Anexo II: MPSE

```
typedef struct _mpse_struct {
    int    method;
    void * obj;
    int    verbose;
    uint64_t bcnt;
    char   inc_global_counter;
} MPSE;
```

Código 1: Estructura MPSE situada en mpse.c

```
void * mpseNew( int method, int use_global_counter_flag, void (*userfree)(void *p),
                void (*optiontreefree)(void **p), void (*neg_list_free)(void **p))
{
    MPSE * p;
    p = (MPSE*)calloc( 1,sizeof(MPSE) ); if( !p ) return NULL;
    p->method = method;  p->verbose = 0;  p->obj = NULL;
    p->bcnt = 0; p->inc_global_counter = (char)use_global_counter_flag;
    switch( method )
    {
        case MPSE_AC_BNFA:
            p->obj=bnfaNew(userfree, optiontreefree, neg_list_free);
            if(p->obj) ((bnfa_struct_t*)(p->obj))->bnfaMethod = 1;
            break;
        case MPSE_AC_BNFA_Q:
            p->obj=bnfaNew(userfree, optiontreefree, neg_list_free);
            if(p->obj) ((bnfa_struct_t*)(p->obj))->bnfaMethod = 0;
            break;
        case MPSE_AC:
            p->obj = acsmNew(userfree, optiontreefree, neg_list_free);
            break;
        case MPSE_ACF:
            p->obj = acsmNew2(userfree, optiontreefree, neg_list_free);
            if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_FULL  );
            break;
        case MPSE_ACF_Q:
            p->obj = acsmNew2(userfree, optiontreefree, neg_list_free);
            if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_FULLQ  );
            break;
        case MPSE_ACS:
            p->obj = acsmNew2(userfree, optiontreefree, neg_list_free);
            if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_SPARSE  );
            break;
        case MPSE_ACB:
            p->obj = acsmNew2(userfree, optiontreefree, neg_list_free);
            if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_BANDED  );
            break;
        case MPSE_ACSB:
            p->obj = acsmNew2(userfree, optiontreefree, neg_list_free);
            if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_SPARSEBANDS  );
            break;
        case MPSE_LOWMEM:
            p->obj = KTrieNew(0,userfree, optiontreefree, neg_list_free);
            break;
        case MPSE_LOWMEM_Q:
            p->obj = KTrieNew(1,userfree, optiontreefree, neg_list_free);
            break;
#ifdef INTEL_SOFT_CPM
        case MPSE_INTEL_CPM:
            p->obj=IntelPmNew(userfree, optiontreefree, neg_list_free); break;
#endif
        default:
            /* p is free'd below if no case */
            break;
    }
    if( !p->obj ) { free(p); p = NULL; }
    return (void *)p;
}
```

Código 2: Función mpseNew situada en mpse.c

```
/*
*   Aho-Corasick State Machine Struct - one per group of pattterns
*/
typedef struct {

    int acsmMaxStates;
    int acsmNumStates;

    ACSM_PATTERN2    * acsmPatterns;
    acstate_t        * acsmFailState;
    ACSM_PATTERN2   ** acsmMatchList;

    /* list of transitions in each state, this is used to build the nfa & dfa */
    /* after construction we convert to sparse or full format matrix and free */
    /* the transition lists */
    trans_node_t ** acsmTransTable;

    acstate_t ** acsmNextState;
    int          acsmFormat;
    int          acsmSparseMaxRowNodes;
    int          acsmSparseMaxZcnt;

    int          acsmNumTrans;
    int          acsmAlphabetSize;
    int          acsmFSA;
    int          numPatterns;
    void         (*userfree)(void *p);
    void         (*optiontreefree)(void **p);
    void         (*neg_list_free)(void **p);
    PMQ q;
    int sizeofstate;
    int compress_states;

}ACSM_STRUCT2;
```

