ABSTRACT

Venugopalan Ullanatt. Summary Representation for Service Discovery Protocols. (Under the direction of Dr.Injong Rhee.)

Recent advances in technology have led to the widespread deployment of computational resources and network-enabled end-devices. This poses new challenges to network engineers: how to locate a particular service or device out of hundreds of thousands of accessible services and devices. One of the major issues involved is the efficient storage, retrieval and dissemination of information about available services. Well known relational database techniques are not very efficient in these situations because our primary concern is the determination of availability of a service, not the retrieval of data. Also, database techniques involve additional overhead for indexing and query processing. We propose a novel scheme for efficient determination of the availability of services called SRDP(Summary Representation for service Discovery Protocols). SRDP makes use of a sub-string search algorithm based on hashing techniques. For this purpose, service descriptions are treated as strings and queries are treated as sub-strings. Information about each service and its attributes is stored as a 128 bit signature in a hash table. To exploit all bits of the signature, a signature creation scheme using the characteristics of the distribution of characters in English language is employed. For the hash table, a *Fibonacci* hash based scheme and a *CRC* hash based scheme using primitive polynomials are tested for their effectiveness as hash functions. Results are presented from tests performed using actual URL data obtained from the Internet. Finally we compare the performance and memory requirements of our scheme with a *Bloom-filter* based approach. Results show that SRDP executes twice as fast, consumes 80% less memory and still provides false drop probabilities comparable to a Bloom filter based approach.

Summary Representation for Service Discovery Protocols

by

Venugopalan Ullanatt

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh

2001

APPROVED BY:

Chair of Advisory Committee

BIOGRAPHY

Venugopalan Ullanatt was born in Kerala, India in 1970. He received his B.Tech degree in Electronics & Communication Engineering from Regional Engineering College,Calicut in 1992. He worked as a Software Engineer before joining the graduate program in the Computer Science department at North Carolina State University in 1999. He is currently working towards completion of M.S. in Computer Science.

To

Achan and Amma

ACKNOWLEDGEMENTS

I would like to thank Dr.Injong Rhee who was primarily responsible for my returning to Graduate School as a full time student. I am very grateful to him for appointing me as his research assistant right from my first semester as a full time student. He has always urged to innovate and refine our ideas which resulted in this thesis. He also encourgaged free thinking and experimentation.

Thanks are also due to Dr.George Rouskas for giving me constructive advice during the course of my thesis work. I would like to thank Dr.Wenke Lee for serving on my thesis committee.

I would like to thank friends Jason Walter, Thomas Alspaugh and Mahmoud Elhaddad for helping me with my LaTeX doubts. Dr.Gary Stelling was always there to help me with the computational needs. A special thanks to John Borwick for helping me to collect URL data from the Internet which proved very helpful during testing.

Last, but not the least, my family has always been with me in all my decisions. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we develop and analyze algorithms for summary representation in service discovery protocols. The major contribution of this thesis is the design of an efficient service discovery protocol using a signature based hashing scheme for summary representation called SRDP (Summary Representation for service Discovery Protocols).

Numerous summary representation techniques were developed in the seventies for applications in data base research. Recently, one such technique called Bloom Filters was successfully applied in Summary Cache [13] which was integrated with the Internet Caching Protocol [9].

1.1 The service discovery problem

In recent years, the Internet has evolved from a tool for sharing data among researchers to a powerful commercial medium for conducting business and sharing all kinds of information. Currently, most Internet transactions are initiated by entering a Uniform Resource Locator (URL). URLs are the foundation of the World Wide Web. However they have proven to be limited in practice. The basic problem is that URLs typically identify a particular path to a file on a particular host. There is no graceful way of changing the path or host once the URL has been assigned. Neither is there a graceful way of replicating the resource located by a URL to achieve better network utilization or fault tolerance.

The decreasing cost of networking technology and network-enabled devices is enabling the large-scale deployment of such networks and devices. Simultaneously, significant computational resources are being deployed within the network infrastructure; this com-

putational infrastructure is being used to offer many new and innovative services to users of these network-enabled devices. We define *services* to be applications with well-known interfaces that perform computation or actions on behalf of client users. For example a remote fax server that allows a PDA user to send fax is a service. Other examples are printers, mp3 servers and stock tickers.

We believe that one major characteristic of services will be that users may not be willing to access them in the traditional way of specifying hostname/IP address. Instead, users would express their *intent* for a service and the network infrastructure should be able to direct the service request to an appropriate server. Just as there are hundreds of web servers, there will be thousands of services available to users.

Thus, clients will require a directory service that enables them to discover services. This will give rise to mainly two problems: i) a protocol to disseminate and locate service information and ii) data structures and algorithms to efficiently represent the large amount of information about services. In this thesis we present ways to tackle the problem of efficient representation of data.

1.2 Summary representation

For the purpose of discussion in this thesis, we will treat service descriptions as being composed of a service name and a set of attributes. We use '/' as the delimiting character to delimit service names, attributes, etc.

For example, *service=printer/building=egrc/room=409/type=laser* describes a service called printer whose attributes are building, room,and type.

Service descriptions could be thought of as free form text strings. Thus, we transform the problem of looking up a service request in a database to the problem of sub-string match in a database. The conventional text search algorithm is to search the string for a sub-string match by doing a character by character comparison. Since service requests can have attributes specified in any order, we need to search for each attribute separately. For example, a user could send a request for *service=printer/type=laser/building=egrc/room=409*. Even though the attributes in the request are the same as that present in the service description, an exact matching of the whole string will not work, because the order of attributes is different from that in the database. We need to match each attribute separately.

Summary representation on the other hand addresses the sub-string search problem

in a different way. The service description string is transformed into a bit vector that we call the summary representation of the service description string. When a query has to be performed, the query string is transformed to a bit vector and compared against the bit vector representing the service description string. This comparison could be performed in a few operations on most computers.

In this thesis, we will examine two summary representation techniques i) Bloom filter based and ii)signature based.

1.2.1 Summary representation using Bloom Filter

A Bloom filter is an encoding of the keys of a file into a single bit string. As the name suggests, it is a filter for quickly eliminating non-matches (those records which do not exist in the file) from further consideration. It was invented by Burton Bloom in 1970 [5].

The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \ldots h_k$ each with a range $1...m$. For each input key, the bit positions $h_1(key), h_2(key) \ldots h_k(key)$ are set to 1. The salient feature of Bloom filter is that the bit vector size m and the number of hash functions k can be tuned to achieve different false drop probabilities.

1.2.2 Summary representation using signatures (SRDP)

In SRDP, keys are transformed into m bit signatures using a hash function. m is usually chosen to be a multiple of the word size of the computer. The difference between Bloom filter and this approach is that there is a separate signature for each key. The Bloom filter on the other hand stores signatures for all keys in a single vector. To reduce the search time, however, a two-step procedure is used in SRDP. As mentioned before, a service description is composed of a service name followed by a list of attributes. We treat the service name part of a service description as a *primary key* and map it into a hash table. The signature for the service description is then inserted into the corresponding hash table slot. Chaining is used in case of collisions. The *hash table size* and the *signature size* are *tunable* parameters to achieve different false drop probabilities.

The hash table size is determined by the number of services. For example if we have around 65000 services, we choose a hash table of size 65536($=2^{16}$).The hash function is carefully chosen according to the nature of the input data to reduce collisions.

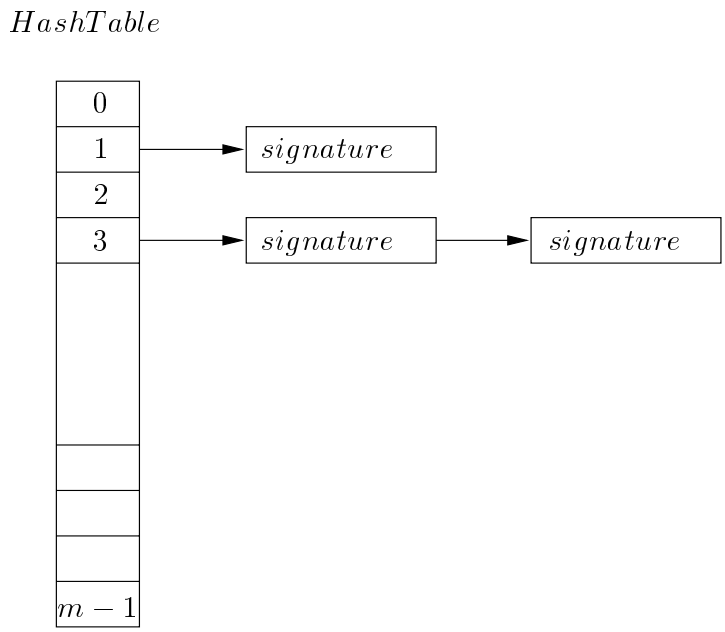

Figure 1.1: Summary representation using signatures

1.3 Searching using summary representation

In order to perform a search using summary representation, the query string is first transformed to a bit vector and compared against the bit vector representing the service description string stored in the database. For illustration, assume that we have just one entry in our database. It is a bit vector, 10011010, representing the service description for the *printer* service and its attributes. When we get an arbitrary request for service, the query service and its attributes are first transformed into a bit vector. If that bit vector has 1s in any position other than 2,4,5 or 8 we will immediately declare that the service is not available. On the other hand, if the transformed query string has 1s in only positions 2, 4, 5 or 8, we will declare that the service is available in the domain.

Now, we will consider how searching is done in Bloom Filter method and signature method.

In the Bloom filter approach, given a query for a string, we compute the values h_1(string),h_2(string),...,h_k(string) and check if the bit vector in database has 1s in those positions. If any of them has a 0 in the database, then the search string is definitely not present in the file. A match is found only if all h_k positions have 1s in the database.

In SRDP, to query a service, the hash table slot and signature have to be calculated. In case of a collision at the particular hash table slot, the whole chain has to be searched before declaring the absence of a service. We will declare a match only if the signature is found at the particular hash slot. A positive match of signatures signals that the service may be present. As done in Bloom Filter approach, a match only means that we have found a signature in our database with 1s in all positions where the request string has 1s. The database string could have either 1 or 0 in those positions where the transformed query string has a 0.

1.3.1 False drop problem

A search technique employing summary representation will declare that a match is found if the database signature has 1s whereever the query signature has 1s. Also, the summary representation is not unique because the transform functions may produce the same output signature for two different input strings. There may be situations in which there is a signature match even though the service is not really present the database. Such an occurence is called a *false drop*. A design challenge is to reduce the false drop probability.

1.4 Design objectives

Our goal in designing SRDP is to provide a computationally efficient and less memory hungry algorithm without increasing false drop probability.

Bloom filters [5] have been proposed as a possible mechanism for lossy aggregation and query routing in Secure Service Discovery Service [8]. A service can have many attributes and in the Bloom filter approach, every combination of attributes has to be treated as a separate record. This approach faces some challenges when the number of attributes for a service is large. For example, if a service has three attributes, the Bloom filter approach is to compute hash values for each combination of zero attributes, one attribute, two attributes and three attributes. The number of combinations for n attributes is 2^n. Even a small number of attributes, say 10, gives rise to 1024 combinations. Also, in order to reduce false drops, the scheme proposed in [8] uses MD5 [35] hashing algorithm to calculate the signature associated with every (service, attribute) combination. As explained in [41], MD5 is a very expensive operation and is very difficult to optimize because of the inter-dependencies among various stages of computation.

We propose a scheme (SRDP) where there is a single calculation done for each service irrespective of the number of attributes. This is made possible by employing a string search algorithm. Unlike the Bloom filter approach, we compute individual bit vectors for each record and since we use a combination of hash table and signature, we do not need a very powerful hash function like MD5.

1.5 Related Work

In recent years there has been a lot of interest in service discovery. Most of the research work has been focused in the area of designing protocols for service discovery. Anycast [4], SLP [18], LDAP [43] etc. are different protocols which try to address the problem of service discovery. At the time of this writing, we haven't seen much work on summary representation for service discovery protocols. Secure Service Discovery Service (SDS) [8] and INS (Intentional Name System) [1] have addressed the issue of service representation and retrieval. There have also been some standardization efforts on how to address resources. We present a survey of the different approaches below.

1.5.1 Service Location and addressing

Uniform Resource Names(URNs) have been proposed to serve as persistent, location independent identifiers for Internet resources overcoming most of the problems associated with URLs. A major motivation for a URN based naming scheme is to decouple naming schemes from resolution schemes. This allows multiple naming approaches and resolution approaches to compete as it allows different protocols and resolvers to be used. DNS is a possible candidate for name resolution because of its ubiquitous deployment. However, it is not appropriate to use DNS to store information on a per-resource basis. But it is possible to use DNS to identify resolvers that can provide information on individual resources. RFC 2168 [10] refers to this as a Resolver Discovery Service (RDS). RDS is implemented by a new DNS Resource Record, NAPTR(Naming Authority PoinTeR), that provides rules for mapping parts of a URN to domain names.

The Service Location Protocol (SLP) [18] provides a flexible and scalable framework for providing hosts with access to information about the existence, location, and configuration of networked resources. Traditionally, users have had to find services by knowing the names of a host. SLP eliminates the need for a user to know the name of a network

host supporting a service. Rather, the user supplies the desired type of service and a set of attributes that describe the service. Based on that description, the SLP resolves the network address of a service for the user. SLP provides a dynamic configuration mechanism for applications in local area networks. Applications are modeled as clients that need to find servers attached to any of the available networks within an enterprise. Servers advertise their services to Directory Agents (DAs). For cases where there are many different clients and/or services available, the protocol is adapted to make use of nearby Directory Agents that offer a centralized repository for advertised services.

Light-weight Directory Access Protocol (LDAP) [43] is designed to provide access to X.500 Directory while not incurring the resource requirements of the Directory Access Protocol. It is specifically targeted at simple management applications and browser applications that provide simple read/write interactive access to the X.500 Directory.

Anycasting [4] provides service location with the help of IP layer. An IP anycast address is used to define a group of servers that provide the same service. A sender desiring to communicate with only one of the servers sends datagrams to the anycast address. The datagram is then routed to at least one of the servers (preferably to the best according to some criteria) identified by the anycast address.

Active Names [42] is another approach where a service is mapped to a chain of mobile programs that can customize how a service is located and how its results are transformed and transported back to the client. Active Names offer a powerful platform for network computation, mobility, composition and location independence.

Prospero [31] also supports extensible naming to support mobile hosts and the integration of multiple wide area information services (e.g. WAIS and gopher). Prospero allows users to customize their own namespaces by grouping related information.

Intentional Naming System (INS) [1] argues that applications use naming to describe their *intent* as opposed to where to find it. INS uses declarative style data structures for maintaining *attribute-value* pairs used to bind a user specified name to an appropriate instance of the target resource. INS represents service descriptions using *name specifiers*. A *name specifier* consists of an *attribute* and a *value*. An *attribute* is a category in which an object can be classfied, for example the location of a printer. A *value* is the object's classification within that category, for example, the library. *Name specifiers* are a hierarchical arrangement of attribute-value pairs. The pairs are arranged in a tree such that a pair that is dependent on another is a descendent of it.

Sun's Jini [29] software provides for both the *Jini* connection technology and the *Jini* distributed system. Clients discover new hardware and software services using the *Jini* Lookup Service. A *Jini* searching mechanism uses the Java serialized object matching mechanism from JavaSpaces which is based on exact matching of serialized objects.

1.5.2 Summary representation

Superimposed coding, signatures etc. have been used in database related research from early seventies. Implementation of the sub-string test by hashing was proposed by Malcom C.Harrison [19] in 1971. Later, Tharp et al.[39] demonstrated the feasibility of Harrison's algorithm for text searching. Our signature creation part is largely adopted from their methodology. But their algorithm was primarily for text searching and had to scan the entire file for matching patterns. In our approach we reduce the search space by hashing service names to a hash table. The number of entries searched is thus reduced to the load factor of the hash table.This is possible because we can treat the service name part of a service description as a primary key to be mapped to a hash table slot.

