Anexo IV: Desarrollo

```
/* Create a new redBorder Structure */
REDB_STRUCT * redbNew()
{
    REDB_STRUCT * redb;
    init_xlatcase ();
    redb = (REDB_STRUCT *)
        RB_MALLOC(sizeof(REDB_STRUCT), REDB_MEMORY_TYPE__RB_STRUCT);
    RB_MEMASSERT (redb, "redbNew: redb");

    if (redb)
    {
        memset (redb, 0, sizeof (REDB_STRUCT));

        /* Default values */
        redb->numPatterns   = 0;
        redb->numNodes      = 0;
        redb->numChars      = 0;
        redb->numSigQgrams  = 0;
        redb->numQgramCover = 0;
                redb->redbPatterns  = NULL;
        redb->qgbf          = NULL;
        redb->root          = NULL;
#ifdef RB_V4
        redb->noqgbf        = NULL;
#endif
        redb->noqg          = NULL;
#ifdef RB_V3
        int i, j;
        for (i=0; i<SIZE_CHAR; i++)
            for (j=0; j<SIZE_CHAR; j++)
                redb->mn[i][j] = NULL;
#endif
        rbInstances++;
        redb->id = rbInstances;
    }
    return redb;
}
```

Código 1: Función redbNew

```
/* redBorder Structure */
typedef struct {

    int id;
    int numPatterns;
    int numNodes;
    int numChars;
    // number of significative qgrams (greater than 0)
    int numSigQgrams;
    // number of qgrams used to create the structure detection
    int numQgramCover;

    REDB_PATTERN    * redbPatterns;
    ROOT_NODE       * root;
    REDB_QGBF       * qgbf;
    REDB_NOQG       * noqg;
#ifdef RB_V4
    REDB_QGBF       * noqgbf;
#endif
#ifdef RB_V3
    void * mn[SIZE_CHAR][SIZE_CHAR];
#endif

} REDB_STRUCT;
```

Código 2: Estructura REDB_STRUCT

```
/*
*   Add Patterns to REDB_STRUCT
*/
int
redbAddPattern(REDB_STRUCT * redb, unsigned char *pat, int n)
{
    REDB_PATTERN * rbplist;

    rbplist = (REDB_PATTERN *)
        RB_MALLOC(sizeof(REDB_PATTERN), REDB_MEMORY_TYPE__PATTERN);
    RB_MEMASSERT (rbplist, "redbAddPattern: rbplist");

    rbplist->next = redb->redbPatterns;
    redb->redbPatterns = rbplist;

    rbplist->len = n;
    rbplist->patrn = (unsigned char *)
        RB_MALLOC(sizeof(unsigned char)*rbplist->len,
                REDB_MEMORY_TYPE__PATTERN);
    RB_MEMASSERT (rbplist->patrn, "redbAddPattern: rbplist->patrn");

    ConvertCaseEx (rbplist->patrn, pat, n);

    rbplist->cardQgram = 0;

    rbChars += rbplist->len;
    if (rbplist->len > 0 && rbplist->len < 21)
        rbLongPat[(rbplist->len)-1]++;
    else
        rbLongPat[20]++;

    rbPatterns++;
    redb->numPatterns++;

    return 0;
}
```

Código 3: Función redbAddPattern

```
/*
*   Pattern Structure
*/
typedef
struct _redb_pattern
{
    struct  _redb_pattern *next;

    unsigned char   *patrn;
    int             len;            // pattern lenght
    int             cardQgram;      // number of qgrams contained in 'patrn'

} REDB_PATTERN;
```

Código 4: Estructura REDB_PATTERN

```
/*
*   Compile Patterns to create the pattern matching structure
*/
int
redbCompile(REDB_STRUCT * redb)
{
    REDB_PATTERN * rbplist;
    REDB_NOQG * noqg;
    int i;

    numQgrams = (unsigned int)pow(SIZE_CHAR,RB_B);

    qgramCount = (unsigned int *)
        RB_MALLOC(sizeof(unsigned int)*numQgrams, REDB_MEMORY_TYPE__QGRAM);
    RB_MEMASSERT (qgramCount, "redbCompile: qgramCount");

    init_qgrams();

// Compile & Create the redBorder structure based on sigMatch

    // Get the qgrams from each pattern.
    for (rbplist = redb->redbPatterns;
         rbplist != NULL;
         rbplist = rbplist->next)
    {
        // very short signature: length < 2 (no q-gram)
        //   example: "T"
        if (rbplist->len < RB_B)
        {
            #ifdef RB_V4
            if (redb->noqgbf == NULL)
            {
                redb->noqgbf = (REDB_QGBF *)
                    RB_MALLOC(sizeof(REDB_QGBF), REDB_MEMORY_TYPE__NOQG);
                RB_MEMASSERT (redb->noqgbf, "redbCompile: noqgbf");

                redb->noqgbf->numnostr = 1;
            }
            #endif
            noqg = (REDB_NOQG *)
                RB_MALLOC(sizeof(REDB_NOQG), REDB_MEMORY_TYPE__NOQG);
            RB_MEMASSERT (noqg, "redbCompile: noqg");

            noqg->next = redb->noqg;
            redb->noqg = noqg;
            rbNoqg++;

            noqg->signlen = rbplist->len;
            noqg->vssign = (unsigned char *)
                RB_MALLOC(sizeof(unsigned char)*noqg->signlen,
                          REDB_MEMORY_TYPE__NOQG);
            RB_MEMASSERT (noqg->vssign, "redbCompile: noqg->vssign");

            memcpy (noqg->vssign, rbplist->patrn, rbplist->len);
        }
        // regular or short signature: length >= 2
        //   example: either "AB123","signature" or "ABC","ID","VERS","XXYY"
        else
        {
            get_qgrams(rbplist);    // Fill qgramCount list
        }
    }
```

