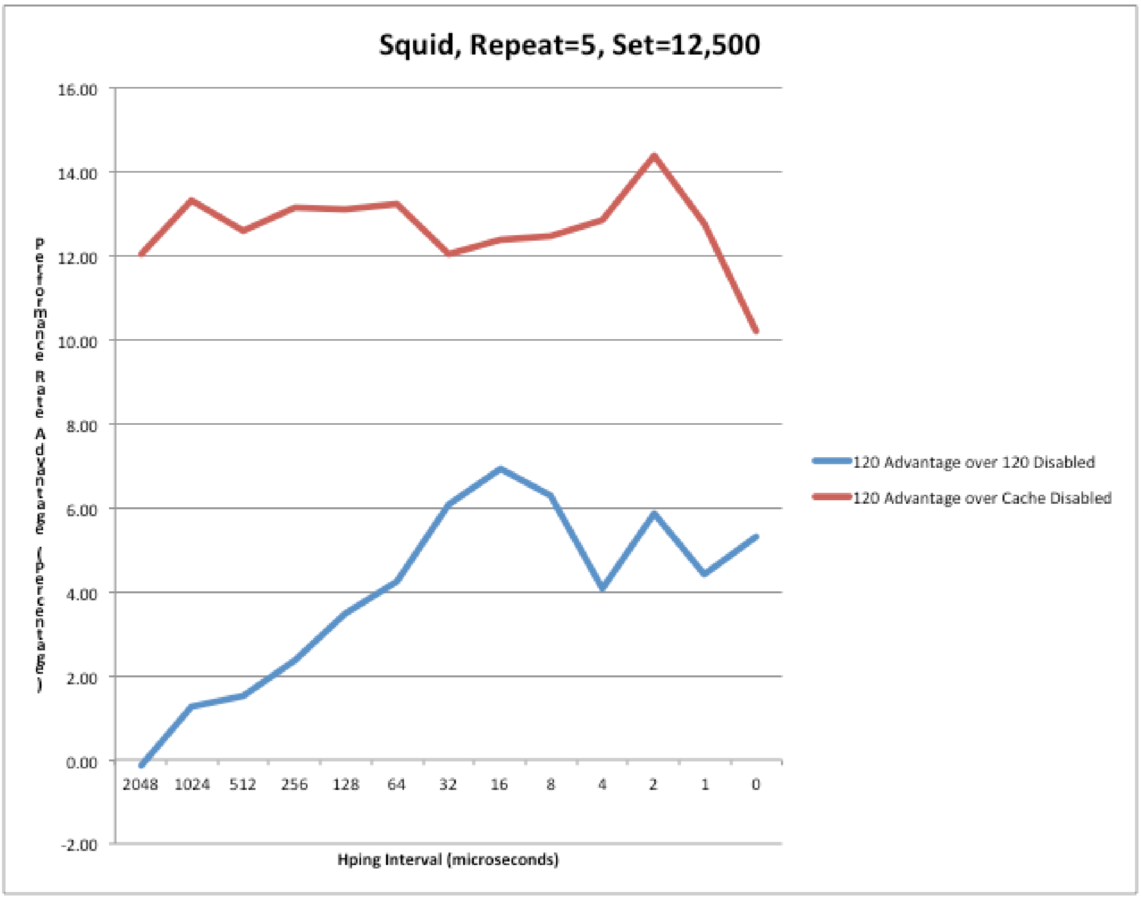

Figure 1: Results for the Squid Application for Repeat=5 and Set=12,500. Details are explained in the experimental results discussion.