Bloom filter was proposed for use in the web context by Marais and Bharat [27] as a mechanism for identifying which pages have associated comments stored within a CommonKnowledge server. They used Bloom filters for representing the server database contents and used this information for supporting cooperative and personal surfing with a desktop assistant.

Bloom Filters were successfully employed in Summary cache [13] for representing the cached set of URLs at each node. This was used in conjunction with MD5 Hashing scheme to reduce the likelihood of false drops. Every node constructs a Bloom Filter for its set of URLs and propagates this information to nearby caches.

SDS [8] also makes use of Bloom filters for service representation. In order to limit the number of attribute combinations, however, they allow only combinations of small number of attributes. For example if a service has five attributes, they allow only combinations of 1, 2 or 3 attributes. SDS uses XML to specify service requests. SDS also provides a secure framework for service discovery.

1.6 Organization

The rest of this document is organized as follows. Chapter 2 lays out the theoretical foundation for some of the principles used in the signature based hashing scheme and Bloom filter based hashing scheme. Chapter 3 describes the implementation of both schemes. Chapter 4 evaluates the performance of signature based summary representation and compares it with a Bloom filter based approach. Chapter 5 presents numerical results from the tests we conducted. Chapter 6 outlines conclusions and future research directions.

Chapter 2

Hashing algorithms and Summary representations

In this chapter we will lay the theoretical foundation for some of the concepts used in this thesis.

2.1 Hashing

A hash function h maps the universe U of keys into the slots of a hash table $T[0..m-1]$. $h : U \rightarrow 0, \ldots, m-1$ where m is the hash table size and is typically less than the size of U. A practical hash function must have the following characteristics

- it must be computationally simple.
- it must distribute keys evenly.

There are many types of hash functions. Some of the widely used hashing methods are

- hashing by division
- hashing by multiplication
- hashing by polynomial division (CRC)
- hashing by XOR folding

In our implementation we use a variant of the multiplicative hashing scheme called Finbonacci hashing.

2.1.1 Multiplicative hashing

The multiplicative method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then we multiply this value by m, the hash table size, and take the $floor$ of the result. In short, the hash functions is

$H(k) = \lfloor m(kA \ mod \ 1) \rfloor$,where $kA \ mod \ 1$ means the fractional part of kA.

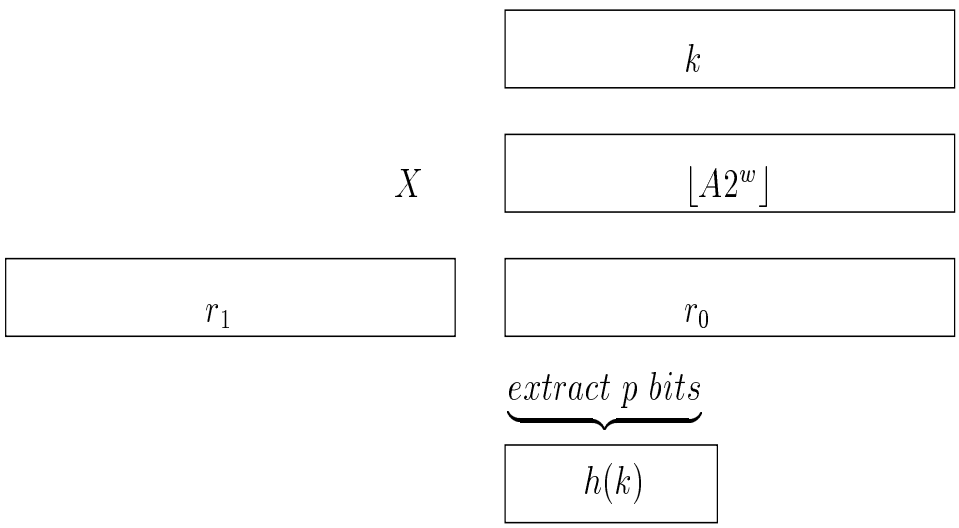

Figure 2.1: Multiplicative hashing

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2, i.e. $m = 2^p$ for some integer p - since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits and that k fits into a single word. Referring to Fugure 2.1, we first multiply k by the w bit integer $A.2^w$. The result is a $2w - bit$ value $r_1 2^w + r_0$ where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

One feature of the multiplicative hash method is that it makes good use of the non-randomness found in many keys. Quite often, actual sets of keys may have a prepon-

derance of arithmetic progressions, where $\{K, K+d, K+2d\ldots\}$ will appear in the set of keys. For example, consider the alphabetic names like $\{PART1, PART2, PART3\}$. The multiplicative hash method converts an arithmetic progression into an approximate arithmetic progression $\{h(k), h(k+d), h(k+2d)\}$ thus reducing the number of collisions from what we would expect in a random situation.

2.1.2 Fibonacci Hashing

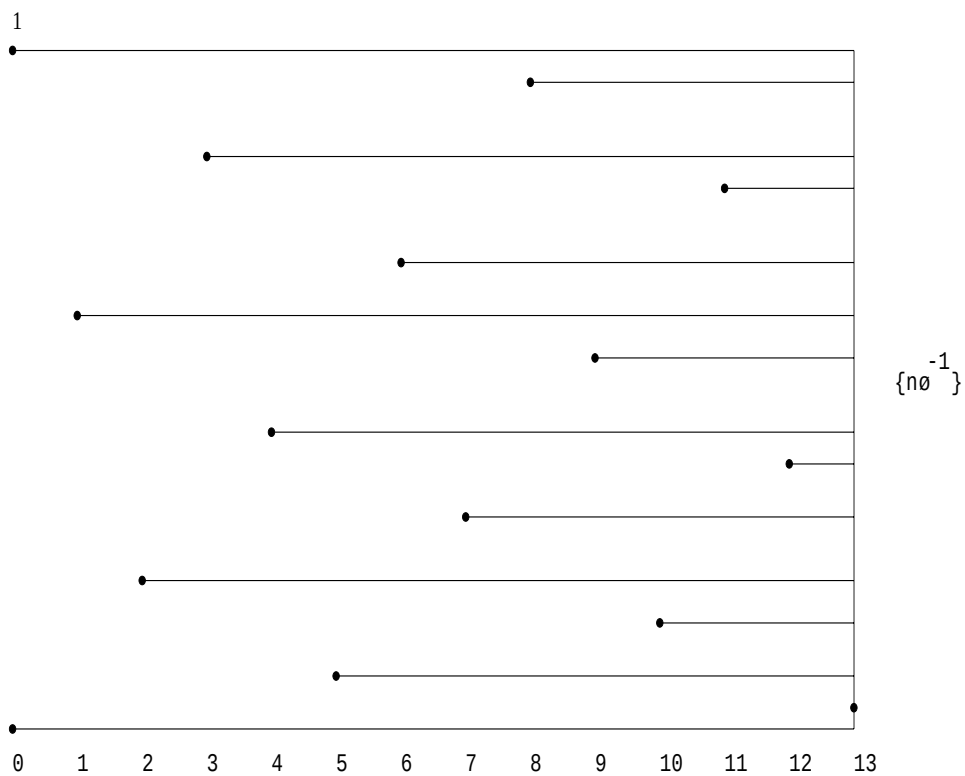

Figure 2.2: Fibonacci hashing

Although the multiplicative hash method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth [24] discusses the choice of A in some detail and suggests that the *golden ratio*, $\phi^{-1} = 0.6180339887$, works reasonably well in many situations. Figure 2.2 illustrates the remarkable properties of this ratio.

The behavior of successive values of $h(k), h(k+1), h(k+2)\ldots$ can be studied by considering the behavior of the successive values of $h(0), h(1), h(2)\ldots$This suggests the

following experiment. Starting with the line segment $[0, 1]$ we successively mark off the points $\{\phi^{-1}\}, \{2\phi^{-1}\}, \{3\phi^{-1}\}$ etc. where $\{x\}$ denotes the fractional part of x, namely, $x \ mod \ 1$. As shown in Figure 2.2, these points are very well separated from each other; in fact, each newly added point falls into one of the largest remaining intervals, and divides it in the golden ratio! What this means is that, say, if we have input keys 1,2,3,4 etc. and a hash table of size say 64, 1 will hash to 40, 2 will hash to 15, 3 will hash to 55, 4 will hash to 30 etc. i.e. each subsequent key falls into one of the largest remaining intervals.

A hashing scheme employing the above technique is called *Fibonacci hashing*. The name comes from the strong role *Fibonacci numbers* play in the proof of the above phenomenon. This hashing scheme will spread out alphabetic keys like {PART1, PART2, PART3} very nicely. The following table shows the multiplier for various key lengths.

key size(bits)	multiplier
16	40503
32	2654435769
64	11400714819323198485

Table 2.1: Golden Ratio Multiplier

2.1.3 MD5 hashing

MD5 [35] is a hashing function with excellent cryptographic properties. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit *fingerprint* or *message digest* of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be *compressed* in a secure manner before being encrypted with a private (secret) key under a public-key crypto-system such as RSA.

MD5 is a block-chained digest algorithm, computed over the data in phases of 512-byte blocks organized as little-endian 32-bit words (Figure 2.3). The first block is processed with an initial seed, resulting in a digest that becomes the seed for the next block. When the last block is computed, its digest is the digest for the entire stream.

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it

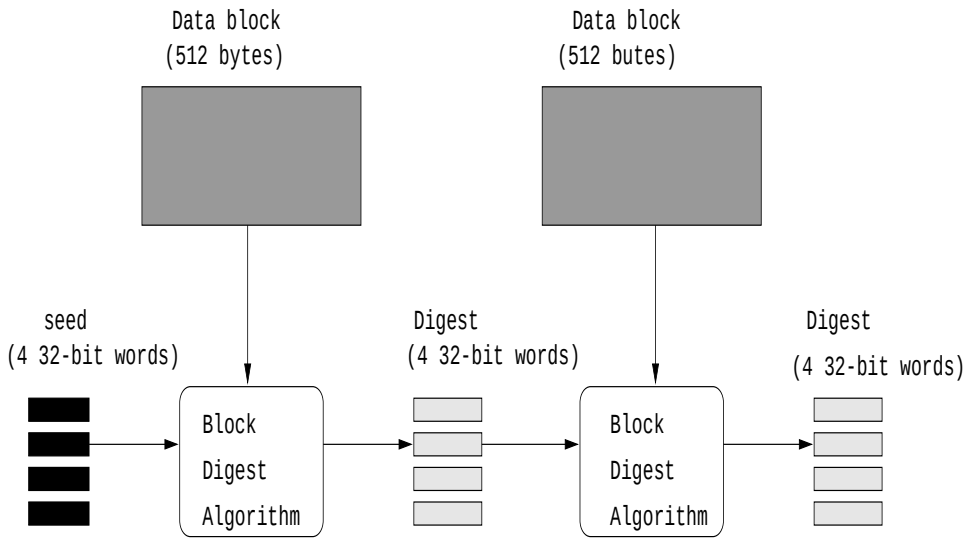

Figure 2.3: MD5 block-chanined digest algorithm

need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows: $m_0 m_1 \ldots m_{b-1}$

The following five steps are performed to compute the message digest of the message.

Step 1. Append Padding Bits

The message is *padded* (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. Padding is performed as follows: a single 1 bit is appended to the message, and then 0 bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

Step 2. Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^{64}, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first) At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let $M[0 \ldots N-1]$

denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD Buffer

A four-word buffer (A, B, C, D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first:

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

Step 4. Process Message in 512 bit Blocks (See [17] for details)

Each 512-byte block is digested in 4 phases. Each phase consists of 16 basic steps, for a total of 64 basic steps. Each step updates one word of a 4-word accumulated digest, using the entire intermediate digest as well as block data and constants. In general, each basic step depends on the output of the prior step. The basic structure of the steps is shown below (<<< denotes rotate).

```
A = B+((A+ F(B,C,D) + X[i++] + k1)<<<k2)

D = A+((D+ F(A,B,C) + X[i++] + k1)<<<k2)

C = D+((C+ F(D,A,B) + X[i++] + k1)<<<k2)

B = C+((B+ F(C,D,A) + X[i++] + k1)<<<k2)
```

There are 16 steps based on each of 4 logical functions; 4 based on F are shown here. The constants k1 and k2 are not necessarily identical in basic steps, and are not relevant to this analysis. The logical functions are(^ denotes xor):

```
F(x, y, z) = (((x) & (y)) | ((~x) & (z)))

G(x, y, z) = (((x) & (z)) | ((y) & (~z)))

H(x, y, z) = ((x) ^ (y) ^ (z))

I(x, y, z) = ((y) ^ ((x) | (~z)))
```

Step 5. Output

The message digest produced as output is A, B, C, D.

2.1.4 CRC Hashing

CRC hashing treats the input message as a polynomial. We represent a block of $(k+1)$ bits by a polynomial of degree k in the dummy variable x, written as $a_k x^k + \ldots + a_1 x^1 + a_0 x^0$ where a_k is 0 if the bit in that position is zero and 1 otherwise.

In order to compute the hash value, the input message polynomial is divided by a pre-defined polynomial called *generator* polynomial. The *coefficients* of the remainder polynomial give the hash value.

CRC codes are popular because they can be efficiently implemented in hardware or software. Hardware implementations require only a shift register and some XOR gates, uses no additional storage and can compute the CRC on the fly [45] . An efficient software implementation is to pre-compute the remainder for each possible data string of a certain length (say 16 bits) and store the result in a lookup table. The algorithm looks up the remainder for each 16-bit chunk of the input and adds these, using modulo addition, to a running sum. It can be shown that this is equivalent to computing the remainder over the entire string [11] . Thus, we can compute the CRC for an arbitrarily long input string on the fly in software, with only one lookup per block of input bits.

2.2 Sub-string test by hashing (Harrison's Method)

As mentioned earlier, we treat service descriptions as strings. Any service request can be treated as a sub-string and compared against the strings in the database for a match.

In this method, a service description string is compressed into an m-bit code referred to as the *signature* of the service description. m is usually chosen to be a multiple of the word size of the computer. The code is obtained by hashing all contiguous k-symbol groups (k is chosen to be 2 in our implementation) into an integer in the $[0..m-1]$ range. For each of these k-symbol groups, the bit position corresponding to the resulting hash value is set to 1.

Thus, a string of length $l+k-1$ can set a maximum of l bits in the signature. If $k=2$, for example, a string of length $l+1$ can set maximum of l bits.

For example, consider a service description:*service=printer/room=409/bldg=egrc*. If we take $k=2$, then all 2-symbol pairs would be hashed. i.e. $h(pr), h(ri), h(in), h(nt), h(te), h(er), h(ro), h(oo), h(om) \ldots h(rc)$ where $h()$) is the hash function. Note that we omit the

key word *service* and the delimiter '/' while hashing. Also, hashing does not occur between characters from two different attributes or between service name and attributes. For example *rr* is not hashed, where *r* is the last character of service name *printer* and *r* is the first character of attribute *room*. This is done to take care of each attribute separately because attributes may appear in any order in a service request. Otherwise if we get a request with attributes specified in a different order as in *service=printer/bldg=egrc/room=409*, the test will fail.

This scheme makes use of three ideas. The *first* is that if the search is likely to be unsuccessful, it can usefully be preceded by a computationally faster test for necessary but not sufficient conditions that the substring be found.

The *second* is that a string can be represented by the set of its substrings (or attributes in our case). In general such a representation is not unique, but it does preserve the substring property in the sense that, if one string has another string as substring, the set of substrings of the first will include the set of substrings of the second.