[1/2]

Código 5: Función redbCompile (parte 1/2)

```
    // Sort qgrams by frequency
    for (i=0; i<numQgrams; i++)
        if (qgramCount[i] > 0)
            redb->numSigQgrams++;

    qgramSorted = (unsigned int **)
        RB_MALLOC(sizeof(unsigned int *)*redb->numSigQgrams,
                  REDB_MEMORY_TYPE__QGRAM);
    RB_MEMASSERT (qgramSorted, "redbCompile: qgramSorted");
    for (i=0; i<redb->numSigQgrams; i++)
    {
        qgramSorted[i] = (unsigned int *)
            RB_MALLOC(sizeof(unsigned int)*4, REDB_MEMORY_TYPE__QGRAM);
        RB_MEMASSERT (qgramSorted[i], "redbCompile: qgramSorted[i]");
        qgramSorted[i][0] = qgramSorted[i][1] = 0;
        qgramSorted[i][2] = qgramSorted[i][3] = 0;
    }

    sort_qgramsHi2Lo();


    // Build the qgram-cover
    build_qgramCover(redb);
    reallocate_qgbf(redb);


    // Build redBorder tree (v4)
    redb->root = build_rbtree4(redb);


    // Build bloom filter
    build_bloomFilter(redb);


    // De-allocate memory that is not useful anymore
    for (i=0; i<redb->numSigQgrams; i++)
    {
        RB_FREE(qgramSorted[i], sizeof(int)*4,REDB_MEMORY_TYPE__QGRAM);
    }
    RB_FREE(qgramSorted,
            sizeof(int *)*redb->numSigQgrams,
            REDB_MEMORY_TYPE__QGRAM);

    RB_FREE(qgramCount,
            sizeof(unsigned int)*numQgrams,
            REDB_MEMORY_TYPE__QGRAM);


    return 0; // if all is right
}
```

[2/2]

Código 6: Función redbCompile (parte 2/2)

```
/*
*   Writes in qgramCount the qgram-frequencies in 'qgramCount'
*/
static
void
get_qgrams(REDB_PATTERN * rbplist)
{
    int i, r, dup;
    unsigned int qgCount;

    // regular signature, example: "AB123", "regular signature"
    if (rbplist->len >= (RB_B+RB_BETA))
    {
        for (i=0; i<=(rbplist->len-(RB_B+RB_BETA)); i++)
        {
            qgCount = get_qgram(rbplist, i);
            dup = 0;
            for (r=0; r<i; r++)
                if (qgCount == get_qgram(rbplist, r))
                    dup = 1;
            if (dup == 0)
            {
                qgramCount[qgCount]++;
                rbplist->cardQgram++;
            }
        }
    }
    // short signature, example: "ABC","ID","VERS","XXYY"
    else if (rbplist->len >= RB_B)
    {
        qgCount = get_qgram(rbplist, 0);
        qgramCount[qgCount]++;
        rbplist->cardQgram++;
    }
}
```

Código 7: Función get_qgrams()

```
/*
*   Very Short Signature Structure
*/
typedef
struct _redb_noqg
{
    struct _redb_noqg *next;

    unsigned char   *vssign;
    int             signlen;

} REDB_NOQG;
```

Código 8: Estructura REDB_NOQG

```
/*
*   Lists the qgram-frequencies from highest to lowest in 'qgramSorted'
*/
static
void
sort_qgramsHi2Lo()
{
    int i;
    unsigned int indmax;
    unsigned int maxqg;
    int tam;
    int match;

    tam = 0;
    do{
        maxqg = 0;
        match = 0;
        for (i=0; i<numQgrams; i++)
        {
            if (maxqg < qgramCount[i])
            {
                maxqg = qgramCount[i];
                indmax = i;
                match = 1;
            }
        }
        if (match == 1)
        {
            qgramCount[indmax] = 0;
            qgramSorted[tam][0] = indmax;
            qgramSorted[tam][1] = maxqg;
            tam++;
        }
    }while (match == 1);
}
```

Código 9: Función sort_qgramsHi2Lo()

```
/* Gets the list of all Representative q-grams */
static void build_qgramCover(REDB_STRUCT * redb)
{
    REDB_PATTERN * rbplist;
    int *cardReq;   // it stores the cardinality required of each signature,
                    //   i.e. the RSs which are needed in each signature (=1)
    int i, j, p, qgramFreq, cardTot;
    unsigned int qgram;

    cardReq = (int *)RB_MALLOC(sizeof(int)*redb->numPatterns,
                               REDB_MEMORY_TYPE__QGRAM);
    RB_MEMASSERT (cardReq, "build_qgramCover: cardReq");

    // Building the qgram cover
    cardTot = 0;
    for (rbplist = redb->redbPatterns, p=0; rbplist != NULL;
         rbplist = rbplist->next, p++)
    {
        // Finally RB_GAMMA is useless. Will always be 1 so in this point
        //   the program will go through 'else'.
        if (rbplist->cardQgram < RB_GAMMA)
            cardReq[p] = rbplist->cardQgram;
        else
            cardReq[p] = RB_GAMMA;
        cardTot += cardReq[p];
    }
    for (j=0; j<redb->numSigQgrams; j++)
    {
        qgramSorted[j][2] = 0; qgramSorted[j][3] = 0;
        qgramFreq = qgramSorted[j][1];
        for (rbplist = redb->redbPatterns, p=0; rbplist != NULL;
             rbplist = rbplist->next, p++)
        {
            if (cardReq[p] > 0)
            {
                // regular signature: length >= 2+3
                //   example: "AB123", "signature"
                if (rbplist->len >= (RB_B+RB_BETA))
                {
                    for (i=0; i<=(rbplist->len-(RB_B+RB_BETA)); i++)
                    {
                        if (qgramSorted[j][0] == get_qgram(rbplist, i))
                        {
                            qgramFreq--; qgramSorted[j][2]++;
                            cardReq[p]--; cardTot--;
                        }
                        if (cardReq[p] <= 0) break;
                    }
                }
                // short signature: length >= 2 && < 2+3
                //   example: "ABC","ID","VERS","XXYY"
                else if (rbplist->len >= RB_B)
                {
                    if (qgramSorted[j][0] == get_qgram(rbplist, 0))
                    {
                        qgramFreq--; qgramSorted[j][3]++;
                        cardReq[p]--; cardTot--;
                    }
                }
            }
            if (qgramFreq <= 0)
                break;
        }
        if (cardTot <= 0)
            break;
    }
```

