The Great Firewall of Santa Cruz

Moore Macauley
Glorious People's Republic of Santa Cruz

November 20, 2022

1 Introduction

Thanks to my appointm- I mean, my victory in the recently held election, I have become the leader of the Glorious People's Republic of Santa Cruz, and I have a duty to my subject- I mean, citizens to keep them safe, happy, and compliant from the terrors of the internet. To do this, I will create a program that will be able to quickly filter out any ungood words, and inform the citizen who produced them that they will be enjoying the next few weeks in joycamp!

Filtering the words of so many loyal subjects will be difficult, as undoubtedly some lonely few will believe my victory was unearned somehow, and may begin to wrongthink. In service of this, I will begin by creating a bloom filter. This filter will, using a series of hash functions and a bit vector, be able to quickly rule out most words that are uncategorized or are already newspeak, so that my system is not bogged down filtering the words of the truly loyal.

Once this bloom filter has ruled out the acceptable and newspeak words, we will move on to filtering the potentially ungood words. To do this, I will then use a chained hash table, filled with all of the oldspeak, badspeak, and newspeak, to quickly test to see if the words used are ungood or not. To implement this properly, I will also need to produce a doubly linked list, as for the hash table to be chained properly, each element in it should be a linked list. This list will then be used to deal with collisions, as badthink or oldthink that map to the same hash value need to be dealt with properly.

Finally, I will need to create a program that processes the text provided by my loyal subject. For this, I will need a file parser, which will provide the next word that my loyal subject typed every time it is called, until there are no more words to provide.

This will all be put together in my main function which will first propagate a hash table and bloom filter with all of the badspeak, along with the oldspeak/newspeak pairs. I will then use the parser to get each word in succession from the input. Using this, I will take the bloom filter to determine if a word is possibly bad/oldspeak, or definitely safe, and if it is potentially ungood, I then use the hash table to determine for sure. If ungood words are contained, I will use the linked list implementation to save all of the ungood words, before informing the user of their wrongthink and taking the appropriate action.

2 Reimplementing Parts of the String Library

In the GPRSC's infinite wisdom, we have decided to depreciate parts of the standard library, as they were obviously the creation of corrupt, capitalist pig-dogs. However, I do need to make use of some of their functionality, and as such will need to reproduce them. Some of these functions will lack some of the functionality seen in the library, but this is because these features go unused in this program. These

functions will be included as statics inside .c files as needed, as we are not allowed to add more/change already existing headers.

2.1 String Length

To begin with, we need a function that will return a string's length. To do so, we will simply iterate over every character until we find a
0, increasing a counter as we go.

```
int str_len(char* str)

Save the first char in str as c
Create the int count, set to 0

While char is not equal to \0
    increment count
    Set char to str[count]
```

2.2 String Duplication

Similarly, we will need to copy a string into another string. To do this, we use str_len, as defined above, to find the length of the original string. We then iterate over every char in the original string, setting the corresponding character in the copy location.

```
void str_copy(char* original, char* copy)

Find the length of original using str_len()

For every value i between 0 and str_len + 1
    set character i in copy to character i in original
```

2.3 String Comparison

To compare two strings, we accept two strings as an argument. We then iterate over every character in these strings. If they are not equal, we return false. Otherwise, we return true.

```
bool str_cmp(char* a, char* b)

Find the length of a using str_len()

For every value i between 0 and str_len + 1
    If character i in a != character i in b
```

```
        Return false

Return true
```

3 Bit Vector

In order to properly implement a bloom filter, I will need to create a bit vector. This will be an array of uint64s, which I will manipulate with bitwise operators to check and see if bits inside those uint64s are set. Doing this will require a number of functions.

All of the bit vectors will be made using a struct. This struct will simply contain an integer, the number of bits it contains, and a pointer to an array of uint64s that makes up the actual vector.