The *third* is that a set S can be represented by a binary string $b_1b_2b_3b_4 \ldots b_n$ in which a value of 1 for b_i indicates that S contains at least one element of some set E_i. The set E_i in our implementation is a set of 2 character symbols ($ab, 12, a1$ etc). In general such a representation is not unique, unless each E_i contains exactly one element and each possible element is contained in some E_i . However, it preserves the subset property in the sense that, if set S_1 is a subset of S_2 , the binary string representing S_2 will have ones in all positions where the string representing S_1 has ones.

Accordingly consider a string S to be represented by a binary string $b_1b_2b_3b_4 \ldots b_m$ constructed as follows.

- Set all bits b_i to zero.
- For each substring s of length k, compute $i = h(s)$ and set b_i to zero.

This uses a hashing function $h()$ which is assumed to give an integer result in the range $[1..m]$. The resulting binary string, which contains a 1-bit only in positions that correspond to certain substrings of length k, is called the hashed k-signature of the string S. A hashed signature is illustrated in Figure 2.4. It shows a 10 character string S and its 5-bit signature.

It is clear that for any particular hashing function, a necessary condition for string S_2 to be a substring of S_1 is that the hashed k-signature of S_2 have 1's wherever the hashed

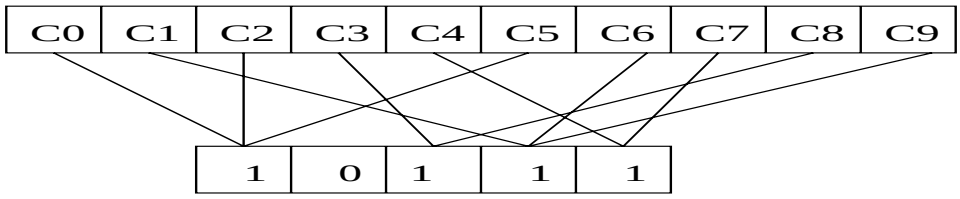

Figure 2.4: String and its signature

k-signature of S_1 has ones. In practice, it is often convenient to choose m such that the signature fits in a single machine word, in which case the signature test can be in one or two machine instructions.

The essential property of these methods is that a complex data object is represented by a simpler data object which contains less information but which retains some of the properties of the original.

2.2.1 Choice of parameters

Two parameters, m and k, are available to be chosen. As mentioned above, it is usually convenient to choose m such that it is a multiple of the number of bits in a machine word. Clearly, the larger m is the more accurate the results will be, in the sense that fewer strings will be incorrectly identified as substrings by the signature test.

The parameter k can be chosen as follows. Clearly if k is 1, no information is included in the signature about the order, so we will ordinarily expect k to be two or more. If there are n possible symbols in the strings , it does not make sense to choose k so small that n^k is less than m since there are only n^k distinct substrings of length k. On the other hand, if k is chosen too large, the number of bits in the signature can become too small, and in fact will be zero if k is larger than the length of the string. Note that maximum information content corresponds to having about half the bits in the signature zero.

2.3 Bloom Filter

A Bloom filter [5] is a method for representing a set $A = \{a_1, a_2, \ldots a_n\}$ of n elements (also called keys) to support membership queries.

The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $\{h_1, h_2, \ldots h_n\}$, each with range $\{1 \ldots m\}$. For each element

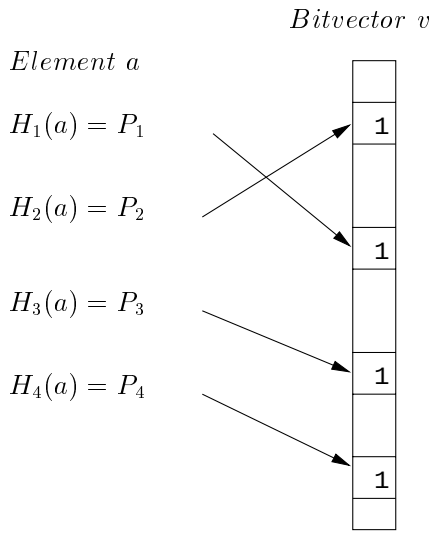

Figure 2.5: Bloom Filter with 4 hash functions

$a \in A$, the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ in v are set to 1. (A particular bit might be set to 1 multiple times.) Given a query for b we check the bits at positions $h_1(b), h_2(b), \ldots, h_k(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a *false positive*, or, for historical reasons, a *false drop*. The parameters k and m should be chosen such that the probability of a false positive (and hence a false hit) is acceptable.

As an example consider two strings $ABCD$ and XYZ and consider a Bloom filter of size 8 bits.[1]Suppose that $h_1(ABCD) = 3$, $h_2(ABCD) = 2$, $h_3(ABCD) = 1$, $h_4(ABCD) = 5$ and $h_1(ZYY) = 7$, $h_2(XYZ) = 3$, $h_3(XYZ) = 6$, $h_4(XYZ) = 0$. The Bloom filter for this set is 11101111. Here bit 4 is not set by either string and bit 3 is set by both strings.

[1]Bloom filters are much longer than 8 bits. 8 bits is shown just for illustative purpose.

Chapter 3

Implementation

In this chapter, we will discuss the implementation details of the signature based summary representation and Bloom filter based summary representation.

For the purpose of this discussion, a service description is specified as a service name followed by its attributes. The services and attributes are delimited by *'/'*. An example is *service=printer/bldg=egrc/room=409/type=laser*. The choice of *'/'* as delimiter was driven by the fact that we can use the huge *URL* name space for testing, treating the *domain name* as *service name* and each *path* in the *URL* as an *attribute*. So, if we have 10 *URL*s from the same domain with each *URL* having say 5 sub-directories, it immediately gives us a service name with 50 attributes.

Appendix A contains source code for all the implementations done in this chapter.

3.1 Overview of signature based scheme

Given a service description S as specified above, the signature based summary representation algorithm proceeds as follows.

1. extract the service name, s.
2. compute $H(s)$, where $H()$ is a hash function. This gives the index into hash table.
3. compute 128 bit signature of the entire service description.
4. insert the 128 bit signature into the hash table. Chaining is used in case of collisions.

In short, we use a two stage hashing procedure. In stage one, a hashing scheme is used for indexing keys to a hash table slot. In stage two, another hashing scheme is used to form a signature of the string. The signature is then inserted into the hash table slot calculated in stage one.

When we deal with strings as inputs to a hash function, special care is needed to avoid the problems caused by the following two common scenarios.

- If the input keys have an arithmetic progression as in *PART1, PART2, PART3* etc. they may hash to same slot or nearby slots giving rise to the problem of clustering
- If the input keys are permutations of the same string, they may hash to the same slot.

A similar problem can occur in the signature creation part also. The method we use for calculating signatures operates on the input string 2-characters at a time as described in Section 2.2. The implication is that strings differing in only few characters will give rise to similar signatures. Unless care is taken to hash similar strings to different slots, these strings may appear together in the hash table thus increasing the probability of false drop. For example strings PART1, PART2 will produce very similar looking signatures using our signature creation method. In order to spread them apart in the hash table, we employ Fibonacci hashing. Thus, the probability of two strings with similar signatures falling into the same hash slot is reduced, reducing the probability of false drop.

3.2 Hashing scheme for service names

When the input keys are strings, we need a mechanism to convert strings to an integer to be used as input for Fibonacci hashing. We had explained before that Fibonacci hashing is very effective in spreading out similar inputs. But, in order to form an integer from a string, if we take an approach of adding up the *ascii* values of characters , it leads to a new problem. For example keys $ABCD$ and $BCDA$ will map to the same hash slot. *Fibonacci* hashing will not help in avoiding collision in the case of permutations of the same string because the input to the hash function is the same value. In order to avoid this problem, we propose the following approach.

Consider two strings $ABCD$ and $BCDA$. If we can take the position of the character in the string into account while forming the sum, then these two strings will produce different sums. That is, instead of summing up $A[i]$, we sum up $A[i] * i$. So strings

AB and BA will give rise to $A.1 + B.2$ and $B.1 + A.2$, which are two different integers. But *multiplication* is a slow operation compared to other operations like *add*, *shift* etc. This leads us to another technique. Before making the sum, we will *rotate* the bits of each character by a number that is dependent upon the character's position in the string. The algorithm is as follows

```
//algorithm to create unique sums from permuted strings
#define ROL(x,n) ( (x << n)|(x >> 32 − n))          //rotate left 'x' by 'n' bits
sum =0;
for(i=0; i<strlen(s); i++) {
    bits_to_rotate = i & 0xf ;              //we need max 31 bit rotation.
    sum += ROL(a[i], bits_to_rotate);
}
```

Figure 3.1: Rotate Hash Function

This scheme combined with the Fibonacci hashing provides an excellent hashing function that reduces the problems of clustering (similar keys mapping to nearby slots), and permuted strings causing collisions. The final algorithms is as follows

```
//hash function
#define GOLDENRATIO 2654435769        //multiplier for 32 bits
#define ROL(x,n) ((x << n)|(x >> 32 − n))       //rotate left 'x' by 'n' bits
COMPUTE_HASH(char *s, len stringlen)
{
    sum =0;
    for(i=0; i<stringlen; i++) {
        bits_to_rotate = i & 0xf ;       //we need max 31 bit rotation.
        sum += ROL(a[i], bits_to_rotate);
    }

    hashvalue = sum * GOLDENRATIO;
    hashvalue = hashvalue >> 16;          //upper 16 bits for hash table of size 65536
}
```

Figure 3.2: Rotate + Fibonacci Hash Function

Note1: Rotation will not have any effect on the same character appearing in 1st and 32nd position. But in reality, we expect that the service names will be less than 32 characters.(service name is the service description minus attributes, for example printer is a service name). In that case 31 bit rotation will create unique sums for permuted strings

with high probability. This assumption is not a requirement, though.

Note2: As an optimization, the most frequently accessed signature may be moved to the top of the chaining list in case that it is at the end of the list. This ensures that, though the collision chain is long, the most frequently accessed keys will get faster responses.

3.2.1 Alternate hashing scheme for service names

During tests, it was found that the above scheme produces long hash chains when the number of services exceeds 64K. We found that CRC hashing scheme works well under these situations. CRC works for smaller number of services as well, but we wanted to evaluate the performance our hash function also. In Chapter 5, we will present a comparison of test results using Fibonacci hashing and CRC hashing.

Figure 3.4 is a simplified version of the *pathalias* [32] hashing function thanks to Steve Belovin and Peter Honeyman. First, a table containing the remainder for each possible data string of 32 bit length is computed. This fast table calculation works only if POLY (refer Figure 3.4) is a prime polynomial in the field of integers modulo 2. It must also be the case that that the coefficients of orders 31 down to 25 are zero. [44] gives one such 31 bit generator polynomial which is shown in Figure 3.3. The value *0x48000000* is obtained by reversing the order and dropping the last bit.

$x^{31} + x^3 + x^0$
01001000000000000000000000000001 — drop the last bit
0x48000000

Figure 3.3: 31bit prime polynomial

Description. *CRC_INIT()* pre-computes a table to be used while forming CRC of the message. *HASH_CRC()* takes as input the message to be hashed and produces 32 bit hash value in array *hash[]*. 16 bit hash, 17bit hash etc. can be extracted from the above value. In our experiments, we found that the lower order bits give a more evenly distributed hash value.

```
#define POLY 0x48000000L          // 31-bit polynomial (avoids sign problems)
static long CrcTable[128];
/*
CRC_INIT - initialize tables for hash function
/
static void CRC_INIT()
{
    for (i = 0; i < 128; ++i) {
        sum = 0L;
        for (j = 7 - 1; j >= 0; –j)
            if (i & (1 << j))
                sum ^ = POLY >> j;
            CrcTable[i] = sum;
    }
}
HASH_CRC(char *name, int size)
{
    while (size–) {
        sum = (sum >> 7)^CrcTable[(sum^(*name++))&0x7f];
    }
    ltouchar(sum, &hash[0]); //mask off 8th bit and copy
}
```

Figure 3.4: CRC hashing

3.3 Sub-string test by hashing

For this discussion, assume that the length of the contiguous symbols input to the hashing function, i.e. k, is 2 and m, the signature length, is 128 bits. Even though 128 bits are allocated for the signature, only 121 bits are used as explained below.

To utilize all bits of the signature, the symbols of English language were placed into classes based upon their frequency of occurrence (as given in Knuth [24],Tharp [40]) so that the likelihood of occurrence is approximately the same for each group. This is because all symbols do not occur with the same frequency in English. The total frequency of occurrence for each class is approximately the same in Table 3.1. We made a count of characters in the 4,00,000 URLs we collected from the Internet and observed that characters W, ., and / have a higher occurrence than as given in Knuth because of the way URLs are formed. Other characters were more or less following a similar distribution. We assigned W a separate class to reduce the effects of its dominance. W's dominance comes from the *www* prefix

many URLs have. Also in our implementation, '/' was treated as the *delimiter* separating attributes in a service description and was not considered in forming the signature value. This is done because we presume that when service names are actually deployed, they may not look like URLs and so the occurrence of '/' character may be very less. Upper and lower cases for the same letter belong to the same class.

For a symbol pair $(y1, y2)$, the hashing function is

$$H(y1, y2) = \text{number-of-classes} \times T(y1) + T(y2)$$

The *number-of-classes* for $k = 2$ is chosen to be the largest integer n such that $n^2 < m$, where m is the signature length.

Therefore, the number-of-classes for 128 bit signature is 11. The corresponding hash function is

$$H(y1, y2) = 11 \times T(y1) + T(y2)$$

Class	Characters
0	W
1	E 1 ! + *
2	T 2 ? < @
3	A G 3 . > /
4	O Y 4 & (
5	I F Q 5 ; =)
6	N M J 6 : - {
7	S U K 7 _ # }
8	R C V X 8 ' ∧ ∼ [
9	H B Z 9 %] .
10	D L P 0 " $ 1

Table 3.1: Classification of characters

Table 3.1 shows the distribution of characters into 11 groups. An array (T) of 128 is constructed in which the index is the integer corresponding to the bit pattern for that character and the entry is the class number to which that symbol belongs.

This hash function maps every symbol pair into an integer between 0 and 120. Due to the construction of array T, all eleven classes have an equal likelihood of being chosen. The hashing function therefore produces numbers in the range 0 to 120 with equal probability under the assumption that the occurrence of a symbol pair is based upon the

distribution of characters given in Table 3.1 . Also, all 121 bits are used. The signature is allocated 128 bits, so 7 bits remain unused.

```
short TABLE_HASH(char x,char y)
{
    return(11 × table[x]+table[y]);
}
```

Figure 3.5: hash function for signature formation

The following points have to be noted about the above mentioned table.

1. Characters from the same row appearing in different strings can lead to false drops. For example, *ET* and *12* both map to the same hash value. But a false drop occurs only if all the character pairs collide.

2. The only way collisions occur is by having characters from the same row appearing in different strings. For example, if *H* appears in a string, a collision will not occur if the characters *H,B,Z,9,%* and *]* are not there in the second string.

3. Only characters appearing in the same order lead to collisions. For example *AB* and *BA* will not cause a collision.

Note: As mentioned before, only 121 out of 128 bits of signature are used in our implementation. The remaining 7 bits can possibly be used for further reduction of false hits. For example, there might be a hash function, that can utilize the fact that we use Fibonacci hashing and a table based hashing (for signature formation) and exploit some other properties of the input to provide a 7 bit hash value of some sort. We haven't worked out any details yet.

3.3.1 Signature creation

A signature is computed for each service description by hashing the adjacent 2 characters using the above mentioned hash function. For example if the service description is 50 bytes long, 49 hash values are calculated and corresponding bit positions are set in the signature. This may not set 49 different bits because some of them may collide.

Description. The algorithm starts by initializing the 128 bit signature to zero. For each character pair in the input string, a hash value is calculated using *TABLE_HASH()*

described in Figure 3.5. Care is taken to skip delimiter character '/' while forming the signature. Also, character pairs spanning a *service name* and *attribute* or one *attribute* and another *attribute* are not hashed. As explained in Section 2.2, this is done so that attributes can be specified in any order in the service request.

