EXHIBIT 16

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

TABLE OF CONTENTS

1. Introduction

1.1 Summary of Opinions

Based on my investigation in this matter, I have concluded that Softlayer Technologies, Inc. ("Softlayer"), Google Inc. ("Google"), Yahoo! Inc. ("Yahoo"), MySpace Inc. ("MySpace"), Amazon.com Inc. ("Amazon"), Match.com, Inc. ("Match.com"), and AOL Inc. ("AOL") have infringed and continue to infringe U.S. Patent No. 5,893,120 ("the '120 patent"). In this report, I explain how I arrived at this opinion as well as other opinions and findings.

1.2 Personal Background

I have been retained by Bedrock Computer Technologies LLC ("Bedrock"), and its counsel, McKool Smith P.C., as an expert in Bedrock's lawsuit as captioned above against Softlayer, Google, Yahoo, MySpace, Amazon, Match.com, and AOL.

I am a Professor of Electrical and Computer Engineering at Virginia Tech in Blacksburg Virginia. I graduated summa cum laude from Clemson University in 1986 with a B.S. in Computer Science and a minor in Computer Engineering while holding a National Merit Scholarship and the R. F. Poole Scholarship. I then graduated from Duke University in 1990 with a PhD in Computer Science while holding the Von Neumann Fellowship.

Upon graduation, I joined the Department of Energy at their Argonne National Laboratory facility. My responsibilities there included the design and use of software for computers with hundreds of processing elements. This software was designed for compatibility with new parallel computer architectures as they became available as well as with other large software components being written in the Department of Energy. While with DOE, I received the IEEE Gordon Bell Prize.

In 1994, I joined the Computer Science faculty at the University of Tennessee. My teaching responsibilities included computer architecture and computer networking. My research

interests included the design and use of software that used the collective power of large groups of workstations. While at the University of Tennessee, I received a CAREER Award from the National Science Foundation.

In 1997, I joined the Electrical and Computer Engineering faculty at Virginia Tech. My teaching responsibilities have included the design of embedded systems, computer organization, computer architecture, a variety of programming courses, and parallel computing. I have been cited multiple times on the College of Engineering's Dean's List for teaching.

In addition to the activities, education, and professional experience listed above, I have been involved in research projects that contribute to my expertise relating to this declaration. While at Virginia Tech, I have been a primary or co-investigator on government and industrial research grants and contracts in excess of five million dollars. As part of these efforts, I am a co-leader of a laboratory housing approximately thirty students performing research, over fifty computers, and hundreds of items of computing-related equipment.

The majority of the research contracts undertaken in the laboratory have involved collaboration and coordination with other groups to build a larger system. My responsibilities under the SLAAC project (a collaborative effort funded by the Defense Advanced Research Projects Agency involving the University of Southern California, Sandia National Laboratory, Los Alamos National Laboratory, Brigham Young University, UCLA, Lockheed-Martin, and the Navy) included the development of a software system for monitoring, configuring, and controlling a networked collection of computers hosting specialized computer hardware. As part of the DSN project (a collaborative effort funded by the Defense Advanced Research Projects Agency involving UCLA and USC), I was responsible for designing algorithms and software for controlling and monitoring a large network of autonomous computer sensor nodes. This software

was integrated with software from several other teams around the country for a set of field demonstrations over a three-year period.

In the TEAMDEC project for the Air Force Research Laboratory, I led an effort to design and construct a collaborative, Internet-based decision making system. This Java-based system provided a geographically diverse team with Internet-based tools to enable collaborative decision-making. On the server side, the system architecture made extensive use of database technology. This work was awarded first prize at the 2002 AOL/CIT Research Day.

Other projects have involved the close coupling of computer hardware and software, including the writing of device drivers and simple operating systems, the design of hardware circuits, the design of new system architectures, architectures for secure computing, the modification of complex operating systems, and software for mediating between complex software packages. A detailed record of my professional qualifications is set forth in the attached Exhibit 1, which is a curriculum vitae, including a list of publications, awards, research grants, and professional activities.

I am being compensated $400 per hour for my time spent working in connection with this case.

In the last four years, I have testified at deposition, hearing or trial in the following matters: (a) *Blackboard Inc. v. Desire2Learn Inc.*, No. 9:06-CV-155, in the Eastern District of Texas, before Judge Clark; (b) *TechRadium Inc. v. Blackboard Connect Inc.*, No. 2:08-cv-214, in the Eastern District of Texas, before Judge Ward; (c) *VirnetX, Inc. v. Microsoft Corporation*, No. 6:07-cv-080, in the Eastern District of Texas before Judge Davis; (d) *Bedrock Computer Techs. LLC v. Softlayer Inc. et al.*, No. 6:09-cv-269 in the Eastern District of Texas before Judges Love and Davis (claim construction deposition); (e) *DVSI v. University of Phoenix et al.*, No.

2:09-cv-555 in the Eastern District of Virginia; and (f) *TechRadium v. Twitter*, No. 4:09-cv-2490 in the Southern District of Texas.

1.3 The Organization of the ’120 Patent

Like other U.S. patents, the first page of the ’120 patent contains the title of the patent: “Methods and Apparatus for Information Storage and Retrieval Using a Hashing Technique with External Chaining and On-the-Fly Removal of Expired Data.” The first page also lists the inventor of the ’120 patent, Dr. Richard Michael Nemes as well as the abstract and a list of cited references (which will be supplemented by issuance of the certificate of reissues to include the references submitted to the USPTO in the re-examination).

Figures cited in the specification are given after the abstract. Following the pages with figures, there are numbered columns in the patent. In the ’120 patent, the numbered columns begin by restating the title of the patent. The ’120 patent then begins a section titled “Background of the Invention” in which the Dr. Nemes describes the state-of-the art in the year 1997. This is followed in the ’120 by a “Brief Summary of the Invention” section. Following this section, a brief description of each of the figures in the patent is given. Next, the ’120 patent contains a section entitled “Detailed Description of the Invention” which will include descriptions of “preferred embodiments” of the invention, which is what Dr. Nemes believed to be the best way to implement his invention.

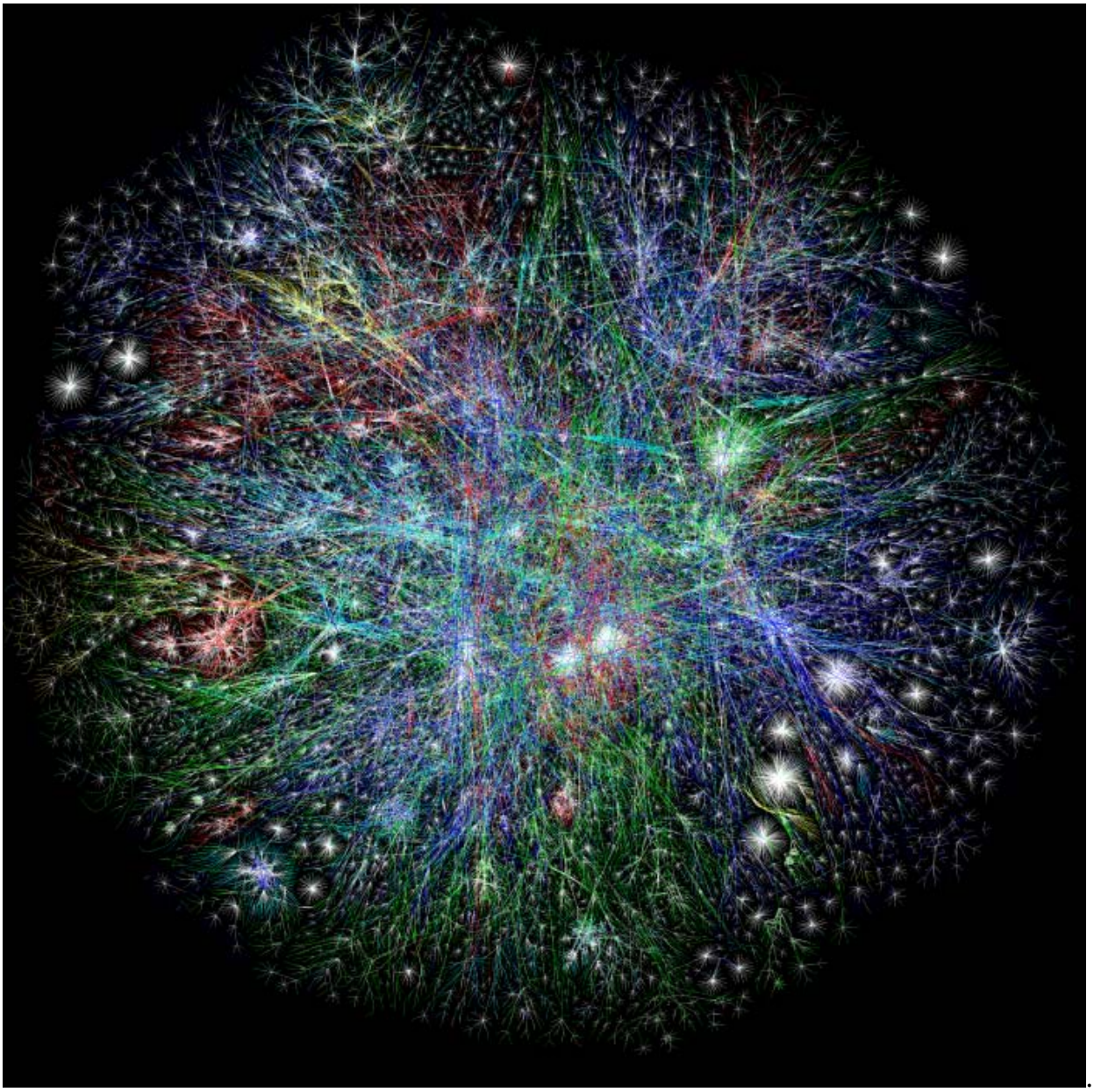
Following the detailed description of the inventions is pseudocode, which is something that computer programmers use to express algorithms in an informal notation to convey the important aspects of the functionality of the algorithm. Many programmers write their programs first in pseudocode in a way that looks like a mixture between English and formal code syntax. Programmers would then take the pseudocode and use it as a blueprint when writing the code completely in formal syntax. Once code is written in formal syntax, it can be compiled to object

code, which can be directly executed by a system's central processing unit (CPU). After the pseudocode in the '120 patent are numbered paragraphs that are the claims of the patents. It is here that Dr. Nemes expresses the boundaries of his intellectual property in terms of limitations, much like a deed expresses boundaries of real property in terms of geographic coordinates and points of reference.

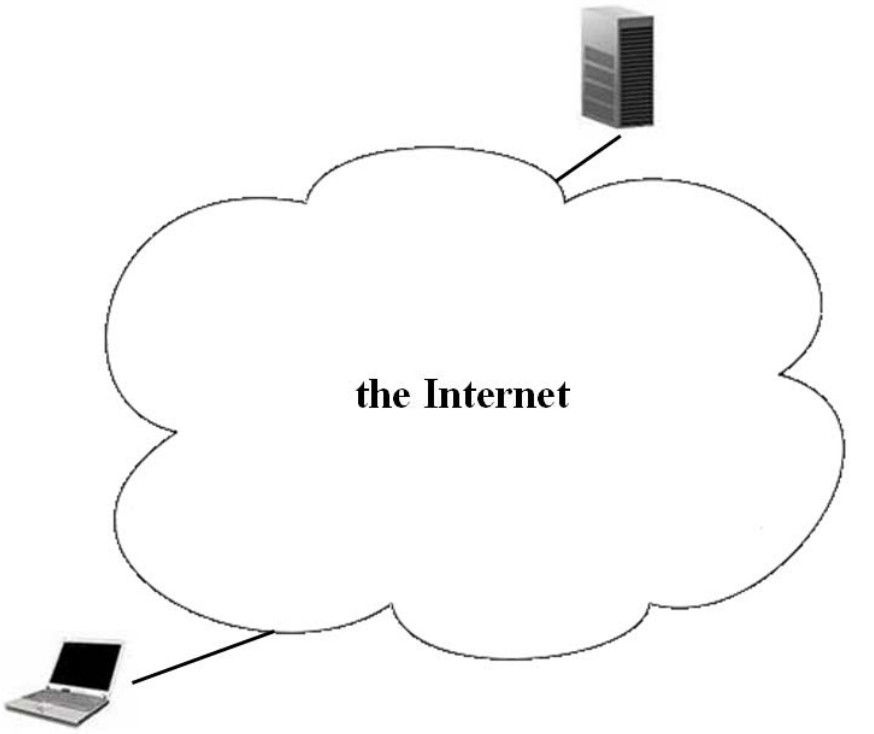
1.4 Technology Background

1.4.1 The Internet

The Internet is the worldwide collection of interconnected computer networks that communicate with each other using the TCP/IP suite of protocols. The Opte Project has rendered the connections between the many computer networks that comprise the Internet as a two dimensional, graphical representation:

 .

Understandably, the Internet between computers (including computers that are servers) is usually represented as a cloud in network diagrams:



The TCP/IP suite of protocols is way of organizing the communication data for a packet of information into layers embedded within each packet where each layer corresponds to an intended communication. The foremost protocol of the TCP/IP suite is the Internet Protocol (IP) which provides addressing systems, known as IP addresses, for computers on the Internet. IP addresses enable internetworking and establish the Internet itself. IP Version 4 (IPv4) was the version of the Internet Protocol to be widely implemented and is still the most dominant addressing scheme on the Internet today. Computers on the Internet know each other by IP addresses, and if a computer wants to talk to another computer, it transmits an IP packet to the Internet with the destination IP address specified in the header of that IP packet.

The Internet is often used synonymously with the World Wide Web, which is the collective body of web pages and web content on the Internet. Web pages are primarily written in html, which tells a user's web browser, such as Microsoft's Internet Explorer or Apple's Safari, how to display the content. Other content, such as videos, pictures, documents, or music,

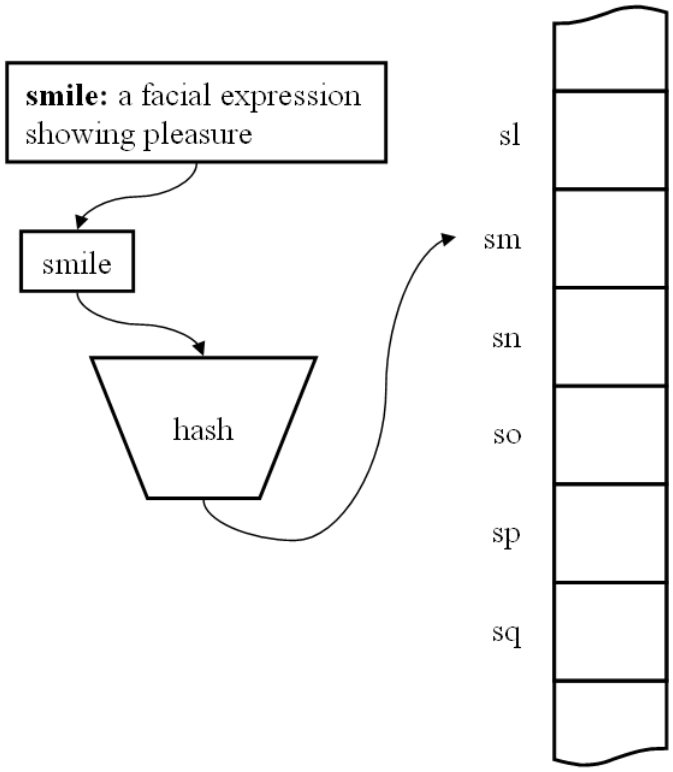
are digitally encoded by other schemes, such as xml, jpeg, mp3, pdf, etc. The content of a page on the World Wide Web—regardless of format—is placed in the payload of an IP packet and transmitted over the Internet. An IP packet is routed on the Internet primarily using the destination IP address of that packet, irrespective of the contents of the payload.

1.4.2 The Invention Claimed in the '120 Patent

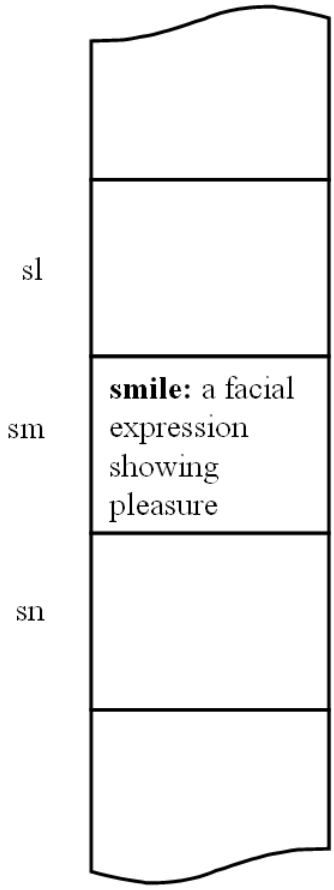
The '120 patent is directed to "on-the-fly removal of expired data" in an information storage and retrieval system. *See* '120::1:1-5. Though the patent addresses a general problem related to providing constantly-available, high-performance storage/retrieval operations while, at the same time, handing expiring data, *see* '120::2:22-26, the '120 patent's solution and claims are directed to a specific type of information storage and retrieval system—namely, one that (i) uses a hashing technique and (ii) uses external chaining. *See id*.

"Hashing" is the translation, via a mapping function (commonly known as hash or hashing function), of a record key value to a hash table array address. *See* '120::1:34-46; *see also* '120::4:53-62. Hashing is advantageous because it provides a very quick and efficient way to isolate the set of records that might be the searched-for key value. A problem with hashing, however, is that two different key values can often map to the same hash table address. This is known as a collision. *See id*.