3.1 Creating, Destroying, and getting the length of the Vector

This function will simply accept a uint32 size as the number of bits it wants. It will then create an array of uint64s using calloc, with a length of size/64 + 1. Because we used calloc, all of the bits should already be initialized to 0. We then create a bit vector, and set the array pointer to the array we just created, and set the length to the passed in size.

To destroy a vector, we will first free the array and set the pointer to NULL, before freeing the bit vector and setting the pointer to the bit vector to NULL as well.

Finally, to get the length of a bit vector, we simply return the uint64 that corresponds to the vector's size

3.2 Setting a bit

This function will accept an int i and a pointer to a bit vector, and should then set that bit to 1. Before anything else, we compare i to the length, just to be sure that the i is a valid bit to set. Once this is done, we determine which uint64 the bit is in by doing i/64, saving this as block, before doing i % 64 to determine which bit inside of that uint64 corresponds to i, and saving it as bit. After this, we create a new uint64 j, set it to one, and left shift it by bit position. This should create a number with a 1 at the bit position, allowing us to use it as a bitmask. We then bitwise or the two uint64s to set the bit to 1 if it was not already.

```
void bv_set_bit(BitVector *bv, uint32_t i)

If i is greater than bv's length, return

Set block to i/64
Set bit to i%64

Create int j, set it to 1
Left shift j by bit
Bitwise or element block in bv's array and j
```

3.3 Clearing a Bit

Same as when setting a bit, we are passed a bit number to clear and a bit vector, and we start by finding the block and the bit in that block, saving them as block and bit respectively. From there, we must once again go about creating a bit mask, which should consist of a 1 at every position except the bit position. To do this, we first create a new uint64 j with a value of 1, and left shift it by bit-1. From there we xor j with UINT64_MAX, a uint64 with every value set to 1, which should make j be all ones except for the value at 0. We then bitwise and the element at block in the passed-in bit vector, maintaining all values except for the one at bit, which should be set to 0.

```
void bv_clr_bit(BitVector *bv, uint32_t i)

If i is greater than bv's length, return

Set block to i/64
Set bit to i%64

Create int j, set it to 1
Left shift j by bit
xor j and UINT64_MAX, and save it in j
Bitwise and element block in bv's array and j
```

3.4 Getting a Bit

Finally, we must be able to get a bit. To do this, we start the same way we have every bit vector function, by finding the block and bit in the same way. We then create a duplicate of block at block, before left shifting it over by bit, which should put bit in the first slot. We then bitwise and this with 1, which should result in the bit. If this results in 1, the bit was set, and we return a uint8 containing a 1. Otherwise, we return a uint8 containing a 0.

```
void bv_get_bit(BitVector *bv, uint32_t i)

If i is greater than bv's length, return

Set block to i/64
Set bit to i%64

Create uint8 rtrn, set to 0

Create int j, set it to the value in element block in bv's array
Right shift j by bit
Bitwise and j and 1

If j is 1
```

```
    set rtrn to 1

return rtrn
```

3.5 Printing a Bit Vector

To print the bit vector, I will iterate over all of the uint64s contained in the bit vector, and I will then iterate over every bit in each uint64. To do this, I will create two for loops, one that iterates over the uint64s, and another that iterates over the bits. Inside the first for loop, I will create a copy of the uint64 being iterated over. I then continuously shift the uint64 right by 1 bit and check to see if the resulting number is odd. If it is, then the least significant must be 1, so we print a 1. Otherwise, we print a 0.

```
void bv_print(BitVector *bv)

int count = 0

int internal size = bv's length/64 + 1

For every value i between 0 and internal size
    int tester = the ith element in bv's array
    For every value j between 0 and 64
        if tester is odd, print 1

        Otherwise, print 0

        Shift tester right by 1
        Increase count by 1

        If count equals 8
            Print a single space

    print a newline
```

4 Bloom Filter

A bloom filter is a filter that will be able to very quickly rule out some standard or newspeak words. To do this, we use a bit vector to store which values have been added, and a hash function to determine where these bits should be stored.