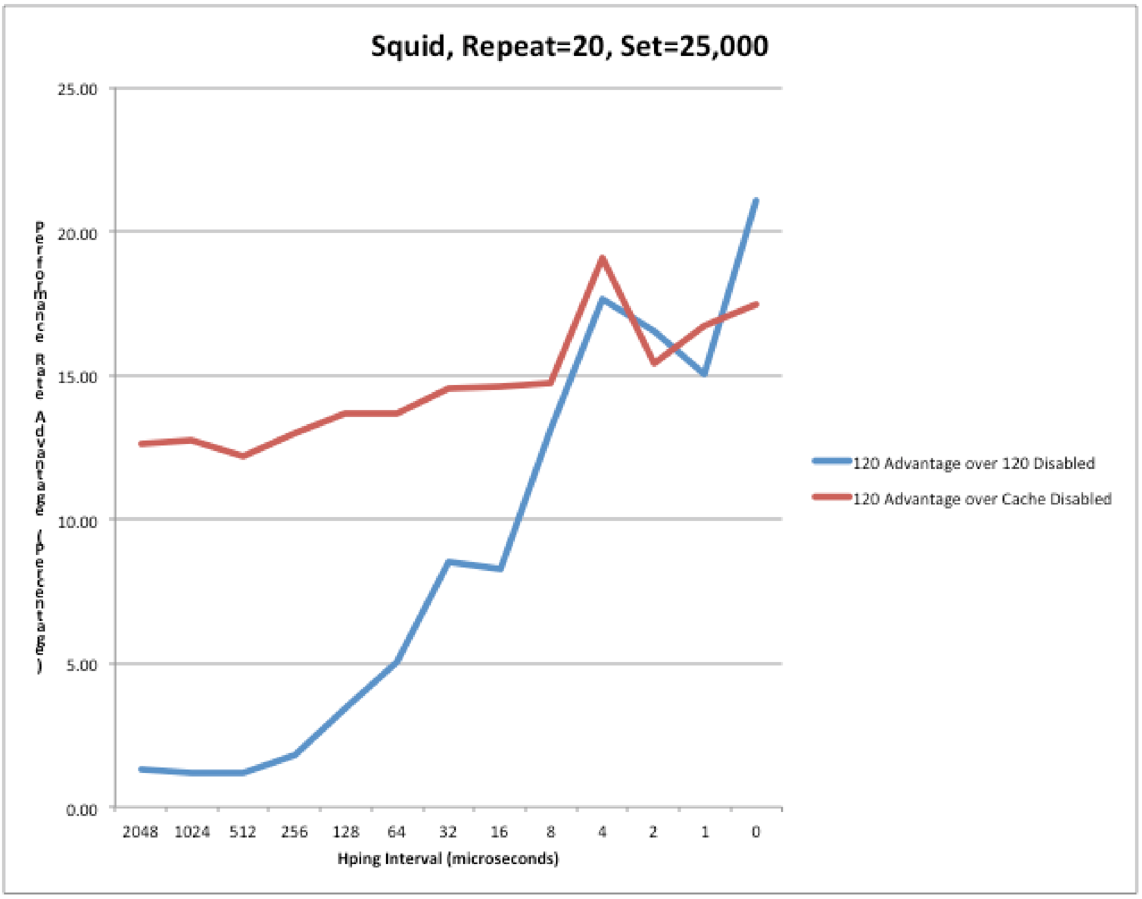

Figure 2: Results for the Squid Application for Repeat=20 and Set=25,000. Details are explained in the experimental results discussion.

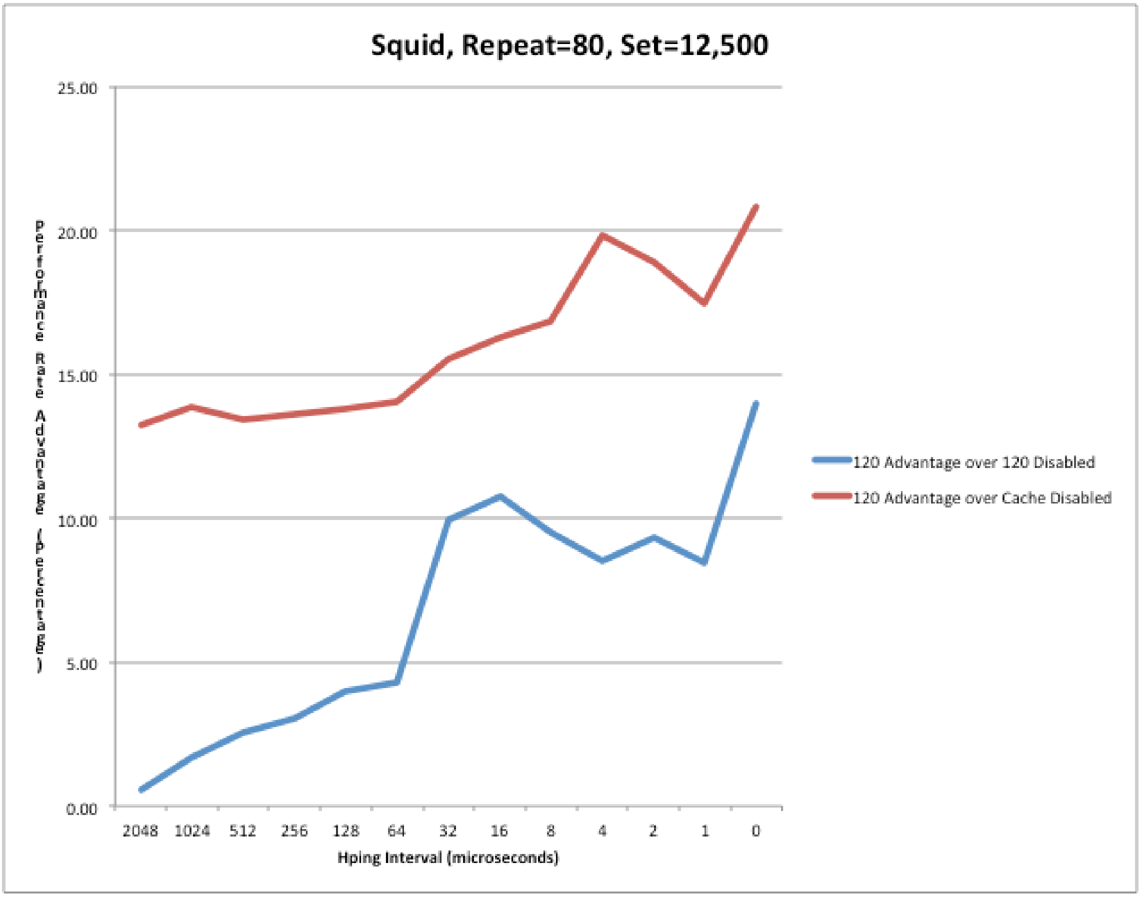

Figure 3: Results for the Squid Application for Repeat=80 and Set=80,000. Details are explained in the experimental results discussion.