[1/2]

Código 10: Función build_qgramCover() (parte 1/2)

```
    // Refining the qgram cover
    cardTot = 0;
    for (rbplist = redb->redbPatterns, p=0; rbplist != NULL;
         rbplist = rbplist->next, p++)
    {
        // Finally RB_GAMMA is useless. Will always be 1 so in this point
        //   the program will go through 'else'.
        if (rbplist->cardQgram < RB_GAMMA)
            cardReq[p] = rbplist->cardQgram;
        else
            cardReq[p] = RB_GAMMA;
        cardTot += cardReq[p];
    }
    for (j=redb->numSigQgrams-1; j>=0; j--)
    {
        if ((qgramSorted[j][2] + qgramSorted[j][3]) > 0)
        {
            qgramSorted[j][2] = 0; qgramSorted[j][3] = 0;
            qgramFreq = qgramSorted[j][1];
            for (rbplist = redb->redbPatterns, p=0; rbplist != NULL;
                 rbplist = rbplist->next, p++)
            {
                if (cardReq[p] > 0)
                {
                    // regular signature: length >= 2+3
                    //   example: "AB123", "signature"
                    if (rbplist->len >= (RB_B+RB_BETA))
                    {
                        for (i=0; i<=(rbplist->len-(RB_B+RB_BETA)); i++)
                        {
                            if (qgramSorted[j][0] ==
                                    (qgram = get_qgram(rbplist, i)))
                            {
                                qgramFreq--; qgramSorted[j][2]++;
                                cardReq[p]--; cardTot--;
                                add_qgbf(redb, rbplist, i, i+RB_B+RB_BETA, j);
                            }
                            if (cardReq[p] <= 0) break;
                        }
                    }
                    // short signature: length >= 2 && < 2+3
                    //   example: "ABC","ID","VERS","XXYY"
                    else if (rbplist->len >= RB_B)
                    {
                        if (qgramSorted[j][0] ==
                                (qgram = get_qgram(rbplist, 0)))
                        {
                            qgramFreq--; qgramSorted[j][3]++;
                            cardReq[p]--; cardTot--;
                            add_qgbf(redb, rbplist, 0, rbplist->len, j);
                        }
                    }
                }
                if (qgramFreq <= 0) break;
            }
            if (cardTot <= 0)
            {
                for (i=j-1; i>=0; i--)
                {
                    qgramSorted[i][2] = 0;
                    qgramSorted[i][3] = 0;
                }
                break;
            }
        }
    }
   RB_FREE(cardReq, sizeof(int)*redb->numPatterns, REDB_MEMORY_TYPE__QGRAM);
}
```

[2/2]

Código 11: Función build_qgramCover() (parte 2/2)

```
/*
*   Puts into a REDB_QGBF Structure the info relatived to a Representative
*   q-gram found. This function is called whenever a representative q-gram
*   must be added. When a q-gram is added, it may bring with him either a
*   regular string or a short string.
*
*   Now, this q-gram may already exist, so a new REDB_QGBF Structure must
*   not be created, just must add the string to the regular strings list,
*   or to the short strings list. It is also possible that the next string
*   already exist, in this case must not even be added the string.
*
*   A string may not come with the q-gram. In this case, after creating, or
*   not, the REDB_STRUCT, the numnostr variable increases.
*/
static void add_qgbf(REDB_STRUCT * redb, REDB_PATTERN * rbplist,
                     int from, int to, unsigned int ind_qgram)
{
    REDB_QGBF * list; REDB_QGBF * qgbf;
    int exist_qgram, exist_substr, i, j;

    // Check if the REDB_QGBF corresponding to the q-gram here already exist
    exist_qgram = 0;
    for (list = redb->qgbf; list != NULL; list = list->next)
        if (qgramSorted[ind_qgram][0] == list->id)
        {
            qgbf = list;
            exist_qgram = 1; break;
        }
    // If the REDB_QGBF struct does not exist yet then go ahead and create it.
    if (!exist_qgram)
    {
        qgbf = (REDB_QGBF *)RB_MALLOC(sizeof(REDB_QGBF),
                                      REDB_MEMORY_TYPE__QGBF);
        RB_MEMASSERT (qgbf, "add_qgbf: qgbf");
        qgbf->next = redb->qgbf;
        redb->qgbf = qgbf;
        redb->numQgramCover++;
        rbQgrams++;
        qgbf->id = qgramSorted[ind_qgram][0];
        qgbf->ind_qgramSorted = ind_qgram;
        qgbf->patlenmax = qgbf->patlenmin = rbplist->len;
        qgbf->patindex = from;
        qgbf->qgram = (unsigned char *) RB_MALLOC (sizeof(unsigned char)*RB_B,
                                                   REDB_MEMORY_TYPE__QGBF);
        RB_MEMASSERT (qgbf->qgram, "add_qgbf: qgbf->qgram");
        for (j=0; j<RB_B; j++)
            qgbf->qgram[j] = rbplist->patrn[from+j];

        // take into account it's been allocating as memory as times the qgram
        //   is found, later those ones that won't be helpful should be
        //   de-allocated (reallocate_qgbf() function).
        qgbf->bfstr = (unsigned char **)RB_MALLOC(
                      sizeof(unsigned char *)*qgramSorted[qgbf->ind_qgramSorted][1],
                      REDB_MEMORY_TYPE__QGBF);
        RB_MEMASSERT (qgbf->bfstr, "add_qgbf: qgbf->bfstr");
        qgbf->numbfstr = 0;
        qgbf->shortstr = (unsigned char **)RB_MALLOC(
                         sizeof(unsigned char *)*qgramSorted[qgbf->ind_qgramSorted][1],
                         REDB_MEMORY_TYPE__QGBF);
        RB_MEMASSERT (qgbf->shortstr, "add_qgbf: qgbf->shortstr");
        qgbf->lenshortstr = (int *)RB_MALLOC(
                            sizeof(int)*qgramSorted[qgbf->ind_qgramSorted][1],
                            REDB_MEMORY_TYPE__QGBF);
        RB_MEMASSERT (qgbf->lenshortstr, "add_qgbf: qgbf->lenshortstr");
        qgbf->numshortstr = 0;
        qgbf->numnostr = 0;
    }
```