Each bloom filter should contain a total of 6 variables. The first will contain the salts used for the hash function, the second the number of keys input into the bloom filter, the third the number of times it found that a bit had been set, the fourth the number of times it found a bit had not been set, and the fifth a count of the total number of bits examined. The 6th and final variable should be a bit vector

4.1 Deleting a Bloom Filter

To delete a bloom filter, we first free the bit vector it contains. We then set the pointer to that vector to NULL, before freeing the pointer pointing to the filter, before setting that to NULL as well.

4.2 Inserting Values into the Bloom Filter

This function will accept a string and a bloom filter pointer, and will set the bit corresponding to a passed in string in the internal bit vector. To insert a corresponding value to that string, we simply iterate over all the salts, and run the provided hash() function using each salt on the string. We then find the size of the bloom filter with the function above, and mod the hash by that value, before using the set bit function defined above to set the corresponding bit in the bit vector to 1. Before the function ends, we must increment the number of keys by 1.

```
void bf_insert(BloomFilter *bf, char *oldspeak)
increment keys in bf by 1

For every value i between 0 and the number of salts
    Set int j to the hash of oldspeak and the ith salt with the provided hash()
    Set j to j mod the size of the bloom filter found with the above function
    Set the jth bit in the bitvector
```

4.3 Bloom Probing

To probe the bloom filter, we effectively perform the same steps as above, iterating over all of the hash functions and calculating j in the same way. Once we have j, however, rather than setting a bit, we instead get a bit. If any of the bits have not been set, we return false. Otherwise, we return true. We also need to increment the hits, misses, and bits examined as we do these things.

```
bool bf_insert(BloomFilter *bf, char *oldspeak)

For every value i between 0 and the number of salts
    Increment bits examined by 1
    Set int j to the hash of oldspeak and the ith salt with the provided hash()
    Set j to j mod the size of the bloom filter found with the above function
    Get the jth bit in the bitvector, and set it to the int bit
    If bit is 0
        Increment misses
        Return false

Increment hits
Return true
```

4.4 Counting the Set Bits in a Bloom Filter

For counting the set bits in a filter, we will mostly be interacting with the bloom filter's internal bit vector. The general idea will be the same as the bit vector print function described above, but instead of printing, we increment a counter when the number is odd.

```
void bf_count(BloomFliter *bf)

int count = 0

For every value i between 0 and bf's bv's length
    int tester = the ith element in bv
    if tester = 1, increase count by 1

return count
```

4.5 Size of a Bloom Filter

For the size, simply return the size of the internal bit vector using the above vector length function.

4.6 Printing a Bloom Filter

To print a broom filter, I simply print out all the internal variables, including the bit vector.

4.7 Bloom Filter Stats

For this function, I am given pointers to variables that correspond to the internal variables inside the bloom filter. I simply set the variables these pointers are pointing to to the values of the interval Bloom Filter variables.

5 Node

To create the doubly linked list, which is needed for my hash table, I first need to create the node that make up these lists.

Each node will contain a work of oldspeak, alongside its newspeak equivalent, provided it has one. If the newspeak string is simply NULL, that is doubleplusungood, as that word is not mere oldspeak, it is instead badspeak. It will also contain a node to the next node, and the previous node.

5.1 Creating a node

When a node is created, we are given two strings, one for oldspeak and another for newspeak, which we need to copy into the corresponding variable. We start by allocating space for the node using malloc. Using the string_copy defined above, we should be able to simply plug in the internal newspeak and oldspeak strings and their passed-in counterparts to str_copy, before returning the pointer of the node.

5.2 Deleting a node

To delete a node, we must make sure that the pointers for the values in front and behind of the node still point to the right place. To do this, we make the node behind the node point forward to the node in front of the node, and visa versa. We then free oldspeak and newspeak, before freeing the node.

```
void node delete(Node **n)

Next node's previous is equal to n's previous
Set previous node's next to n's next

Free oldspeak and newspeak, and set both to NULL
Free node n, then set n to NULL
```

5.3 Printing a node

This will be a very simple function. If oldspeak is NULL, we print nothing. If newspeak is NULL, we print oldspeak. If neither are NULL, we print both.