As an example, a computer could act as a storage and retrieval system for a dictionary:



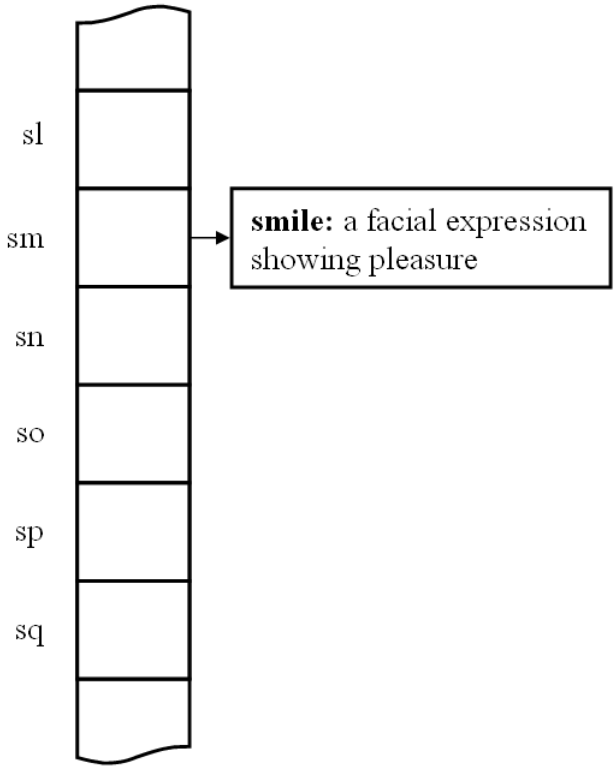
In this example, the record is the word “smile” along with its definition. The defined word is the key word, which input to the hash function, and the hash function could work by dropping all letters of the key word after the first two letters. The output of the hash function for the word “smile” would be “sm,” which is an address of the hash table. A difference between this example hash function and real hash functions is that real hash functions “are designed to translate the universe of keys into addresses uniformly distributed throughout the hash table.” See ’120::1:47-49. The example hash function does not satisfy that design goal since more words in English begin with sl, sm, sn, so, and sp rather than sq. An information storage and retrieval system uses the “sm” hash table address in its storage of the word “smile” and its definition, e.g.,:



Information storage and retrieval systems that use a hashing technique must somehow handle collisions so that collided records are not lost and are still retrievable using the hash function.

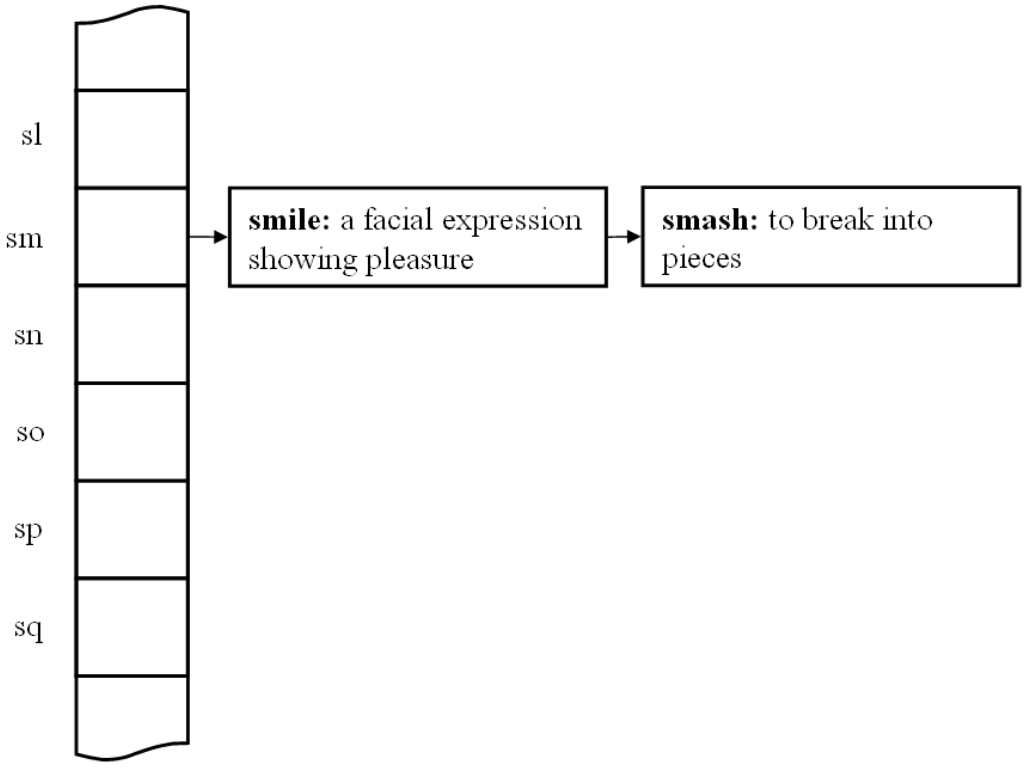
Going back to my dictionary example, a hash collision would occur if the computer were to store the of the word “smash” and its definition since “smash” would hash to the same hash value as “smile.” External chaining, in which linked lists chain from of the hash table, is one way to handle such collisions, *see* ’120:1:58-59; *see also* ’120::5:16-17, and the ’120 patent is directed exclusively to information storage and retrieval systems that use hashing with external chaining.

In a system that uses external chaining, a hash table entry does not contain the record itself; instead, the hash table entry contains a pointer to a linked list of records that have collided at that location. *See* '120::1:59-6; *see also* '120::5:16-20. Thus, when the information system is asked to store a new record, the record is inserted into a linked list that chains off of the hash table entry corresponding to the new record. In the dictionary example, the record "smile" would be stored as



,

and the collision of "smash" and "smile" would be handled as



The records "smile" and "smash" are connected by a pointer contained in the "smile" record. A pointer, in computer science, is "a variable that contains the memory location (address) of some data rather than the data itself." *See* Microsoft Computer Dictionary 4ed. (Microsoft Press, 1999). The hash table itself contains pointers to the first record in the linked list. The last record in the linked list contains a NIL pointer, which is a reserved sequence of bits, usually all zeros, to signify that there is no next record in the linked list.

In the example above, if the computer needed to retrieve the definition of the word "smash," the computer would first hash the word "smash," which would output "sm." Then, the computer would access the hash table array at the offset corresponding to the "sm" value. At that location is a pointer that is the address of the first record in the linked list. The computer would follow that pointer to the first record. The computer would then compare the key word, "smash," against the key word of that first record, which is "smile." Because "smile" does not

match “smash,” the computer would use the pointer in the “smile” record to go to the next record. The computer would then compare “smash” and “smash,” which indicates that it has found the searched-for record, and the computer would return the definition for the word “smash” to whoever requested it.

While accumulating collided records on linked lists solves the problem of hash collisions, the accumulation of records can, itself, become a problem.

Specifically, the ’120 patent teaches that some data records, after a limited period of time, become obsolete, and their presence in the storage system is no longer needed or desired. *See* ’120::2:7-10. If these expired records are not removed from the information system, they can seriously degrade the performance of the information system. *See* ’120::5:41-44. The ’120 patent describes two ways in which expired data can burden the performance of an information storage and retrieval system: (i) “the presence of expired records lengthens search times since they cause the external chains to be longer than they otherwise would be” and (ii) “expired records occupy dynamically allocated memory storage that could be returned to the system memory pool for useful allocation.” *See* ’120:5:44-49. As stated in the specification of the ’120 patent, the objective of the invention, then, is to “provide the speed of access of hashing techniques for large, heavily used information storage systems having expiring data and, at the same time, prevent the performance degradation resulting from the accumulation of many expired records.” *See* ’120::2:22-26.

Removal of unneeded records in an information system is commonly referred to as garbage collection. Before the ’120 patent, most system invoked a garbage collection routine when necessary, as part of a separate, standalone process. I might refer to these prior art garbage collection as “on-demand” or “standalone” garbage collection, in contrast to “on-the-fly”

garbage collection. The biggest disadvantage to on-demand garbage collection routines is that they required that the system effectively be taken off-line while they execute. *See* '120::2:64-67. In contrast, the garbage collection technique disclosed and claimed in the '120 patent executes "on-the-fly," that is, while other types of access to the linked lists take place. *See* '120::2:54-63. As the patent emphasizes, on-the-fly garbage collection "has the decided advantage of automatically eliminating unneeded records without requiring that the information storage system be taken off-line for such garbage collection. This is particularly important for information storage systems requiring rapid access and continuous availability to the user population." *See* '120::2:64-3:3.

1.4.3 The Linux Operating System

Linux is software. Software is a set of instruction or commands that tell a computer to do something. As stated above, software is usually expressed and understood in its source code form, but the source code must be compiled into object code so that it can work with a system's CPU. Linux is a specific category of software called an operating system, like Microsoft Windows. An operating system is the software that controls the hardware of a computer, such as the memory, the CPU, and peripheral devices. An operating system provides an environment on which software applications run, so that each application does not have to know how to control the computer hardware itself. Examples of applications are Microsoft Word and Microsoft Excel, which directly interact with a computer user. Other applications indirectly interact with a computer user, such as applications that serve requested web pages or web content. Also, some applications do not interact with a computer user at all but instead interact with other computers, such as applications that perform load balancing. The ability of applications to perform their tasks depends on the ability of an operating system to receive instructions from the application and in turn direct the computer hardware to perform the appropriate function.

Linux is an operating system that began essentially as a rewrite of an operating system called Unix. Linux 1.0 was released in 1994. *See* http://www.linux.org/info/index.html. Linux is what is known as "open source" software, which means that anyone can obtain the source code for Linux. This is in contrast to closed source software. A well-known example of closed source software is Microsoft Windows. Anyone can buy the object code for Microsoft Windows, but Microsoft keeps the source code for Windows concealed from the public. There is an official release of the Linux kernel maintained at kernel.org, which is controlled by The Linux Kernel Organization, Inc. (The kernel of an operating system is the core of the operating system. It is the system that manages memory, files, and allocates system resources.)

Because Linux is open source software, anyone can review the code and suggest changes or patches to it. "Since 2005, over 6100 individual developers from over 600 different companies have contributed to the kernel." *See* BTEX0752238. Although anyone can participate in this process, there is a very organized way in which submitted changes to the kernel are reviewed:

> Patches do not normally pass directly into the mainline kernel; instead, they pass through one of over 100 subsystem trees. Each subsystem tree is dedicated to a specific part of the kernel (examples might be SCSI drivers, x86 architecture code, or networking) and is under the control of a specific maintainer. When a subsystem maintainer accepts a patch into a subsystem tree, he or she will attach a "Signed>off>by" line to it.

See BTEX0752238. Also, it is worth noting that:

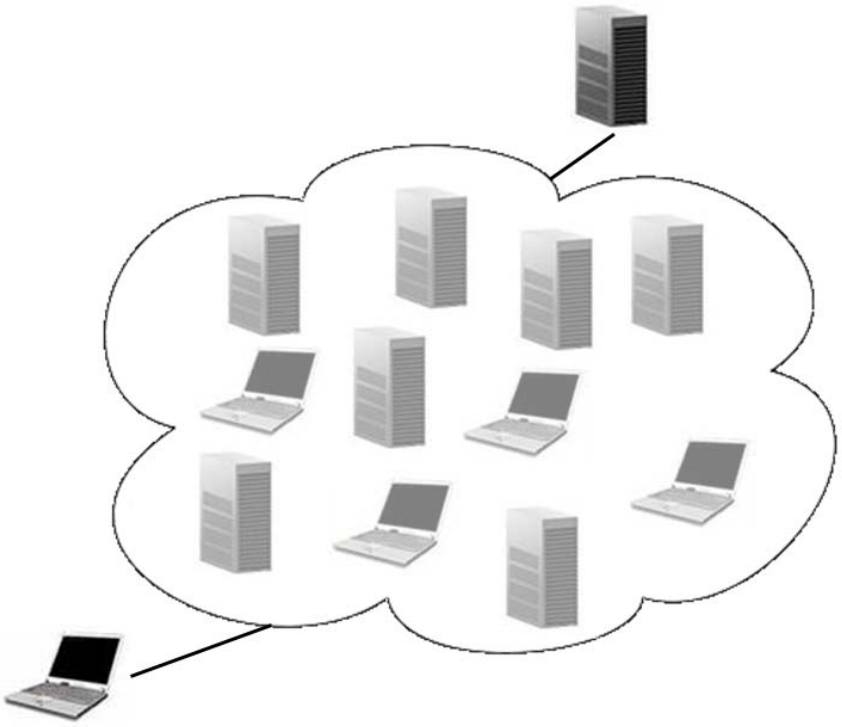
> Despite the large number of individual developers, there is still a relatively small number who are doing the majority of the work. In any given development cycle, approximately 1/3 of the developers involved contribute exactly one patch. Over the past 5.5 years, the top 10 individual developers have contributed 10% of the total changes and the top 30 developers have contributed almost 22% of the total.

See BTEX0752238. Moreover, a contributor to the Linux kernel is considered affiliated with the corporation he or she works for. *See* BTEX0752238. It is worth noting that contributors with no corporate affiliation contribute 18.9% of the changes to the Linux kernel, and no corporation contributes more. For comparison, contributors working for Google only contribute 0.8% of the changes, whereas contributors affiliated with universities, such as professors, contribute 1.3% of the changes. *See* BTEX0752238. The other Defendants contribute so few changes to the Linux kernel that they do not appear in the list of contributors, where the lowest contributor in the list contributes just 0.6%. I may refer to the collection of individuals who participate in submitting or review patches to the Linux kernel, irrespective of the individual's affiliation, as the "Linux community" or the "Linux development community." As of the time of this report, the latest release of the Linux kernel is version 2.6.37.

1.4.4 IP Packet Routing and the Routing Cache in Linux

The routing cache part of the Linux kernel. "The main job of the cache is to store information that allows the routing subsystem to find destinations for packets, and to offer this information through a set of function to higher layers." *See Understanding Linux Network Internals* (O'Reilly 2006) at 861 (DEF00008677). Like the dictionary example, the routing cache in Linux is an information storage and retrieval system. But instead of storing definitions of words, the records in the routing cache store connection information.

Revisiting the discussion of the Internet, an IP packet contains a source and destination IP address. When an IP packet is transmitted, many computers are involved in routing that IP packet to the correct destination:

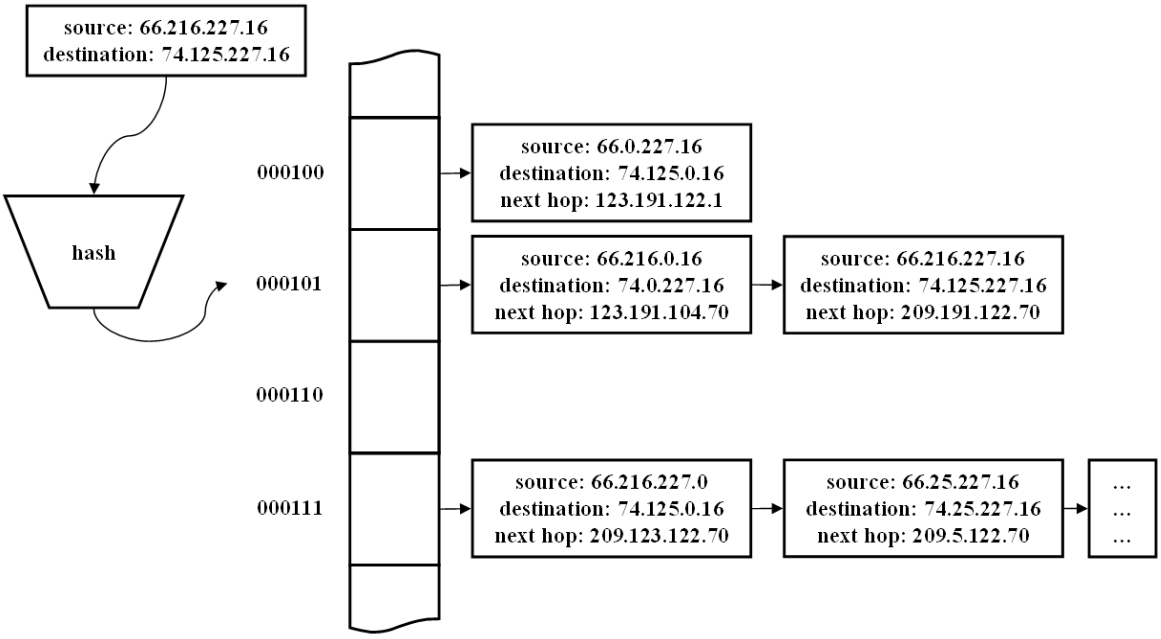


.

While a computer may not talk directly to the computer associated with the destination IP address of a particular computer, computers on the Internet know the appropriate next hop for an IP packet based on where the IP packet is trying to go:

.

The routing cache in Linux contains next hop information as well as other network connection information. The routing cache uses hashing with external chaining:



.

1.4.5 On-Demand Garbage Collection Routines in Linux

1.4.5.1 Overview of the On-Demand Garbage Collection Routines in the Linux Routing Cache

In the Linux routing cache, there are three on-demand garbage collection routines. These routines are designed to keep the routing cache from growing in size without bounds.

The first routine is named *rt_garbage_collect()*. It potentially can run each time a new routing cache entry is allocated. Based on the number of entries in the routing cache and when the routine was last executed, *rt_garbage_collect()* will delete old and expired cache entries from the routing cache. On a typical system, *rt_garbage_collect()* will execute only twice a second and only if there are relatively large number of entries in routing cache.