I have performed the experiments on the 2.6.31 version of the Linux kernel because it is relatively recent and contains both mechanisms of on-the-fly garbage collection discussed in the infringement section of this report. I have used an Ubuntu distribution (upgraded to version 9.10). For all tests, I have built the kernel with the server option for version 2.6.31-22 of the source code.

I have selected two applications for these experiments. One is the Apache2[20] web server, one of the most widely used web servers on the Internet.[21] The other is Squid,[22] which I have

[20] I installed Apache2 using the "apt-get install apache2" command on the server. I am using a version that is current as of January 2011. I am using apache2.2-common 2.2.12-1ubuntu2.4.

[21] See, for example, www.netcraft.com

configured as a reverse proxy server in a manner that is designed to increase the efficiency of web servers that are located behind Squid. Squid is used by many web sites to improve efficiency.[23] For both applications, I used a single web page that was 3594 bytes long. Even though Squid was directed to a web server, I visited the web page prior to running the tests to ensure Squid had the web page in its cache.

To perform the experiments, I used a Dell PowerEdge R300[24] running one of the two applications. To generate the loads, I used two identical client computers, Client A and Client B.[25] Both client computers were running Ubuntu 10.10. These computers were connected using a NetGear ProSafe 8-Port Gigabit Smart Switch. All three computers were configured with addresses on a 71.1.x.x network[26] and the server computer was configured with a default route with client B as the gateway.[27]

To generate part of the load, Client A ran curl-loader, an open source program.[28] For the squid experiments, I ran curl-loader with the command line options "-t 4 –l 2 –r –i 1". The configuration file used is attached to this report as Appendix K. Essentially, I used 800 virtual clients sharing four different IP addresses, all loading the same page from Squid. For the apache experiments, I ran the curl-loader with the command line options "-l 2 –r –i 1". The

---

[22] I installed Squid using the "apt-get install squid" command on the server. I am using a version that is current as of January 2011. I am using Squid 2.7.STABLE6-2ubuntu2.2.

[23] See, for example, www.squid-cache.org

[24] See Appendix I.

[25] See Appendix J.

[26] This network was not directly connected to the Internet.

[27] Client B was configured to drop packets other than those for which it was the destination.

[28] I used "curl-loader-0.52" downloaded from http://sourceforge.net/projects/curl-loader/files/curl-loader-stable/.

configuration file is attached to this report as Appendix L. Essentially, I used 200 virtual clients each with a different IP address, all loading the same page from Apache.

The other portion of the load was generated using a modified version of the hping3 utility. As written, hping3 has a large number of options for generating network packets, including an option to generate packets as if they are originating from a random source address. However, for the range of experiments that I believed appropriate to carry out, a new random source address for every packet was not suitable. I modified the hping3 source code to have two new options "—repeat <integer>" and "—interleave <integer>". These two commands are intended to operate together with "—random-source".[29] The interleave number indicates the size of an array that is filled up with randomly generated addresses. The modified hping3 program steps through the addresses, sending them one by one. When the program has reached the end of the array. The repeat number is used to compute the number of addresses in the array that will be replaced with new random addresses before hping3 starts back at the beginning of the array.[30] This testing can be used to approximate a set of ongoing connections, where each connection is approximately "repeat" packets in length (from the sender to the server) and there are approximately "interleave" connections in the timeframe. In the experimental setup, hping3 was not used to set up a TCP connection to the server, but it did accept the packets that the server sent to it (one for each packet sent to the server). Note that the packets in hping3 are to be sent at

[29] Note that I have included in the Appendix a "diff" of my version against the original version of hping3 to indicate where I have made changes. I note that the modifications are intended to perform for the range of parameters I have used and are not in form that has been robustly tested for a wider range.

[30] The number to replace is ~(interleave #)/(repeat #) – see diff file for details.

intervals of the number of microseconds specified on the command line[31] – I note that due to loading and other factors, the actual packet rate sent from the computer does not precisely scale by factors of two, particularly for very small intervals, therefore, I have recorded the observed average packet rates for each interval value in my experiments in Appendix M.

10.3.1 Squid Experiments

In the Squid-based experiments, I kept the load from curl-loader (800 virtual clients) constant and then varied the traffic generated by hping3. The hping3 load was used to represent a variety of mixes of IP address ranges to test the response of the server under these mix of address ranges. This was done in an identical fashion for each of the three conditions ('120 enabled, '120 disabled, cache disabled) and the results are given in Appendix H.[32, 33, 34] I collected two performance results: (1) the number of URL successes for curl-loader per second (Column F) and (2) the percentage of time the CPU was busy (column G).[35] Taken together as (number of successes)/(CPU busy %) gives a metric of the amount of "useful work" the CPU

---

[31] hping3 -Z -n -I eth0 71.1.1.7 --rand-source --repeat <number> --interleave <number> -i <number> -q -d 200

[32] A '1' in "Cache Enabled" indicates that the route cache is enabled, a '0' indicates it is disabled. A '1' in "120 Enabled" indicates that all Linux kernel code is left unmodified. A '0' indicates that the modified route.c [attached in an appendix] has been used, where that modified version has the "on-the-fly garbage collection" is removed in rt_intern_hash().