[1/3]

Código 12: Función add_qgbf() (parte 1/3)

```
    // If not, just change some required values
    else
    {
        if (qgbf->patlenmax < rbplist->len) qgbf->patlenmax = rbplist->len;
        if (qgbf->patlenmin > rbplist->len) qgbf->patlenmin = rbplist->len;
        if (qgbf->patindex < from)          qgbf->patindex = from;
    }
    // add bfpatterns (regular strings)
    if ((to-from-RB_B) == RB_BETA && qgbf->numnostr == 0)
    {
        // check if there is the same regular string before
        exist_substr = 0;
        for (i=0; i<qgbf->numbfstr; i++)
        {
            for (j=0; j<RB_BETA; j++)
            {
                if (rbplist->patrn[from+RB_B+j] == qgbf->bfstr[i][j])
                    exist_substr = 1;
                else { exist_substr = 0; break; }
            }
            if (exist_substr) break;
        }
        // If the regular string does not exist yet then create it.
        if (!exist_substr)
        {
            qgbf->bfstr[qgbf->numbfstr] =
                (unsigned char *)RB_MALLOC(sizeof(unsigned char)*RB_BETA,
                                           REDB_MEMORY_TYPE__QGBF);
            RB_MEMASSERT (qgbf->bfstr[qgbf->numbfstr],
                          "add_qgbf: qgbf->bfstr[qgbf->numbfstr]");
            for (j=0; j<RB_BETA; j++)
                qgbf->bfstr[qgbf->numbfstr][j] = rbplist->patrn[from+RB_B+j];
            qgbf->numbfstr++;
            rbQgBfstr++;
        }
    }
    // add shortpatterns (short string)
    else if ((to-from-RB_B) > 0 && qgbf->numnostr == 0)
    {
        // check if there is the same short string before
        exist_substr = 0;
        for (i=0; i<qgbf->numshortstr; i++)
        {
            if (qgbf->lenshortstr[i] == to-from-RB_B)
                for (j=0; j<qgbf->lenshortstr[i]; j++)
                {
                    if (rbplist->patrn[from+RB_B+j] == qgbf->shortstr[i][j])
                        exist_substr = 1;
                    else {    exist_substr = 0; break; }
                }
                if (exist_substr)
                    break;
        }
        // If the short string does not exist yet then create it.
        if (!exist_substr)
        {
            qgbf->lenshortstr[qgbf->numshortstr] = to-from-RB_B;
            qgbf->shortstr[qgbf->numshortstr] = (unsigned char *)RB_MALLOC(
                sizeof(unsigned char)*(qgbf->lenshortstr[qgbf->numshortstr]),
                REDB_MEMORY_TYPE__QGBF);
            RB_MEMASSERT (qgbf->shortstr[qgbf->numshortstr],
                          "add_qgbf: qgbf->shortstr[qgbf->numshortstr]");
            for (j=0; j<(to-from-RB_B); j++)
                qgbf->shortstr[qgbf->numshortstr][j] = rbplist->patrn[from+RB_B+j];
            qgbf->numshortstr++;
            rbQgShstr++;
        }
    }
```


Código 13: Función add_qgbf() (parte 2/3)

```
    else
    {
        qgbf->numnostr++;
        rbQgNostr++;
        if (qgbf->numbfstr)
        {
            for (i=0; i<qgbf->numbfstr; i++)
                RB_FREE(qgbf->bfstr[i],
                        sizeof(unsigned char)*RB_BETA,
                        REDB_MEMORY_TYPE__QGBF);
            RB_FREE(qgbf->bfstr,
                    sizeof(unsigned char *)*qgbf->numbfstr,
                    REDB_MEMORY_TYPE__QGBF);
            qgbf->numbfstr = 0;
        }
        if (qgbf->numshortstr)
        {
            for (i=0; i<qgbf->numshortstr; i++)
                RB_FREE(qgbf->shortstr[i],
                        sizeof(unsigned char)*(qgbf->lenshortstr[i]),
                        REDB_MEMORY_TYPE__QGBF);

            RB_FREE(qgbf->shortstr,
                    sizeof(unsigned char *)*qgbf->numshortstr,
                    REDB_MEMORY_TYPE__QGBF);
            RB_FREE(qgbf->lenshortstr,
                    sizeof(int)*qgbf->numshortstr,
                    REDB_MEMORY_TYPE__QGBF);
            qgbf->numshortstr = 0;
        }
    }
}
```