The second garbage collection routine of the Linux routing cache is named *rt_check_expire()*. This routine runs in the background and also deletes old and expired cache entries. By default, *rt_check_expire()* runs much less frequently than *rt_garbage_collect()* at once a minute. However, *rt_check_expire()* is less sensitive to the size of the routing cache.

The third garbage collection routine of the Linux routing cache is named *rt_run_flush()*. The routine is very basic. It walks the entire routing cache, freeing all entries of the routing cache at once.

All three garbage collection routines, *rt_garbage_collect()*, *rt_check_expire()*, and *rt_run_flush()*, suffer from the same problem: When they execute, they can consume considerable CPU cycles all at one time. For a real-time system, such as the Linux TCP/IP stack, this means that the processing of real-time events can be blocked while a routine executes. With the Linux TCP/IP stack, this problem can cause incoming network packets to be missed by the TCP/IP stack and lost, requiring costly packet retransmissions.

Performance problems are especially acute with rt_run_flush since it deletes the entire routing cache at once. The Linux community came to realize the problems that rt_run_flush causes and the routine was eliminated starting in Linux version 2.6.25. In this and subsequent releases, flushing of the entire routing cache was instead done by the introduction of a type of on-the-fly garbage removal, discussed below.

In addition, classic garbage collectors do not deal very well with DoS attacks or DoS-like traffic that creates very long cache chains. The issue is that the classic garbage collectors do not execute frequently enough to deal the problem (or, if they do, the cost in CPU resources is high).

1.4.5.2 rt_garbage_collect()

The routine *dst_alloc()* is called by the Linux TCP/IP stack to allocate a new routing cache entry. The routine *dst_alloc()* also includes a check for the routing cache becoming relatively large in which case the cache becomes a candidate for garbage collection. *dst_alloc()* checks to see if the current number of entries in the cache exceeds a garbage collection threshold. By default this threshold is set to the number of buckets in the cache. If the threshold is exceeded, then a protocol-dependent garbage collector routine is called. For IPv4, this routine is named *rt_garbage_collect()*.

Because garbage collection can be a very expensive operation in terms of CPU time, *rt_garbage_collect()* includes additional tests that attempt to limit how often garbage collector executes and how many routing cache entries are freed during one run of the garbage collector.

For example, if the garbage collector has run recently, *rt_garbage_collect()* will not run it again unless the routing cache has become very large. The global variable that controls the minimum amount of time between two garbage collections is named *ip_rt_gc_min_interval*. By default this number is set to half of a second.

The global variable named *ip_rt_max_size* will override this minimum. It specifies a maximum size for the routing cache where garbage collection should always be done. By default, this limit is set to 16 times the number of buckets in routing cache.

If a garbage collection is to be done by *rt_garbage_collect()*, the routine then calculates the number of cache entries to be expired in the one run of the garbage collector. This entry count is stored in the local variable named "goal".

The routine deals with two basic scenarios. If the routing cache has become very large—by default more than 8 times the number of hash buckets—then the garbage collector becomes aggressive and attempts to free up many entries at once. By default the number of entries to be freed in this scenario will be the number of hash buckets or half of the number of buckets in excess of 8 times the number of hash buckets, whichever is greater. The 8-times limit is controlled by the *ip_rt_gc_elasticity* parameter.

In the alternate scenario, where the number of entries in the routing cache is less than 8 times of number of hash buckets, the number of entries to be garbage collected is much smaller. *rt_garbage_collect()* also attempts to spread out the number of entries between multiple garbage collections using the variable named "equilibrium".

The actual garbage collection code walks the external chains of sequential hash buckets looking for expired routing cache entries. An expired entry is removed from a hash chain and its memory freed by calling *rt_free()*. The garbage collection process stops when the garbage collection goal has been reached or when all routing cache entries have been examined.

So that the garbage collection process occurs evenly distributed over the hash table over time, *rt_garbage_collect()* maintains a static variable named "rover" that remembers that last

bucket list which was examined. The garbage collector will restart in its next run at the next bucket pointed to be rover.

It is possible that the goal for freeing routine cache is not met. In this case, the global variable *gc_goal_miss* is incremented by one. If the number of entries in the routing cache is still 16 times the number hash buckets, then the garbage collector considers the routing cache overflowed and also increments the global variable *gc_dst_overflow*. It may also attempt to do garbage collection again.

The following table summarizes the conditions for garbage collection of the Linux routing cache:

n = Number of entries in the routing cache

B = Number of hash buckets in the routing cache

e = Elasticity (default is 8)

Rating	Condition	Action	Frequency
"Green"	n < B	No GC	Never
"Yellow"	n >= B and n < e * B	Modest GC	Every .5 sec.
"Orange"	n >= e * B and n < 16 * B	Aggressive GC with goal = max(B, (n – B *e)/2)	Every .5 sec.
"Red"	n >= 16 * B	Aggressive GC with goal = max(B, (n – B *e)/2)	Immediate

1.4.5.3 rt_check_expire()

The routing code also runs a background process that calls the routine *rt_check_expire()* that periodically deletes expired cache entries. The frequency that *rt_check_expire()* runs at is controlled by the variable *ip_rt_gc_interval*. The default value for this variable is 60 seconds.

The maximum number of expired entries deleted by *rt_check_expire()* in a single run is dynamic and is determined by the size of the cache and how often *rt_check_expire()* runs.

rt_check_expire() does not delete all expired cache entries in a single run in order to not use too much CPU time all at once.

The code for *rt_garbage_collect()* and *rt_check_expire()* are similar. Both functions are designed to discard outdated routing cache entries. There are three differences between the routines:

1. *rt_garbage_collect()* is called only when a new entry needs to be added to the routing cache. However, in most cases, *rt_garbage_collect()* does nothing when it is called. By default, it only executes twice a second when the number of routing cache entries goes above a threshold. On the other hand, *rt_check_expire()* runs once a minute (by default) regardless of the number entries in the routing cache.

2. *rt_garbage_collect()* is designed to free up a certain number of routing cache entries. The number of entries is dependent on the size of the routing cache and how active the garbage collector has been in the past. On the other hand, *rt_check_expire()* runs for a fixed amount of time, freeing up as many outdated entries that it can find.

3. *rt_check_expire()* discards routing cache entries that have been marked to expire after a certain amount of time. In contrast, *rt_garbage_collect()* does not deal with entries that are marked for expiration by time.

1.4.5.4 rt_run_flush()

The implementation of the rt_run_flush routine is straight-forward. The routine simply visits all buckets of the routing cache and deletes all cache entries of each bucket list. Execution times for rt_run_flush are relatively long since execution times are a function of the number of buckets in a routing cache plus the number of entries in the cache at the time of the flush operation.

1.4.5.5 Changes to rt_garbage_collect() Over Time

Between Linux kernel versions 2.4.20 and 2.6.35 (the latest Linux kernel release as of the summer of 2010), there have been only small modifications made to *rt_garbage_collect()*, none of which changed significantly how the routines operates. The changes of note are:

1. In kernel version 2.6.12, code was added for multipath caching. This code was then removed in version 2.6.23.1. This code was conditioned compiled if the #define symbol CONFIG_IP_ROUTE_MULTIPATH_CACHED

2. Beginning in kernel version 2.6.25, *rt_garbage_collect()* frees up entries that have *rt_genid*'s which are no longer current.

2. How the Linux Kernel Came to Infringe the '120 Patent

On-the-fly garbage collection was added to the Linux routing cache code in 2003 to solve a serious performance problem when the Linux routing cache was operating under heavy load. The problem was the on-demand garbage collection routines in Linux, discussed in § 1.4.5. Specifically, under heavy TCP/IP loads, servers running Linux were performing poorly and dropping TCP/IP packets. The Linux community traced the problem to the routing cache garbage collector, which was keeping the routing cache locked for long periods of time during high TCP/IP loads.

The routing cache performance problem is memorialized in a Linux-net email list of Linux developers, which is publicly available. The message thread was entitled "Route cache performance under stress." Someone with the forum handle "CIT/Paul" identified the problem:

> Try forwarding packets generated by juno-z.101f.c and it adds EVERY packet to the route cache.. Every one. And at 30,000 pps It destroys the cache because every single packet coming in is NOT in the route cache because it's random ips. Nothing you can do About that except make the cache and everthing related to it wicked faster, OR remove the per packet additions to the cache.

See BTEX0748686-88 (http://marc.info/?l=linux-net&m=105513656320859&w=2).

David Miller, a Red Hat employee, first attempted to solve the problem by shortening the length of a hash chain in an independent routine, rt_hash_shrink. Seeing Mr. Miller's work, Robert Olsson suggested that the same idea be imported into rt_intern_hash. *See* BTEX0751091. Mr. Miller agreed and relayed the on-the-fly garbage collection solution to the routing cache performance problems to the thread identifying the problem:

> Here is a simple idea, make the routing cache miss case steal an entry sitting at the end of the hash chain this new one will map to. It only steals entries which have not been recently used.

See BTEX0748689.

> My main current quick idea is to make rt_intern_hash() attempt to flush out entries in the same hash chain instead of allocating new entries.

See BTEX0748690.

> We have to walk the entire destination hash chain _ANYWAYS_ to verify that a matching entry has not been put into the cache while we were procuring the new one. During this walk we can also choose a candidate rtcache entry to free.

See BTEX0748691-92.

> The problem is that GC cannot currently keep up with DoS like traffic pattern. As a result, routing latency is not smooth at all, you get spikes because each GC run goes for up to an entire jiffie because it has so much work to do. Meanwhile, during this expensive GC processing, packet processing is frozen on UP system.

See BTEX0748693-94.

In this way, the Linux community in 2003 identified that on-demand garbage collection routine in Linux was the essentially taking servers off-line, just as described by Dr. Nemes in the '120 Patent in 1999. To solve the problem of On-the-fly garbage collection was first added to *rt_intern_hash()* in Linux kernel version 2.5.72, which is a development version, in June 2003 by David Miller and Alexey Kuznetsov, another Linux networking developer. The changes became

a permanent part of the Linux kernel beginning on version 2.4.22 and going forward. I refer to this type of on-the-fly removal as the "candidate" removal.

In 2008, another type of on-the-fly garbage removal was implemented into the Linux kernel. Eric Dumazet proposed adding this type of on-the-fly garbage removal, which I refer to as the "genid" removal, to the Linux community:

> Current ip route cache implementation is not suited to large caches.
>
> We can consume a lot of CPU when cache must be invalidated, since we currently need to evict all cache entries, and this eviction is sometimes asynchronous. min_delay & max_delay can somewhat control this asynchronism behavior, but whole thing is a kludge, regularly triggering infamous soft lockup messages. When entries are still in use, this also consumes a lot of ram, filling dst_garbage.list.
>
> A better scheme is to use a generation identifier on each entry, so that cache invalidation can be performed by changing the table identifier, without having to scan all entries.
>
> No more delayed flushing, no more stalling when secret_interval expires.
>
> Invalidated entries will then be freed at GC time (controled by ip_rt_gc_timeout or stress), or when an invalidated entry is found in a chain when an insert is done.
>
> Thus we keep a normal equilibrium.
>
> This patch :
>
> - renames rt_hash_rnd to rt_genid (and makes it an atomic_t)
> - Adds a new rt_genid field to 'struct rtable' (filling a hole on 64bit)
> - Checks entry->rt_genid at appropriate places :
> --- Readers have to ignore invalidated entries.
> --- Writers can delete invalidated entries.
> - Removes rt_flush_timer timer
> - Removes unused /proc/sys/net/ipv4/{min_delay,max_delay}

See BTEX0750859. David Miller commented that Mr. Dumazet suggestion, "looks really nice" and implemented the patch into the Linux kernel beginning with version 2.6.25. *See* BTEX0750869; *see also* KTS0000244.

While the messages that I specifically discuss are the key messages that evidence the history of how Linux came to infringe the '120 patent, I have reviewed other documents that place these discussions in full context.[1]

3. Preface to My Infringement Analysis

[1] Specifically, documents BTEX0748686, BTEX0748689, BTEX0748690, BTEX0748691, BTEX0748693, BTEX0748695, BTEX0750801, BTEX0750804, BTEX0750806, BTEX0750808, BTEX0750810, BTEX0750812, BTEX0750814, BTEX0750815, BTEX0750818, BTEX0750820, BTEX0750821, BTEX0750822, BTEX0750823, BTEX0750824, BTEX0750826, BTEX0750828, BTEX0750830, BTEX0750837, BTEX0750845, BTEX0750848, BTEX0750850, BTEX0750852, BTEX0750854, BTEX0750856, BTEX0750858, BTEX0750871, BTEX0750874, BTEX0750877, BTEX0750878, BTEX0750879, BTEX0750881, BTEX0750882, BTEX0750883, BTEX0750884, BTEX0750885, BTEX0750887, BTEX0750889, BTEX0750890, BTEX0750891, BTEX0750893, BTEX0750894, BTEX0750895, BTEX0750896, BTEX0750897, BTEX0750898, BTEX0750900, BTEX0750901, BTEX0750902, BTEX0750903, BTEX0750906, BTEX0750907, BTEX0750909, BTEX0750910, BTEX0750911, BTEX0750912, BTEX0750913, BTEX0750914, BTEX0750916, BTEX0750917, BTEX0750918, BTEX0750919, BTEX0750920, BTEX0750921, BTEX0750922, BTEX0750923, BTEX0750924, BTEX0750925, BTEX0750931, BTEX0750932, BTEX0750933, BTEX0750934, BTEX0750935, BTEX0750936, BTEX0750940, BTEX0750942, BTEX0750946, BTEX0750947, BTEX0750949, BTEX0750957, BTEX0750958, BTEX0750959, BTEX0750984, BTEX0750985, BTEX0750987, BTEX0750988, BTEX0750990, BTEX0750991, BTEX0750992, BTEX0750994, BTEX0750995, BTEX0750997, BTEX0751000, BTEX0751001, BTEX0751002, BTEX0751004, BTEX0751005, BTEX0751006, BTEX0751008, BTEX0751009, BTEX0751012, BTEX0751013, BTEX0751016, BTEX0751017, BTEX0751019, BTEX0751020, BTEX0751021, BTEX0751022, BTEX0751023, BTEX0751024, BTEX0751025, BTEX0751026, BTEX0751027, BTEX0751028, BTEX0751029, BTEX0751030, BTEX0751031, BTEX0751032, BTEX0751033, BTEX0751034, BTEX0751035, BTEX0751036, BTEX0751039, BTEX0751040, BTEX0751041, BTEX0751042, BTEX0751043, BTEX0751044, BTEX0751045, BTEX0751046, BTEX0751047, BTEX0751048, BTEX0751049, BTEX0751050, BTEX0751051, BTEX0751052, BTEX0751053, BTEX0751054, BTEX0751055, BTEX0751056, BTEX0751057, BTEX0751058, BTEX0751059, BTEX0751060, BTEX0751061, BTEX0751062, BTEX0751063, BTEX0751064, BTEX0751065, BTEX0751066, BTEX0751069, BTEX0751070, BTEX0751071, BTEX0751072, BTEX0751078, BTEX0751079, BTEX0751080, BTEX0751081, BTEX0751082, BTEX0751083, BTEX0751085, BTEX0751086, BTEX0751089, BTEX0751090, BTEX0751091, BTEX0751092, BTEX0751093, BTEX0751094, BTEX0751095, BTEX0751100, BTEX0751115, BTEX0751119, BTEX0751120, BTEX0751121, BTEX0751123, BTEX0751176, BTEX0751214, BTEX0751219, BTEX0751533, BTEX0751536, and BTEX0751538.

3.1 Organization

I have organized my analysis into two sections. In the first section, I explain why the candidate removal in the Linux kernel versions 2.4.22 and onwards infringes the claims of the '120 patent. In the second section, I explain why the genid removal in the Linux kernel versions 2.6.25 and onwards infringes the claims of the '120 patent. I have reviewed numerous versions of the Linux kernel and have found that, while there have been many changes to other parts of the kernel, the changes to the code responsible for the candidate removal and the genid removal has only been changed superficially, that is, the functionality has not substantively changed since the introduction of the functionality into the kernel. I have also reviewed many distributions, including the distributions used by the Defendants, and the modifications that Google and Yahoo have made to the Linux kernel. With the exception of Google, the functionality of the Linux kernel that infringes the claims of the '120 patent has not substantively changed since the introduction of the functionality into the kernel. The names of the variables, routines, and data structures that I specifically discuss in this report is come from the official Linux kernel version 2.6.31, but my opinions for the candidate removal apply to Linux versions 2.4.22 and onward, and my opinions for the genid removal apply to Linux versions 2.6.25 and onward. My use of the word "Linux" in these sections applies to these associated Linux versions respectively. Some of the names of the variables, routines, and data structures that I discuss in this report have been changed across the various versions of Linux, but the core functionality has not. Further, it should be readily apparent to the Defendants (at least their experts) what I am referring to.

3.2 Hardware Components

My analysis does not detail the hardware components of the Defendants' servers or computers for two reasons: (i) it is my opinion that computers and servers meet these limitations; and (ii) no Defendant contested that these aspects of the structure for 112 ¶ 6 claim terms were

missing. Specifically, all 112 ¶ 6 claim terms require CPU 10, RAM 11, portions of application software, user access software or operating system software. *See* Claim Construction Opinion (Dkt. No. 369) at 27 n.27. CPU and RAM are fundamental hardware components of any computer. *See* '120::3:52-56 ("Figure 1 of the drawings shows a general block diagram of a computer hardware system comprising a Central Processing Unit (CPU) 10 and a Random Access Memory (RAM) unit 11. Computer programs stored in the RAM 11 are accessed by CPU 10 and executed, one instruction at a time, by CPU 10.") Additionally, the Linux operating system is an operating system as operating systems are described in the '120 patent in that the Linux operating system "coordinates the activities of all of the hardware components of [a] computer system and provides a number of utility programs." *See* '120::4:30-34.; *see also* DEF00008677. If any Defendant supplements its response to Bedrocks' noninfringement interrogatory to include new noninfringement theories on this basis, I reserve my right to supplement this analysis. I also note that, while the source code does not run on any of the Defendants' servers, the source code completely describes and expresses the functionality of object code running on the Defendants' servers and computers.