[33] It is important to note that on the hping3 column (and the x-axis in the Figures above), the value 2048 for the interval actual denotes zero load from hping3 (the 2048 is used for convenient plotting) and the value 1 is actually the "—faster" command line argument to hping3 and the value 0 is actually the "—flood" command line argument to hping3.

[34] I note that the experiments related to Squid and Apache2 are conservative in several respects. For example, I operated the server with a very simple, two-entry routing table and nothing entered in the routing policy database. As another example, neither application was required to make any connections to other computers or programs in order to service the requests; e.g., Squid did not need to contact any Web servers.

[35] The busy time was calculated at 100%-(CPU idle %). I note that when not performing tests (or running other programs), the "top" command reported an idle percentage of 100%.

does as a function of its time devoted to processing (as opposed to idle capacity) and is shown in Column I. To quantify the performance difference between the ‘120 enabled condition and the other two conditions (column J), I computed [(Work Rate ‘120 from Column I) – (Work Rate for Other Condition from Column I)]/(Work Rate ‘120 from Column I). This value is in Column J of the Table for each of the other two conditions – note, of course, that there is no number in the Rows for the ‘120 enabled condition=1. I note that for the experiments that generated the results in Appendix H, I used the default settings for all kernel parameters other than: (a) To generate results that accurately represented the difference between the ‘120 enabled and ‘120 disabled, I effectively prevented the system from turning off caching (a feature available in 2.6.31) by setting the parameter “rt_cache_rebuild_count” to a very high number so that the system would not turn off the cache[36] and (b) the same parameter was set to “-1” to effectively disable the route cache.

To collect the results for Squid and Apache, I ran the experiments using the following sequence:

(1) Start hping3
(2) Start curl-loader
(3) Wait 5 seconds
(4) Begin collecting load information on the server at 1 second intervals using the “top” utility
(5) Wait 250 second
(6) Stop curl-loader and hping3
(7) Keep last 180 samples for curl and all samples for the CPU rates
(8) Compute the average values across all of these samples.

[36] Note that in those cases where the ’120 disabled performs worse than with caching turned off, it would be reasonable to say that ’120 disabled could simply default to the cache turned off and, therefore, should not be worse than the cache disabled case.

The results in Appendix H are for the default number of hash chains, which for this computer is 2^{16} with a default value of gc_elasticity of 8. I note I observed results consistent with those in this Appendix for a range of parameter values in my experiments.

10.3.2 Apache Experiments

For the Apache-based experiments, I examined the same three configurations and used similar methods to those described for the Squid-based experiments. The primary difference, as explained above, is that the load is smaller and that it is sent from 200 different IP addresses. These results, given in Appendix N[37] begin with a higher hping3 load because the load from curl-loader is low for these experiments.[38] The results were collected and analyzed in the same way as for the Squid experiments.

The results in Appendix N are for the default number of hash chains, which for this computer is 2^{16} with a default value of gc_elasticity of 8. To investigate other values for these (and other) parameters, I give results other experiments for just the '120 enabled and '120 disabled conditions (the cache parameter values do not affect the cache disabled case) in the Appendix O (number of hash chains = 2^{18}, gc_elasticity=4) and Appendix P (number of hash chains = 2^{20}, gc_elasticity=4).[39] I further note I observed results consistent with those in these three Appendices for a range of parameter values in my experiments.

I note that the results from the Apache experiments are consistent with the results in the Squid experiments.

[37] I may present figures at trial from these two Appendices that are formatted similarly to the figures presented above.

[38] Note that on the hping3 column, the value 1 is actually the "—faster" command line argument to hping3 and the value 0 is actually the "—flood" command line argument to hping3.

[39] I may present figures at trial from these two Appendices that are formatted similarly to the figures presented above.

10.3.3 Chain length experiments

One of the benefits of using a hash table data structure with chaining is that the average number of entries in each chain is alpha=(number of entries)/(number of chains), leading to an average search time proportional to alpha.[40] Thus, if the program using the data structure is able to keep alpha constant, the expected time to search for any entry is constant. However, in many systems, one is not interested in just the average search time, but also in the expected maximum search time. For example, in a real time system (or a portion of a system that operates according to real time constraints), the expected maximum search time is very important. While the average search time is important for performance, the system can still fail or have performance degraded when a single search takes too long. For example, it is important to keep the time spent in interrupt handlers low and predictable. So even though the average time is important, the expected maximum time is important as well. There are well-known results in computer science regarding the expected maximum chain length. For example, if alpha is 1, the expected maximum chain length in a hash table with n entries is proportional to log(n)/loglog(n). To illustrate this behavior experimentally for a range of parameter values, I wrote a program that randomly and uniformly distributes entries to hash buckets and tracked the statistical behavior of the results.[41]

[40] Under the assumption of a hash function that uniformly and randomly distributes the entries across the chains.

[41] I note that this program is conservative in its estimates because it uses random number for the keys to achieve the desired uniform and random distribution of keys to hash buckets.