[3/3]

Código 14: Función add_qgbf() (parte 3/3)

```
/*
*   Q-Gram / Bloom Filter Structure
*/
typedef
struct _redb_qgbf
{
    struct _redb_qgbf *next;

    int             id;
    int             ind_qgramSorted;
    short int       patlenmax;
    short int       patlenmin;
    short int       patindex;
    unsigned char   *qgram;
    int             numbfstr;
    unsigned char   **bfstr;
    int             numshortstr;
    int             *lenshortstr;
    unsigned char   **shortstr;
    int             numnostr;
    int             bfsize;
    int             bfsizeLongSide;
    int             bfsizeShortSide;
    unsigned char   *bfmatrix;

} REDB_QGBF;
```

Código 15: Estructura REDB_QGBF

```
/*
*   Reallocates the memory used by the REDB_QGBF Structures to save memory
*/
static
void
reallocate_qgbf(REDB_STRUCT * redb)
{
    REDB_QGBF * list;

    for (list = redb->qgbf; list != NULL; list = list->next)
    {
        if (list->numbfstr == 0)
        {
            RB_FREE(list->bfstr,
                    sizeof(unsigned char *)*
                        qgramSorted[list->ind_qgramSorted][1],
                    REDB_MEMORY_TYPE__QGBF);
        }
        else if (list->numbfstr < qgramSorted[list->ind_qgramSorted][1])
        {
            list->bfstr = (unsigned char **)
                RB_REALLOC(list->bfstr,
                           sizeof(unsigned char *)*list->numbfstr,      //new
                           sizeof(unsigned char *)*
                               qgramSorted[list->ind_qgramSorted][1],   //old
                           REDB_MEMORY_TYPE__QGBF);
            RB_MEMASSERT (list->bfstr, "reallocate_qgbf: list->bfstr");
        }

        if (list->numshortstr == 0)
        {
            RB_FREE(list->shortstr,
                    sizeof(unsigned char *)*
                        qgramSorted[list->ind_qgramSorted][1],
                    REDB_MEMORY_TYPE__QGBF);
            RB_FREE(list->lenshortstr,
                    sizeof(int)*qgramSorted[list->ind_qgramSorted][1],
                    REDB_MEMORY_TYPE__QGBF);
        }
        else if (list->numshortstr < qgramSorted[list->ind_qgramSorted][1])
        {
            list->shortstr = (unsigned char **)
                RB_REALLOC(list->shortstr,
                           sizeof(unsigned char *)*list->numshortstr,   //new
                           sizeof(unsigned char *)*
                               qgramSorted[list->ind_qgramSorted][1],   //old
                           REDB_MEMORY_TYPE__QGBF);
            RB_MEMASSERT (list->shortstr, "reallocate_qgbf: list->shortstr");

            list->lenshortstr = (int *)
                RB_REALLOC(list->lenshortstr,
                           sizeof(int)*list->numshortstr,               //new
                           sizeof(int)*
                               qgramSorted[list->ind_qgramSorted][1],   //old
                           REDB_MEMORY_TYPE__QGBF);
            RB_MEMASSERT (list->lenshortstr,
                          "reallocate_qgbf: list->lenshortstr");
        }
    }
}
```

Código 16: Función reallocate_qgbf()

```
/*
*   Builds the b-height trie (implementation: 4)
*/
static
ROOT_NODE *
build_rbtree4(REDB_STRUCT * redb)
{
    ROOT_NODE * root;
    REDB_QGBF * qlist;
    REDB_NOQG * nlist;
    int i;
    int depth;      // how deep is going the tree (height)

    if (redb->qgbf != NULL || redb->noqg != NULL)
    {
        rbNodes++;
        root = (ROOT_NODE *) RB_MALLOC (sizeof(ROOT_NODE),
                                        REDB_MEMORY_TYPE__TREE);
        RB_MEMASSERT (root, "build_rbtree4: root");
        // init root node
           for (i=0; i<SIZE_CHAR; i++)
                  root->next[i] = NULL;
        if (root)
            memset (root, 0, sizeof (ROOT_NODE));
    }
    else
        return NULL;

    for (nlist = redb->noqg; nlist != NULL; nlist = nlist->next)
    {
        depth = 0;
        root->next[nlist->vssign[0]] =
            create_nbranch4(root, nlist, &depth, redb->noqgbf, 0);
    }

    for (qlist = redb->qgbf; qlist != NULL; qlist = qlist->next)
    {
        depth = 0;
        if (RB_B == 1)
            root->next[qlist->qgram[0]] = (LEAF_NODE *) qlist;
        else
            root->next[qlist->qgram[0]] =
                create_qbranch4(root, qlist, &depth);
    }

    return root;
}
```

Código 17: Función build_rbtree4()

```
/* Root node */
typedef struct _root_node
{
    LEAF_NODE *next[SIZE_CHAR];
} ROOT_NODE;
```

Código 18: Estructura ROOT_NODE

```
/* Leaf node */
typedef struct _leaf_node
{
    void *next[SIZE_CHAR];
} LEAF_NODE;
```