To set the bit corresponding to the computed hash value, the following technique is used. The 128 bit signature is actually four 32 bit words. Since the hash code is a value between 0 and 120, *hashcode/32* tells us which one of these four words will have a bit set. The particular bit to be set within this word is given by the value, *hashcode - 32 * (hashcode/32)*.

```
#define DELIMITER /
#define ONE 1
CREATESIGNATURE(char *str,int string_len,unsigned int *signature)
{
    signature[0]=signature[1]=signature[2]=signature[3]=0;
    while( i < string_len-1) {
        if(str[i+1] != DELIMITER) //do not process delimiter
            hashcode = tablehash(str[i],str[i+1]);
        else {
            i+=2; //jump over the DELIMITER
            continue;
        }
//set the appropriate bit in the appropriate word (there are four 32 bit words
in the signature).
        byte = hashcode/32;
        shift = hashcode - (byte*32);
         signature[byte] | = (ONE << shift);
        i++;
    }
}
```

Figure 3.6: Algorithm to compute signature

3.3.2 Signature search

In order to search for a service request, the first step is to extract the *service name* and compute its *hash* value using the *Fibonacci hash + Rotate* algorithm. If that slot is not present in the hash table, the algorithm terminates. Otherwise, the signature of the service description is calculated and the *chaining* list at this hash table slot is examined for

a match until a match is found or the chain is exhausted.

Description. The algorithm starts by calling *COMPUTE_HASH()* to compute the hash table slot. After checking that the hash table slot is not empty, it computes the signature for the service description string by calling *CREATE_SIGNATURE()*. Then it compares the computed signature against the signatures present in the hash table chain by checking each of the 4 words. To do this, it performs an AND of its signature and that of the hash table signature. If the result is the same as its own signature, we know that it did not have a 1 in any position where the hash table signature had a zero. This is the signature test we intended to perform.

```
SEARCHPATTERN(char *pattern,int len)
{
    Extract only service name from pattern;
    hashcode = COMPUTE_HASH(service,strlen(service));
    if (hashtable[hashcode] == NULL) { //if slot is empty
        printf("Service not found");
        return;
    }
    else {
        createsignature(pattern,len,pattern_signature);
        //Scan the chaining list for a match
        while (LIST != NULL) {
          if ( ((pattern_signature[0] & LIST->signature[0]) == pattern_signature[0]) &&
               ((pattern_signature[1] & LIST->signature[1]) == pattern_signature[1]) &&
               ((pattern_signature[2] & LIST->signature[2]) == pattern_signature[2]) &&
                ((pattern_signature[3] & LIST->signature[3]) == pattern_signature[3]) ) {
                   printf("Service found
                   return ;
          }

          else
                   LIST = LIST->next; //increment the list
        }
    }
}
```

Figure 3.7: Algorithm for signature search

3.4 Bloom Filter

We also implemented a Bloom filter based search algorithm to compare its performance with our signature based algorithm. MD5 is used as the hashing function in Bloom filter implementation.

The Bloom filter size is m bits. The number of hash functions k is chosen to be 4. This is the number of bits set by each record. Instead of using 4 hash functions, the following technique is used.

Once MD5 produces the 128 bit *digest*, we treat each of the 32 bit words A,B,C,D (see Figure 3.8) as a separate 32 bit unsigned integers and take *modulo* m for each of the 32 bit integers. This will give rise to 4 values in the range $[0..m-1]$. We set bits at these 4 positions in the Bloom filter to 1.

A	B	C	D

Figure 3.8: MD5 Digest

3.4.1 Filter creation

The algorithm proceeds as follows

- Compute the 128 bit digest of the input string using MD5
- Divide the 128 bit digest into four separate 32 bit words.
- Compute *mod* m for each of these 4 words where m is the filter size.
- Set those bit positions to 1.

MD5 hash function

The MD5 hash function implementation details can be found in [35]. Here is a brief description of the function calls implemented in [35].

MDString() is simply a driver function to invoke the actual hashing functions. *MDInit()* loads the initial seed values for hashing algorithm into the MD buffer and also

does some other initial housekeeping operations like setting counters etc. *MDUpdate()* is the crux of the algorithm and processes the message in 512 bit blocks to produce the MD5 signature. *MDFinal()* copies this signature to the final *digest* output by the algorithm. (This operation is not a simple copy(). It performs some encoding, decoding etc. Please refer [35] for details)

```
/* Produces a MD5 digest of the string
/
static void MDString (char *string,unsigned char *digest)
{
MD_CTX context;
unsigned int len = strlen (string);

MDInit (&context);
MDUpdate (&context, string, len);
MDFinal (digest, &context);
}
```

Figure 3.9: MD5 function calls

Update

The algorithm (Figure 3.10) takes the MD5 digest, which is an array of 128 characters, as input. It treats this input as four 32 bit words. After computing the 32 bit integer value for each word, it takes *modulo FILTERSIZE* to produce an integer in the range $[0..FILTERSIZE]$ and sets that bit to 1. This procedure is repeated for each of the remaining 3 words.

3.4.2 Search

The search algorithm performs exactly similar operations on the service request as that performed during Bloom filter creation. It computes the MD5 digest of the service request and computes the 4 bit positions this digest will give rise to. It then checks if Bloom filer has 1's in those 4 positions. If any of the tests comes out negative, the service is declared to be not available. Only if all four tests are successful is the service declared as found.

Description: The algorithm starts by calling *MDString*(Figure 3.11) to compute

```
#define ONE 1
SET_BITS(unsigned char *digest)
{
    value = ( digest[0] | (digest[1] << 8) | (digest[2] << 16) | (digest[3] << 24));
     hashvalue = value%FILTERSIZE;
     byte = hashvalue/32;
     shift = hashvalue - byte*32;
     bitarray[byte] | = (ONE<<shift);

     value = ( digest[4] | (digest[5] << 8) |(digest[6] << 16) | (digest[7] << 24));
     hashvalue = value%FILTERSIZE;
     byte = hashvalue/32;
     shift = hashvalue - byte*32;
     bitarray[byte] | = (ONE<<shift);

     value = ( digest[8] | (digest[9] << 8) |(digest[10] << 16) |(digest[11] << 24));
     hashvalue = value%FILTERSIZE;
     byte = hashvalue/32;
     shift = hashvalue - byte*32;
     bitarray[byte] | = (ONE<<shift);

     value = ( digest[12] |(digest[13] << 8) |(digest[14] << 16) |(digest[15] << 24));
     hashvalue = value%TABLESIZE;
     byte = hashvalue/32;
     shift = hashvalue - byte*32;
     bitarray[byte] | = (ONE<<shift);
}
```

Figure 3.10: Bloom filter update algorithm

the 128 bit digest of the service description. It then calls *TEST_BITS()* with this 128 bit digest as parameter. *TEST_BITS()* operations are similar to those in *SET_BITS()* and so are self-explanatory.

```
#define ONE 1
TEST_BITS(unsigned char *digest)
{
    value = ( digest[0] |(digest[1] << 8) |(digest[2] << 16) |(digest[3] << 24));
    hashvalue = value%FILTERSIZE;
    byte = hashvalue/32;
    shift = hashvalue - byte*32;
    bitarray[byte] | = (ONE<<shift);
    if( (bitarray[byte] & (ONE<<shift)) == 0) {
        printf("no match");
        return;
    }
    value = ( digest[4] |(digest[5] << 8) |(digest[6] << 16) |(digest[7] << 24));
    hashvalue = value%FILTERSIZE;
    byte = hashvalue/32;
    shift = hashvalue - byte*32;
    bitarray[byte] | = (ONE<<shift);
    if( (bitarray[byte] & (ONE<<shift)) == 0) {
        printf("no match");
        return;
    }
    value = ( digest[8] |(digest[9] << 8) |(digest[10] << 16) $—(digest[11] << 24));
    hashvalue = value%FILTERSIZE;
    byte = hashvalue/32;
    shift = hashvalue - byte*32;
    bitarray[byte] | = (ONE<<shift);
    if( (bitarray[byte] & (ONE<<shift)) == 0) {
        printf("no match");
        return;
    }
    value = ( digest[12] |(digest[13] << 8) |(digest[14] << 16) |(digest[15] << 24));
    hashvalue = value%TABLESIZE;
    byte = hashvalue/32;
    shift = hashvalue - byte*32;
    bitarray[byte] | = (ONE<<shift);
    if( (bitarray[byte] & (ONE<<shift)) == 0) {
        printf("no match");
        return;
    }
   printf("match found");
}
```

Figure 3.11: Bloom filter search algorithm

3.5 Update handling

It is conceivable that services provided by a domain may undergo changes. New services may get added, existing services may get dropped and existing services may add, update or drop some attributes.

Bloom filter implementation in Summary Cache [13] makes use of a 4-bit count field per bit to take care of updates. This field stores the number of different records that have set this bit to 1. When updates occur, this field is adjusted. If the counter is zero, then the bit is reset to 0. This approach will involve additional memory and processing overhead.

Secure Service Discovery [8] requires that the source send an updated Bloom filter whenever change occurs.

In our scheme, we resort to re-transmitting the signature when changes happen. This is because, in the hash table we do not keep a mapping of service names to signatures. All we have is a linked list of signatures all of which map to that particular hash slot. When a change occurs, the sender will send the signatures for all records that map to the same hash slot as that of the service being changed. On the average, it will have to re-transmit α (load factor) signatures.

Chapter 4

Performance Analysis

In this chapter, we will derive analytical expressions for the false drop probabilities of SRDP and the Bloom filter based scheme. Later, we will compare the memory requirements and running time for both these schemes.

4.1 False drop probability for SRDP

Let the signature length be m bits. From Chapter 2, section 2.2 we have seen that a string of length $l+1$ can set a maximum of l bits in the signature.

To calculate the false drop probability, we will proceed as follows. Suppose we received a service request for a string of length $l+1$. Let this string have k 1's in its signature where $1 \leq k \leq l$.

We want to calculate the probability that there is a signature in our database that has k 1's in exactly the same positions as the signature of the request string. The remaining $m-k$ positions could be either 0 or 1 because so long as there is a match in the same k bit positions, we will declare it as a match.

If we denote the signature for request string as r and that of the data base string as d, we want to calculate the conditional probablity of d having the same k 1's as r. i.e. $P(d\ has\ same\ k\ 1's \mid r\ has\ k\ 1's)$.

There are a total of 2^m different signatures in the database. There are 2^{m-k} signatures with a particular set of k bits set to 1.

Therefore, $P(d\ has\ same\ k\ 1's \mid r\ has\ k\ 1's) = \frac{2^{m-k}}{2^m} = 2^{-k}$.

Since we have a hashing stage before the signature formation stage, the false drop

probability $= \alpha 2^{-k}$ where α is the load factor of the hash table.

4.1.1 Calculation of expected false drop probability

The probability derived above gives an expression in terms of an arbitrary number of bits k set by a string. In order to get an idea about the numerical values expected from our algorithm, we will derive an expression in terms of the lengths of strings.

Probability that a bit is set to 1 $= \frac{1}{m}$

Probability that bit not set ,ie it is still 0 $= 1 - \frac{1}{m}$

Let's focus on one particular bit (say the ith bit).

The probability that this particular bit is not set(is still 0) after l insertions

$$= (1 - \frac{1}{m})^{l} \tag{4.1}$$

Therefore, the expected number of zeros in the signature after l insertions

$$= m(1 - \frac{1}{m})^{l} \tag{4.2}$$

Now consider that the database has a signature for a string of length l_1. We receive a request for a string of length l_2. A false drop depends on the chances of the bits set to 1 by l_2 being set to 1 by l_1 also.

From (4.2), the expected number of bits still 0 after l_2 insertions

$$= m(1 - \frac{1}{m})^{l_2} \tag{4.3}$$

Therefore, the expected number of bits set to 1 after l_2 insertions

$$= m - m(1 - \frac{1}{m})^{l_2} \tag{4.4}$$

From (4.1), the probability that one particular bit is set to 1 after l_1 insertions

$$= 1 - (1 - \frac{1}{m})^{l_1} \tag{4.5}$$

Therefore, from (4.4) and (4.5) , the probability of a false drop (false drop occurs if all bits set to 1 by l_2 are also set to 1 b y l_1)

$$= \left(1 - (1 - \frac{1}{m})^{l_1}\right)^{(m - m(1-\frac{1}{m})^{l_2})} \tag{4.6}$$

Table 4.1 shows the probability that a string of length $l_2 + k - 1$ will be identified as a sub-string of a string of length of length $l_1 + k - 1$.

l_1=5	1.44e-07	3.94e-14	1.81e-26	2.85e-48	2.19e-82	2.0e-124	5.0e-137
10	3.94e-06	2.57e-11	4.62e-21	2.75e-38	3.09e-65	1.89e-98	2.1e-108
20	9.75e-05	1.38e-08	8.11e-16	1.33e-28	1.33e-48	3.01e-73	1.21e-80
40	1.99e-03	5.12e-06	6.92e-11	1.71e-19	5.89e-33	1.50e-49	1.59e-54
80	2.85e-02	9.40e-04	1.53e-06	1.83e-11	3.63e-19	1.16e-28	1.66e-31
160	2.19e-01	5.14e-02	3.34e-03	2.68e-05	1.40e-08	1.27e-12	7.84e-14
200	3.54e-01	1.30e-01	2.01e-02	7.39e-04	4.18e-06	7.12e-09	1.05e-09
	l_2=5	10	20	40	80	160	200

Table 4.1: Expected false drop under various l_1 and l_2 combinations

4.2 False drop probability for Bloom filter based scheme

For a Bloom filter of size m bits storing n records with each record setting k bits(load factor), the probability of false drop is given by $P(n, m, k) = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k$

Proof: Probability that a bit is set to 1 $= \frac{1}{m}$

Probability that the bit is 0 $= 1 - \frac{1}{m}$

After inserting n keys into the table, the probability that a particular bit is still 0

$$= (1 - \frac{1}{m})^{kn} \tag{4.7}$$

Hence the probability of a false drop

$$= \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{\frac{kn}{m}})^k \tag{4.8}$$

This function is minimized for $k = ln2 * \frac{m}{n}$. k is chosen to be the nearest integer. It is plotted in column 2 of tables 4.2 thru 4.4. The salient feature of Bloom filters is that there is a clear tradeoff between m and the probability of a false positive.

Tables 4.2 though 4.4 lists the false positive ratios for common combinations of $\frac{m}{n}$ and k.