6 Doubly Linked List

A doubly linked list is a fairly simple construction. Using the nodes created above, it creates a series of connected nodes, with each node connecting to the last. The first and last node should both have NULL set to both their newspeak and oldspeak strings, as they will serve as the head and tail of the list. There will also be two variables, seeks and links, which will track the number of times something is looked up in a linked list and the number of links traversed in that lookup, respectively.

6.1 Creating a Linked List

To create a new linked list, we first create two nodes, the head and the tail, with NULL as both their oldspeak and newspeak strings. We then make the head point forward to the tail, and the tail point backward to the head. We leave the remaining pointers pointing at NULL. Finally, we set the internal mtf variable to the passed in value, and the length to 0.

```
LinkedList *ll_create(bool mtf)

Create the linked list
Create the node head
Create the node tail
Make head's next point to tail
Make tail's previous point to head
```

6.2 Deleting a Linked List

To delete a linked list, we start by setting the current node to whatever head's next is, before using node delete to delete head. So long as the current node's next is not NULL, we get the next node in the list, then delete it with node delete and setting the pointer's value to NULL. Once the current node points to null, we finally delete the tail, before freeing the list and setting the pointer to NULL.

6.3 Returning the Length of a List

To get the length of a linked list, we simply return the internal length variable.

6.4 Moving a Node to the Front

Moving a node to the front is fairly simple. We start by making the provided node's next point back to the provided node's previous, and vice versa. Then, we make the provided node point back to the head, and forward to head's next. Finally, we make head's next point back to the provided node, and head point forward to the provided node.

```
static void ll_mtf(LinkedList *ll, Node *node)

Save node's next as n, and node's previous as p
Set n's previous to p
Set p's next to n

Save ll's head as head.
Set node's next to head's next
Set node's previous to head

Set head's next's previous to node
Set head's next to node
```

6.5 Looking Up a Node in the List

To find a node, we iterate over the whole list, by continuing to find the next node until either the oldspeak in the current node is equal to the provided string (compared with str_cmp, defined above), or the current node's oldspeak is NULL. If we find a node containing the correct oldspeak, we return, otherwise we return NULL. If we find a node and the internal variable mtf is set to true, we then run ll_mtf on that node. We will also have to increase the two external variables mentioned above.

```
void ll_lookup(LinkedList *ll, char *oldspeak)

Increase seeks by 1

Set current node to ll's head
```

```
While next node's oldspeak is not NULL
    Set the current node to current node's next
    Increment links
    Compare oldspeak and current node's oldspeak
    If oldspeak and current node's oldspeak are equal with str_cmp
        If mtf is true
            Use ll_mtf on current node
        return current node]

return NULL
```

6.6 Inserting a New Node

To insert a new node, we first run ll_lookup on the provided oldspeak string. If that does not return NULL, we then create a new node with the provided oldspeak and newspeak strings. We then set the new node to point back to head and forward to head's next. Then, we set head's next to point back to the new node, and head to point forward to the new node.

```
void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)

If ll_lookup(ll, oldspeak) is not NULL, then
    create a new node n
    Set n's next to head's next (from ll)
    Set n's previous to head

    Set head's next's previous to n
    Set head's next to n
```

6.7 Printing the List

To print the list, we first test to see if the linked list is null, printing out nothing if it is. If it is not, we simply walk over the entire linked list, and print every node with node_print(). While this will result in node_print() being run on the head and tail, with how my node print function is set up, nothing will actually be printed as head and tail have a NULL in their oldspeak string.

6.8 Getting Stats

To return the stats, we simply set the respective variable to the corresponding external variables.

7 Hash Table

The hash table is, in theory, a fairly simple function. On creation we give it a size, and it creates an array of pointers to linked list structs that is size long. From here, when passed a string of oldspeak, we run a

hash function on that string to determine where in the array it should be placed. The hash table struct itself will contain the salt used for the hash table, the size of the table, the number of keys added to it, the number of times a lookup command found what it was looking for, the number of times a lookup command didn't find what it was after, and finally, the number of times elements in the linked list was examined.