Código 3: Estructura ACSM_STRUCT2 situada en acsmx2.h

```
/*
*  Create a new AC state machine
*/
ACSM_STRUCT2 * acsmNew2 (void (*userfree)(void *p),
                         void (*optiontreefree)(void **p),
                         void (*neg_list_free)(void **p))
{
  ACSM_STRUCT2 * p;
  init_xlatcase ();
  p = (ACSM_STRUCT2 *)AC_MALLOC(sizeof (ACSM_STRUCT2), ACSM2_MEMORY_TYPE__NONE);
  MEMASSERT (p, "acsmNew");

  if (p)
  {
    memset (p, 0, sizeof (ACSM_STRUCT2));

    /* Some defaults */
    p->acsmFSA               = FSA_DFA;
    p->acsmFormat            = ACF_FULL;//ACF_BANDED;
    p->acsmAlphabetSize      = 256;
    p->acsmSparseMaxRowNodes = 256;
    p->acsmSparseMaxZcnt     = 10;
    p->userfree              = userfree;
    p->optiontreefree        = optiontreefree;
    p->neg_list_free         = neg_list_free;
  }

  return p;
}
```

Código 4: Función acsmNew2 situada en acsmx2.c

```
int  mpseAddPattern ( void * pvoid, void * P, int m,
                      unsigned noCase, unsigned offset, unsigned depth,
                      unsigned negative, void* ID, int IID )
{
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )
   {
     case MPSE_AC_BNFA:
     case MPSE_AC_BNFA_Q:
       return bnfaAddPattern( (bnfa_struct_t*)p->obj, (unsigned char *)P, m,
              noCase, negative, ID );

     case MPSE_AC:
       return acsmAddPattern( (ACSM_STRUCT*)p->obj, (unsigned char *)P, m,
              noCase, offset, depth, negative, ID, IID );

     case MPSE_ACF:
     case MPSE_ACF_Q:
     case MPSE_ACS:
     case MPSE_ACB:
     case MPSE_ACSB:
       return acsmAddPattern2( (ACSM_STRUCT2*)p->obj, (unsigned char *)P, m,
              noCase, offset, depth, negative, ID, IID );

     case MPSE_LOWMEM:
     case MPSE_LOWMEM_Q:
       return KTrieAddPattern( (KTRIE_STRUCT *)p->obj, (unsigned char *)P, m,
                               noCase, negative, ID );
#ifdef INTEL_SOFT_CPM
     case MPSE_INTEL_CPM:
       return IntelPmAddPattern((IntelPm *)p->obj, (unsigned char *)P, m,
               noCase, negative, ID, IID);
#endif
     default:
       return -1;
   }
}
```

Código 5: Función mpseAddPattern situada en mpse.c

```
typedef
struct _acsm_pattern2
{
    struct  _acsm_pattern2 *next;

    unsigned char         *patrn;
    unsigned char         *casepatrn;
    int      n;
    int      nocase;
    int      offset;
    int      depth;
    int      negative;
    void *udata;
    int      iid;
    void   * rule_option_tree;
    void   * neg_list;

} ACSM_PATTERN2;
```

Código 6: Estructura ACSM_PATTERN2 situada en acsmx2.h

```
/*
*   Add a pattern to the list of patterns for this state machine
*
*/
int
acsmAddPattern2 (ACSM_STRUCT2 * p, unsigned char *pat, int n, int nocase,
        int offset, int depth, int negative, void * id, int iid)
{
  ACSM_PATTERN2 * plist;

  plist = (ACSM_PATTERN2 *)
      AC_MALLOC(sizeof (ACSM_PATTERN2), ACSM2_MEMORY_TYPE__PATTERN);
  MEMASSERT (plist, "acsmAddPattern");

  plist->patrn =
      (unsigned char *)AC_MALLOC(n, ACSM2_MEMORY_TYPE__PATTERN);
  MEMASSERT (plist->patrn, "acsmAddPattern");

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

  plist->casepatrn =
      (unsigned char *)AC_MALLOC(n, ACSM2_MEMORY_TYPE__PATTERN);
  MEMASSERT (plist->casepatrn, "acsmAddPattern");

  memcpy (plist->casepatrn, pat, n);

  plist->n      = n;
  plist->nocase = nocase;
  plist->offset = offset;
  plist->depth  = depth;
  plist->negative = negative;
  plist->iid    = iid;
  plist->udata = id;

  plist->next     = p->acsmPatterns;
  p->acsmPatterns = plist;
  p->numPatterns++;

  return 0;
}
```