3.3 Discussions of the Functionality of the Linux Code

This report focuses on the rationale underlying my opinions of infringement. When I refer to the Linux source code, I incorporate by reference the source necessary to understand the code, including the definitions and invocations of the referenced source code as well as the analogous functionality across the versions of Linux. I also incorporate by reference Bedrock's infringement contentions into this report.

3.4 Equivalence Theories

My use of the word "equivalence" is in reference to statutory equivalence under 35 U.S.C. § 112 ¶ 6, and I have added alphanumeric designators to the steps and terms of the claims.

3.5 Amended Method Claims

Bedrock has proposed amendments to the method claims of the '120 patent. After reviewing the parties' claim construction briefs, the '120 patent, the file history of the '120 patent, as the re-examination history, it is my opinion that these amendments did not change the scope of the method claims as originally drafted. In this report, I use the amended claim language, but my analysis and opinions apply equally to the claims as originally drafted. The fact that my infringement analysis would be unchanged under the claims as amended or originally drafted further confirms my belief that the scope of the method claims has not changed in re-examination.

4. The Court's Constructions of the Claims of the '120 Patent

I have reviewed the claim construction briefs submitted by Bedrock and the Defendants. I have also reviewed the Court's Provisional Claim Construction Order as well as the Court's final Memorandum Opinion and Order on claim construction (Dkt. No. 369). I have applied the Court's constructions to my analysis in this report. From my review, it appears that the Defendants attempted to incorporate noninfringement theories into their proposed constructions, but the Court rejected every one of those attempts.

5. "Candidate" On-The-Fly Removal (Linux versions 2.4.22 and Onwards)

5.1 Independent Claim 1

5.1.1 Preamble

An information storage and retrieval system, the system comprising:

5.1.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of Claim 1 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation.

5.1.1.2 Analysis

Computer equipment configured with or utilizing software based on Linux version 2.4.22 and onwards is an information storage and retrieval system and therefore meets this limitation literally. I will refer to such a system in this report as a "Linux System," but my usage of this term for these versions of Linux only relates to this section, i.e., § 5. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining.

5.1.2 Term 1(a)

a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring

5.1.2.1 Court's Construction

The Court has construed the following from term 1(a):

Claim Term	Court's Construction
a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

5.1.2.2 Analysis

A Linux System makes, uses, and is capable of making and using a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring, and I therefore find that this limitation is literally met. As discussed in § 1.4.4, the routing cache in Linux uses hashing with external chaining.

In the Linux source code, the routing cache records are C structs of type *rtable* (defined in route.h). These records contain a field named *u.dst.rt_next* (defined in dst.h), which is a pointer to the next record in the list. If there is no next record, then the *u.dst.rt_next* field contains a null pointer. The Linux System uses *u.dst.rt_next* to link *rtable* records together to form a list, and this list is capable of containing two or more *rtable* records. When the Linux System needs to create a new record on a linked list, memory is allocated for that record, and the routing cache records are stored in the RAM of the Linux System. The linked lists of *rtable* records chains off of the hash table *rt_hash_table*.

The records in the linked list automatically expire per the Court's construction. A Linux System scores the desirability or need for routing cache records based on the following criteria: (i) the age of the routing cache record; (ii) the type of route, e.g., whether the route is multicast, broadcast, and local; and (iii) whether the route has been redirected. The function *rt_score()* is the function that scores the desirability or need for records in the Linux System. Further, the *__refcnt* of the record is examined to see if the record is currently in use; if so, a record is not scored and is not eligible for deletion.

5.1.3 Term 1(b)

a record search means utilizing a search key to access the linked list

5.1.3.1 Court's Construction

The Court has construed term 1(b):

Claim Term	Court's Construction
a record search means utilizing a search key to access the linked list	Function: utilizing a search key to access the linked list Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52- 56 and portions of the application software, user access software or operating system software, as described at col. 4 lines 22-48, programmed with software instructions as described in Boxes 31-36 and Boxes 39-41 of FIG. 3 and in col. 5 line 53-col. 6 line 4 and col. 6 lines 14-20, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

5.1.3.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means utilizing a search key to access the linked list as construed by the Court. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term. A Linux System is programmed with software instructions that are identical to and at the very least equivalent to the structure specifically cited in the Court's construction.

A Linux System performs hashing using *rt_hash()*, which takes source IP address and destination IP address as inputs, among other inputs. The output of *rt_hash()* is assigned to variable hash, which is passed into *rt_intern_hash()*.[2] *rt_intern_hash()* then uses the value of

[2] *See*, *e.g.*:
unsigned hash = rt_hash(daddr, skeys[i], ikeys[k], rt_genid(net));
if (!rt_intern_hash(hash, rt, &rt, NULL);
---
hash = rt_hash(daddr, saddr, dev->ifindex, rt_genid(dev_net(dev)));
return rt_intern_hash(hash, rth, NULL, skb);
---
hash = rt_hash(daddr, saddr, fl->iif, rt_genid(dev_net(rth->u.dst.dev)));
return rt_intern_hash(hash, rth, NULL, skb);

hash as an offset of the hash table, *rt_hash_table*, and assigns the address of the head of the linked list chaining off of that hash table address to variable *rthp*. The Linux System then inspects each record, one-by-one, in the linked list using a while loop starting at the record at *rthp*, and searches for a record by comparing keys. The Linux System compares keys using the function *compare_keys()*, which uses *fl*, a struct of type *flowi*. fl contains source IP address and destination IP address, among other things. If the key comparison yields a match, the Linux System passes back the searched-for record to the caller of *rt_intern_hash()*.

In this way, this structure in a Linux System is identical to—or at the very least equivalent to—the structure in the Alternate Version of the Search Table Procedure. Additionally, a Linux System performs a function identical to the function construed by the Court, i.e., the Linux System utilizes a search key to access the linked list. A Linux System performs this function in substantially the same way as the various structures in the Court's construction. Namely, both a Linux System and the structures in the Court's construction hash a search key to obtain the head of the target list, traverse through the list looking for a record, and pass back the searched-for record, if found. A Linux System and the structures in the Court's construction yield the identical—and at the very least substantially the same—result, which is that the searched-for record is passed back to the caller of the search procedure. Put a different way, the differences between the structure in a Linux System and the structures in the Court's construction are not substantial and simply amount to implementation choices; i.e., (i) *rt_hash()* is called outside of *rt_intern_hash()* rather than inside of the function (ii) the eventual "user" of

---

---

hash = rt_hash(daddr, saddr, fl.iif, rt_genid(net));
err = rt_intern_hash(hash, rth, NULL, skb);
hash = rt_hash(oldflp->fl4_dst, oldflp->fl4_src, oldflp->oif, rt_genid(dev_net(dev_out)));
err = rt_intern_hash(hash, rth, rp, NULL);

the search structure in *rt_intern_hash()* is actually inside of *rt_intern_hash()* in the case where the record is not found; and (iii) in the case where the record is found, the updating of the record (e.g., updating the use time via *dst_use(&rth->u.dst,now)*) is done inside of *rt_intern_hash()*. Again, it is my opinion that these differences are insubstantial as they amount to implementation choices and do not change the functionality of the code.

5.1.4 Term 1(c)

> *the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and*

5.1.4.1 Court's Construction

The Court has construed term 1(c):

Claim Term	Court's Construction
the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	Function: identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4 lines 22-48, programmed with software instructions as described in Boxes 33-42 of FIG. 3 and in col. 5 line 53-col. 6 line 34, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

5.1.4.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term.

A Linux System is programmed with software instructions that are identical to—and at the very least equivalent to—the structure specifically cited in the Court's construction. As described for the previous claim term, the Linux system (in *rt_intern_hash()*) accesses the linked list using the search key starting with using the head of the hash chain that is located by hashing the search key. For each record in the list that is not a match, it scores the desirability of that record with the function *rt_score()* (if that record is not in use, as indicated by the *__refcnt* field of the record). The output of *rt_score()* is stored in variable score, and if score is less than the running minimum score, which is kept in *min_score*, then the current record is the least desirable of all the records scored up to that point and its address is stored in variable *cand*. At the end of the while loop, a pointer to the least desirable record (if any) is stored in *cand*. The Linux System checks to see whether the variable cand stores anything other than zero, which is the default setting of *cand* and indicates that no candidate for on-the-fly removal was found. If *cand* is not zero, i.e., if *cand* holds the address of the lowest scoring routing cache record, then it will check to see whether the chain length, which was tallied by *chain_length* exceeds a limit set by *ip_rt_gc_elasticity*. If *ip_rt_gc_elasticity* is exceeded, then the Linux System will perform pointer adjustment to bypass the lowest scoring record and will pass the address of that record to *rt_free()*, which is a deallocation routine. If the key comparison yields a match, the Linux System passes back the searched-for record to the caller of *rt_intern_hash()* as described in § 5.1.3.

It is: (i) the scoring the desirability of the routing cache records; (ii) the keeping track of the record with the lowest score; and (iii) the *if(cand)* statement that identify an expired record. It is the pointer adjustment following the test of chain length against gc_elasticity that removes the expired record. Both the identifying and removing occur when the linked list is accessed. In

this way, a Linux System performs a function identical to the function in the Court's construction, i.e., identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. A Linux System performs this function in substantially the same way as the various structures in the Court's construction. Namely, both a Linux System and the structures in the Court's construction use the linked list access used for record searching to identify and remove an expired record. A Linux System and the structures in the Court's construction yield the identical—and at the very least substantially the same—result, which is the removal of an expired record at the completion of record searching on a linked list. The differences between the structure in a Linux System and the structures in the Court's construction are not substantial and are a direct result of the fact that the implementation of on-the-fly garbage removal for candidate removal is based on a dynamic determination to remove at most one record.

The Defendant have pointed to differences between the structures in the court's construction and the structures in route.c As indicated above, any such differences are not substantial and arise from (a) differences in coding style and (b) the efficient implementation of a structure for removing at most one record due to *cand* deletion. For example, the location of the *if (cand)* statement that performs the removal comes immediately after the end of the while loop and, further, it is not called if a match is found. This is because the code is evaluating a combination of conditions to determine whether or not to remove a record, and one of those conditions is the length of the linked list. If a match is found, then code knows that it will not be increasing the length of the list and is, therefore, content to leave the list unchanged. The *if (cand)* test could easily be moved into the while loop and augmented with a check to test whether the end of the list was reached.

5.1.5 Term 1(d)

means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.

5.1.5.1 Court's Construction

The Court has construed term 1(d):

Claim Term	Court's Construction
means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list	Function: utilizing the record search means, accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions that provide the insert, retrieve, or delete record capability as described in the flowchart of FIG. 5 and col. 7 line 65 – col. 8 line 32, FIG. 6 and col. 8 lines 33-44, or FIG. 7 and col. 8 lines 45-59, respectively, and/or programmed with software instructions that provide the insert, retrieve or delete record capability as described in the pseudo-code of Insert Procedure (cols. 9 and 10), Retrieve Procedure (cols. 9, 10, 11, and 12), or Delete Procedure (cols. 11 and 12), respectively, and equivalents thereof.

5.1.5.2 Analysis

A Linux System makes, uses, and is capable of making and using a means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term.

A Linux System is programmed with software instructions that are identical to and at the very least equivalent to the structure specifically cited in the Court's construction, and it is

therefore my opinion that a Linux System literally meets this limitation. Within the function *rt_intern_hash()*, a Linux System will insert a record if the searched-for record was not found by the key comparison. This insertion utilizes the record search means discussed in §§ 5.1.3.2 and 5.1.4.2. The Linux System inserts a record by assigning the address of the current head of the linked list to the to-be-inserted record's *rt_next* field. Then the Linux System will store the to-be-inserted record's address to the hash table for the linked list so that the inserted record is the new head of the linked list. Also within the function *rt_intern_hash()*, a Linux System will retrieve a record if the record is found by key comparison. The Linux System retrieves a record by utilizing the record search means discussed in §§ 5.1.3.2 and 5.1.4.2. If the record search means encounters a key match, the record is passed back to the caller of *rt_intern_hash()* either by address or by attaching it to an *SKB* data structure, depending on how the caller invoked *rt_intern_hash()*. The Linux System will also update the usage information of a searched-for record if found.

In this way, these structures in a Linux System are equivalent to the structures in the Court's construction. A Linux System performs a function identical to the function construed by the Court, i.e., the Linux System utilizes the record search means, accesses the linked list and, at the same time, removes at least some of the expired ones of the records in the linked list. A Linux System performs this function in substantially the same way to yield substantially the same result as the various structures in the Court's construction, as discussed below.

Insertion. Both the Linux System and the structures in the Court's construction corresponding to record insertion utilize the record search means to see if a record exists in a linked list. If the record is found, then the to-be-inserted record is passed back to the caller of the record search means and the usage information for a record is updated; otherwise, both insert the

record at the head of the list. A Linux System and the structures in the Court's construction corresponding to record insertion yield the identical—and at the very least substantially the same—result, which is that the record to be inserted is either passed back to the called of the search procedure or is inserted at the head of the linked list and expired record(s) are removed. The differences between the structure in a Linux System and the structures in the Court's construction are not substantial; in fact, the only missing structural component is the check to determine whether memory is available, which is performed before the invocation of *rt_intern_hash()*.

Retrieval. Both the Linux System and the structures in the Court's construction corresponding to record retrieval utilize the record search means to return a record to the caller of the record search means. If the record is found, then the searched-for record is passed back to the caller. A Linux System and the structures in the Court's construction corresponding to record insertion yield the identical—and at the very least substantially the same—result, which is that the caller of the record search means has the searched-for record, and expired record(s) are removed. The difference between the two is that the Linux System does not return success or failure based on the success of the record searching but instead inserts the searched-for record if it is not found. In other words, the differences between the structure in the Linux System and the Retrieval structure in the Court's construction is due to the placement of the Retrieval and Insertion routines primarily within one subroutine, *rt_intern_hash()*, within the Linux System.

5.2 Dependent Claim 2

5.2.1 Claim language

> *The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.*

5.2.1.1 Court's Construction

The Court has construed the following within claim 2:

Claim Term	Court's Construction
means for dynamically determining maximum number	Function: dynamically determining maximum number for the record search means to remove in the accessed linked list of records Structure: CPU 10, and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions to dynamically determine a maximum number of records to remove by choosing a search strategy of removing all expired records from a linked list or removing some but not all of the expired records as described in col. 6 line 56 – col. 7 line 15 and/or programmed with software instructions to dynamically determine a maximum number of records to remove by choosing between the pseudo-code of the Search Table Procedure (cols. 11 and 12) or Alternative Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

5.2.1.2 Analysis

A Linux System makes, uses, and is capable of making and using means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. As claim 2 is dependent on claim 1, I incorporate my discussion of claim 1 from § 5.1.

As a Linux System traverses a linked list in the routing cache, it keeps track of the length of the linked list with the variable *chain_length*. The Linux System compares the ending value of *chain_length* against the value stored in *ip_rt_gc_elasticity*, which is a routing cache tuning parameter that controls how long linked lists in the routing cache can grow. The gc stands for

garbage collection. If *chain_length* exceeds *ip_rt_gc_elasticity*, then the on-the-fly garbage collection removal, discussed in § 5.1.4, will execute. If *chain_length* does not exceed *ip_rt_gc_elasticity*, then the on-the-fly garbage collection removal will not execute.

In this way, this structure in a Linux System is identical to—and at the very equivalent to—the structures in the Court's construction, which incorporates the discussion in the '120 patent disclosing the dynamically determining algorithm. *See* '120::6:66–7:10 ("The implementor even has the prerogative of choosing among these strategies dynamically at the time the search table is invoked by the caller, thus sometimes removing all expired records, at other times removing some but not all of them, and yet at other times choosing to remove none of them. Such a dynamic runtime decision might be based on factors such as, for example, how much memory is available in the system storage pool, general system load, time of day, the number of records currently residing in the information system, and other factors both internal and external to the information storage and retrieval system.") The Linux System performs a function identical to the function construed by the Court, i.e., the Linux System dynamically determines maximum number for the record search means to remove in the accessed linked list of records. The Linux System performs this function in an identical way and to achieve the identical result as this structure of the Court's construction. Namely, the Linux System removes an expired record or no records at all based on the dynamic runtime factor of the number or records in the linked list. The result is that dynamic runtime criteria (or a criterion) control the tolerance for expired records in the information storage system.

This structure in the Linux System is also an equivalent to the structure in the Court's construction that is the choice between software instructions to dynamically determine a maximum number of records to remove by choosing between the pseudo-code of the Search

Table Procedure (cols. 11 and 12) or Alternative Version of Search Table Procedure (cols. 11, 12, 13, and 14). Again, the Linux System performs a function identical to the function construed by the Court, i.e., the Linux System dynamically determines maximum number for the record search means to remove in the accessed linked list of records. The difference between the Search Table Procedure and the Alternate Search Table Procedure is that the Alternate Search Table Procedure terminates when the searched-for record is found. Put another way, the Search Table Procedure will continue on-the-fly removal of expired records where the Alternate Search Table would stop. In this way, the Linux System performs this function in substantially the same way as structure in the Court's construction, namely, by making a dynamic runtime decision to continue on-the-fly removal of expired records. Regardless of the comparison of *chain_length* to *ip_rt_gc_elasticity*, the Linux System will begin identifying expired records; it is only when *chain_length* exceeds *ip_rt_gc_elasticity* that the Linux System will continue onto the removal of the expired record. The result of both the Linux System and the structures in the Court's construction is that dynamic runtime criteria (or a criterion) control the tolerance for expired records in the information storage system.