7.1 Deleting a Hash Table

To delete a hash table, we iterate over all of the linked list pointers in the internal array. If the pointer is not NULL, then we run the linked list destructor on it. Once all the linked lists have been freed and deleted, we then free the array itself and the pointer to the hash table. Finally, we set the passed in pointer to NULL.

7.2 Size of a Hass Table

To get the size of a hash table, we simply return the internal size variable

7.3 Looking Up an Item in the Hash Table

When looking up a hash table, we first hash the inputted string to find the location of the array it should be at. If the linked list in that slot is NULL, we immediately return NULL, and increment misses. Otherwise, we get the linked list stats by running ll_stats, before using ll_lookup, incrementing misses if it returns NULL, and hits if it does not. We then get ll_stats again, and increase the number of time elements in the linked list were examined by the difference. We then return the result of ll_lookup.

```
Node *ht_lookup(HashTable *ht, char *oldspeak)

Set hash value to the hash of oldspeak mod ht's size

Set ll to element hash value of ht's array of linked lists
If ll is NULL
    Increment ht's misses
    return NULL

Set seeks and links using ll_stats
Set node returnable to ll_lookup
Set new seeks and new links with ll_stats
Increase ht's examined by links - new links

If returnable is NULL
    increment misses
else
    increment hits

return returnable
```

7.4 Inserting a New Value

To insert a new value, we first find the hash value of the passed in oldspeak, and find the linked list at that element of the array. If it is currently NULL, we create it with ll_create. We then use ll_insert to add the node to the list.

7.5 Finding the Usage of a Hashtable

To find the number of used linked lists in a hash table, we simply create a new int count, before iterating over all the linked lists in the internal array. If their value is not NULL, we add one to count, before returning count at the end of the loop.

7.6 Printing a Hash Table

To print a hash table, we simply iterate over all of the linked lists, printing each one out. Some of these lists will be null, but due to the way I've coded the linked list print function, a NULL linked list prints out nothing, so I don't need to worry about that.

7.7 Getting the Stats from a Hash Table

For this function, all I need to do is copy the interval variables from the hash table over to the passed in pointers.

8 Parser

We will also need functions to read text out of a file, so for this, we will create a parser. Every parser struct will contain a stream that it reads from, a string of characters read from the file, and an int for where the next word starts. Acceptable characters will be checked by testing to see if isalnum() returns a non 0 value, or if the char is a ', or a -.

8.1 Creating a Parser

To create a parser, we start by allocating the memory for the parser. We then simply set the passed in stream to the internal stream. We then save the first line from the provided file into the line string with fgets(). Finally, we use the function next_accceptable_char to set the line offset to where the first word starts

```
Parser *ht_loopup(FILE f)

Create a Parser

Parser's f = f

line offset = 0

Read the next line of parser's f into parser's current line
```

```
Use next_acceptable_char on Parser
```

8.2 Deleting a Parser

To delete a parser, we should simply free the parser, then set the passed pointer to NULL

8.3 Getting the Next Word from a Parser

To get the next word from the parser, we first set a char equal to the value at that parser's line offset. We then iterate over the characters in the current line until we find the next acceptable charcter. Once we find that character, we continue to iterate over the characters in the current line, writing them to a buffer until we find the next unacceptable character. Now that the whole word has been written, we add a null terminator to our buffer to make it a proper string. End by returning true.

```
bool next_word(Parser *p, char *word)

next char = element line offset in current line

While element line offset in current line is not acceptable or \0
    next char = element line offset in current line
    if next char = \n
        read in the next line from parser's f with fgets
        If fgets returns NULL, return false
        line offset = 0


Set char c to element p's line offset of p's line
uint32 buffer = 0

While c is an acceptable character
    Set element buffer of word to tolower(p's line offset)
    Increment buffer and p's line offset
    Set c to element p's line offset of p's line

Set element buffer pos in word to \0

return true
```