The graph in Figure 4.1 shows the probability of a false positive as a function of the number of bits allocated for each entry, that is, the ratio m/n. The curve above is for the case of 4 hash functions. The curve below is for the optimum number of hash functions. The scale is logarithmic so the straight line observed corresponds to an exponential decrease. It is clear that Bloom filters require very little storage per key at the slight risk of some false positives. For instance for a bit array 10 times larger than the number of entries, the probability of a false positive is 1.2% for 4 hash functions, and 0.9% for the optimum case

m/n	k	k=1	k=2	k=3	k=4	k=5	k=6	k=7
2	1.39	0.393	0.400					
3	2.08	0.283	0.237	0.253				
4	2.77	0.221	0.155	0.147	0.160			
5	3.46	0.181	0.109	0.092	0.092	0.101		
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638	
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364	
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702
17	11.8	0.0571	0.0123	0.00423	0.00193	0.00107	0.000692	0.000499
18	12.5	0.054	0.0111	0.00362	0.00158	0.000839	0.000519	0.00036
19	13.2	0.0513	0.00998	0.00312	0.0013	0.000663	0.000394	0.000264
20	13.9	0.0488	0.00906	0.0027	0.00108	0.00053	0.000303	0.000196
21	14.6	0.0465	0.00825	0.00236	0.000905	0.000427	0.000236	0.000147
22	15.2	0.0444	0.00755	0.00207	0.000764	0.000347	0.000185	0.000112
23	15.9	0.0425	0.00694	0.00183	0.000649	0.000285	0.000147	8.56e-05
24	16.6	0.0408	0.00639	0.00162	0.000555	0.000235	0.000117	6.63e-05
25	17.3	0.0392	0.00591	0.00145	0.000478	0.000196	9.44e-05	5.18e-05
26	18	0.0377	0.00548	0.00129	0.000413	0.000164	7.66e-05	4.08e-05
27	18.7	0.0364	0.0051	0.00116	0.000359	0.000138	6.26e-05	3.24e-05
28	19.4	0.0351	0.00475	0.00105	0.000314	0.000117	5.15e-05	2.59e-05
29	20.1	0.0339	0.00444	0.000949	0.000276	9.96e-05	4.26e-05	2.09e-05
30	20.8	0.0328	0.00416	0.000862	0.000243	8.53e-05	3.55e-05	1.69e-05
31	21.5	0.0317	0.0039	0.000785	0.000215	7.33e-05	2.97e-05	1.38e-05
32	22.2	0.0308	0.00367	0.000717	0.000191	6.33e-05	2.5e-05	1.13e-05

Table 4.2: False drop rate under various $\frac{m}{n}$ and k combinations

of 5 hash functions. The probability of false positives can be easily decreased by allocating more memory.

m/n	k	k=8	k=9	k=10	k=11	k=12	k=13	k=14
9	6.24	0.0145						
10	6.93	0.00846						
11	7.62	0.00509	0.00531					
12	8.32	0.00314	0.00317	0.00334				
13	9.01	0.00199	0.00194	0.00198	0.0021			
14	9.7	0.00129	0.00121	0.0012	0.00124			
15	10.4	0.000852	0.000775	0.000744	0.000747	7.78e-04		
16	11.1	0.000574	0.000505	0.00047	0.000459	4.66e-04	4.88e-04	
17	11.8	0.000394	0.000335	0.000302	0.000287	2.84e-04	2.91e-04	
18	12.5	0.000275	0.000226	0.000198	0.000183	1.76e-04	1.76e-04	1.82e-04
19	13.2	0.000194	0.000155	0.000132	0.000118	1.11e-04	1.09e-04	0.00011
20	13.9	0.00014	0.000108	8.89e-05	7.77e-05	7.12e-05	6.79e-05	6.71e-05
21	14.6	0.000101	7.59e-05	6.09e-05	5.18e-05	4.63e-05	4.31e-05	4.17e-05
22	15.2	7.46e-05	5.42e-05	4.23e-05	3.5e-05	3.05e-05	2.78e-05	2.63e-05
23	15.9	5.55e-05	3.92e-05	2.97e-05	2.4e-05	2.04e-05	1.81e-05	1.68e-05
24	16.6	4.17e-05	2.86e-05	2.11e-05	1.66e-05	1.38e-05	1.2e-05	1.08e-05
25	17.3	3.16e-05	2.11e-05	1.52e-05	1.16e-05	9.42e-06	8.01e-06	7.1e-06
26	18	2.42e-05	1.57e-05	1.1e-05	8.23e-06	6.52e-06	5.42e-06	4.7e-06
27	18.7	1.87e-05	1.18e-05	8.07e-06	5.89e-06	4.56e-06	3.7e-06	3.15e-06
28	19.4	1.46e-05	8.96e-06	5.97e-06	4.25e-06	3.22e-06	2.56e-06	2.13e-06
29	20.1	1.14e-05	6.85e-06	4.45e-06	3.1e-06	2.29e-06	1.79e-06	1.46e-06
30	20.8	9.01e-06	5.28e-06	3.35e-06	2.28e-06	1.65e-06	1.26e-06	1.01e-06
31	21.5	7.16e-06	4.1e-06	2.54e-06	1.69e-06	1.2e-06	8.93e-07	7e-07
32	22.2	5.73e-06	3.2e-06	1.94e-06	1.26e-06	8.74e-07	6.4e-07	4.92e-07

Table 4.3: False drop rate under various $\frac{m}{n}$ and k combinations

M/n	k	k=15	k=16	k=17	k=18	k=19	k=20	k=21
19	13.2	.000114						
20	13.9	6.84e-05						
21	14.6	4.16e-05	4.27e-05					
22	15.2	2.57e-05	2.59e-05	2.67e-05				
23	15.9	1.61e-05	1.59e-05	1.61e-05				
24	16.6	1.02e-05	9.87e-06	9.84e-06	1e-05			
25	17.3	6.54e-06	6.22e-06	6.08e-06	6.11e-06	6.27e-06		
26	18	4.24e-06	3.96e-06	3.81e-06	3.76e-06	3.8e-06	3.92e-06	
27	18.7	2.79e-06	2.55e-06	2.41e-06	2.34e-06	2.33e-06	2.37e-06	
28	19.4	1.85e-06	1.66e-06	1.54e-06	1.47e-06	1.44e-06	1.44e-06	1.48e-06
29	20.1	1.24e-06	1.09e-06	9.96e-07	9.35e-07	9.01e-07	8.89e-07	8.96e-07
30	20.8	8.39e-06	7.26e-06	6.5e-07	6e-07	5.69e-07	5.54e-07	5.5e-07
31	21.5	5.73e-07	4.87e-07	4.29e-07	3.89e-07	3.63e-07	3.48e-07	3.41e-07
32	22.2	3.95e-07	3.3e-07	2.85e-07	2.55e-07	2.34e-07	2.21e-07	2.13e-07

Table 4.4: False drop rate under various $\frac{m}{n}$ and k combinations

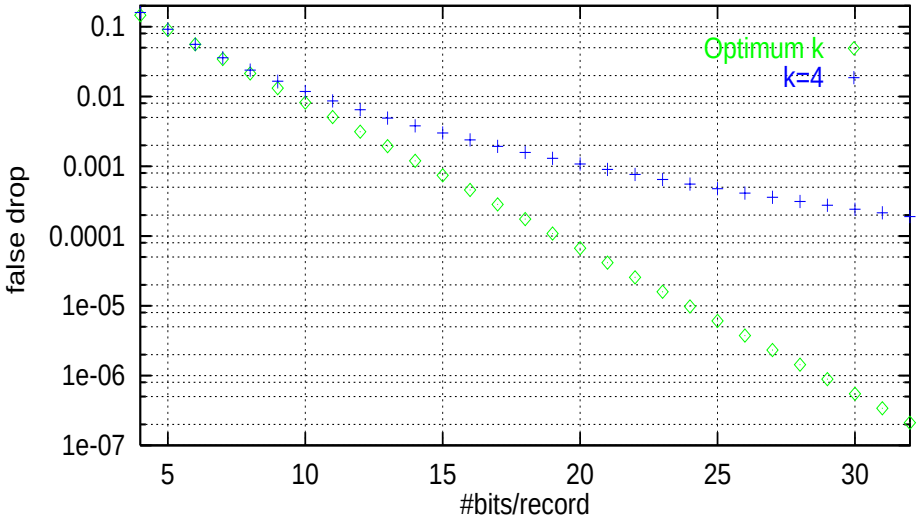

Figure 4.1: Bloom filter: probability of false drops(log scale)

4.3 Memory requirements

As mentioned before, the memory requirements for Bloom filer will increase as the number of attributes increases. For our examples, we will take a very conservative figure for the number of attributes, say 5. This gives rise to 32 records for the <service,attribute> pair in a Bloom filter. On the other hand, SRDP will treat the whole service description as a single record. It is possible to test SRDP for services having a large number of attributes, but the number of records to be created for Bloom filter becomes quite large.

SRDP requires 4 bytes per hash table slot (pointer to struct), 16 bytes for signature and another 4bytes for chaining. The hash table size is chosen to be the nearest power of 2 corresponding to the number of records.

For the Bloom filter method, we will consider a load factor of 32(The load factor is the number of bits allocated per record). This load factor will provide a false drop probability of .000191 for k =4.

Let N represent the number of services, with each service having 5 attributes.Let N' denote the hash table size. Total memory requirements are as follows:

For Bloom Filter: $N \times 32 \times 32$ bits, where one 32 stands for the load factor and the other stands for the number of possible combination of attributes.

For SRDP : $N' \times 4 + N \times 20$ bytes where N' represents the nearest power of 2 for this particular N. N' may be $> N$ or $< N$ as illustrated in the following table.

Table 4.5 summarizes memory requirements for some sample scenarios.

N	Bloom filter(in Mbytes)	SRDP(in Mbytes)	N'
10000	1.2	0.24	8192
30000	3.6	0.73	32768
50000	6.0	1.27	65536
70000	8.4	1.66	65536

Table 4.5: Theoritical estimate of memory requirements for 5 attributes

It can be seen that SRDP consumes 80% less memory than SDS.

4.4 Computational overhead

Now we will discuss the computational overhead involved in performing search for a service request.

SRDP using Fibonacci Hashing: This method involves computation of hash table index, computation of signature and checking for a signature match.

1. Computation of hash table index

   sum ← 0

   For each character c in the service request

   > Compute c & $0xf$ (to calculate shift quantity)
   > Rotate each character by the shift quantity
   > Add to sum

   End

   sum ← sum * GOLDEN RATIO

   Extract desired number of bits (depending upon hash table size)

2. Computation of 128 bit signature

   For each adjacent character pair

   > Perform table lookups twice
   > Perform one multiplication and one addition
   > Set a bit (involves SHIFT and AND)

   End

3. Compare the 128 bit signature with hash table signature. In case of mismatch, traverse the chaining list until match is found or list is exhausted.

SRDP using CRC Hashing: Steps consist of computation of hash table index, computation of signature and checking for a signature match.

1. Computation of CRC hash

   For each character c in the service request

   > Perform 2 XORs, 1 AND, 1 SHIFT and pointer incrementing, de-referencing and subscript operation.

   End

   Extract desired number of bits (depending upon hash table size)

2. Computation of 128 bit signature

   For each adjacent character pair

Perform table lookups twice
Perform one multiplication and one addition
Set a bit (involves SHIFT and AND)

End

3. Compare the 128 bit signature with hash table signature. In case of mismatch, traverse the chaining list until match is found or list is exhausted.

Bloom filter method: The steps involve the calculation of MD5 digest and checking the bit positions in Bloom filter.

1. Compute MD5 digest (128 bits) for the string.
   Each 512-byte block is digested in 4 phases. Each phase consists of 16 basic steps, for a total of 64 basic steps. Each step updates one word of a 4-word accumu lated digest, using the entire intermediate digest as well as block data and con stants. In general, each basic step depends on the output of the prior step. The basic structure of the steps is shown below (<<< denotes rotate).

```
A = B+((A+ F(B,C,D) + X[i++] + k1)<<<k2)

D = A+((D+ F(A,B,C) + X[i++] + k1)<<<k2)

C = D+((C+ F(D,A,B) + X[i++] + k1)<<<k2)

B = C+((B+ F(C,D,A) + X[i++] + k1)<<<k2)
```

   There are 16 steps based on each of 4 logical functions; 4 based on F are shown here. The constants k1 and k2 are not necessarily identical in basic steps, and are not relevant to this analysis. The logical functions are(ĉdenotes xor):

```
F(x, y, z) = (((x) & (y)) | ((~x) & (z)))

G(x, y, z) = (((x) & (z)) | ((y) & (~z)))

H(x, y, z) = ((x) ^ (y) ^ (z))

I(x, y, z) = ((y) ^ ((x) | (~z)))
```

2. Convert signature to four 32 bit words
3. Calculate Modulo Filtersize for each of them.
4. Check the bit positions corresponding to each of the 4 modulo values.

As seen above, the Bloom filter approach involves significantly more number of computations owing to the presence of MD5 hashing algorithm.

Chapter 5

Numerical Results

5.1 Test data preparation

Currently we do not have a large volume of service descriptors available for testing. We made use of the large database of URLs available on the Internet. For this purpose, we convert a URL into a service description as follows. Consider a URL that has the format *www.domain.com/path1/path2/path3/path4*. We treat the *www.domain.com* as the *service name* and each of the paths, *path1, path2, path3*, etc., as *attributes*.

As mentioned in previous sections, n attributes will give rise to 2^n attribute combinations in Bloom filter based representation. Because of this, we limited our testing to 5 attributes which give rise to 32 combinations of service descriptions. Note that some of the the service descriptions may have less than 5 attributes because of the nature of the data we collected.

The *URL* names were collected from search engines *Altavista* and *Webcrawler* using a *perl* script. Once the *URL*s were collected, they were sorted on *domain names*. The *URL*s from the same domain were then concatenated to form a single string. From this string, duplicate path names were eliminated and a new string with domain name and 5 paths was formed. This forms the *service description* string with 5 *attributes*. From this string, 32 new strings were formed using a permutation program. These 32 strings are used as the service description strings for Bloom filter based representation.

Example:

Let www.ncsu.edu/tracs/spring/index.html and www.ncsu.edu/students/tution/summer/index.html be the *URL*s pulled from the search engines.

Step1. Sort URLs

Step2. Concatenate URLs from same domain.

www.ncsu.edu/tracs/spring/index.html/students/tuition/summer/index.html

Step3. Eliminate duplicate paths. index.html appears twice. One of them is removed.

www.ncsu.edu/tracs/spring/index.html/students/tuition/summer/

Step4. Choose domain name and 5 paths. This is our service description string.

www.ncsu.edu/tracs/spring/index.html/students/tuition/

Step5. Create 32 combinations of the string made in Step4. These are the different service strings for Bloom filter.

www.ncsu.edu/

www.ncsu.edu/tracs/

www.ncsu.edu/tracs/spring/

.....

www.ncsu.edu/tracs/spring/index.html/students/tuition/

In all, we collected 400,000 URLs. These were from 139,507 different domains. These 139,507 service descriptions were used in all the tests performed.

5.1.1 False drop probability

For this experiment, we used 69,768 URLs as service requests originating from clients. The remaining 69,739 URLs were used to mimic the database of available services at a domain. The 69,768 URLs were further converted using a permutation program to produce various attribute combinations for requests. This results in 313,426 service requests. We made sure that none of these requests are present in the list of 69,739 available services. So, ideally the false drop should be zero. Table 5.1 summarizes the results. We conducted 6 experiments, each time creating a database of available services using a subset of the 69739 URLs considered as available services. Each time, 313426 service requests representing the total attribute combinations for 69768 different services were sent and the number of false drops noted. Columns I & II in Table 5.1 denote the number of available services and the number of different attribute combinations it gives rise to. (Column II is applicable only to Bloom filter where each attribute combination is treated as a separate record)

We note that SRDP using Fib Hashing + signature based scheme performs better than Bloom filter in all cases. SRDP using CRC + signature scheme performs better than

# client requests = 313426							
#services	#combinations	# of false drops & %					
		Rotate + Fib hashing		CRC Hashing		MD5 hashing	
6974	31298	10	.000032	17	.000054	59	.000188
8718	38942	18	.000057	24	.000076	56	.000179
13948	62352	20	.000064	31	.000099	69	.000220
17435	78371	23	.000073	42	.000134	51	.000163
34870	154812	25	.000079	45	.000144	55	.000176
69739	308769	49	.000156	86	.000274	57	.000182

Table 5.1: False drop values

Bloom filter except in one case. The Bloom filter false drop probability is as predicted in Table 4.2 (=.000191)

5.2 Running Time

The times shown in Table 5.2 are just the time taken to search the records. The experiment is exactly same as that conducted in Section 5.2. The time taken to build initial table, access files etc. is excluded. In order to achieve this, the entire URL file was read into memory prior to searching. The times were collected using *getrusage()* instead of *gettimeofday()* for better accuracy. Tests were run on a Pentium III 550 Mhz PC running Linux. (Note that the Linux stack size has to be set to 96M using *ulimit()* for this test to run because of the large number of records read into memory)

# client requests = 313426			
#services	Running time(μsec)		
	Rotate + Fib hashing	CRC Hashing	MD5 hashing
6974	1480000	1460000	3680000
8718	1620000	1680000	3670000
13948	1590000	1490000	3680000
17435	1630000	1590000	3680000
34870	1600000	1700000	3880000
69739	1590000	1550000	3710000

Table 5.2: Running time

From Table 5.2, the average search time for 1 request is roughly 5μsec. for SRDP and 11.5μsec. for SDS.