5.3 Independent Claim 3

5.3.1 Preamble

> *A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:*

5.3.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of Claim 3 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation. The Court has construed the following from the preamble:

Claim Term	Court's Construction
a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

5.3.1.2 Analysis

A Linux System uses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining. Further, as discussed in § 5.1.2, the *rtable* routing cache records in Linux form linked lists chaining from the *rt_hash_table* hash table, and the routing cache records automatically expire.

5.3.2 Step 3(a)

accessing the linked list of records to search for a target record,

5.3.2.1 Court's Construction

The Court has construed the following from step 3(a):

Claim Term	Court's Construction
linked list of records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record

5.3.2.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of accessing the linked list of records to search for a target record. As discussed in § 5.1.3, a Linux System accesses a linked list of the routing cache by hashing a search key, and the Linux System searches for a target record using key comparison.

5.3.3 Step 3(b)

identifying at least some of the automatically expired ones of the records while searching for the target record, and

5.3.3.1 Court's Construction

The Court has construed the following from step 3(b):

Claim Term	Court's Construction
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

5.3.3.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records while searching for the target record. As discussed in § 5.1.4, a Linux System identifies expired records using the *rt_score()* function to keep track of the least desirable record while searching for a target record using key comparison discussed in § 5.1.3.

5.3.4 Step 3(c)

removing at least some of the automatically expired records from the linked list when the linked list is accessed.

5.3.4.1 Court's Construction

The Court has construed the following from term 3(c):

Claim Term	Court's Construction
removing . . . from the linked list	adjusting the pointer in the linked list to bypass the previously identified expired records
linked list	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

5.3.4.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of removing at least some of the automatically expired records from the linked list when the linked list is accessed. As discussed in § 5.1.4, when a Linux System identifies an expired record, it will remove that record from the linked list via pointer adjustment if *chain_length* exceeds *ip_rt_gc_elasticity*, and the identification and removal of the expired record occurs during the same access of the linked list.

5.3.5 Ordering of the Steps

5.3.5.1 Court's Construction

The Court has construed the ordering of the steps of claim 3, specifically that the identifying step must begin before the removing step can begin.

5.3.5.2 Analysis

When a Linux System searches for a route in the routing cache, the identifying step of claim 3 begins before the removing step begins. As discussed in § 5.1.4, the identifying of expired records using *rt_score()* begins before the removal of the expired records, which is

performed using the pointer adjustment. In fact, in a Linux System, the identifying step is completed before the removal begins.

5.4 Dependent Claim 4

5.4.1 Claim Language

The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.

5.4.1.1 Court's Construction

The Court has construed the following from claim 4:

Claim Term	Court's Construction
dynamically determining	making a decision based on factors internal or external to the information storage and retrieval system
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

5.4.1.2 Analysis

When a Linux System searches for a route in the routing cache, the Linux System performs a method that includes the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. As claim 4 is dependent on claim 3, I incorporate my discussion of claim 3 from § 5.3.

As discussed in § 5.2, a Linux System compares *chain_length* against *ip_rt_gc_elasticity* to decide whether to remove the identified expired record. *chain_length* is a factor internal to the information storage and retrieval system as it measures the length of the object linked list and is subject to change during runtime.

5.5 Independent Claim 7

5.5.1 Preamble

A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:

5.5.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of claim 7 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation. The Court has construed the following from the preamble:

Claim Term	Court's Construction
external chaining	a technique for resolving hash collisions using a linked list(s)
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

5.5.1.2 Analysis

A Linux System uses method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining. As discussed in § 5.1.2, the *rtable* routing cache records in Linux form linked lists chaining from the *rt_hash_table* hash table, and the routing cache records automatically expire. The linked list of *rtable* records contains *rtable*

records that hash to the same hash value; in this way, the linked list of *rtable* records resolves hash collisions.

5.5.2 Step 7(a)

accessing a linked list of records having same hash address to search for a target record,

5.5.2.1 Court's Construction

The Court has construed the following from term 7(a):

Claim Term	Court's Construction
a linked list of records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record

5.5.2.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of accessing a linked list of records having same hash address to search for a target record. As discussed in § 5.3.2, a Linux System accesses a linked list of the routing cache by hashing a search key, and the Linux System searches for a target record using key comparison. Further, the *rtable* records for a given linked list should have the same hash value as *rtable* records are inserted into the linked list of *rtable* records via the hashing function *rt_hash()*.

5.5.3 Step 7(b)

identifying at least some of the automatically expired ones of the records while searching for the target record,

5.5.3.1 Court's Construction

The Court has construed the following from term 7(b):

Claim Term	Court's Construction
automatically expired	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of

	time

5.5.3.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records while searching for the target record. This step is the same step of claim 3(b), and so I incorporate my discussion from § 5.3.3.

5.5.4 Step 7(c)

removing at least some of the automatically expired records from the linked list when the linked list is accessed, and

5.5.4.1 Court's Construction

The Court has construed the following from term 7(c):

Claim Term	Court's Construction
removing . . . from the linked list	adjusting the pointer in the linked list to bypass the previously identified expired records
linked list	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

5.5.4.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records

when the linked list is accessed. This step is the same step of claim 3(c), and so I incorporate my discussion from § 5.3.4.

5.5.5 Step 7(d)

inserting, retrieving or deleting one of the records from the system following the step of removing.

5.5.5.1 Court's Construction

The Court has construed the following from term 1(a):

Claim Term	Court's Construction
a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

5.5.5.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of inserting, retrieving or deleting one of the records from the system following the step of removing. The removing of an expired record discussed in § 5.1.4 executes before the insert step discussed in § 5.1.5 executes.

5.5.6 Ordering of the Steps

5.5.6.1 Court's Construction

The Court has construed the following order of the steps of claim 7: the identifying step must begin before the removing step can begin, and the ultimate step of claim 7 must follow, or at least partially follow, the penultimate step of claim 7.

5.5.6.2 Analysis

When a Linux System searches for a route in the routing cache, the identifying step of claim 7 begins before the removing step begins. As discussed in § 5.1.4, the identifying of expired records using *rt_score()* begins before the removal of the expired records, which is performed using the pointer adjustment. In fact, in a Linux System, the identifying step is completed before the removal begins.

5.6 Dependent Claim 8

5.6.1 Claim Language

The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.

5.6.1.1 Court's Construction

The Court has construed the following from claim 8:

Claim Term	Court's Construction
dynamically determining	making a decision based on factors internal or external to the information storage and retrieval system
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

5.6.1.2 Analysis

When a Linux System searches for a route in the routing cache, the Linux System performs a method that includes the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. As claim 8 is dependent on claim 7, I incorporate my discussion of claim 7 from § 5.5.

As discussed in § 5.2, a Linux System compares *chain_length* against *ip_rt_gc_elasticity* to decide whether to remove the identified expired record. *chain_length* is a factor internal to the information storage and retrieval system as it measures the length of the object linked list and is subject to change during runtime.

6. "GenID" On-The-Fly Removal (Linux versions 2.6.25 and Onwards)

6.1 Independent Claim 1

6.1.1 Preamble

An information storage and retrieval system, the system comprising:

6.1.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of Claim 1 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation.

6.1.1.2 Analysis

Computer equipment configured with or utilizing software based on Linux version 2.6.25 and onwards is an information storage and retrieval system and therefore meets this limitation literally. I will refer to such a system in this report as a "Linux System," but my usage of this term for these versions of Linux only relates to this section, i.e., § 6. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining.

6.1.2 Term 1(a)

a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring

6.1.2.1 Court's Construction

The Court has construed the following from term 1(a):

Claim Term	Court's Construction
a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.1.2.2 Analysis

A Linux System makes, uses, and is capable of making and using a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring, and I therefore find that this limitation is literally met. As discussed in § 1.4.4, the routing cache in Linux uses hashing with external chaining.

In the Linux source code, the routing cache records are C structs of type *rtable* (defined in route.h). These records contain a field named *u.dst.rt_next* (defined in dst.h), which is a pointer to the next record in the list. If there is no next record, then the *u.dst.rt_next* field contains a null pointer. The Linux System uses *u.dst.rt_next* to link *rtable* records together to form a list, and this list is capable of containing two or more *rtable* records. When the Linux System needs to create a new record on a linked list, memory is allocated for that record, and the routing cache records are stored in the RAM of the Linux System. The linked lists of *rtable* records chains off of the hash table *rt_hash_table*.

The records in the linked list automatically expire per the Court's construction. An *rtable* record in the routing cache has a field called *rt_genid*. The *rt_genid* field of an *rtable* record indicates which generation a record belongs to when the record was created, and it is set using

the network genid variable *ipv4.rt_genid*. The Linux System periodically ages records in the routing cache by modifying the network genid variable *ipv4.rt_genid*.[3] An *rtable* record is considered to be expired if its *rt_genid* field does not equal the current network genid variable *ipv4.rt_genid*.

6.1.3 Term 1(b)

a record search means utilizing a search key to access the linked list

6.1.3.1 Court's Construction

The Court has construed term 1(b):

Claim Term	Court's Construction
a record search means utilizing a search key to access the linked list	Function: utilizing a search key to access the linked list Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52- 56 and portions of the application software, user access software or operating system software, as described at col. 4 lines 22-48, programmed with software instructions as described in Boxes 31-36 and Boxes 39-41 of FIG. 3 and in col. 5 line 53-col. 6 line 4 and col. 6 lines 14-20, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.1.3.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means utilizing a search key to access the linked list as construed by the Court. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term. A Linux

[3] The *rt_genid* value is modified whenever *rt_cache_invalidate()* is called. This can happen periodically when *rt_secret_rebuild()* is called, but can also occur elsewhere, for example, when *rt_cache_flush()*.

System is programmed with software instructions that are identical to and at the very least equivalent to the structure specifically cited in the Court's construction.

A Linux System performs hashing using *rt_hash()*, which takes source IP address and destination IP address as inputs, among other inputs. The output of *rt_hash()* is assigned to variable hash, which is passed into *rt_intern_hash()*.[4] *rt_intern_hash()* then uses the value of hash as an offset of the hash table, *rt_hash_table*, and assigns the address of the head of the linked list chaining off of that hash table address to variable *rthp*. The Linux System then inspects each record, one-by-one, in the linked list using a while loop starting at the record at *rthp*, and searches for a record by comparing keys. The Linux System compares keys using the function *compare_keys()*, which uses *fl*, a struct of type *flowi*. fl contains source IP address and destination IP address, among other things. If the key comparison yields a match, the Linux System passes back the searched-for record to the caller of *rt_intern_hash()*.

In this way, this structure in a Linux System is identical to—or at the very least equivalent to—the structure in the Alternate Version of the Search Table Procedure. Additionally, a Linux System performs a function identical to the function construed by the Court, i.e., the Linux System utilizes a search key to access the linked list. A Linux System

---

[4] *See*, *e.g.*:
unsigned hash = rt_hash(daddr, skeys[i], ikeys[k], rt_genid(net));
if (!rt_intern_hash(hash, rt, &rt, NULL);
---
hash = rt_hash(daddr, saddr, dev->ifindex, rt_genid(dev_net(dev)));
return rt_intern_hash(hash, rth, NULL, skb);
---
hash = rt_hash(daddr, saddr, fl->iif, rt_genid(dev_net(rth->u.dst.dev)));
return rt_intern_hash(hash, rth, NULL, skb);
---
hash = rt_hash(daddr, saddr, fl.iif, rt_genid(net));
err = rt_intern_hash(hash, rth, NULL, skb);
hash = rt_hash(oldflp->fl4_dst, oldflp->fl4_src, oldflp->oif, rt_genid(dev_net(dev_out)));
err = rt_intern_hash(hash, rth, rp, NULL);

performs this function in substantially the same way as the various structures in the Court's construction. Namely, both a Linux System and the structures in the Court's construction hash a search key to obtain the head of the target list, traverse through the list looking for a record, and pass back the searched-for record, if found. A Linux System and the structures in the Court's construction yield the identical—and at the very least substantially the same—result, which is that the searched-for record is passed back to the caller of the search procedure. Put a different way, the differences between the structure in a Linux System and the structures in the Court's construction are not substantial and simply amount to implementation choices; i.e., (i) *rt_hash()* is called outside of *rt_intern_hash()* rather than inside of the function (ii) the eventual "user" of the search structure in *rt_intern_hash()* is actually inside of *rt_intern_hash()* in the case where the record is not found; and (iii) in the case where the record is found, the updating of the record (e.g., updating the use time via *dst_use(&rth->u.dst,now)*) is done inside of *rt_intern_hash()*. Again, it is my opinion that these differences are insubstantial as they amount to implementation choices and do not change the functionality of the code.

6.1.4 Term 1(c)

the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and

6.1.4.1 Court's Construction

The Court has construed term 1(c):

Claim Term	Court's Construction
the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	Function: identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4 lines 22-48, programmed with software

	instructions as described in Boxes 33-42 of FIG. 3 and in col. 5 line 53-col. 6 line 34, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.1.4.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term.

A Linux System is programmed with software instructions that are identical to—and at the very least equivalent to—the structure specifically cited in the Court's construction. As described for the previous claim term, the Linux system (in *rt_intern_hash()*) accesses the linked list using the search key starting with using the head of the hash chain that is located by hashing the search key. For each record in the list, the Linux System uses function *rt_is_expired()* to check the *rtable* record's *rt_genid* field against the current network genid variable *ipv4.rt_genid*. If the *rt_genid* field for an *rtable* record is not the same value of the current network genid variable *ipv4.rt_genid*, then the Linux System will first adjust the pointers to bypass that record and then pass the address of that record to *rt_free()*,which is a deallocation routine. If the *rt_genid* field for an *rtable* record is the same value of the current network genid variable *ipv4.rt_genid*, then the Linux System will proceed to search for a record using key comparison as discussed in § 6.1.3, and if the key comparison yields a match, the Linux System passes back the searched-for record to the caller of *rt_intern_hash()*.

It is the check the of the *rtable* record's *rt_genid* field against the current network genid variable *ipv4.rt_genid* using *rt_is_expired()* that identifies an expired record. It is the pointer adjustment following this check for expiration that removes the expired record. Both the identifying and removing occur when the linked list is accessed, and both occur within the while loop utilized by the key comparison. In this way, a Linux System performs a function identical to the function in the Court's construction, i.e., identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. This Linux System structure has an identical structure to the Alternate Record Search Procedure as described in prose and expressed as pseudocode. The Linux System also performs this function in substantially the same way as the other structures in the Court's construction. Namely, both a Linux System and the structures in the Court's construction use the linked list access used for record searching to identify and remove an expired record. A Linux System and the structures in the Court's construction yield the identical—and at the very least substantially the same—result, which is the removal of expired records at the completion of record searching on a linked list.

6.1.5 Term 1(d)

means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.

6.1.5.1 Court's Construction

The Court has construed term 1(d):

Claim Term	Court's Construction
means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list	Function: utilizing the record search means, accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col.

	4, lines 22-48, programmed with software instructions that provide the insert, retrieve, or delete record capability as described in the flowchart of FIG. 5 and col. 7 line 65 – col. 8 line 32, FIG. 6 and col. 8 lines 33-44, or FIG. 7 and col. 8 lines 45-59, respectively, and/or programmed with software instructions that provide the insert, retrieve or delete record capability as described in the pseudo-code of Insert Procedure (cols. 9 and 10), Retrieve Procedure (cols. 9, 10, 11, and 12), or Delete Procedure (cols. 11 and 12), respectively, and equivalents thereof.

6.1.5.2 Analysis

A Linux System makes, uses, and is capable of making and using a means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. I incorporate by reference my discussion in § 3 regarding the hardware components of this claim term.

A Linux System is programmed with software instructions that are identical to and at the very least equivalent to the structure specifically cited in the Court's construction, and it is therefore my opinion that a Linux System literally meets this limitation. Within the function *rt_intern_hash()*, a Linux System will insert a record if the searched-for record was not found by the key comparison. This insertion utilizes the record search means discussed in §§ 6.1.3.2 and 6.1.4.2. The Linux System inserts a record by assigning the address of the current head of the linked list to the to-be-inserted record's *rt_next* field. Then the Linux System will store the to-be-inserted record's address to the hash table for the linked list so that the inserted record is the new head of the linked list. Also within the function *rt_intern_hash()*, a Linux System will retrieve a record if the record is found by key comparison. The Linux System retrieves a record by utilizing the record search means discussed in §§ 6.1.3.2 and 6.1.4.2. If the record search

means encounters a key match, the record is passed back to the caller of *rt_intern_hash()* either by address or by attaching it to an *SKB* data structure, depending on how the caller invoked *rt_intern_hash()*. The Linux System will also update the usage information of a searched-for record if found.

In this way, these structures in a Linux System are equivalent to the structures in the Court's construction. A Linux System performs a function identical to the function construed by the Court, i.e., the Linux System utilizes the record search means, accesses the linked list and, at the same time, removes at least some of the expired ones of the records in the linked list. A Linux System performs this function in substantially the same way to yield substantially the same result as the various structures in the Court's construction, as discussed below.