Código 7: Función acsmAddPattern2 situada en acsmx2.c

```
int  mpsePrepPatterns  ( void * pvoid,
                         int ( *build_tree )(void *id, void **existing_tree),
                         int ( *neg_list_func )(void *id, void **list) )
{
  int retv;
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )
   {
     case MPSE_AC_BNFA:
     case MPSE_AC_BNFA_Q:
       retv = bnfaCompile( (bnfa_struct_t*) p->obj, build_tree, neg_list_func );
     break;

     case MPSE_AC:
       retv = acsmCompile( (ACSM_STRUCT*) p->obj, build_tree, neg_list_func );
     break;

     case MPSE_ACF:
     case MPSE_ACF_Q:
     case MPSE_ACS:
     case MPSE_ACB:
     case MPSE_ACSB:
       retv = acsmCompile2( (ACSM_STRUCT2*) p->obj, build_tree, neg_list_func );
     break;

     case MPSE_LOWMEM:
     case MPSE_LOWMEM_Q:
       return KTrieCompile( (KTRIE_STRUCT *)p->obj, build_tree, neg_list_func );

#ifdef INTEL_SOFT_CPM
     case MPSE_INTEL_CPM:
       return IntelPmFinishGroup((IntelPm *)p->obj, build_tree, neg_list_func);
#endif

     default:
       retv = 1;
     break;
   }

  return retv;
}
```

Código 8: Función mpsePrepPatterns situada en mpse.c

```
/*
*   Compile State Machine - NFA or DFA and Full or Banded or Sparse or SparseBands
*/
int acsmCompile2(ACSM_STRUCT2* acsm, int (*build_tree)(void* id, void** existing_tree),
        int (*neg_list_func)(void* id, void** list))
{
    ACSM_PATTERN2* plist;

    /* Count number of possible states */
    for (plist = acsm->acsmPatterns; plist != NULL; plist = plist->next)
        acsm->acsmMaxStates += plist->n;

    acsm->acsmMaxStates++; /* one extra */

    /* Alloc a List based State Transition table */
    acsm->acsmTransTable =
        (trans_node_t**)AC_MALLOC(sizeof(trans_node_t*) * acsm->acsmMaxStates,
                ACSM2_MEMORY_TYPE__TRANSTABLE);
    MEMASSERT(acsm->acsmTransTable, "acsmCompile");

    /* Alloc a MatchList table -
       this has a lis tof pattern matches for each state, if any */
    acsm->acsmMatchList =
        (ACSM_PATTERN2 **)AC_MALLOC(sizeof(ACSM_PATTERN2*) * acsm->acsmMaxStates,
                ACSM2_MEMORY_TYPE__MATCHLIST);
    MEMASSERT(acsm->acsmMatchList, "acsmCompile");

    /* Initialize state zero as a branch */
    acsm->acsmNumStates = 0;

    /* Add each Pattern to the State Table - This forms a keywords state table  */
    for (plist = acsm->acsmPatterns; plist != NULL; plist = plist->next)
    {
        summary.num_patterns++;
        summary.num_characters += plist->n;
        AddPatternStates(acsm, plist);
    }

    /* Add the 0'th state */
    acsm->acsmNumStates++;

    if (acsm->compress_states)
    {
        if (acsm->acsmNumStates < UINT8_MAX)
        {
            acsm->sizeofstate = 1;
            summary.num_1byte_instances++;
        }
        else if (acsm->acsmNumStates < UINT16_MAX)
        {
            acsm->sizeofstate = 2;
            summary.num_2byte_instances++;
        }
        else
        {
            acsm->sizeofstate = 4;
            summary.num_4byte_instances++;
        }
    }
    else
        acsm->sizeofstate = 4;
```

[1/2]