One of the major concerns of SRDP is the length of hash chain in case of collisions. To study how our scheme compares with Bloom filter when collisions occur, the following test was performed. We mapped a list of 90000 URLs to a hash table of size 65535. This was done to create long collision chains. Then we picked 1500 URLs that were appearing towards the end of hash chain and had collision chains of lengths 19 to 21. (Actually there are not 1500 URLs with hash chain lengths of 19 to 21 in the table. We picked 5 URLs that had collision chain lengths of 19 to 21 and repeated the search 300 times). Table 5.3 shows the results.

#client requests = 1500	Time(μsec)
Signature	5000
BloomFilter	15000

Table 5.3: Worst case running time

5.3 Effectiveness of hashing schemes

The following experiment tests the effectiveness of 3 different hashing schemes, Rotate+Fibonacci, CRC and MD5 hashing. Note that this test is only to check how effectively these hashing schemes distribute the keys. We do not compute signature or check for a string match.

The basic idea is to allocate a hash table of size very close to the number of keys. The size is chosen to be the nearest power of 2. Note that in some cases the hash table size is less than the number of keys. This will increase collisions. But recall that the hashing stage is followed by a signature matching stage in the string search algorithm. Because of this, it is acceptable to have some amount of collisions.

Note also that MD5 hashing produces 128 bit signatures, but we will not use all 128 bits in this experiment. From the 128 bits, only enough bits to address the hash table size in each experiment is extracted. The same is true for CRC hashing where we extract the required number of bits from the 32 bits produced by the CRC implementation described in Chapter 3.

For each experiment, we create an input file of service names and run the 3 hashing algorithms. Hash values are extracted and checked to see as to how many different service names produce the same hash value.

Tables 5.4 thru 5.6 give collision occurrences for each scheme. The different fields in the table are as follows:

Total # keys → the number of service names in the input file
#slots → number of hash table slots that had this many collisions
collisions → number of keys mapped to the same hash table slot - 1.[1]

As an example, in Table 5.4, a file with 7700 service names was hashed to a hash table of size 8192. 2984 slots had unique keys, 1375 slots had 2 keys mapped to it, 469 slots had 3 keys mapped to it etc. For illustration, we present only 3 tables here. The complete list of experiment results can be seen in Appendix B.

Total # keys = 7700 Hash table size = 8192					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
2984	0	3033	0	2991	0
1375	1	1375	1	1388	1
469	2	435	2	451	2
108	3	111	3	115	3
22	4	23	4	22	4
3	5	9	5	3	5
1	6	1	6		

Table 5.4: Collision occurrences for Fibonacci, CRC and MD5 hashing

Total # keys = 10000 Hash table size = 8192					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
2926	0	2869	0	2948	0
1791	1	1791	1	1868	1
756	2	801	2	794	2
297	3	266	3	259	3
69	4	75	4	55	4
22	5	19	5	14	5
5	6	6	6	1	6
		1	7	1	7

Table 5.5: Collision occurrences for Fibonacci, CRC and MD5 hashing

[1]Note the -1 here. This is to emphasize that when the number of records at a slot is 1, there is no collision

Total # keys = 94000 Hash table size = 65536					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
21443	0	22417	0	22266	0
13476	1	15699	1	15967	1
6524	2	7473	2	7762	2
2813	3	2753	3	2728	3
1167	4	875	4	823	4
539	5	273	5	228	5
242	6	91	6	34	6
153	7	22	7	8	7
96	8	5	8	1	8
66	9	3	9		
27	10	3	10		
20	11	1	11		
14	12				
6	13				
3	14				
1	15				
1	17				
1	18				
1	21				

Table 5.6: Collision occurrences for Fibonacci, CRC and MD5 hashing

It can be seen from Tables 5.4 thru 5.6 that, the Fibonacci scheme performs well for small set of keys. We observed that upto 64K number of keys, the scheme performs well. After that the collision occurrences increase. On the other hand, CRC scheme is equally good as MD5 hashing.

5.4 Effect of bigger hash table

In order to study the effect of a bigger hash table than the number of keys, we performed the experiment with 94000 input keys for hash table sizes of 2^{17}(=131072), 2^{18} and 2^{19}.

Total # keys = 94000 Hash table size = 2^{17}					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
40122	0	45479	0	45533	0
13884	1	16315	1	16550	1
4202	2	3950	2	4005	2
1488	3	806	3	727	3
573	4	157	4	114	4
273	5	23	5	11	5
153	6	10	6		
99	7	2	7		
71	8				
27	9				
20	10				
8	11				
6	12				
2	13				
2	14				
1	16				

Table 5.7: Collision occurrences for Fibonacci, CRC and MD5 hashing

It can be seen that the CRC and MD5 hashing schemes have significantly less collisions as we allocate more space for the hash table. As pointed out in earlier chapters, The *Rotate + Fibonacci* scheme does not perform well when the number of input keys exceeds 64K, even if we allocate more space for the hash table.

Total # keys = 94000 Hash table size = 2^{18}					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
54397	0	65047	0	65497	0
11031	1	11846	1	11918	1
2690	2	1546	2	1428	2
946	3	172	3	136	3
419	4	18	4	5	4
219	5	5	5	1	5
116	6	1	6		
86	7	1	7		
49	8				
18	9				
15	10				
7	11				
2	12				
3	13				
1	14				
1	16				

Table 5.8: Collision occurrences for Fibonacci, CRC and MD5 hashing

Total # keys = 94000 Hash table size = 2^{19}					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
62433	0	78180	0	78668	0
8655	1	7223	1	7045	1
2002	2	479	2	450	2
786	3	31	3	21	3
365	4	1	4	1	4
207	5			1	5
104	6				
83	7				
42	8				
15	9				
15	10				
4	11				
3	12				
2	13				
1	14				
1	16				

Table 5.9: Collision occurrences for Fibonacci, CRC and MD5 hashing

5.5 Effect of signature size

Now we will present the motivation for a 128 bit signature in our implementation. In order to calculate the effect of signature size in false drop probability, we conducted the experiment done in Section 5.2 using 64 bit signatures instead of 128 bit signatures. The 128 bit signature dramatically reduces the number of false drops.As you may recall from Chapter 2, a different signature size requires a different lookup table. Appendix C lists the lookup table for 64 bit signature.

# service requests searched = 313426 # services = 69768			
# services	#requests	# false drops	
		128 bit signature	64 bit signature
6974	31298	10	1828
8718	38942	18	2632
13948	62352	20	3027
17435	78371	23	3643
34870	154812	25	3728
69739	308769	49	5096

Table 5.10: Effect of signature

Chapter 6

Conclusion

As the Internet grows to include more services, it is increasingly necessary to be able to identify these services in a more flexible way than specifying their network location. To this end, we have proposed a novel scheme called SRDP for representing service descriptions in a compact form. We have designed, implemented and evaluated this summary representation scheme.

We have demonstrated that by using a combination of hash table and signature, service descriptions can be represented in a compact format and searched for availability efficiently without compromising reliability.

A novel hashing scheme involving Fibonacci hashing and rotation of bits is evaluated for its strength and compared against CRC hashing and MD5 hashing. We observed that this scheme can perform well for hash tables of size less than 64K.

Finally we compared SRDP with a Bloom filter based representation. We demonstrated that a less computationally intensive method involving CRC hashing (or Fibonacci hashing) and signature based representation of services can be used to achieve comparable or even better false drop probabilities. We have also shown that the signature based scheme consumes less memory and executes faster.

All these conclusions were drawn from tests performed on data generated from URLs. These assertions need to be verified by deployment in an actual service discovery environment.

6.1 Future directions

In our current research, we have focused on representing data available within a domain. We plan to enhance this scheme for wide area deployment. One obvious disadvantage of our scheme over Bloom filters is that we can not aggregate signatures from two different domains in a simple way. Bloom filter has the advantage that Bloom filters from two different domains can be ORed to form a new Bloom filter, though increasing the probability of false drop.

One of the schemes we are contemplating is an approach similar to adaptive web caching [28] where the information is shared only with a limited set of neighbors. In this approach, neighbors form *multicast* groups and share information only with members of the same groups. Every node will typically be a member of more than one multicast group.

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, and H. Balakrishnan. The design and implementation of an intentional naming system. in proc. acm sosp, 1999.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly & Associates, Oct 1992.

[4] Samrat Bhattacharjee, Mostafa Ammar, Ellen Zegura, Viren Shah, and Zongming Fei. Application-layer anycasting. Technical Report GIT-CC-96-25.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[6] Doug S. Comer. *Internetworking with TCP/IP*. Prentice-Hall, 1999.

[7] Corman, Leiserson, and Rivest. Introduction to algorithms. mcgraw-hill, 1992.

[8] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[9] K. Claffy D. Wessels. RFC 2286: Internet caching protocol v2, 1997.

[10] R. Daniel and M. Mealling. Rfc 2168: Resolution of uniform resource identifiers using the domain name system, June 1997.

[11] D.Sarwate. Computation of cyclic redundancy check via table lookup. *Communications of the ACM*, pages 1008–1013, August 1988.

[12] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. acm transactions on office information systems, 2(4):267–288, oct. 1984.

[13] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[14] William Feller. *An introduction to probability theory and its applications.* John Wiley & Sons, 1968.

[15] Michael L. Fredman, Janos Komlos, and Endre Szemeredi. Storing a sparse table with $o(1)$ worst case access time. journal of assoc. comput. mach., 31(3):538-544, 1984., 1984.

[16] Lee L. Gremillion. Designing a bloom filter for differential file access. *Communications of the Association for Computing Machinery*, 25(9):600–604, 1982.

[17] A. Gulbrandsen and P. Vixie. Rfc 2052 - a dns rr for specifying the location of services (dns srv), october 1996.

[18] E. Guttman. Rfc 2608: Service location protocol v.2 draft. technical report, sun microsystems., 1999.

[19] M. C. Harrison. Implementation of the substring test by hashing. *Communications of the Association for Computing Machinery*, 14(12):777–779, 1971.

[20] P. Indyk. Deterministic superimposed coding with applications to pattern matching, proceedings of the38th IEEE annual symposium on foundations of computer science. pages 127–136. IEEE Computer Society Press, 1997.

[21] H.L. Montgomery I.Niven, H.S. Zuckerman. *The theory of Numbers.* John Wiley & Sons, 1991.

[22] R. Jain. A comparison of hashing schemes for address lookup in computer networks. *IEEE Transactions on Communications*, 40(3):1570–1573, 1992.

[23] S. Keshav. *An Engineering Approach to Computer Networking.* Addison-Wesley, 1997.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Addison-Wesley, Reading, MA, USA, 1973.

[25] Donald E. Knuth. *Selected papers on Analysis of Algorithms*. Addison-Wesley, 2000.

[26] Daniel K. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. *Master's thesis, MIT.*, 1996.

[27] J. Marais and K. Bharat. Supporting cooperative and personal surfing with a desktop assistant. *Proceedings of 10th ACM symposium on UIST*, 1997.

[28] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang. Url forwarding and compression in adaptive web caching. in proc. of ieee infocom 2000, volume 2, pages 670–678, mar. 2000.

[29] Sun microsystems. Jini technology architectural overview. http://www.sun.com/jini/whitepapers/architecture.html.

[30] R. Moats. RFC 2141: URN syntax, 1997.

[31] Clifford Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, applications and policy*, pages 30–37, 1992.

[32] S.M.Bellovin P. Honeyman. *Pathalias* or the care and feeding of relative addresses. *Usenix Conference Proceedings*, pages 126–141, 1986.

[33] M. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. communications of the acm, volume 32, number 10, pages 1237–1239, 1989.

[34] I. Rhee, G. Rouskas, and N.Balaguru. Mtcp: Scalable tcp-like congestion control for reliable multicast. *NCSU Technical report, TR-98-01.*

[35] R.L.Rivest. Rfc 1321, the MD5 message-digest algorithm, 1992.

[36] Kenneth H. Rosen. *Elelmentary number theory and its applications*. Addison-Wesley, 1992.

[37] K. Sollins. RFC 2276: Architectural principles of Uniform Resource Name resolution, 1998.

[38] Richard Stevens. *TCP/IP Illustrated Volume 1.* Addison-Wesley, 1994.

[39] A. L. Tharp and K.-C. Tai. The practicality of text signatures for accelerating string searching. *Software Practice and Experience*, 12(1):35–44, Jan 1982.

[40] Alan L. Tharp. *File Organization and Processing.* John Wiley & Sons, 1988.

[41] Joseph D. Touch. Performance analysis of MD5. In *Proceedings SIGCOMM*, volume 25, pages 77–86, 1995.

[42] A. Vahdat and M. Dahlin. Active names: Flexible location and transport of wide-area resources. in proc. usenix usits (oct. 1999).

[43] M. Wahl, T. Howes, and S. Kille. Rfc 2251- lightweight directory access protocol version 3 (ldapv3). ietf, december 1997.

[44] E. J. Watson. Primitive polynomials. *Mathematics of Computation*, 16(79):368–369, jul 1979.

[45] W.Stallings. *Data and Computer Communications.* Macmillan, 1988.