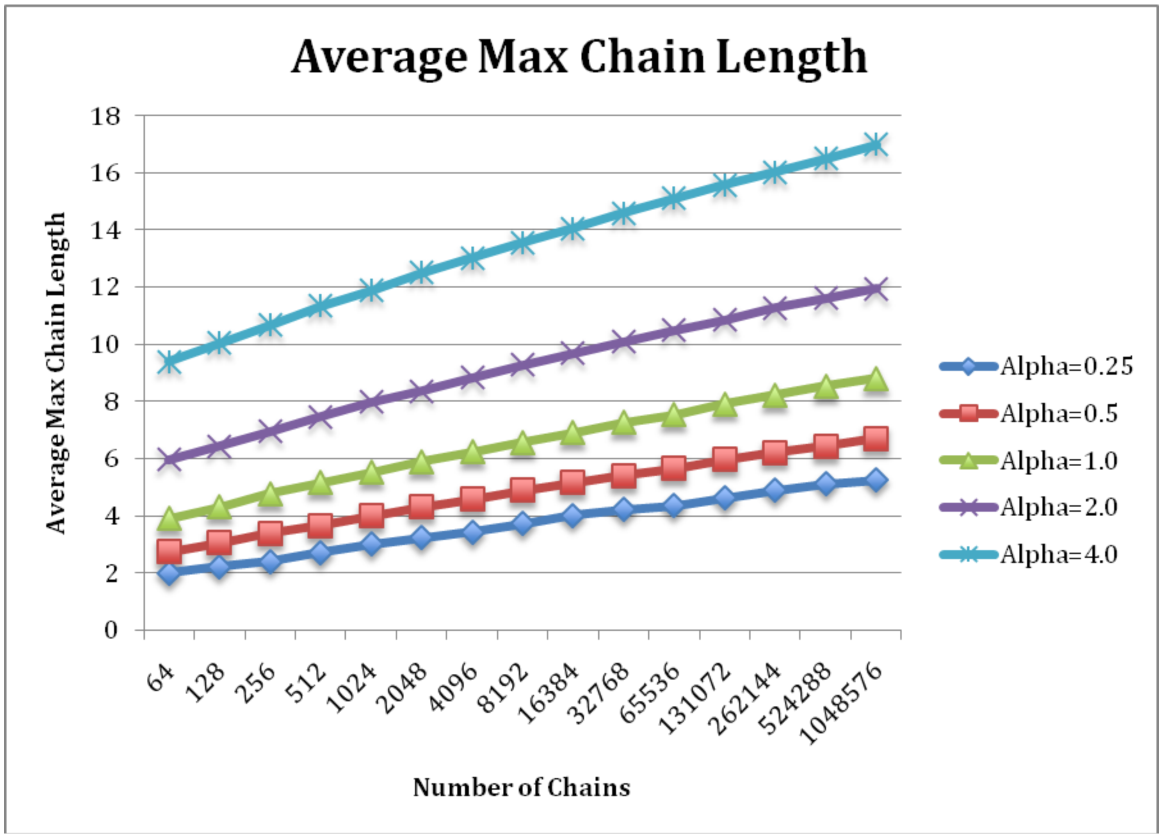

Figure 4 shows the average maximum chain length as a function of the number of chains for several different values of alpha; these results were computed from 1000 runs for each value of alpha. For the same conditions, Figure 5, Figure 6, and Figure 7 show the percentage of runs for which the maximum chain length exceeded a specific value. These results can be used to determine, for a given alpha and number of chains, the likelihood that at least one chain in a hash table will exceed a specified length. Note that even for alpha=1.0 (i.e., the average number of entries in a chain is 1), it is more likely than not that at least one chain will exceed a length of 8.

In the context of the Linux route cache, this result indicates that even when the number of entries is as desired (see rt_garbage_collection() in route.c, for example), the probability of at least one chain being longer than gc_elasticity (at the default value of 8) is high. Further, the ongoing operation of the system can be seen as a series of opportunities to "redo" the experiment and retest whether a chain exceeds gc_elasticity because (a) in several versions of the code, the hash function is changed regularly and the cache is effectively flushed and restarted and (b) in some contexts, there will be an ongoing series of new IP addresses to store in the cache. So, while the results in the Figures below indicate the likelihood of a single experiment, the operation over time will result in an increasing likelihood that at least one of the experiments will result in a chain that exceeds gc_elasticity. If each experiment is considered an independent event, which is a reasonable assumption if the hash function is being changed regularly, then the probability of at least one chain exceeding the specified length in N such experiments is (1 – (1-P(exceeding in one experiment)^N) which approaches one as N goes to infinity if the probability is non-zero. N will grow with the number of servers as well as over time. As an example, consider the case where the probability of a chain length exceeding gc_elasticity is 0.001, but that experiment is performed every ten minutes for 30 days (N=4320), then the probability of at least one experiment having a chain length exceeding gc_elasticity exceeds 0.98. Thus, I conclude that not only is the Linux route cache code designed to infringe the method claims of the '120 (as analyzed in this report), it is likely to do so when used over a long period of time (e.g., 30 days)[42] even for a relatively small number of chains (e.g., 32768) and a value of alpha as low as 0.5[43]. Obviously, as indicated above, the likelihood rises dramatically for higher values of alpha.

[42] Where either the hash function is regularly changing and the computer connects to a moderate number of computers (e.g., ¼ of the number of hash chains) or the mix of IP addresses is regularly changing such as what would occur at computers that are providing services (e.g., Web server) over the Internet that can be accessed by large numbers of computers.