Código 9: Función acsmCompile2 situada en acsmx2.c (parte 1/2)

```
   /* Alloc a failure table - this has a failure state,
      and a match list for each state */
    acsm->acsmFailState =
        (acstate_t*)AC_MALLOC(sizeof(acstate_t) * acsm->acsmNumStates,
                ACSM2_MEMORY_TYPE__FAILSTATE);
    MEMASSERT(acsm->acsmFailState, "acsmCompile");

    /* Alloc a separate state transition table == in state 's' due to event 'k',
       transition to 'next' state */
    acsm->acsmNextState =
        (acstate_t**)AC_MALLOC_DFA(acsm->acsmNumStates * sizeof(acstate_t*),
                acsm->sizeofstate);
    MEMASSERT(acsm->acsmNextState, "acsmCompile-NextState");

    if ((acsm->acsmFSA == FSA_DFA) || (acsm->acsmFSA == FSA_NFA))
        Build_NFA(acsm);

    if (acsm->acsmFSA == FSA_DFA)
        Convert_NFA_To_DFA(acsm);

    /* Select Final Transition Table Storage Mode */
    if (acsm->acsmFormat == ACF_SPARSE)
    {
        /* Convert DFA Full matrix to a Sparse matrix */
        [...]
    }
    else if (acsm->acsmFormat == ACF_BANDED)
    {
        /* Convert DFA Full matrix to a Sparse matrix */
        [...]
    }
    else if (acsm->acsmFormat == ACF_SPARSEBANDS)
    {
        /* Convert DFA Full matrix to a Sparse matrix */
        [...]
    }
    else if ((acsm->acsmFormat == ACF_FULL) || (acsm->acsmFormat == ACF_FULLQ))
    {
        if (Conv_List_To_Full(acsm))
            return -1;
        /* Don't need the FailState table anymore */
        AC_FREE(acsm->acsmFailState, sizeof(acstate_t) * acsm->acsmNumStates,
                ACSM2_MEMORY_TYPE__FAILSTATE);
        acsm->acsmFailState = NULL;
    }

    /* load boolean match flags into state table */
    acsmUpdateMatchStates(acsm);

    /* Free up the Table Of Transition Lists */
    List_FreeTransTable(acsm);

    /* Accrue Summary State Stats */
    summary.num_states += acsm->acsmNumStates;
    summary.num_transitions += acsm->acsmNumTrans;
    summary.num_instances++;
    memcpy(&summary.acsm, acsm, sizeof(ACSM_STRUCT2));

    if (build_tree && neg_list_func)
        acsmBuildMatchStateTrees2(acsm, build_tree, neg_list_func);

    return 0;
}
```


Código 10: Función acsmCompile2 situada en acsmx2.c (parte 2/2)

```
int mpseSearch(void *pvoid, const unsigned char *T, int n,
               int (*action)(void *id,void *tree,int index,void *data,void *neg_list),
               void *data, int *current_state )
{
  MPSE * p = (MPSE*)pvoid;
  int ret;
  PROFILE_VARS;

  PREPROC_PROFILE_START(mpsePerfStats);

  p->bcnt += n;

  if(p->inc_global_counter)
    s_bcnt += n;

  switch( p->method )
   {
     case MPSE_AC_BNFA:
     case MPSE_AC_BNFA_Q:
      /* return is actually the state */
      ret = bnfaSearch((bnfa_struct_t*) p->obj, (unsigned char *)T, n,
                       action, data, 0 /* start-state */, current_state );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

     case MPSE_AC:
      ret = acsmSearch( (ACSM_STRUCT*) p->obj, (unsigned char *)T, n,
                        action, data, current_state );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

     case MPSE_ACF:
     case MPSE_ACF_Q:
     case MPSE_ACS:
     case MPSE_ACB:
     case MPSE_ACSB:
      ret = acsmSearch2( (ACSM_STRUCT2*) p->obj, (unsigned char *)T, n,
                         action, data, current_state );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

     case MPSE_LOWMEM:
     case MPSE_LOWMEM_Q:
        ret = KTrieSearch( (KTRIE_STRUCT *)p->obj, (unsigned char *)T, n, action, data);
        *current_state = 0;
        PREPROC_PROFILE_END(mpsePerfStats);
        return ret;

#ifdef INTEL_SOFT_CPM
     case MPSE_INTEL_CPM:
        ret = IntelPmSearch((IntelPm *)p->obj, (unsigned char *)T, n, action, data);
        *current_state = 0;
        PREPROC_PROFILE_END(mpsePerfStats);
        return ret;
#endif

     default:
       PREPROC_PROFILE_END(mpsePerfStats);
       return 1;
   }
}
```

Código 11: Función mpseSearch situada en mpse.c

```
/*
*   Search Function
*/
int
acsmSearch2(ACSM_STRUCT2 * acsm, unsigned char *Tx, int n,
           int (*Match)(void * id, void *tree, int index, void *data, void *neg_list),
           void *data, int* current_state )
{

    switch( acsm->acsmFSA )
    {
        case FSA_DFA:

        if( acsm->acsmFormat == ACF_FULL )
        {
            return acsmSearchSparseDFA_Full( acsm, Tx, n, Match, data,
                    current_state );
        }
        else if( acsm->acsmFormat == ACF_FULLQ )
        {
            return acsmSearchSparseDFA_Full_q( acsm, Tx, n, Match, data,
                    current_state );
        }
        else if( acsm->acsmFormat == ACF_BANDED )
        {
            return acsmSearchSparseDFA_Banded( acsm, Tx, n, Match, data,
                    current_state );
        }
        else
        {
            return acsmSearchSparseDFA( acsm, Tx, n, Match, data,
                    current_state );
        }

        case FSA_NFA:

            return acsmSearchSparseNFA( acsm, Tx, n, Match, data,
                    current_state );

        case FSA_TRIE:

            return 0;
    }
    return 0;
}
```