Appendix A

Source Code

```
# @ident : www.perl
# perl script to pull URLs from Search engine AltaVista
#!/usr/bin/perl
use strict;
use warnings;
$|++;

use WWW::Search;
my $search = new WWW::Search('AltaVista');

open(DICT,"dictionary") or die $!;
my $url = 0;
SEARCH:
while(<DICT>) {
        chomp(my $word = $_);
        $search->native_query(WWW::Search::escape_query($word));
        my $num_responses = 0;
        while (my $result = $search->next_result()
                and $num_responses++<10
              ) {
                last SEARCH if $url++>100000;
                print $result->url,"\n";
        }
}
```

```
/*
  Fibonacci Hash + signature creation for service location
  Author : Venu Ullanatt
*/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include"table.128bit.h"
#include<sys/time.h>
#include<sys/resource.h>
#include<unistd.h>

void create_hashtable(char *);
unsigned int computehash(char *array,int len);
void create_signature(char *str,int len,unsigned int *signature_);
void modify_signature(char *attributes,int len,unsigned int *signature);
void insert(unsigned int hashcode,unsigned int *signature);
short hashfunction(char x,char y);
void search_pattern(char *pattern,int len);
void printbinary(int num);
void search_string(char *file);

#define MAXSTRLEN 201
#define MINBITSET 16
#define DELIMITER 47
//#define HASHTBLSIZE 8192
//#define HASHTBLSIZE 16384
//#define HASHTBLSIZE 32768
#define HASHTBLSIZE 65536
#define TESTRECORDS 313436
struct payload {
unsigned int signature[4];
struct payload *next;
};

int falsedrop = 0;
struct payload *hashtable[HASHTBLSIZE];

main(int argc, char **argv)
{
char array[MAXSTRLEN];
int i;
for(i=0;i<HASHTBLSIZE;i++)
hashtable[i] = NULL;   //initialize pointers
```

```
create_hashtable(argv[1]);
search_string(argv[2]);
printf("false drop= %d\n",falsedrop);
}

void create_hashtable(char *attrfile)
{
FILE *fp2;
char servicename[MAXSTRLEN];
char attributes[MAXSTRLEN];
unsigned int hashcode;
unsigned int signature[4];
short i;
int recordcnt = 0;

fp2=fopen(attrfile,"r");
while(!feof(fp2)) {
fscanf(fp2,"%s\n",attributes);
recordcnt++;
i=0;
                while(attributes[i] != '/' && attributes[i] != 0) {
                        servicename[i] = attributes[i];
                        i++;
                }
                servicename[i] = 0;

hashcode = computehash(servicename,strlen(servicename));
create_signature(attributes,strlen(attributes),signature);
// modify_signature(attributes,strlen(attributes),signature);
insert(hashcode,signature);
}
fclose(fp2);
printf("Number of records read = %d\n",recordcnt);
}

void modify_signature(char *attributes,int len,unsigned int *signature)
{
int i;
unsigned int temp=0;

temp = attributes[4];
temp <<= 25;
signature[3] |= temp;
}
```

```
unsigned int computehash(char *array,int len)
{
unsigned int hashcode;
short numbits;
int i;
hashcode= 0;
for(i=0;i<len;i++) {
// array[i] = array[i] | 0x20; //to change case;
numbits = i&0xf;  //max 31 bits rotate
hashcode+= ((array[i] << numbits) | (array[i] >> 32-numbits));
}
hashcode= hashcode * 2654435769;
hashcode = (hashcode >> 16);
#ifdef DEBUGHASHCODE
printf("%011u %-50s\n",hashcode,array);
#endif
return hashcode;
}

void create_signature(char *str,int len,unsigned int *signature_)
{
        short hashcode;
        unsigned int one = 1;
        unsigned int shift;
        int i;
int reverse;
short byte;


        signature_[0]=signature_[1]=signature_[2]=signature_[3]=0;
i = 0;
while(i < len-1) {
                one = 1;
if(str[i+1] != DELIMITER)
                 hashcode = hashfunction(str[i],str[i+1]);
else {
i+=2;     //jump over the DELIMITER
continue;
}

#ifdef DEBUGSIGNATURE
printf("%c%c=%d ",str[i],str[i+1],hashcode);
#endif
```

```
byte = hashcode/32;
shift = hashcode - (byte*32);
                signature_[byte] |= (one << shift);
i++;
        }

#ifdef DEBUGSIGNATURE
printf("\n",hashcode);
#endif

#ifdef DEBUGSIGNATURE
printbinary(signature_[3]);
printf("\n");
printbinary(signature_[2]);
printf("\n");
printbinary(signature_[1]);
printf("\n");
printbinary(signature_[0]);
printf("\n");
#endif
}

void insert(unsigned int hashcode,unsigned int *signature)
{
struct payload *tmp;
struct payload *newnode;


if( (newnode = (struct payload *)malloc(sizeof(struct payload))) ==
(struct payload *)NULL )  {
printf("Insert : malloc error\n");
exit(1);
}

newnode->signature[0] = signature[0];
newnode->signature[1] = signature[1];
newnode->signature[2] = signature[2];
newnode->signature[3] = signature[3];
newnode->next = NULL;

if(hashtable[hashcode] == NULL) {
hashtable[hashcode] = newnode;
return;
}
```

```
tmp = hashtable[hashcode];
while(tmp->next != NULL)
tmp = tmp->next;
tmp->next = newnode;
return;
}

void search_pattern(char *pattern,int len)
{
        unsigned int pattern_signature[4];
unsigned int hashcode;
struct payload *tmp;
char service[MAXSTRLEN];
int i =0;
char *strptr;
static int cnt=0;

while( pattern[i] != 0) {
service[i] = pattern[i];
i++;
if (pattern[i] == '/')     /* '/' used as delim */
break;
}
service[i] = 0;

hashcode = computehash(service,strlen(service));
if(hashtable[hashcode] == NULL) {
#ifdef DEBUGNOTFND
printf("Service = %s not found: bucket = %u,no bucket\n",pattern,hashcode);
#endif
return;
}

        create_signature(pattern,len,pattern_signature);
// modify_signature(pattern,strlen(pattern),pattern_signature);

tmp = hashtable[hashcode];
while(tmp != (struct payload *)NULL) {
#ifdef DEBUGPERFORMANCE
printf("Trversing chain\n");
#endif
if( ((pattern_signature[0] & tmp->signature[0]) == pattern_signature[0]) &&
    ((pattern_signature[1] & tmp->signature[1])  == pattern_signature[1]) &&
```

```
   ((pattern_signature[2] & tmp->signature[2])  == pattern_signature[2]) &&
  ((pattern_signature[3] & tmp->signature[3])  == pattern_signature[3])) {
#ifdef DEBUGFOUND
 //         printf("Service=%s found, bucket = %u\n",pattern,hashcode);
// printbinary(pattern_signature[3]);
// printf("\n");
// printbinary(pattern_signature[2]);
// printf("\n");
// printbinary(pattern_signature[1]);
// printf("\n");
// printbinary(pattern_signature[0]);
// printf("\n");
falsedrop++;
// printf("false drop= %d\n",falsedrop);
#endif
         return ;
        }
tmp = tmp->next;
        }  //end of while

#ifdef DEBUGNOTFND
        printf("Service = %s not found : signature mismatch \n",pattern);
#endif
return;
}

short hashfunction(char x,char y)
{
return(11*table[x]+table[y]);
}

#define GETBIT(x,y)  (x << y) & 0x80000000
void printbinary(int num)
{
int i;

for(i=0;i<32;i++) {
if(GETBIT(num,i))
printf("1");
else
printf("0");
}
}
```

```
#ifdef DEBUGUSAGE
void search_string(char *file)
{
FILE *fp;
char pattern[TESTRECORDS][MAXSTRLEN];
// char *pattern[MAXSTRLEN];
int numrecords;
        struct rusage startUsage,endUsage;
unsigned long int userTime;
unsigned long int systemTime;
int i;

// for(i=0;i<TESTRECORDS;i++){
// pattern[i] = (char *)malloc((sizeof(char *)));
//if( (pattern[i] = (char *)malloc((sizeof(char)))) == NULL )  {
// printf("Insert : malloc error\n");
// exit(1);
//}
// }

numrecords = 0;
fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern[numrecords]);
numrecords++;
}
fclose(fp);

getrusage(RUSAGE_SELF,&startUsage);
while(--numrecords >= 0 ) {
search_pattern(pattern[numrecords],strlen(pattern[numrecords]));
}

getrusage(RUSAGE_SELF,&endUsage);

userTime=(endUsage.ru_utime.tv_sec*1000000 + endUsage.ru_utime.tv_usec) -
 (startUsage.ru_utime.tv_sec*1000000 + startUsage.ru_utime.tv_usec);
systemTime=(endUsage.ru_stime.tv_sec*1000000 + endUsage.ru_stime.tv_usec) -
   (startUsage.ru_stime.tv_sec*1000000 + startUsage.ru_stime.tv_usec);
printf("UserTime = %ld microseconds, systemTime = %ld microseconds\n",
userTime,systemTime);
}
#else
void search_string(char *file)
```

```
{
FILE *fp;
char pattern[MAXSTRLEN];

fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern);
search_pattern(pattern,strlen(pattern));
}
fclose(fp);
}
#endif
```

```
/*
  CRC Hash + signature creation for service location
  Author : Venu Ullanatt
*/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include"table.128bit.h"
#include<sys/time.h>
#include<sys/resource.h>
#include<unistd.h>

void create_hashtable(char *);
unsigned int computehash(char *array,int len);
void create_signature(char *str,int len,unsigned int *signature_);
void insert(unsigned int hashcode,unsigned int *signature);
short hashfunction(char x,char y);
void search_pattern(char *pattern,int len);
void printbinary(int num);
void search_string(char *file);

#define MAXSTRLEN 201
#define MINBITSET 16
#define DELIMITER 47
//#define HASHTBLSIZE 8192
//#define HASHTBLSIZE 16384
//#define HASHTBLSIZE 32768
#define HASHTBLSIZE 65536
#define TESTRECORDS 314436
struct payload {
unsigned int signature[4];
struct payload *next;
};

#define DBZ_HASH_SIZE   4
typedef struct {
    unsigned char hash[DBZ_HASH_SIZE];
} hash_t;

hash_t dbzhash_crc(char *name, int size_in);
#define POLY 0x48000000L  /* 31-bit polynomial (avoids sign problems) */
static long CrcTable[128];

struct payload *hashtable[HASHTBLSIZE];
```

```
int falsedrop = 0;

main(int argc, char **argv)
{
char array[MAXSTRLEN];
int i;
for(i=0;i<HASHTBLSIZE;i++)
hashtable[i] = NULL;   //initialize pointers
create_hashtable(argv[1]);
search_string(argv[2]);
printf("false drop= %d\n",falsedrop);
}


void create_signature(char *str,int len,unsigned int *signature_)
{
        short hashcode;
        unsigned int one = 1;
        unsigned int shift;
        int i;
int reverse;
short byte;


        signature_[0]=signature_[1]=signature_[2]=signature_[3]=0;
i = 0;
while(i < len-1) {
                one = 1;
if(str[i+1] != DELIMITER)
                 hashcode = hashfunction(str[i],str[i+1]);
else {
i+=2;     //jump over the DELIMITER
continue;
}

#ifdef DEBUGSIGNATURE
printf("%c%c=%d ",str[i],str[i+1],hashcode);
#endif
byte = hashcode/32;
shift = hashcode - (byte*32);
                signature_[byte] |= (one << shift);
i++;
        }
```

```
#ifdef DEBUGSIGNATURE
printf("\n",hashcode);
#endif

#ifdef DEBUGSIGNATURE
printbinary(signature_[3]);
printf("\n");
printbinary(signature_[2]);
printf("\n");
printbinary(signature_[1]);
printf("\n");
printbinary(signature_[0]);
printf("\n");
#endif
}

void insert(unsigned int hashcode,unsigned int *signature)
{
struct payload *tmp;
struct payload *newnode;


if( (newnode = (struct payload *)malloc(sizeof(struct payload))) ==
(struct payload *)NULL )  {
printf("Insert : malloc error\n");
exit(1);
}

newnode->signature[0] = signature[0];
newnode->signature[1] = signature[1];
newnode->signature[2] = signature[2];
newnode->signature[3] = signature[3];
newnode->next = NULL;

if(hashtable[hashcode] == NULL) {
hashtable[hashcode] = newnode;
return;
}

tmp = hashtable[hashcode];
while(tmp->next != NULL)
tmp = tmp->next;
tmp->next = newnode;
return;
```

```
}

void search_pattern(char *pattern,int len)
{
        unsigned int pattern_signature[4];
unsigned int hashcode;
struct payload *tmp;
char service[MAXSTRLEN];
int i =0;
char *strptr;
int cnt;
        hash_t hash;

while( pattern[i] != 0) {
service[i] = pattern[i];
i++;
if (pattern[i] == '/')     /* '/' used as delim */
break;
}
service[i] = 0;

        hash = dbzhash_crc(service, strlen(service));
hashcode = (hash.hash[0]  | ((hash.hash[1] & 0xff) << 8));
if(hashtable[hashcode] == NULL) {
#ifdef DEBUGNOTFND
printf("Service = %s not found: bucket = %u,no bucket\n",pattern,hashcode);
#endif
return;
}

        create_signature(pattern,len,pattern_signature);
// modify_signature(pattern,strlen(pattern),pattern_signature);

tmp = hashtable[hashcode];
while(tmp != NULL) {
#ifdef DEBUGPERFORMANCE
printf("Trversing chain\n");
#endif
if( ((pattern_signature[0] & tmp->signature[0]) == pattern_signature[0]) &&
     ((pattern_signature[1] & tmp->signature[1])  == pattern_signature[1]) &&
     ((pattern_signature[2] & tmp->signature[2])  == pattern_signature[2]) &&
     ((pattern_signature[3] & tmp->signature[3])  == pattern_signature[3])) {
#ifdef DEBUGFOUND
falsedrop++;
```

```
//      printf("Service=%s found, bucket = %u\n",pattern,hashcode);
#endif
         return ;
        }
tmp = tmp->next;
        }  //end of while

#ifdef DEBUGNOTFND
        printf("Service = %s not found : signature mismatch \n",pattern);
#endif
}
short hashfunction(char x,char y)
{
return(11*table[x]+table[y]);
}

#define GETBIT(x,y)  (x << y) & 0x80000000
void printbinary(int num)
{
int i;

for(i=0;i<32;i++) {
if(GETBIT(num,i))
printf("1");
else
printf("0");
}
}

#ifdef DEBUGUSAGE
void search_string(char *file)
{
FILE *fp;
char pattern[TESTRECORDS][MAXSTRLEN];
int numrecords;
        struct rusage startUsage,endUsage;
unsigned long int userTime;
unsigned long int systemTime;


numrecords = 0;
fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern[numrecords]);
```

```
numrecords++;
}
fclose(fp);

getrusage(RUSAGE_SELF,&startUsage);
while(--numrecords >= 0 ) {
search_pattern(pattern[numrecords],strlen(pattern[numrecords]));
}

getrusage(RUSAGE_SELF,&endUsage);

userTime=(endUsage.ru_utime.tv_sec*1000000 + endUsage.ru_utime.tv_usec) -
  startUsage.ru_utime.tv_sec*1000000 + startUsage.ru_utime.tv_usec);
systemTime=(endUsage.ru_stime.tv_sec*1000000 + endUsage.ru_stime.tv_usec) -
 (startUsage.ru_stime.tv_sec*1000000 + startUsage.ru_stime.tv_usec);
printf("UserTime = %ld microseconds, systemTime = %ld microseconds\n",
userTime,systemTime);
}
#else
void search_string(char *file)
{
FILE *fp;
char pattern[MAXSTRLEN];

fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern);
search_pattern(pattern,strlen(pattern));
}
fclose(fp);
}
#endif

/*
 - crcinit - initialize tables for hash function
 */
static void
crcinit()
{
            register int i, j;
            register long sum;

            for (i = 0; i < 128; ++i) {
                    sum = 0L;
```

```
                    for (j = 7 - 1; j >= 0; --j)
                            if (i & (1 << j))
                                    sum ^= POLY >> j;
                    CrcTable[i] = sum;
            }
}

static void ltouchar(long sum, unsigned char *str)
{
        int i;
        for (i=0; i<4; i++) {
            str[i] = sum & 0xff;
            sum >>= 8;
        }
}

    /*
     - dbzhash_crc - Extended Honeyman's nice hashing function
     */

hash_t dbzhash_crc(char *name, int size_in)
{
            long sum = 0L;
            int size;
            static hash_t hash;
            unsigned char tmp;

            size = size_in;
            while (size--) {
                 sum = (sum >> 7) ^ CrcTable[(sum ^ (*name++)) & 0xff];
            }
            ltouchar(sum, &hash.hash[0]);


            /* move 7-bit hash of the first word to the tail */
/*
            tmp = hash.hash[3];
           hash.hash[3] = hash.hash[6];
          hash.hash[6] = tmp;
*/
         return hash;
}

void create_hashtable(char *attrfile)
```

```
{
FILE *fp2;
char servicename[MAXSTRLEN];
char attributes[MAXSTRLEN];
unsigned int hashcode;
unsigned int signature[4];
short i;
        hash_t hash;
int recordcnt = 0;

fp2=fopen(attrfile,"r");
crcinit();
while(!feof(fp2)) {
fscanf(fp2,"%s\n",attributes);
recordcnt++;
i=0;
                while(attributes[i] != '/' && attributes[i] != 0) {
                        servicename[i] = attributes[i];
                        i++;
                }
                servicename[i] = 0;

                hash = dbzhash_crc(servicename, strlen(servicename));
hashcode = (hash.hash[0]  | ((hash.hash[1] & 0x7f) << 8));
// hashcode = ((hash.hash[0] << 8) | (hash.hash[1] & 0x1f));
// hashcode = ((hash.hash[0] << 16) | (hash.hash[1] << 8) | \
//        (hash.hash[2] & 0x0f));
create_signature(attributes,strlen(attributes),signature);
insert(hashcode,signature);
}
fclose(fp2);
printf("Number of records read = %d\n",recordcnt);
}
```

```
/*
  Bloom filter implementation
*/
#include <stdio.h>
#include <time.h>
#include <string.h>
#include<sys/time.h>
#include<sys/resource.h>
#include<unistd.h>
#include "global.h"
#include "md5.h"
#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final
#define MAXSTRLEN 201
#define MINBITSET 16
#define DELIMITER 47
#define LOADFACTOR 32  //if load factor changes change table size also.
#define TESTRECORDS 313436
#define NUMRECORDS 308769
#define TABLESIZE 9880608
#define MD 5

static void MDString PROTO_LIST ((char *));
void search_string(char *);
void create_bitarray(char *);
void search_pattern(char *pattern,int len);
void setbits(unsigned char *digest);
void testbits(unsigned char *digest);
void printbinary(char num);

unsigned int bitarray[NUMRECORDS];
int falsedrop = 0;

main(int argc, char **argv)
{
char array[MAXSTRLEN];
int i;
for(i=0;i<NUMRECORDS;i++)
bitarray[i] = 0;   //initialize bit array
create_bitarray(argv[1]);
search_string(argv[2]);
printf("false drop= %d\n",falsedrop);
```

```
}

void create_bitarray(char *attrfile)
{
FILE *fp2;
char attributes[MAXSTRLEN];
unsigned char digest[16];
int recordcnt = 0;

fp2=fopen(attrfile,"r");
while(!feof(fp2)) {
fscanf(fp2,"%s\n",attributes);
recordcnt++;
      MDString (attributes,digest);
setbits(digest);
}

fclose(fp2);
printf("Records  read= %d\n",recordcnt);
}

/* Digests a string:sets 4 bit positions in the bit array.
 */
static void MDString (char *string,unsigned char *digest)
{
int i;
MD_CTX context;
unsigned int len = strlen (string);

MDInit (&context);
MDUpdate (&context, string, len);
MDFinal (digest, &context);

#ifdef DEBUG
for(i=0;i<16;i++) {
printf("%u %x ",digest[i],digest[i]);
printbinary(digest[i]);
printf("\n");
}
printf("-------------------------------------\n");
#endif


}
```

```
void setbits(unsigned char *digest)
{
unsigned int byte;
unsigned int shift;
unsigned int hashvalue;
unsigned int one=1;
unsigned int value=0;

one = 1;
value = ( digest[0] | (digest[1] << 8) | (digest[2] << 16) | \
  (digest[3] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
bitarray[byte] |= (one<<shift);

one = 1;
value = ( digest[4] | (digest[5] << 8) | (digest[6] << 16) | \
  (digest[7] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
bitarray[byte] |= (one<<shift);

one = 1;
value = ( digest[8] | (digest[9] << 8) | (digest[10] << 16) | \
  (digest[11] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
bitarray[byte] |= (one<<shift);

one = 1;
value = ( digest[12] | (digest[13] << 8) | (digest[14] << 16) | \
```

```
  (digest[15] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
bitarray[byte] |= (one<<shift);

}

void testbits(unsigned char *digest)
{
unsigned int byte;
unsigned int shift;
unsigned int hashvalue;
unsigned int one=1;
unsigned int value=0;

one = 1;
value = ( digest[0] | (digest[1] << 8) | (digest[2] << 16) | \
  (digest[3] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;

#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
if( (bitarray[byte] & (one<<shift)) == 0) {
#ifdef DEBUGNOTFND
printf("no match\n");
#endif
return;
}

one = 1;
value = ( digest[4] | (digest[5] << 8) | (digest[6] << 16) | \
  (digest[7] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
```

```
#endif
if( (bitarray[byte] & (one<<shift)) == 0) {
#ifdef DEBUGNOTFND
printf("no match\n");
#endif
return;
}

one = 1;
value = ( digest[8] | (digest[9] << 8) | (digest[10] << 16) | \
  (digest[11] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
if( (bitarray[byte] & (one<<shift)) == 0) {
#ifdef DEBUGNOTFND
printf("no match\n");
#endif
return;
}

one = 1;
value = ( digest[12] | (digest[13] << 8) | (digest[14] << 16) | \
  (digest[15] << 24));
hashvalue = value%TABLESIZE;
byte = hashvalue/32;
shift = hashvalue - byte*32;
#ifdef DEBUG1
printf("%x %u %u %u %d\n",value,value,hashvalue,byte,shift);
#endif
if( (bitarray[byte] & (one<<shift)) == 0) {
#ifdef DEBUGNOTFND
printf("no match\n");
#endif
return;
}


#ifdef DEBUGFOUND
falsedrop++;
// printf("match found\n");
```

```
#endif
}

void search_pattern(char *pattern,int len)
{
unsigned char digest[16];
     MDString (pattern,digest);
testbits(digest);
}

#ifdef DEBUGUSAGE
void search_string(char *file)
{
FILE *fp;
char pattern[TESTRECORDS][MAXSTRLEN];
int numrecords;
        struct rusage startUsage,endUsage;
unsigned long int userTime;
unsigned long int systemTime;


numrecords = 0;
fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern[numrecords]);
numrecords++;
}
fclose(fp);

getrusage(RUSAGE_SELF,&startUsage);
while(--numrecords >= 0 ) {
search_pattern(pattern[numrecords],strlen(pattern[numrecords]));
}

getrusage(RUSAGE_SELF,&endUsage);

userTime=(endUsage.ru_utime.tv_sec*1000000 + endUsage.ru_utime.tv_usec) -
 (startUsage.ru_utime.tv_sec*1000000 + startUsage.ru_utime.tv_usec);
systemTime=(endUsage.ru_stime.tv_sec*1000000 + endUsage.ru_stime.tv_usec) -
 (startUsage.ru_stime.tv_sec*1000000 + startUsage.ru_stime.tv_usec);
printf("UserTime = %ld microseconds, systemTime = %ld microseconds\n",
userTime,systemTime);
}
#else
```

```
void search_string(char *file)
{
FILE *fp;
char pattern[MAXSTRLEN];

fp = fopen(file,"r");
while(!feof(fp)) {
fscanf(fp,"%s\n",pattern);
search_pattern(pattern,strlen(pattern));
}
fclose(fp);
}
#endif


#define GETBIT(x,y)  (x << y) & 0x80
void printbinary(char num)
{
int i;

for(i=0;i<8;i++) {
if(GETBIT(num,i))
printf("1");
else
printf("0");
}
}
```

```
/* MD5C.C - RSA Data Security, Inc., MD5 message-digest algorithm
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
 */

#include "global.h"
#include "md5.h"

/* Constants for MD5Transform routine.
 */

#define S11 7
#define S12 12
#define S13 17
#define S14 22
#define S21 5
#define S22 9
#define S23 14
#define S24 20
#define S31 4
#define S32 11
#define S33 16
#define S34 23
#define S41 6
```

```
#define S42 10
#define S43 15
#define S44 21

static void MD5Transform PROTO_LIST ((UINT4 [4], unsigned char [64]));
static void Encode PROTO_LIST
  ((unsigned char *, UINT4 *, unsigned int));
static void Decode PROTO_LIST
  ((UINT4 *, unsigned char *, unsigned int));
static void MD5_memcpy PROTO_LIST ((POINTER, POINTER, unsigned int));
static void MD5_memset PROTO_LIST ((POINTER, int, unsigned int));

static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G, H and I are basic MD5 functions.
 */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
 (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define GG(a, b, c, d, x, s, ac) { \
 (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define HH(a, b, c, d, x, s, ac) { \
 (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
```

```
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define II(a, b, c, d, x, s, ac) { \
 (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }

/* MD5 initialization. Begins an MD5 operation, writing a new context.
 */
void MD5Init (context)
MD5_CTX *context;                                        /* context */
{
  context->count[0] = context->count[1] = 0;
  /* Load magic initialization constants.
*/
  context->state[0] = 0x67452301;
  context->state[1] = 0xefcdab89;
  context->state[2] = 0x98badcfe;
  context->state[3] = 0x10325476;
}

/* MD5 block update operation. Continues an MD5 message-digest
  operation, processing another message block, and updating the
  context.
 */
void MD5Update (context, input, inputLen)
MD5_CTX *context;                                        /* context */
unsigned char *input;                                 /* input block */
unsigned int inputLen;                      /* length of input block */
{
  unsigned int i, index, partLen;

  /* Compute number of bytes mod 64 */
  index = (unsigned int)((context->count[0] >> 3) & 0x3F);

  /* Update number of bits */
  if ((context->count[0] += ((UINT4)inputLen << 3))

   < ((UINT4)inputLen << 3))
 context->count[1]++;
  context->count[1] += ((UINT4)inputLen >> 29);
```

```
  partLen = 64 - index;

  /* Transform as many times as possible.
*/
  if (inputLen >= partLen) {
 MD5_memcpy
   ((POINTER)&context->buffer[index], (POINTER)input, partLen);
 MD5Transform (context->state, context->buffer);

 for (i = partLen; i + 63 < inputLen; i += 64)
   MD5Transform (context->state, &input[i]);

 index = 0;
  }
  else
 i = 0;

  /* Buffer remaining input */
  MD5_memcpy
 ((POINTER)&context->buffer[index], (POINTER)&input[i],
  inputLen-i);
}

/* MD5 finalization. Ends an MD5 message-digest operation, writing the
  the message digest and zeroizing the context.
 */
void MD5Final (digest, context)
unsigned char digest[16];                        /* message digest */
MD5_CTX *context;                                      /* context */
{
  unsigned char bits[8];
  unsigned int index, padLen;

  /* Save number of bits */
  Encode (bits, context->count, 8);

  /* Pad out to 56 mod 64.
*/
  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
  padLen = (index < 56) ? (56 - index) : (120 - index);
  MD5Update (context, PADDING, padLen);

  /* Append length (before padding) */
  MD5Update (context, bits, 8);
```

```
  /* Store state in digest */
  Encode (digest, context->state, 16);

  /* Zeroize sensitive information.
*/
  MD5_memset ((POINTER)context, 0, sizeof (*context));
}

/* MD5 basic transformation. Transforms state based on block.
 */
static void MD5Transform (state, block)
UINT4 state[4];
unsigned char block[64];
{
  UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

  Decode (x, block, 64);

  /* Round 1 */
  FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
  FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
  FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
  FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
  FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
  FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
  FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
  FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
  FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
  FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
  FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
  FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
  FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
  FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
  FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
  FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

 /* Round 2 */
  GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
  GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
  GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
  GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
  GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
  GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */
```

```
GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */

GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */

/* Round 4 */
II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
```

```
  II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
  II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
  II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;

  /* Zeroize sensitive information.

*/
  MD5_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len is
  a multiple of 4.
 */
static void Encode (output, input, len)
unsigned char *output;
UINT4 *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4) {
 output[j] = (unsigned char)(input[i] & 0xff);
 output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
 output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
 output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
  }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len is
  a multiple of 4.
 */
static void Decode (output, input, len)
UINT4 *output;
unsigned char *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4)
```

```
 output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
   (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
 */

static void MD5_memcpy (output, input, len)
POINTER output;
POINTER input;
unsigned int len;
{
  unsigned int i;

  for (i = 0; i < len; i++)

 output[i] = input[i];
}

/* Note: Replace "for loop" with standard memset if possible.
 */
static void MD5_memset (output, value, len)
POINTER output;
int value;
unsigned int len;
{
  unsigned int i;

  for (i = 0; i < len; i++)
 ((char *)output)[i] = (char)value;
}
```

```
/* MD5.H - header file for MD5C.C
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
 */

/* MD5 context. */
typedef struct {
  UINT4 state[4];                                   /* state (ABCD) */
  UINT4 count[2];        /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                         /* input buffer */
} MD5_CTX;

void MD5Init PROTO_LIST ((MD5_CTX *));
void MD5Update PROTO_LIST
  ((MD5_CTX *, unsigned char *, unsigned int));
void MD5Final PROTO_LIST ((unsigned char [16], MD5_CTX *));
```

```
/* GLOBAL.H - RSAREF types and constants
 */

/* PROTOTYPES should be set to one if and only if the compiler supports
  function argument prototyping.
The following makes PROTOTYPES default to 0 if it has not already

  been defined with C compiler flags.
 */
#ifndef PROTOTYPES
#define PROTOTYPES 0
#endif

/* POINTER defines a generic pointer type */
typedef unsigned char *POINTER;

/* UINT2 defines a two byte word */
typedef unsigned short int UINT2;

/* UINT4 defines a four byte word */
typedef unsigned long int UINT4;

/* PROTO_LIST is defined depending on how PROTOTYPES is defined above.
If using PROTOTYPES, then PROTO_LIST returns the list, otherwise it
  returns an empty list.
 */
#if PROTOTYPES
#define PROTO_LIST(list) list
#else
#define PROTO_LIST(list) ()
#endif
```