Insertion. Both the Linux System and the structures in the Court's construction corresponding to record insertion utilize the record search means to see if a record exists in a linked list. If the record is found, then the to-be-inserted record is passed back to the caller of the record search means and the usage information for a record is updated; otherwise, both insert the record at the head of the list. A Linux System and the structures in the Court's construction corresponding to record insertion yield the identical—and at the very least substantially the same—result, which is that the record to be inserted is either passed back to the called of the search procedure or is inserted at the head of the linked list and expired record(s) are removed. The differences between the structure in a Linux System and the structures in the Court's construction are not substantial; in fact, the only missing structural component is the check to determine whether memory is available, which is performed before the invocation of *rt_intern_hash()*.

Retrieval. Both the Linux System and the structures in the Court's construction corresponding to record retrieval utilize the record search means to return a record to the caller of the record search means. If the record is found, then the searched-for record is passed back to the caller. A Linux System and the structures in the Court's construction corresponding to record insertion yield the identical—and at the very least substantially the same—result, which is that the caller of the record search means has the searched-for record, and expired record(s) are removed. The difference between the two is that the Linux System does not return success or failure based on the success of the record searching but instead inserts the searched-for record if it is not found. In other words, the differences between the structure in the Linux System and the Retrieval structure in the Court's construction is due to the placement of the Retrieval and Insertion routines primarily within one subroutine, *rt_intern_hash()*, within the Linux System.

6.2 Dependent Claim 2

6.2.1 Claim language

The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.

6.2.1.1 Court's Construction

The Court has construed the following within claim 2:

Claim Term	Court's Construction
means for dynamically determining maximum number	Function: dynamically determining maximum number for the record search means to remove in the accessed linked list of records Structure: CPU 10, and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions to dynamically determine a maximum number of records to remove by choosing a search strategy of removing all

	expired records from a linked list or removing some but not all of the expired records as described in col. 6 line 56 – col. 7 line 15 and/or programmed with software instructions to dynamically determine a maximum number of records to remove by choosing between the pseudo-code of the Search Table Procedure (cols. 11 and 12) or Alternative Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.2.1.2 Analysis

A Linux System makes, uses, and is capable of making and using means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. As claim 2 is dependent on claim 1, I incorporate my discussion of claim 1 from § 6.1.

As a Linux System traverses a linked list in the routing cache, it keeps track of the length of the linked list with the variable *chain_length*. The Linux System compares the ending value of *chain_length* against the value stored in *ip_rt_gc_elasticity*, which is a routing cache tuning parameter that controls how long linked lists in the routing cache can grow. The gc stands for garbage collection. If *chain_length* exceeds *ip_rt_gc_elasticity*, then the candidate on-the-fly garbage collection removal, discussed in §§ 5.1.4 and 5.2, will execute. If *chain_length* does not exceed *ip_rt_gc_elasticity*, then the candidate on-the-fly garbage collection removal will not execute.

This structure in the Linux System is an equivalent to the structure in the Court's construction that is the choice between software instructions to dynamically determine a maximum number of records to remove by choosing between the pseudo-code of the Search Table Procedure (cols. 11 and 12) or Alternative Version of Search Table Procedure (cols. 11, 12, 13, and 14). The Linux System performs a function identical to the function construed by the

Court, i.e., the Linux System dynamically determines maximum number for the record search means to remove in the accessed linked list of records. The difference between the Search Table Procedure and the Alternate Search Table Procedure is that the Alternate Search Table Procedure terminates when the searched-for record is found. Put another way, the Search Table Procedure will continue on-the-fly removal of expired records where the Alternate Search Table would stop. In this way, the Linux System performs this function in substantially the same way as structure in the Court's construction, namely, by removing records on-the-fly with the genid removal and by making a dynamic runtime decision to execute candidate on-the-fly removal of expired records as well. Regardless of the comparison of *chain_length* to *ip_rt_gc_elasticity*, the Linux System will identify and remove expired records with the genid on-the-fly removal; it is only when *chain_length* exceeds *ip_rt_gc_elasticity* that the Linux System will continue onto the removal of the expired records using the candidate on-the-fly removal. The result of both the Linux System and the structures in the Court's construction is that dynamic runtime criteria (or a criterion) control the tolerance for expired records in the information storage system.

6.3 Independent Claim 3

6.3.1 Preamble

A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:

6.3.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of Claim 3 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation. The Court has construed the following from the preamble:

Claim Term	Court's Construction

a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.3.1.2 Analysis

A Linux System uses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining. Further, as discussed in § 6.1.2, the *rtable* routing cache records in Linux form linked lists chaining from the *rt_hash_table* hash table, and the routing cache records automatically expire.

6.3.2 Step 3(a)

accessing the linked list of records to search for a target record,

6.3.2.1 Court's Construction

The Court has construed the following from step 3(a):

Claim Term	Court's Construction
linked list of records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record

6.3.2.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of accessing the linked list of records to search for a target record. As discussed in § 6.1.3, a Linux System accesses a linked list of the routing cache by hashing a search key, and the Linux System searches for a target record using key comparison.

6.3.3 Step 3(b)

identifying at least some of the automatically expired ones of the records while searching for the target record, and

6.3.3.1 Court's Construction

The Court has construed the following from step 3(b):

Claim Term	Court's Construction
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.3.3.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records while searching for the target record. As discussed in § 6.1.4, a Linux System identifies expired records using the *rt_is_expired()* function to check an *rtable* record's *rt_genid* field against the current network genid variable *ipv4.rt_genid* while searching for a target record using key comparison discussed in § 6.1.3.

6.3.4 Step 3(c)

removing at least some of the automatically expired records from the linked list when the linked list is accessed.

6.3.4.1 Court's Construction

The Court has construed the following from term 3(c):

Claim Term	Court's Construction

removing . . . from the linked list	adjusting the pointer in the linked list to bypass the previously identified expired records
linked list	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

6.3.4.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of removing at least some of the automatically expired records from the linked list when the linked list is accessed. As discussed in § 6.1.4, when a Linux System identifies an expired record, it will remove that record from the linked list via pointer adjustment if an *rtable* record's *rt_genid* field does not equal the current network genid variable *ipv4.rt_genid*, and the identification and removal of the expired record occurs during the same access of the linked list.

6.3.5 Ordering of the Steps

6.3.5.1 Court's Construction

The Court has construed the ordering of the steps of claim 3, specifically that the identifying step must begin before the removing step can begin.

6.3.5.2 Analysis

When a Linux System searches for a route in the routing cache, the identifying step of claim 3 begins before the removing step begins. As discussed in § 5.1.4, the identifying of expired records using *rt_is_expired()* begins before the removal of the expired records, which is

performed using the pointer adjustment. In fact, in a Linux System, the identifying step is completed before the removal begins.

6.4 Dependent Claim 4

6.4.1 Claim Language

The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.

6.4.1.1 Court's Construction

The Court has construed the following from claim 4:

Claim Term	Court's Construction
dynamically determining	making a decision based on factors internal or external to the information storage and retrieval system
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

6.4.1.2 Analysis

When a Linux System searches for a route in the routing cache, the Linux System performs a method that includes the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. As claim 4 is dependent on claim 3, I incorporate my discussion of claim 3 from § 6.3.

As discussed in § 6.2, a Linux System compares *chain_length* against *ip_rt_gc_elasticity* to decide whether to perform candidate on-the-fly removal in addition to genid on-the-fly removal. *chain_length* is a factor internal to the information storage and retrieval system as it measures the length of the object linked list and is subject to change during runtime.

6.5 Independent Claim 5

6.5.1 Preamble

An information storage and retrieval system, the system comprising:

6.5.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of Claim 1 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation.

6.5.1.2 Analysis

As discussed in § 1.4.4, the routing cache in a Linux System is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h and route.h contain source code that defines, manipulates, and uses the *rt_hash_table* hash table.

6.5.2 Term 5(a)

a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,

6.5.2.1 Court's Construction

The Court has construed term 5(a):

Claim Term	Court's Construction
a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring	Function: to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address at least some of the records automatically expiring Structure: CPU 10, and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions to provide a hash table having a pointer to the head of a linked list of externally

	chained records as described in col. 5 lines 16-26 and/or programmed with software instructions as described in the pseudo-code of Definitions, definition number 4, and equivalents thereof.

6.5.2.2 Analysis

A Linux System makes, uses, and is capable of making and using a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.

As discussed in § 6.1.2, the linked list of *rtable* routing cache records chains off of the *rt_hash_table* hash table. *rt_hash_table* contains a pointer to the head of the linked list of *rtable* records. This structure in a Linux System is identical to—and at the very least equivalent to—the structure in the Court's construction. Further, the *rt_hash_table* hash table is allocated as an array of *rt_hash_bucket* C structs, where each *rt_hash_bucket* contains a pointer to the first *rtable* record of a linked list of *rtable* records. This structure is to—and at the very least equivalent to—the structure in the Court's construction.

6.5.3 Term 5(b)

a record search means utilizing a search key to access a linked list of records having the same hash address,

6.5.3.1 Court's Construction

The Court has construed term 5(b):

Claim Term	Court's Construction
a record search means utilizing a search key to access a linked list of records having the same hash address	Function: utilizing a search key to access the linked list Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52- 56 and portions of the application software, user access software or

	operating system software, as described at col. 4 lines 22-48, programmed with software instructions as described in Boxes 31-36 and Boxes 39-41 of FIG. 3 and in col. 5 line 53-col. 6 line 4 and col. 6 lines 14-20, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.5.3.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means utilizing a search key to access a linked list of records having the same hash address. The Court construed this term to have the same meaning as term 1(b); therefore, I incorporate my analysis in § 6.1.3.

6.5.4 Term 5(c)

> *the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and*

6.5.4.1 Court's Construction

The Court has construed term 5(c):

Claim Term	Court's Construction
the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed	Function: identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4 lines 22-48, programmed with software instructions as described in Boxes 33-42 of FIG. 3 and in col. 5 line 53-col. 6 line 34, and/or programmed with software instructions as described in the pseudo-code of Search Table Procedure (cols. 11 and 12) or Alternate

	Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.5.4.2 Analysis

A Linux System makes, uses, and is capable of making and using a record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. The Court construes this term the same as term 1(c); therefore I incorporate my analysis of § 6.1.4.

6.5.5 Term 5(d)

means, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.

6.5.5.1 Court's Construction

The Court has construed term 5(d):

Claim Term	Court's Construction
means, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records	Function: utilizing the record search means, inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records Structure: CPU 10 and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions that provide the insert, retrieve, and delete record capability as described in the flowchart of FIG. 5 and col. 7 line 65 – col. 8 line 32, FIG. 6 and col. 8 lines 33-44, or FIG. 7 and col. 8 lines 45-59, respectively, and/or programmed with software instructions that provide the insert, retrieve and delete record capability as described in the pseudo-code of Insert Procedure (cols. 9 and 10), Retrieve Procedure (cols. 9, 10, 11, and 12), and Delete Procedure (cols. 11 and 12), respectively, and

	equivalents thereof.

6.5.5.2 Analysis

A Linux System makes, uses, and is capable of making and using a means, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. The difference between the Court's construction for this term and the Court's construction for term 1(d) is that this structure requires inserting, retrieving, ***and*** deleting as opposed discussed to inserting, retrieving, or deleting. As I discussed inserting and retrieving in § 6.1.5, I incorporate that section by reference here. Additionally, both the Linux System and the structures in the Court's construction corresponding to record deletion perform the identical function, i.e., utilizing the record search means, deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. *rt_del()* contains substantially the same code as the genid on-the-fly removal code in *rt_intern_hash()* discussed in § 6.1.4; therefore these structures are performing the function in substantially the same way as the structure in the Court's construction for this term. A Linux System and the structures in the Court's construction corresponding to record insertion yield the identical—and at the very least substantially the same—result, which is that a searched-for record is deleted and expired record(s) are removed.

The difference between the Linux System and the structures in the Court's construction corresponding to record deletion is the identifying and removal code is essentially copied into two separate subroutines, *rt_del()* and *rt_intern_hash()* rather than being in a single subroutine. This difference amounts to an implementation choice and is a result of the placement of the

Retrieval and Insertion routines primarily within one subroutine, *rt_intern_hash()* within the Linux System.

6.6 Dependent Claim 6

6.6.1 Claim language

The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.

6.6.1.1 Court's Construction

The Court has construed the following within claim 6:

Claim Term	Court's Construction
means for dynamically determining maximum number	Function: dynamically determining maximum number for the record search means to remove in the accessed linked list of records Structure: CPU 10, and RAM 11 of FIG. 1 and col. 3 lines 52-56 and portions of the application software, user access software or operating system software, as described at col. 4, lines 22-48, programmed with software instructions to dynamically determine a maximum number of records to remove by choosing a search strategy of removing all expired records from a linked list or removing some but not all of the expired records as described in col. 6 line 56 – col. 7 line 15 and/or programmed with software instructions to dynamically determine a maximum number of records to remove by choosing between the pseudo-code of the Search Table Procedure (cols. 11 and 12) or Alternative Version of Search Table Procedure (cols. 11, 12, 13, and 14), and equivalents thereof.

6.6.1.2 Analysis

A Linux System makes, uses, and is capable of making and using means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. As claim 6 is dependent on claim 5, I incorporate my discussion of claim 5 from

§ 6.5. Further, as the Court has construed this claim the same as claim 2, I incorporate my discussion of § 6.2.

6.7 Independent Claim 7

6.7.1 Preamble

A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:

6.7.1.1 Court's Construction

I do not believe that the Court made a determination regarding whether or not the Preamble of claim 7 should be considered as a limitation of that claim. In this expert report I have nonetheless treated that Preamble as if it were a limitation. The Court has construed the following from the preamble:

Claim Term	Court's Construction
external chaining	a technique for resolving hash collisions using a linked list(s)
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.7.1.2 Analysis

A Linux System uses method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. As discussed in § 1.4.4, the routing cache in Linux is an information storage and retrieval system for network connection information, including next hop information, for given source and destination IP addresses. The files route.c, dst.h, and route.h contain source code that defines, manipulates, and uses a hash table using external chaining. As discussed in § 6.1.2, the *rtable*

routing cache records in Linux form linked lists chaining from the *rt_hash_table* hash table, and the routing cache records automatically expire. The linked list of *rtable* records contains *rtable* records that hash to the same hash value; in this way, the linked list of *rtable* records resolves hash collisions.

6.7.2 Step 7(a)

accessing a linked list of records having same hash address to search for a target record,

6.7.2.1 Court's Construction

The Court has construed the following from term 7(a):

Claim Term	Court's Construction
a linked list of records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record

6.7.2.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of accessing a linked list of records having same hash address to search for a target record. As discussed in § 6.3.2, a Linux System accesses a linked list of the routing cache by hashing a search key, and the Linux System searches for a target record using key comparison. Further, the *rtable* records for a given linked list should have the same hash value as *rtable* records are inserted into the linked list of *rtable* records via the hashing function *rt_hash()*.

6.7.3 Step 7(b)

identifying at least some of the automatically expired ones of the records while searching for the target record,

6.7.3.1 Court's Construction

The Court has construed the following from term 7(b):

Claim Term	Court's Construction
automatically expired	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.7.3.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records while searching for the target record. This step is the same step of claim 3(b), and so I incorporate my discussion from § 6.3.3.

6.7.4 Step 7(c)

removing at least some of the automatically expired records from the linked list when the linked list is accessed, and

6.7.4.1 Court's Construction

The Court has construed the following from term 7(c):

Claim Term	Court's Construction
removing . . . from the linked list	adjusting the pointer in the linked list to bypass the previously identified expired records
linked list	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

6.7.4.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of identifying at least some of the automatically expired ones of the records when the linked list is accessed. This step is the same step of claim 3(c), and so I incorporate my discussion from § 6.3.4.

6.7.5 Step 7(d)

inserting, retrieving or deleting one of the records from the system following the step of removing.

6.7.5.1 Court's Construction

The Court has construed the following from term 1(a):

Claim Term	Court's Construction
a linked list to store and provide access to records	a list, capable of containing two or more records, in which each record contains a pointer to the next record or information indicating there is no next record
automatically expiring	becoming obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time

6.7.5.2 Analysis

When a Linux System searches for a route in the routing cache, it performs a method that includes the step of inserting, retrieving or deleting one of the records from the system following the step of removing. The removing of an expired record discussed in § 6.1.4 executes before the insert step discussed in § 6.1.5 executes.

6.7.6 Ordering of the Steps

6.7.6.1 Court's Construction

The Court has construed the following order of the steps of claim 7: the identifying step must begin before the removing step can begin, and the ultimate step of claim 7 must follow, or at least partially follow, the penultimate step of claim 7.

6.7.6.2 Analysis

When a Linux System searches for a route in the routing cache, the identifying step of claim 7 begins before the removing step begins. As discussed in § 6.1.4, the identifying of expired records using *rt_is_expired()* begins before the removal of the expired records, which is performed using the pointer adjustment. In fact, in a Linux System, the identifying step is completed before the removal begins.

6.8 Dependent Claim 8

6.8.1 Claim Language

> *The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.*

6.8.1.1 Court's Construction

The Court has construed the following from claim 8:

Claim Term	Court's Construction
dynamically determining	making a decision based on factors internal or external to the information storage and retrieval system
expired	obsolete and therefore no longer needed or desired in the storage system because of some condition, event, or period of time
when the linked list is accessed	both identification and removal of the automatically expired record(s) occurs during the same access of the linked list

6.8.1.2 Analysis

When a Linux System searches for a route in the routing cache, the Linux System performs a method that includes the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. As claim 8 is dependent on claim 7, I incorporate my discussion of claim 7 from § 6.7.

As discussed in § 6.2, a Linux System compares *chain_length* against *ip_rt_gc_elasticity* to decide whether to remove the record identified by the candidate on-the-fly removal. *chain_length* is a factor internal to the information storage and retrieval system as it measures the length of the object linked list and is subject to change during runtime.