Código 12: Función acsmSearch2 situada en acsmx2.c

```
/*
 *  Matching states are queued, duplicate matches are dropped, and after the complete
 *  buffer scan, the queued matches are processed. This improves cacheing performance,
 *  and reduces duplicate rule processing. The queue is limited in size and is flushed
 *  if it becomes full during the scan. This allows simple insertions. Tracking queue
 *  ops is optional, as this can impose a modest performance hit of a few percent.
 */
#define AC_SEARCH_Q \
    for (; T < Tend; T++) \
    { \
        ps = NextState[state]; \
        sindex = xlatcase[T[0]]; \
        if (ps[1]) \
            if (MatchList[state]) \
                if (_add_queue(&acsm->q,MatchList[state])) \
                    if (_process_queue(&acsm->q, Match,data)) \
                    { \
                        *current_state = state; \
                        return 1; \
                    } \
        state = ps[2 + sindex]; \
    }

static inline int acsmSearchSparseDFA_Full_q(ACSM_STRUCT2 *acsm, unsigned char *T,
      int n, int (*Match)(void *id, void *tree, int index, void *data, void *neg_list),
      void *data, int *current_state)
{
    unsigned char *Tend;
    int sindex;
    acstate_t state;
    ACSM_PATTERN2 **MatchList = acsm->acsmMatchList;

    Tend = T + n;
    if (current_state == NULL) return 0;
    _init_queue(&acsm->q);
    state = *current_state;

    switch (acsm->sizeofstate)
    {
        case 1:
            {
                uint8_t *ps;
                uint8_t **NextState = (uint8_t **)acsm->acsmNextState;
                AC_SEARCH_Q;
            }
            break;
        case 2:
            {
                uint16_t *ps;
                uint16_t **NextState = (uint16_t **)acsm->acsmNextState;
                AC_SEARCH_Q;
            }
            break;
        default:
            {
                acstate_t *ps;
                acstate_t **NextState = acsm->acsmNextState;
                AC_SEARCH_Q;
            }
            break;
    }
    *current_state = state;
    if (MatchList[state])
        _add_queue(&acsm->q,MatchList[state]);
    _process_queue(&acsm->q,Match,data);

    return 0;
}
```

Código 13: Función acsmSearchSparseDFA_Full_q situada en acsmx2.c

```
/* uniquely insert into q, should splay elements for performance */
static
inline
int
_add_queue(PMQ * b, void * p  )

{
    int i;

#ifdef ACSMX2_TRACK_Q
    snort_conf->tot_inq_inserts++;
#endif

    for(i=(int)(b->inq)-1;i>=0;i--)
        if( p == b->q[i] )
            return 0;

#ifdef ACSMX2_TRACK_Q
    snort_conf->tot_inq_uinserts++;
#endif

    if( b->inq < AC_MAX_INQ )
    {
        b->q[ b->inq++ ] = p;
    }

    if( b->inq == AC_MAX_INQ )
    {
#ifdef ACSMX2_TRACK_Q
        b->inq_flush++;
#endif
        return 1;
    }
    return 0;
}

static
inline
unsigned
_process_queue( PMQ * q,
               int (*Match)(void * id, void *tree, int index, void *data, void
*neg_list),
               void *data )
{
    ACSM_PATTERN2 * mlist;
    unsigned int    i;

#ifdef ACSMX2_TRACK_Q
    if( q->inq > snort_conf->max_inq )
        snort_conf->max_inq = q->inq;
    snort_conf->tot_inq_flush += q->inq_flush;
#endif

    for( i=0; i<q->inq; i++ )
    {
        mlist = q->q[i];
        if (mlist)
        {
            if (Match(mlist->udata,mlist->rule_option_tree,0,data,mlist->neg_list) > 0)
            {
                q->inq = 0;
                return 1;
            }
        }
    }
    q->inq=0;
    return 0;
}
```

Código 14: Funciones _add_queue y _process_queue situadas en acsmx2.c