Código 19: Estructura LEAF_NODE

```
/*
*   Builds a branch of short and/or regular signatures
*/
static
LEAF_NODE *
create_qbranch4(void * node, REDB_QGBF * qgbf, int * depth)
{
    ROOT_NODE * rootNode;
    LEAF_NODE * leafNode; LEAF_NODE * leaf = NULL;
    int i;

    if (*depth == 0)
    {
        rootNode = (ROOT_NODE *) node;
        if (rootNode->next[qgbf->qgram[0]] == NULL)
        {
            rbNodes++;
            leaf = (LEAF_NODE *)RB_MALLOC(sizeof(LEAF_NODE), REDB_MEMORY_TYPE__TREE);
            // init leaf node
            for (i=0; i<SIZE_CHAR; i++)
                leaf->next[i] = NULL;
        }
        else
            leaf = rootNode->next[qgbf->qgram[0]];
    }
    else if (*depth < RB_B)
    {
        leafNode = (LEAF_NODE *) node;
        if (leafNode->next[qgbf->qgram[*depth]] == NULL)
        {
            rbNodes++;
            leaf = (LEAF_NODE *)RB_MALLOC(sizeof(LEAF_NODE), REDB_MEMORY_TYPE__TREE);
            // init leaf node
            for (i=0; i<SIZE_CHAR; i++)
                leaf->next[i] = NULL;
            if (leaf)
                memset (leaf, 0, sizeof (LEAF_NODE));
       }
        else
            leaf = leafNode->next[qgbf->qgram[*depth]];
    }
    (*depth)++;
    if (*depth < RB_B-1)
        leaf->next[qgbf->qgram[*depth]] = create_qbranch4(leaf, qgbf, depth);
    else
        leaf->next[qgbf->qgram[*depth]] = qgbf;

    return leaf;
}
```

Código 20: Función create_qbranch4()

```
/* Builds a branch of very short signatures */
static LEAF_NODE *
create_nbranch4(void * node, REDB_NOQG * noqg, int * depth, REDB_QGBF * noqgbf, int k)
{
    ROOT_NODE * rootNode; LEAF_NODE * leafNode;
    LEAF_NODE * leaf = NULL; int i;

    if (*depth == 0)
    {
        rootNode = (ROOT_NODE *) node;
        if (rootNode->next[noqg->vssign[0]] == NULL)
        {
            rbNodes++;
            leaf = (LEAF_NODE *)RB_MALLOC(sizeof(LEAF_NODE),
                                          REDB_MEMORY_TYPE__TREE);
            // init leaf node
            for (i=0; i<SIZE_CHAR; i++)
                leaf->next[i] = NULL;
        }
        else
            leaf = rootNode->next[noqg->vssign[0]];
    }
    else if (*depth < noqg->signlen)
    {
        leafNode = (LEAF_NODE *) node;
        if (leafNode->next[noqg->vssign[*depth]] == NULL)
        {
            rbNodes++;
            leaf = (LEAF_NODE *)RB_MALLOC(sizeof(LEAF_NODE), REDB_MEMORY_TYPE__TREE);
            // init leaf node
            for (i=0; i<SIZE_CHAR; i++)
                leaf->next[i] = NULL;
        }
        else
            leaf = leafNode->next[noqg->vssign[*depth]];
    }
    // fills the rest of nodes until 'b'
    else if (*depth < RB_B)
    {
        leafNode = (LEAF_NODE *) node;
        if (leafNode->next[k] == NULL)
        {
            rbNodes++;
            leaf = (LEAF_NODE *)RB_MALLOC(sizeof(LEAF_NODE), REDB_MEMORY_TYPE__TREE);
            // init leaf node
            for (i=0; i<SIZE_CHAR; i++)
                leaf->next[i] = NULL;
            if (leaf)
                memset (leaf, 0, sizeof (LEAF_NODE));
        }
        else
            leaf = leafNode->next[k];
    }
    (*depth)++;
    if (*depth < noqg->signlen)
    {
        leaf->next[noqg->vssign[*depth]] =
            create_nbranch4(leaf, noqg, depth, noqgbf, 0);
    }
    // 'RB_B' instead of 'noqg->signlen' because all nodes must be filled until height b
    else if (*depth < RB_B-1)
        for (i=0; i<SIZE_CHAR; i++)
            leaf->next[i] = create_nbranch4(leaf, noqg, depth, noqgbf, i);
    else
        for (i=0; i<SIZE_CHAR; i++)
            leaf->next[i] = noqgbf;

    return leaf;
}
```

Código 21: Función create_nbranch4()

```
/*
*   Builds the bloom filter:
*       first creates the matrix  and then fill it
*/
static
void
build_bloomFilter(REDB_STRUCT * redb)
{
    REDB_QGBF * qlist;

    for (qlist = redb->qgbf; qlist != NULL; qlist = qlist->next)
    {
        if (qlist->numnostr == 0 && qlist->numbfstr > 0)
        {
            bloom_createMatrix(qlist);
            bloom_addStrings(qlist);
        }
        else
            qlist->bfmatrix = NULL;
    }
}
```