Appendix B

Hash collision tables

Total # keys = 7700 Hash table size = 8192					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
2984	0	3033	0	2991	0
1375	1	1375	1	1388	1
469	2	435	2	451	2
108	3	111	3	115	3
22	4	23	4	22	4
3	5	9	5	3	5
1	6	1	6		

Total # keys = 10000 Hash table size = 8192					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
2926	0	2869	0	2948	0
1791	1	1791	1	1868	1
756	2	801	2	794	2
297	3	266	3	259	3
69	4	75	4	55	4
22	5	19	5	14	5
5	6	6	6	1	6
		1	7	1	7

Total # keys = 15000 Hash table size = 16384					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
5761	0	5909	0	6035	0
2670	1	2756	1	2851	1
925	2	889	2	860	2
255	3	221	3	178	3
59	4	42	4	49	4
19	5	18	5	10	5
3	6	4	6		
1	7	2	7		

Total # keys = 17000 Hash table size = 16384					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
5780	0	5909	0	6035	0
3078	1	2756	1	2851	1
1146	2	889	2	860	2
378	3	221	3	178	3
108	4	42	4	49	4
33	5	18	5	10	5
8	6	4	6		
5	7	2	7		

Total # keys = 34000 Hash table size = 32768					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
11397	0	11932	0	12084	0
5742	1	6132	1	6477	1
2203	2	2192	2	2143	2
682	3	618	3	562	3
266	4	169	4	111	4
79	5	35	5	17	5
25	6	11	6	3	6
17	7	2	7	1	7
3	8	1	8		
3	9				
1	11				

Total # keys = 68000 Hash table size = 65536					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
22384	0	23886	0	24205	0
10792	1	12207	1	12605	1
4070	2	4353	2	4329	2
1502	3	1237	3	1135	3
563	4	298	4	226	4
222	5	69	5	41	5
120	6	14	6	7	6
61	7	6	7	1	7
31	8	2	8		
21	9				
12	10				
2	11				
4	12				
1	16				

Total # keys = 94000 Hash table size = 65536					
Rotate + Fib Hashing		CRC Hashing		MD5 hashing	
#slots	Collisions	#slots	Collisions	#slots	Collisions
21443	0	22417	0	22266	0
13476	1	15699	1	15967	1
6524	2	7473	2	7762	2
2813	3	2753	3	2728	3
1167	4	875	4	823	4
539	5	273	5	228	5
242	6	91	6	34	6
153	7	22	7	8	7
96	8	5	8	1	8
66	9	3	9		
27	10	3	10		
20	11	1	11		
14	12				
6	13				
3	14				
1	15				
1	17				
1	18				
1	21				

Appendix C

Character distribution table for 64bit signatures

Class	Characters
0	W
1	E B 6 : & ' "
2	T X Z W G 5 ; /
3	A F Y P ,) ! > ∧
4	O L C 3 . (@ [_
5	I K D M J Q 2 9 #] —
6	N V S 1 8 - $
7	H U R 0 7 + % }