7. Response to the Defendants' Non-Infringement Positions

I have reviewed the Defendants' positions on non-infringement. I disagree with the Defendants' positions and give my reasons below. I note that it is my understanding that Bedrock has accused all versions of the Linux operating system from 2.4.22 onwards.

7.1 "When the Linked List is Accessed"

The Defendants state, "The accused code does not remove expired records 'when the linked list is accessed.'" The Defendants essentially argue that the removal occurs during a separate a distinct access of the linked list. I disagree. The Defendants appear to read in a requirement that "when the linked list is accessed" means "during the same while-loop." The removal action is on the same linked list being operated upon in the while loop. As explained in the infringement analysis, this is during the same access of the linked list.

7.2 "Expired"

The Defendants state, "In the accused code, the record that is removed is not 'expired.'" They argue that because the record could contain a valid IP route that the record is not "expired." Further, they argue that the process of using *rt_score()* to select a candidate for deletion does not identify an expired record. I disagree. The Defendants appear to require that "expired" essentially means "contains no information that may be useful in the future." Yet, even the information the Defendants point to (e.g., "time the record was last used") indicates the idea that the record can be "no longer needed or desired." As explained in my analysis above, this process does identify an expired record and, in fact, the identification of the lowest scoring record (or,

depending on the linked list, no record) for deletion is because route.c is infringing claim 2 as well. Further, the cache is not a permanent location for store, it is a convenient area to keep a reasonably small set of information for fast access; the decision to remove a record can be based on the desire to keep it in the cache as weighed against other entries in the cache.

7.3 "Dynamically Determining"

The Defendants state, "No version of the Accused Linux Kernels contains code for 'dynamically determining maximum number of records to remove.'" The Defendants argue that determining whether to remove 0 or 1 records is not determining a maximum number of records to remove. I disagree and have explained those reasons in my infringement analysis. It is my opinion that 0 and 1 are numbers and that, by choosing 0 or 1, a maximum number has been chosen. Further, contrary to the Defendants' argument, this is a dynamic decision based on internal or external conditions, including, in part, the length of the hash chain.

7.4 "No Evidence of Execution of the Method Claims"

The Defendants state "There is no evidence to show that the accused code has executed on the Defendant's systems." I disagree. I have addressed this statement in this report through my experimental results and my analysis regarding the individual Defendants.

8. Acts of Infringement under 35 U.S.C. § 271(a)

35 U.S.C. § 271(a) reads:

> Except as otherwise provided in this title, whoever without authority makes, uses, offers to sell, or sells any patented invention, within the United States, or imports into the United States any patented invention during the term of the patent therefor, infringes the patent.

Based on my analysis, computer equipment configured with or utilizing software based on Linux version 2.4.22 and onwards practices claims 1-4 and 7-8 of the '120 patent and therefore embodies Bedrock's patented invention. Further, computer equipment configured with

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.

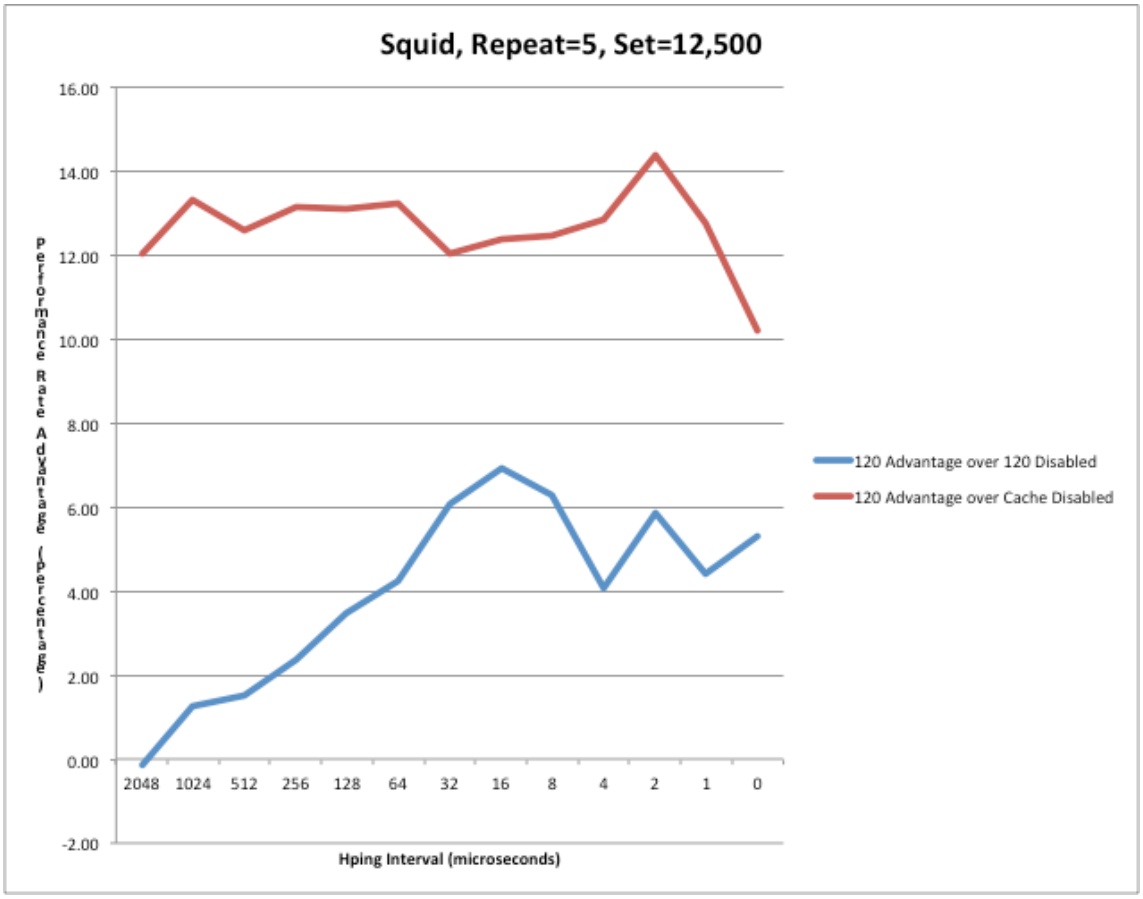


Figure 1: Results for the Squid Application for Repeat=5 and Set=12,500. Details are explained in the experimental results discussion.

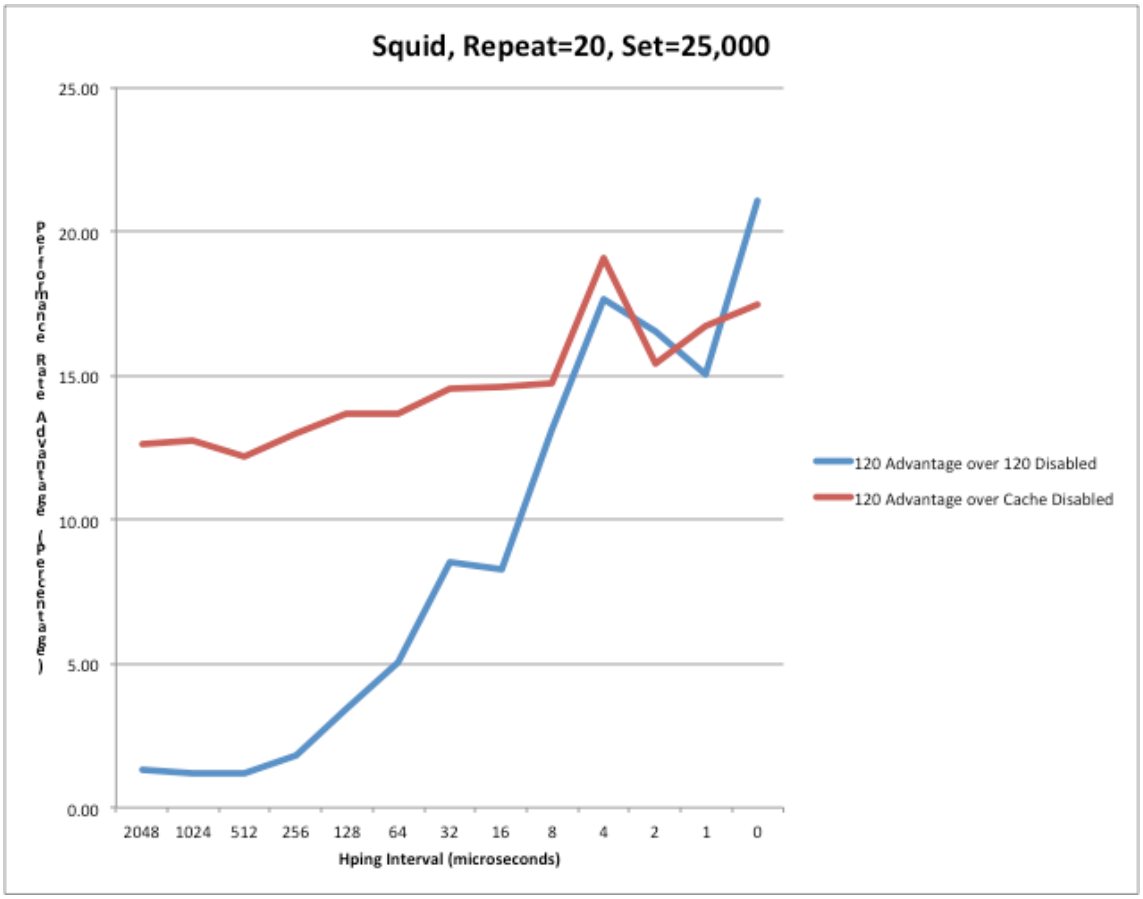


Figure 2: Results for the Squid Application for Repeat=20 and Set=25,000. Details are explained in the experimental results discussion.

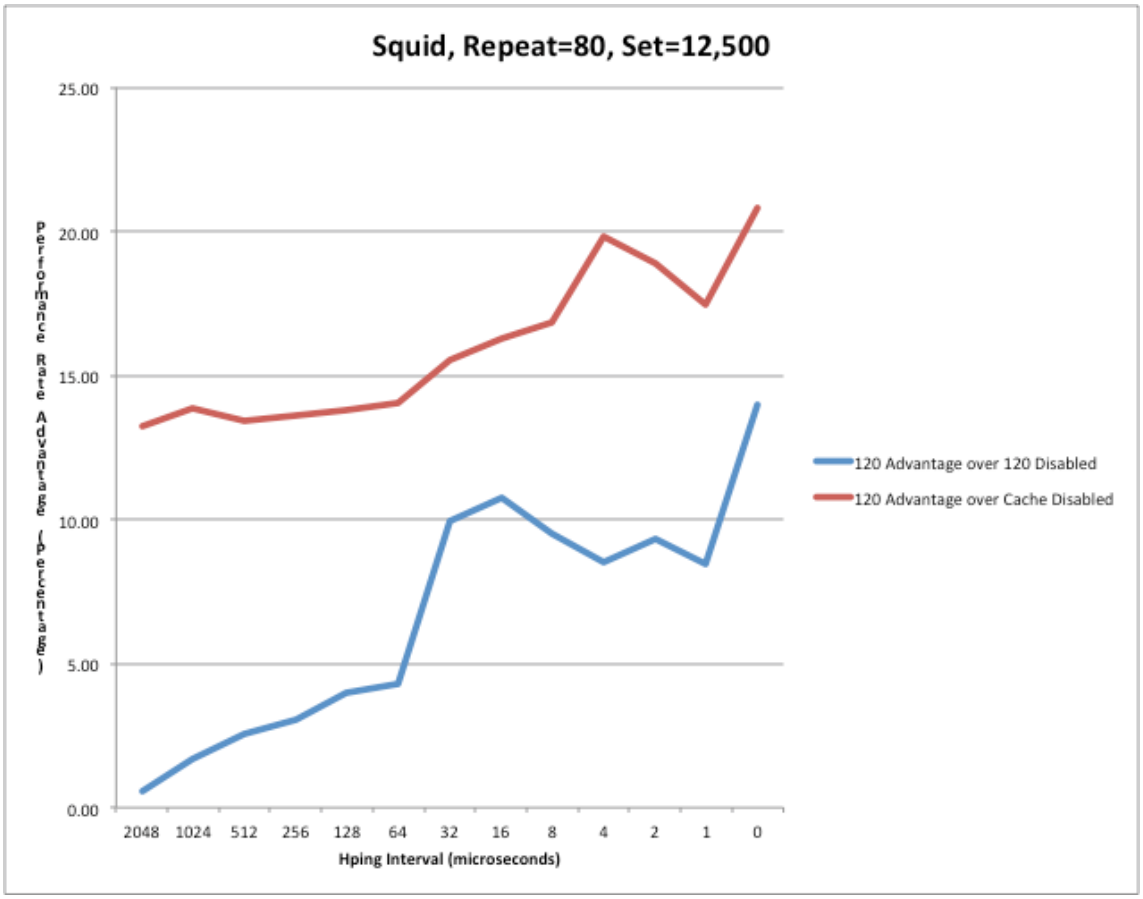


Figure 3: Results for the Squid Application for Repeat=80 and Set=80,000. Details are explained in the experimental results discussion.

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.

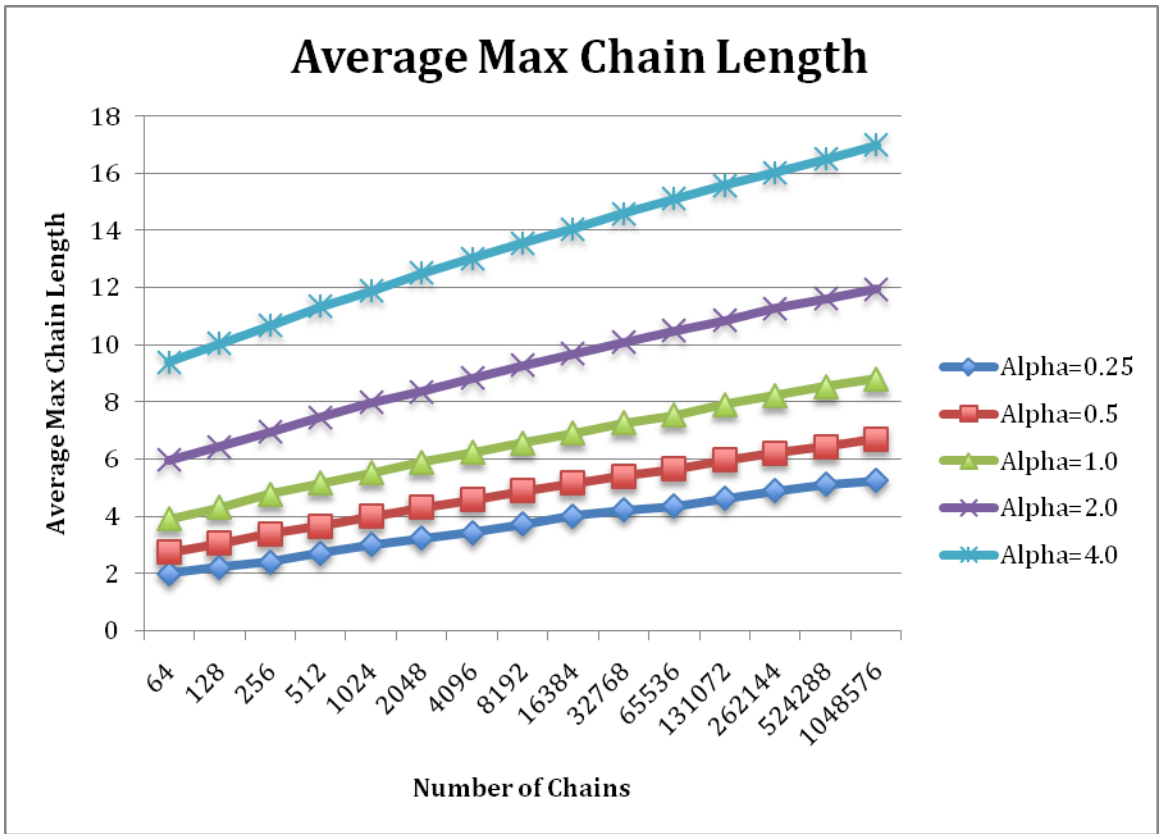


Figure 4 shows the average maximum chain length as a function of the number of chains for several different values of alpha; these results were computed from 1000 runs for each value of alpha. For the same conditions, Figure 5, Figure 6, and Figure 7 show the percentage of runs for which the maximum 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.