Código 22: Función build_bloomFilter()

```
/*
*   Creates Bloom Filter matrix
*/
static
void
bloom_createMatrix(REDB_QGBF * qgbf)
{
    #ifdef RB_BFFACTOR
    int bfFactor = RB_BFFACTOR;

    // minimum value for RB_BFFACTOR
    if (bfFactor < 3)
        bfFactor = 3;

    qgbf->bfsize = (int) pow(2,bfFactor);
    if (bfFactor % 2)
    {
        qgbf->bfsizeLongSide = (int) pow(2,(bfFactor+1)/2);
        qgbf->bfsizeShortSide = qgbf->bfsizeLongSide/2;
    }
    else
        qgbf->bfsizeLongSide = qgbf->bfsizeShortSide = (int)pow(2,bfFactor/2);
    #endif

    #ifdef RB_BFRATIO
    int bfRatio = RB_BFRATIO;

    // minimum value for RB_BFFACTOR
    if (bfRatio < 5)
        bfRatio = 5;

    qgbf->bfsize = qgbf->numbfstr * RB_BFRATIO;

    qgbf->bfsizeLongSide = qgbf->numbfstr;
    qgbf->bfsizeShortSide = RB_BFRATIO;
    #endif

    qgbf->bfmatrix = (unsigned char *)
        RB_MALLOC(sizeof(unsigned char)*qgbf->bfsize/SIZE_BYTE,
                  REDB_MEMORY_TYPE__BLOOMF);
    RB_MEMASSERT (qgbf->bfmatrix, "bloom_createMatrix: qgbf->bfmatrix");
}
```

Código 23: Función bloom_createMatrix()

```
/* Fill the Bloom Filter Matrix */
static void bloom_addStrings(REDB_QGBF * qgbf)
{
    #ifdef RB_HF_RS
    unsigned int hashRS;
    #endif
    #ifdef RB_HF_XOR
    unsigned int hashXOR;
    #endif
    #ifdef RB_HF_SAX
    unsigned int hashSAX;
    #endif
    #ifdef RB_HF_SDBM
    unsigned int hashSDBM;
    #endif
    #ifdef RB_HF_AM
    unsigned int hashAM;
    #endif
    #ifdef RB_HF_AA
    unsigned int hashAA;
    #endif
    int i; int numhashs;
    for (i=0; i<qgbf->numbfstr; i++)
    {
        numhashs = 0;
        #ifdef RB_HF_RS
        hashRS = hashFunction_RS(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashRS/SIZE_BYTE] & (1<<(hashRS % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashRS/SIZE_BYTE] |= (1<<(hashRS % SIZE_BYTE));
        numhashs++; if (numhashs >= RB_NUM_HF) continue;
        #endif
        #ifdef RB_HF_XOR
        hashXOR = hashFunction_XOR(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashXOR/SIZE_BYTE] & (1<<(hashXOR % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashXOR/SIZE_BYTE] |= (1<<(hashXOR % SIZE_BYTE));
        numhashs++; if (numhashs >= RB_NUM_HF) continue;
        #endif
        #ifdef RB_HF_SAX
        hashSAX = hashFunction_SAX(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashSAX/SIZE_BYTE] & (1<<(hashSAX % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashSAX/SIZE_BYTE] |= (1<<(hashSAX % SIZE_BYTE));
        numhashs++; if (numhashs >= RB_NUM_HF) continue;
        #endif
        #ifdef RB_HF_SDBM
        hashSDBM = hashFunction_SDBM(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashSDBM/SIZE_BYTE] & (1<<(hashSDBM % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashSDBM/SIZE_BYTE] |= (1<<(hashSDBM % SIZE_BYTE));
        numhashs++; if (numhashs >= RB_NUM_HF) continue;
        #endif
        #ifdef RB_HF_AM
        hashAM = hashFunction_AM(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashAM/SIZE_BYTE] & (1<<(hashAM % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashAM/SIZE_BYTE] |= (1<<(hashAM % SIZE_BYTE));
        numhashs++; if (numhashs >= RB_NUM_HF) continue;
        #endif
        #ifdef RB_HF_AA
        hashAA = hashFunction_AA(qgbf->bfstr[i]) % qgbf->bfsize;
        if (qgbf->bfmatrix[hashAA/SIZE_BYTE] & (1<<(hashAA % SIZE_BYTE)))
            rbMatrixCollisions[numhashs]++;
        qgbf->bfmatrix[hashAA/SIZE_BYTE] |= (1<<(hashAA % SIZE_BYTE));
        numhashs++;
        #endif
    }
}
```

Código 24: Función bloom_addStrings()

```
/*
**
**   REDB.H
**
**   Version 1.0
**
**   Author: Pablo Cantos Polaino <pablocantos@gmail.com>
**
**          Developed for ENEO Tecnología S.L.
**
*/
[...]
////////////////////////////////////////////////////////////////////////////

/*
*   DEFINES and Typedef's
*/
#define MAX_ALPHABET_SIZE 256

#define RB_B                2   // bytes (allowed values: 1, 2)
#define RB_BETA             3   // bytes (allowed values: ¿1?, 2, 3, 4, ...?)

// Just one of the next two can be enabled
    //#define RB_BFFACTOR   8   // size of bloom filter matrix (2^RB_BFFACTOR)
    #define RB_BFRATIO     90   // size of bloom filter matrix
                                //   (number of bloom strings * RB_BFRATIO)
[...]
// Hash functions: it will be used all hash functions defined
#define RB_NUM_HF           6   // How many hash functions will be used
                                //   (this value must match the number of
                                //   enabled hash functions below)
#define RB_HF_RS                // Robert Sedgwicks Algorithm
#define RB_HF_XOR               // XOR operations
#define RB_HF_SAX               // Scrolling operations
#define RB_HF_SDBM              // used in the open source SDBM project
#define RB_HF_AM                // Addition and Multiplication
#define RB_HF_AA                // Addition and Addition

#define SIZE_CHAR         256
#define SIZE_BYTE           8

/*

RB_V1:
   Fixed b-height trie. If there are very short signatures, with length
    shorter than b, then such signatures must be searched in the NOQG linked
    list.

RB_V2:
   Variable height trie. Thus the very short signatures can be inside the
    trie. This way makes the trie slower because must check in any state if
    it is a final node. So this implementation is not finished.

RB_V3 :
   Search 2-dimensional matrix (b = 2). This implementation is not finished
    either because it turned out slower than trie-based solutions.

RB_V4 :
   Development of the first version. Instead of searching in NOQG linked
    list, the very short signatures will be included in the trie structure,
    but the trie will have a fixed b-height. To achieve this the trie will be
    filled until the end by creating all possible nodes. For this reason it is
    advisable to use a height 2.

*/

#define RB_V4
////////////////////////////////////////////////////////////////////////////
[...]
```