[43] Or, for example, 262144 and alpha=1/3.

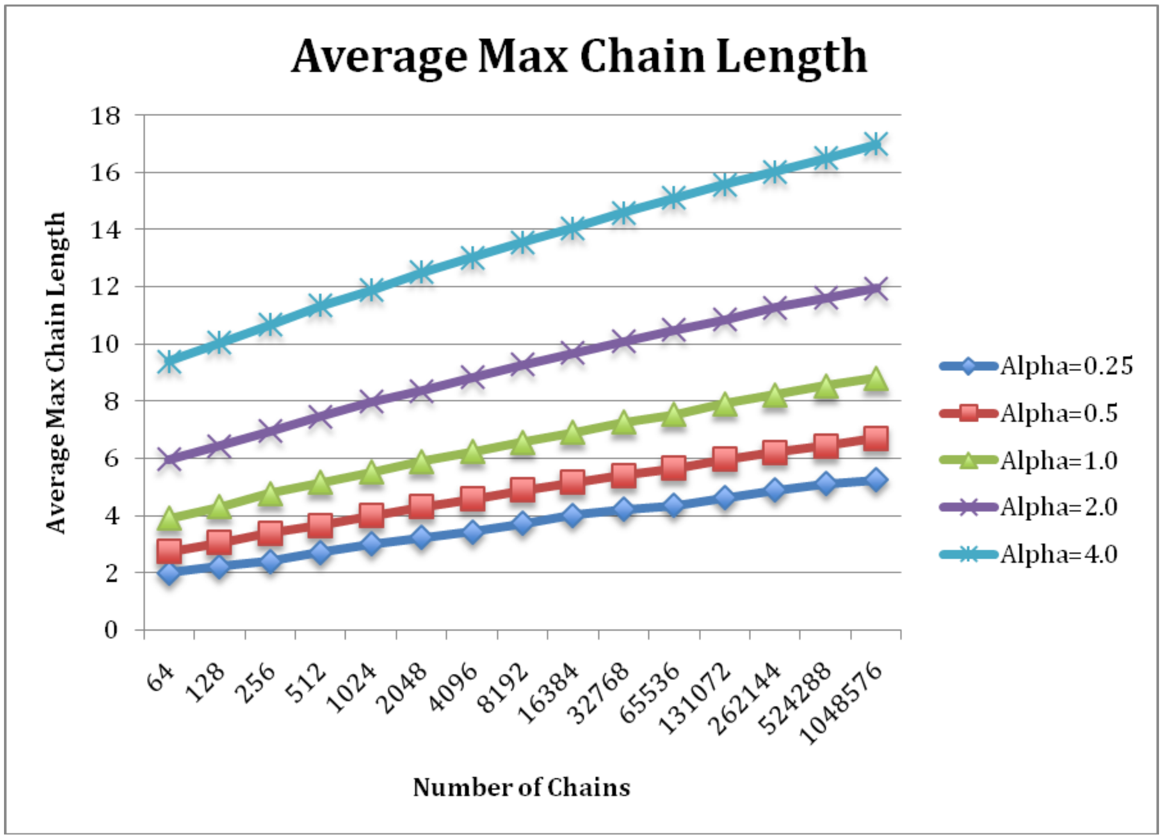

Figure 4: Experimental results from 1000 runs for each value of alpha

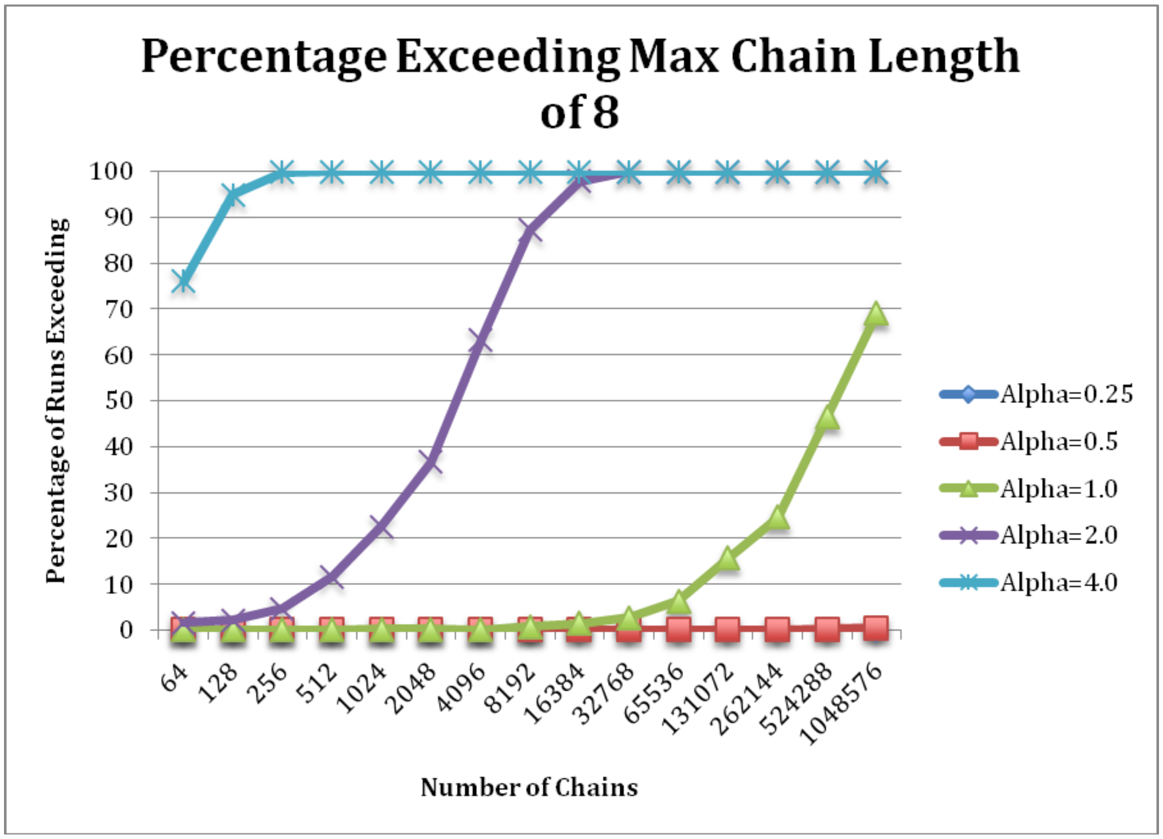

Figure 5: Percentage of experiments where the maximum chain length exceeds 8

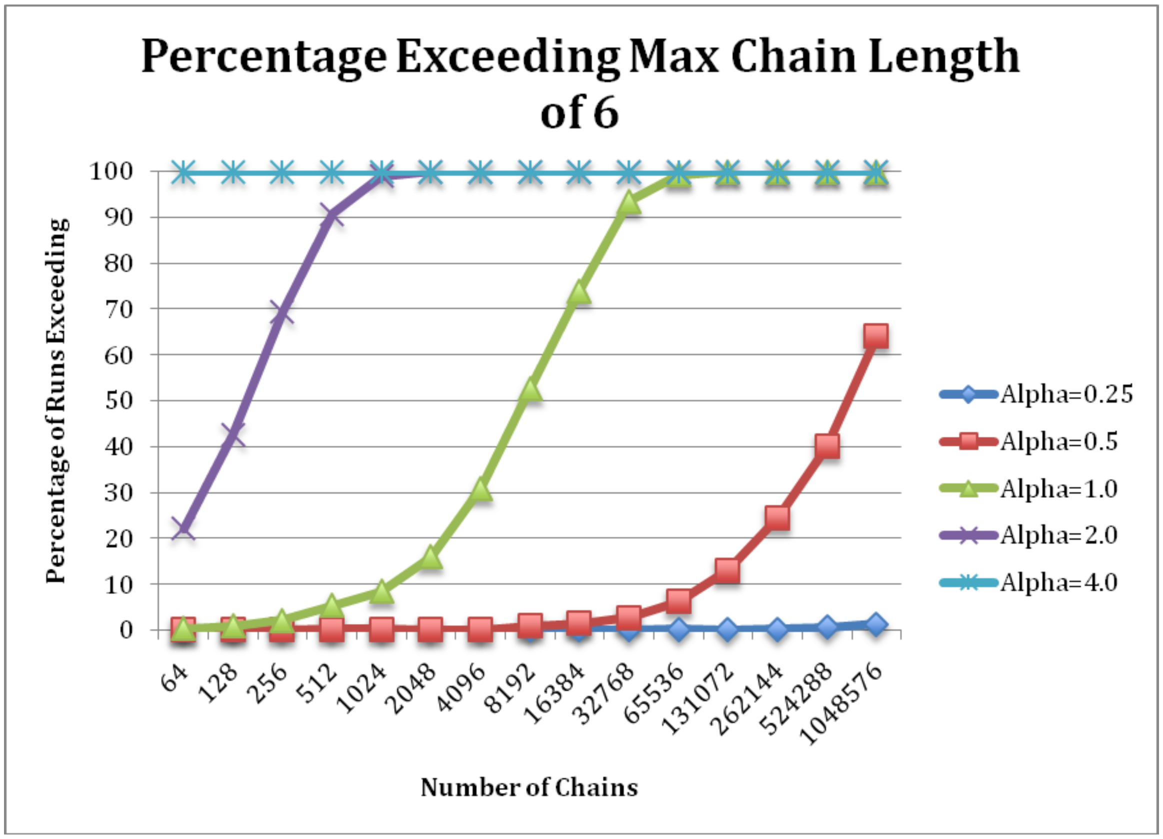

Figure 6: Percentage of experiments where the maximum chain length exceeds 6

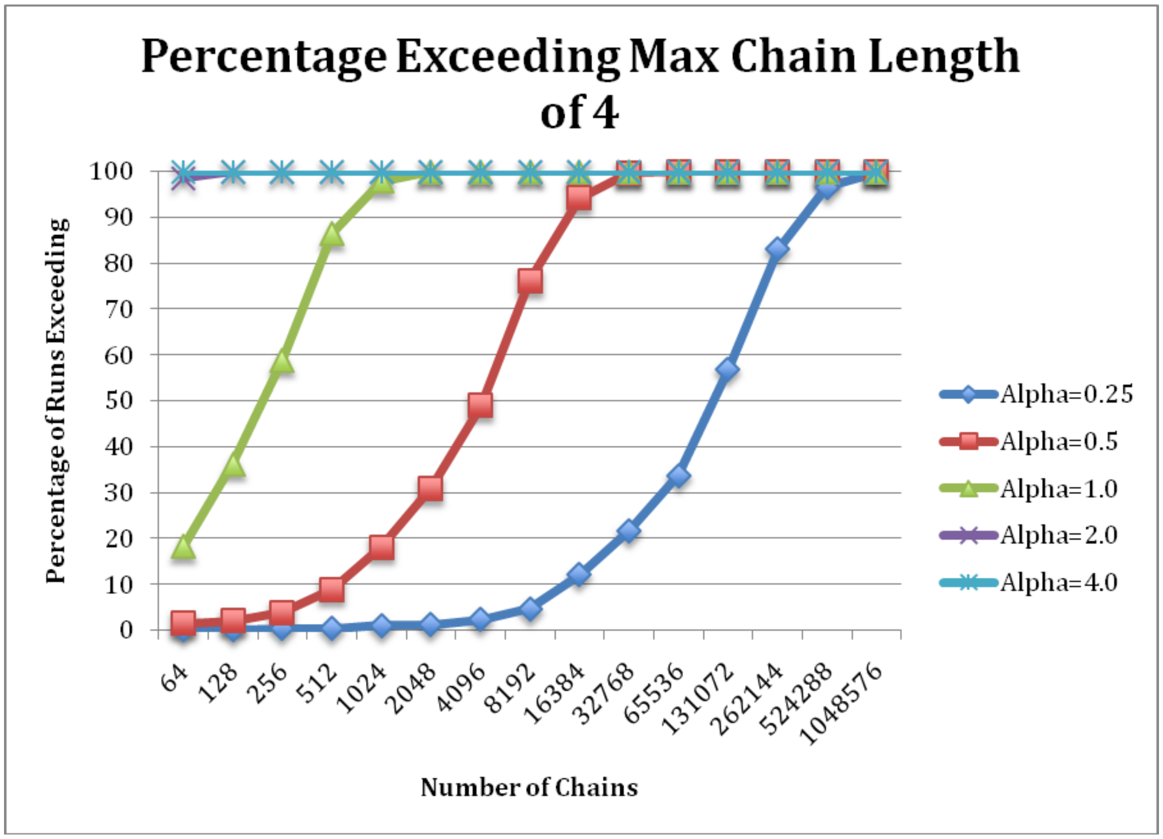

Figure 7: Percentage of experiments where the maximum chain length exceeds 4

In addition to those experiments, I performed experiments directly on the Linux kernel (v2.6.31 as noted above), where I modified the kernel to count the number of "candidate" deletions in *rt_intern_hash*(), the number of "genid" deletions in *rt_intern_hash*() and the number of times the cache was invalidated. These values were printed every time *rt_check_expire*() was called, which was 60 second intervals. For these experiments, used the modified version of hping3 and ran under a variety of load conditions using the method below:

(1) Start recording the log file
(2) Flush the cache
(3) Wait 5 seconds
(4) Start hping3
(5) Wait 300 seconds
(6) Flush the cache
(7) Wait 130 seconds
(8) Stop hping
(9) Retain the log file (counting only those entries after the first one indicating that the cache was flushed).