9 Main

Now that all of the necessary functions have been created, we can finally create the filter that will protect my subjects from the dangers of old and badspeak. To do this, we start by using optarg to parse the

command line arguments. We then create a hash table and bloom filter, before filling these with the list of oldspeak, newspeak, and badspeak provided to me. Once this is done, we use the parsing module created above to read in words, checking to see if it has been added to the bloom filter. If it has been, we then check to see if it is in the hash table. If it is, we then check to see if it is badspeak, or only merely oldspeak. If the word used was merely oldspeak, we set an oldspeak boolean to true, and the oldspeak and corresponding newspeak to an oldspeak linked list. If it was badspeak, we set the badspeak boolean to true, and add the badspeak to a badspeak linked list. Once this is done and -s was not specified at the command line, we print out the corresponding message and linked lists based on if badspeak was used, oldspeak was used, or both were used. If -s was specified, we print out the use statistics instead.

```
Set the default values for bloom filter size, hash table size, move to front for
the linked lists, and if statistics should be printed

Use getopt to parse command line inputs, and change the defaults to what the
user specified

If help was specified, print the help option, and return 0

Create a hash table and bloom filter

Open badspeak.txt with fopen

Create a parser p for badspeak, along with a buffer that will hold the words

While next_word(p, buffer)
    Add buffer to the hash table and bloom filter

Open newspeak.txt with fopen
Create a parser q for newspeak, along with a second buffer for newspeak

While next_word(q, buffer) and next_word(q, newspeak_buffer)
    Add buffer and newspeak_buffer to the hash table and bloom filter

Create a parser with stdin
Create a bool used old and used bad, set both to false

Create a linked list old, and a linked list bad

While next word(buffer) of stdin is true
    If probing the bloom filter with buffer returns true
        Set Node node to the return of ht_lookup on buffer
        If node is not NULL
            If node's newspeak is not NULL
                Set old used to true
                Add node's oldspeak and newspeak to old
```

```
            Else
                Set bad used to true
                Add node's oldspeak to bad

If there was an error, print the help option and return 1

If the statistics command line option was specified
    Print the keys, hits, misses, and probes from the hash table
    Print the keys, hits, misses, and bits examined from the bloom filter
    Calculate and print the bits examined per miss, the number of false positives,
    the total load, and the seek length for the bloom filter
Otherwise
    If old used and bad used are true, print out the mixspeak message, the both
    old and bad
    If only old was used, print out the goodspeak message, and only old
    Finally, if only bad was used, print out the badspeak message, and only bad

Free everything, deconstruct the hash table and linked list, and close all the
steams.

return 0
```

10 Writeup Graphs

Finally, I will need to create a number of graphs that will be included in my final writeup. The first graph I create will be an analysis of the number of bloom filter bits examined per miss vs the size of the bloom filter. To do this, I will simply iterate over a large number of bloom filter sizes, printing the size of the bloom filter and getting the number of bits examined, then plot this as a line.

```
for i between 0 and 50000,
    add i and a space to data.dat
    run banhammer -f i on test.txt
    get the bottom 4 lines from above with tail
    get the top line from that with head
    ignore text in that, and add the result to data.dat

plot data.dat with gnuplot
```

To go along with this graph, I will also

The next graph will be a graph of the false positive rate of the bloom filter, ht misses, and ht hits vs the size of the bloom filter. The point of this graph is to show how the number of lookups in a hash table changes as the size of the bloom filter, which I will do by showing how the hits and misses change as the

false positive rates change. The graph will be created in a very similar way as the previous, with the only major differences being that I will be collecting data from different lines.

Finally, I need to graph the number of links examined by the hash table vs the size of the hash table, generated once with mtf turned on, and a second time with mtf turned off. The script will, once again, be almost identical, with banhammer accepting the -t i argument instead of -f i, and the line I collect data from being, once again, different.

While there is another question that I need to answer for the writeup, I should be able to answer it with the above graph, and as such will not need to create another one for this question.