Código 25: Fichero redb.h

```
/*
*   Search candidate packets for sending them to the Search Engine
*    # fourth implementation
*
*     return value:
*       1 -> pass the packet through search engine
*       0 -> discard the packet
*/
int
redbSearch4(REDB_STRUCT * redb, unsigned char * payload, int len, int *w)
{
    LEAF_NODE * leaf;
    REDB_QGBF * qgbf;
    int i, j, k, p;
    int flag_detect = 0;
    unsigned char bfpayload[RB_BETA+1];
    #ifdef RB_HF_XOR
    unsigned int hashXOR = 0;
    #endif
    [...]
    #ifdef RB_HF_AM
    unsigned int hashAM = 0;
    #endif
    unsigned int bi;

    if (redb->root == NULL)
        return 0;

    // covers the full payload
    for (i=0; i<len; i++)
    {
        if (redb->root->next[xlatcase[payload[i]]] != NULL)
        {
            leaf = redb->root->next[xlatcase[payload[i]]];
            if ((leaf->next[(xlatcase[payload[i+1]])]) != NULL)
            {
                qgbf = (REDB_QGBF *) leaf->next[xlatcase[payload[i+1]]];
                if (qgbf->numnostr)
                {
                    (i-qgbf->patindex < 0) ?
                        (w[0]=0) : (w[0]=i-qgbf->patindex);
                    return 1;
                }
```

[1/2]

Código 26: Función redbSearch4 (parte 1/2)

```
                if (qgbf->numbfstr)
                {
                    // passes it through the bloom filter
                    for (j=0, p=i+RB_B; j<RB_BETA; j++, p++)
                        bfpayload[j] = xlatcase[payload[p]];
                    bfpayload[RB_BETA] = '\0';
                    #ifdef RB_HF_XOR
                    hashXOR = 0;
                    #endif
                    [...]
                    #ifdef RB_HF_AM
                    hashAM = 0;
                    #endif
                    for (bi = 0; bi<RB_BETA; bi++)
                    {
                        #ifdef RB_HF_XOR
                        hashXOR ^= *((unsigned int *) (bfpayload+bi));
                        #endif
                        [...]
                        #ifdef RB_HF_AM
                        hashAM += (bi+1) * bfpayload[bi];
                        #endif
                    }

                    if (qgbf->bfmatrix != NULL
                        #ifdef RB_HF_XOR
                        && (qgbf->bfmatrix[(hashXOR%qgbf->bfsize)/SIZE_BYTE]
                            & (1 << ((hashXOR % qgbf->bfsize) % SIZE_BYTE)))
                        #endif
                        [...]
                        #ifdef RB_HF_AM
                        && (qgbf->bfmatrix[(hashAM%qgbf->bfsize)/SIZE_BYTE]
                            & (1 << ((hashAM % qgbf->bfsize) % SIZE_BYTE)))
                        #endif
                        )
                    {
                        (i-qgbf->patindex < 0) ?
                            (w[0]=0) : (w[0]=i-qgbf->patindex);
                        return 1;
                    }
                }
                if (qgbf->numshortstr)
                {
                    for (k=0; k<qgbf->numshortstr; k++)
                    {
                        flag_detect = 1;
                        for (j=0, p=i+RB_B; j<qgbf->lenshortstr[k]; j++, p++)
                            if (xlatcase[payload[p]] != qgbf->shortstr[k][j])
                            {
                                flag_detect = 0;
                                break;
                            }
                        if (flag_detect)
                        {
                            (i-qgbf->patindex < 0) ?
                                (w[0]=0) : (w[0]=i-qgbf->patindex);
                            return 1;
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

[2/2]

Código 27: Función redbSearch4 (parte 2/2)

```
/*
*   Hash Functions
*/

static inline unsigned int hashFunction_RS(unsigned char * str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;
    int i;

    for(i=0; i<RB_BETA; str++, i++)
    {
        hash = hash*a + (*str);
        a = a*b;
    }
    return hash;
}

static inline unsigned int hashFunction_XOR(unsigned char * str)
{
    unsigned int hash = 0;
    int i;

    for (i=0; i<RB_BETA; i++)
        hash ^= *((unsigned int *) (str+i));
    return hash;
}

static inline unsigned int hashFunction_SAX(unsigned char * str)
{
    unsigned int hash = 0;

    while (*str)
        hash ^= (hash<<5)+(hash>>2) + *str++;
    return hash;
}

static inline unsigned int hashFunction_SDBM(unsigned char * str)
{
    unsigned int hash = 0;

    while (*str)
        hash = *str++ + (hash<<6) + (hash<<16) - hash;
    return hash;
}

static inline unsigned int hashFunction_AM(unsigned char * str)
{
    unsigned int hash = 0;
    int j;

    for (j=0; j<RB_BETA; j++)
        hash += (j+1) * str[j];
    return hash;
}

static inline unsigned int hashFunction_AA(unsigned char * str)
{
    unsigned int hash = 0, hashaux = 0;
    int j;

    for (j=0; j<RB_BETA; j++)
    {
        hashaux += str[j];
        hash += hashaux;
    }
    return hash;
}
```

Código 28: Funciones hash