I collected these results under three route cache configurations: (1) *rhash_entries*=1048576, *gc_elasticity*=4, *secret_interval*=0 (2) *rhash_entries*=262144, *gc_elasticity*=4, *secret_interval*=0 and (3) *rhash_entries*=65536, *gc_elasticity*=8, *secret_interval*=600. In the first two cases, there is no periodic change of genid, so the only genid deletion will occur when the cache is flushed. This would be equivalent to the action that would occur when genid is periodically changed to invalidate the cache, but I have done so under controlled conditions in these two cases. In the third case, I am operating under the default settings on the server computer which includes periodic cache invalidations. In these results, I have not counted deletions due to the first cache flush and I have only counted cand deletions up to (but not including) the second cache flush.

These results, given in Appendix Q, indicate that, even for these short time periods, both cand deletion and genid deletion take place under a variety of conditions. Each time the cache was invalidated (either directly in my experiments or under the default conditions), the genid deletion was invoked. I note that in some cases, where the load is very light, no cand deletions were observed in this short time period which is consistent with the behavior described and analyzed above.

10.3.4 Sample traffic from a simple visit to a Defendant's web page

To examine the type of HTTP traffic that can be generated when accessing a Defendant's server, I visited pages at each defendant and examined the resulting traffic. I do not assert that this traffic is representative of all traffic at the Defendant's server, but rather this is traffic that can be generated in a simple visit to the Defendant's website. In the traffic summaries in Appendix R, I have highlighted in yellow the IP addresses which I have confirmed as being code is still present and can be quickly reactivated; thus, a Defendant can reap a benefit from even the disabled route cache by having a server than can rapidly respond to a new workload.

10.4.1 Information Specific for each Defendant

Based on my analysis of the Defendants' networks, which was limited by level of detail that the Defendants actually disclosed, I have not seen anything that has led me to conclude that any specific Defendant's systems are different enough to change the conclusions given above. My discussions of each Defendant's networks are attached as Appendices A-G to this report.

Executed on January 25, 2011

[illegible]

Mark T. Jones