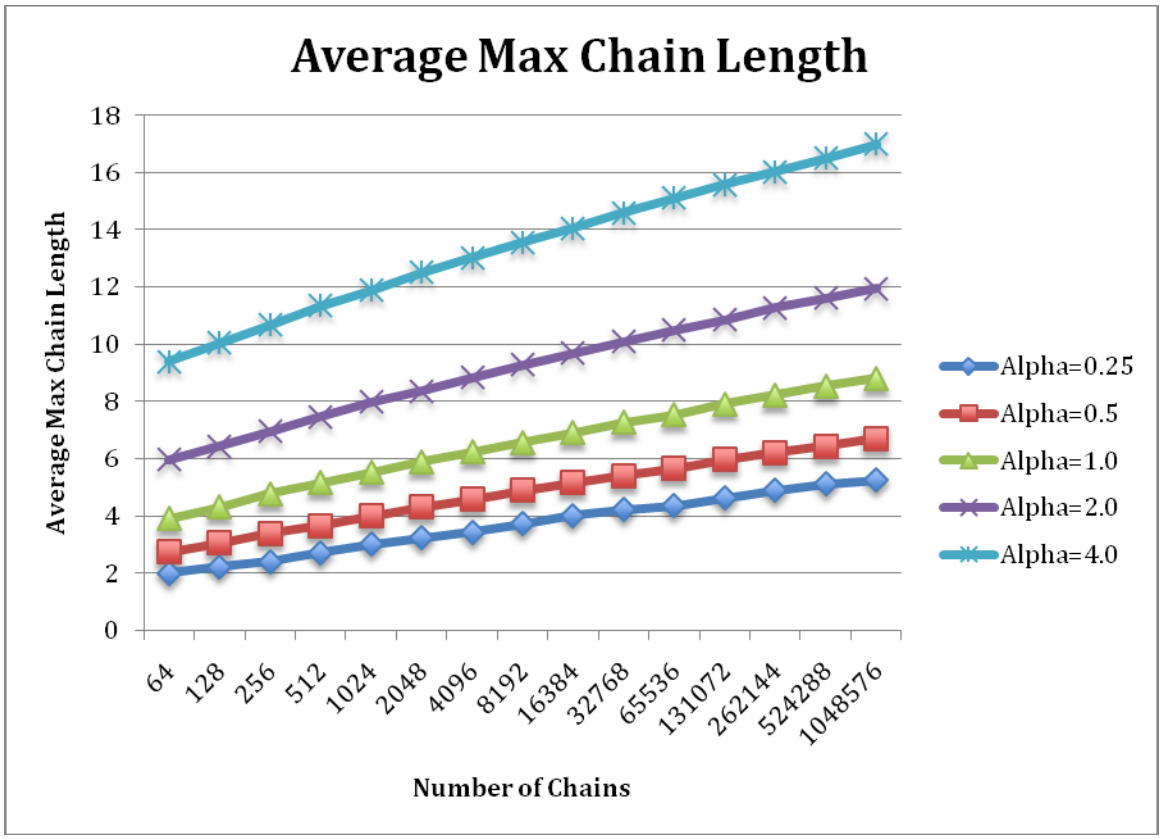


Figure 4: Experimental results from 1000 runs for each value of alpha

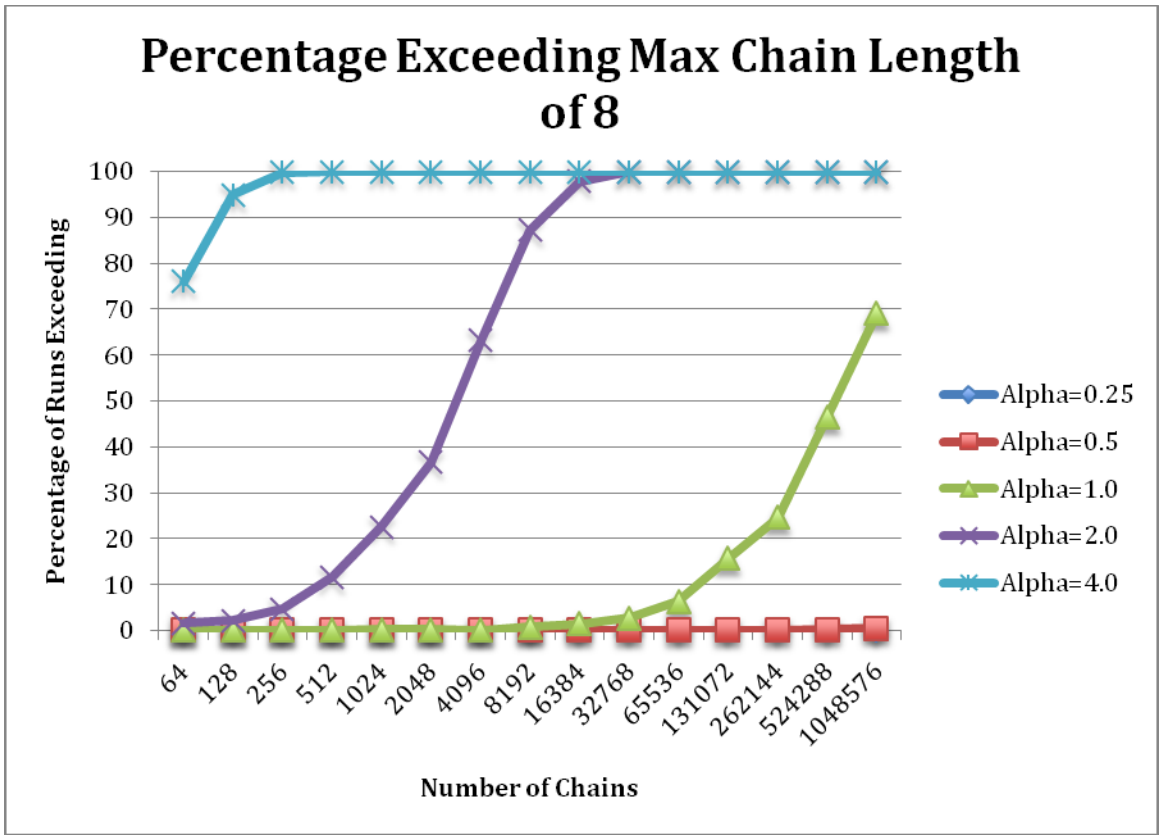


Figure 5: Percentage of experiments where the maximum chain length exceeds 8

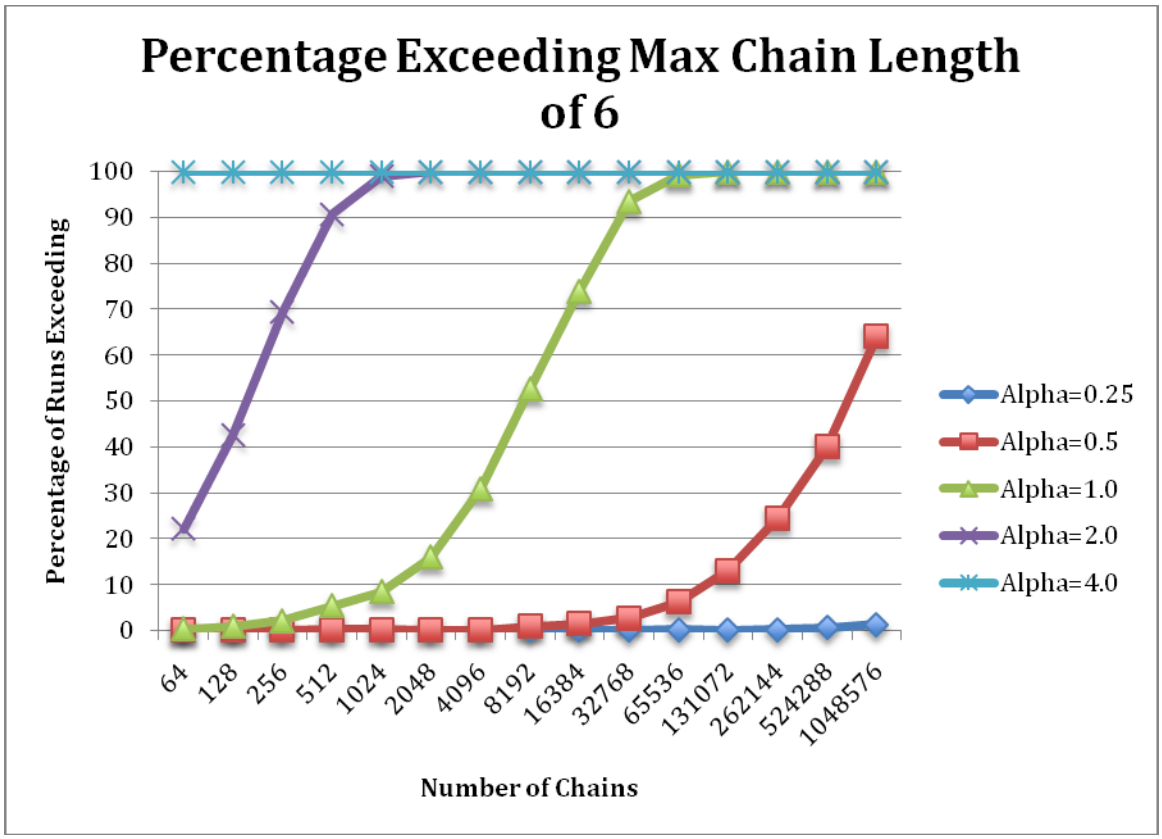


Figure 6: Percentage of experiments where the maximum chain length exceeds 6

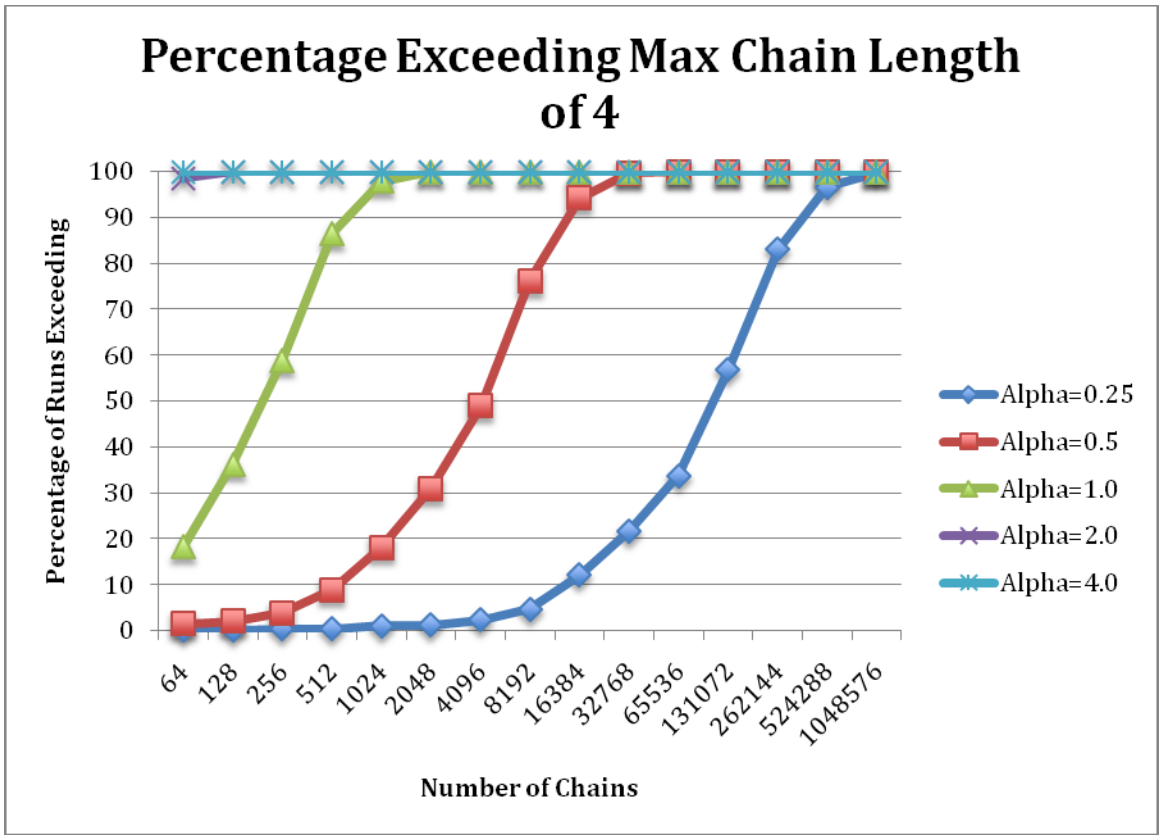


Figure 7: Percentage of experiments where the maximum chain length exceeds 4

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

owned by the Defendants based on checking the owner of the IP address.[44] This chart is the summary of HTTP connections between my computer[45] and sites on the Internet from the time I start the operation by hitting "enter" at the browser until the time the browser indicates that it has completed loading the new page. I performed these captures using WireShark, a commonly used tool for analyzing computer network traffic. The results indicate that, even for these fairly simple queries, there is a wide variety of characteristics in the traffic.[46] As one would expect, the volume of traffic from my computer (the client) to the servers (shown as "Bytes A->B") is lower than the traffic returned by the servers (shown as "Bytes A<-B"). The typical number of packets sent from my computer to the servers has a wide range both per connection and per server. The number of bytes per packet sent from my computer to the servers is typically in the 100-300 bytes range. The observed transmission rates ("bps A->B") exhibit a wide range as well.[47]

The following is a brief summary of the operation I performed for each Defendant.

- Google: I performed a search on the phrase "Google Chrome" at www.google.com (instant search was turned off) in my web browser. In a second search, I searched for "Using linux with an Ubuntu distribution download from Google" (instant search was turned on). Note that these are just two of the wide range of services and pages that Google provides.

[44] This check was done using sites such as www.domaintools.com

[45] In the titles of the chart, my computer would be Address A, which is not shown as it does not change.

[46] Note that this discussion is for the confirmed IP addresses highlighted in yellow.

[47] Note that the values are in units of bits per second and are not the rate at which a single packet is transmitted, but rather the number of bits divided by the duration of the connection. Note that I have tested the uplink speed of my connection to the Internet and found it to have a maximum of approximately 700Kbps.

- Yahoo: I went to "yahoo.com" in my web browser. Note that this is the page that Yahoo serves to a user who has not logged in as a registered Yahoo user. Yahoo offers a wider range of services.
- MySpace: I went to myspace.com in my web browser. Note that several of the IP addresses appear to be Akamai servers, which is consistent with MySpace's testimony regarding their operations. Note that this is the page that MySpace serves to a user who has not logged in as a registered MySpace user. MySpace offers a wide array of different pages and activities.
- AOL: I went to aol.com in my web browser. Note that this is the page that AOL serves to a user who has not logged in as a registered AOL user. AOL offers a wide array of different pages and activities.
- Amazon: At the Amazon.com site, I searched in the "Books" category for "Ubuntu 10.10 server". This is just one of many pages that Amazon serves to users. Further, Amazon hosts a wide variety of sites through its "EC2" facility.
- Softlayer: I visited softlayer.com. Note that Softlayer's primary business is hosting for its customers; this access did not visit those customers' sites.
- Match.com: I visited Match.com. This was a visit to Match.com's main page and did not access the services that Match.com provides to its registered users.

10.4 Implications for the Defendants' Server Computers

In this section, I summarize the advantages discussed above for companies using servers as part of a datacenter operating on the Internet. Each of the defendants operates many servers[48] with a variety of workloads and server configurations where those workloads and configurations have changed over time.[49] In some cases, the defendants have not been able to track those workloads and configurations (and are, in some cases, currently unable to do so). It is apparent that, in general, the current specific peak workload of a server is not important to the Defendants

[48] Note that when I use the term server here, I am referring to computers running the accused versions of Linux.

[49] Details for each defendant are given in the subsections below.

because no Defendant's corporate representative was able to answer with specificity basic questions regarding the peak traffic loads encountered on their servers (note that this is the peak, not, for example, the average peak or the peak during a small observed window in time).

The Defendants rely upon a large numbers of servers with a very small number of configurations to service the changing workloads to which their datacenters are subjected. Each Defendant has sets of servers that encounter different types of network traffic; one set of servers may encounter very high volumes of IP traffic from across the Internet[50] while another set of servers, for example those hosting database services for a Web server, may see no traffic that arrives directly from the Internet. Defendants state that routinely change the role of server from one to another set over time. For Defendants who provide "hosting" services,[51] such as Amazon and Softlayer, the workload for a particular server will vary depending on the client being hosted on that server and may, when hosting virtual servers, vary depending on which virtual server(s) is currently running on that server.[52] Essentially, Defendants that provide hosting services are renting their general server capacity to their clients and claim to provide efficient systems to their clients.[53] The performance and value provided by hosting services is evaluated and reviewed in publications by third parties.

To summarize the discussion above, the Defendants all rely on general-purpose servers that run the accused versions of Linux. The defendants have used a variety of hardware and

[50] See, for example, the discussion below regarding Amazon and its Blackfoot servers.

[51] Note that both Amazon and Softlayer have sets of servers that are not used for hosting clients.

[52] Both Amazon and Softlayer indicated that hosting clients can configure/reconfigure a guest virtual machine (and, in the case of Softlayer, can reconfigure the operating system of the server). Softlayer and Amazon state that they do not have knowledge of their hosting clients specific workloads and configurations and that these vary by client.

[53] See discussions below for specific clients, including their representation of their hosting services.

operating system configurations and the servers are subjected (and have been over time) to a wide variety of workloads. It is important to the defendants that they be able to efficiently operate their servers across a variety of workloads.[54] As discussed above, the Linux route cache with the '120 invention as claimed provides for increased efficiency across a wide range of operating conditions. Based on the evidence that I have reviewed, the overwhelming majority, if not all, of the Defendants' servers receive network traffic, whether that traffic is exchanged directly with the Internet or is between computers in the datacenter. In my experiments, under a range of operating conditions, the route cache with the claims of the '120 invention generally provides an advantage of 10% or more over not using the route cache. Further, the route cache with the claims of the '120 invention is more efficient than the route cache without the claims of the '120 invention across a range of workloads and avoids large degradations in performance see without the claims of the '120 invention seen in certain workloads. Thus, the claims of the '120 invention allows the advantages of the route cache to be achieved under a wider range of conditions, allowing for more efficient operation of the Defendants' servers.

I note that the default condition for every public version Linux containing the claims of the '120 invention which I have examined has the route cache enabled by default.[55] Disabling the claims of the '120 invention is not an option in any public n version and is only possible through source code modification. One Defendant, AOL, has indicated that they plan to do a phased rollout of disabling the route cache on their servers (see discussion below). Even if the route cache is disabled via the sysctl mechanism, that does not remove infringement because the

[54] *See* BTEX0752263.

[55] In some versions, there is no option to disable the route cache.

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

_____

Mark T. Jones