Code Generation & Patching

Gadi Haber

Steps for applying Binary Translated Code

- **Step 0:** Check execution environment
  - Check CPU ID to gather available hardware support
  - Check image permissions and modes
  - Read profile data if available
- **Step 1:** allocate required memory.
  - Requires a calculation of the size of the executable sections and code
- **Step 2:** find candidate routines
  - Analyze to find candidate routines for translation and copy analysis content into appropriate Internal Representation (IR)
- **Step 3:** chaining – resolve target of direct branches and calls in the IR
  - go over the branch target table and check for targets that need to branch to new target addresses
- **Step 4:** fix encoding of translated direct branch and call offsets in the routine:
  - fix all rip-based, direct branch and direct call displacements
  - May require several steps until the displacements of all long and short branches are resolved
  - Need to have an option to rollback when routine cannot be translate
- **Step 5:** write translated routines to new tc
- **Step 6:** Commit the translated functions:
  - Go over the candidate functions and replace the original ones by their new successfully translated ones.

Main Image routine with optimization steps

```
VOID ImageLoad(IMG img, VOID *v)
{
   // Step 0: Check the image and the CPU:
  if (!IMG_IsMainExecutable(img))
   return;

  int rc = 0;

  // step 1: Check size of executable sections and allocate required
memory:
  rc = allocate_and_init_memory(img);
  if (rc < 0)
    return;

  cout << "after memory allocation" << endl;

  // Step 2: go over all routines and identify candidate routines and
copy their code into the instr map IR:
  rc = find_candidate_rtns_for_translation(img);
  if (rc < 0)
    return;

  cout << "after identifying candidate routines" << endl;
```

```
 // Step 3: Chaining - calculate direct branch and call instructions
to point to corresponding target instr entries:
  rc = chain_all_direct_br_and_call_target_entries();
  if (rc < 0 )
    return;

  cout << "after calculate direct br targets" << endl;


  // Step 4: fix rip-based, direct branch and direct call
displacements:
  rc = fix_instructions_displacements();
  if (rc < 0 )
    return;

  cout << "after fix instructions displacements" << endl;

  // Step 5: write translated routines to new tc:
  rc = copy_instrs_to_tc();
  if (rc < 0 )
    return;

  cout << "after write all new instructions to memory tc" << endl;


  // Step 6: Commit the translated routines:
  //Go over the candidate functions and replace the original ones by
their new successfully translated ones:
  commit_translated_routines();

  cout << "after commit translated routines" << endl;

}
```

Data structures – instr_map structure & table

```
// instruction map with an entry for each new instruction:
typedef struct {
    ADDRINT orig_ins_addr;
    ADDRINT new_ins_addr;
    ADDRINT orig_targ_addr;
    bool hasNewTargAddr;
    char encoded_ins[XED_MAX_INSTRUCTION_BYTES];
    xed_category_enum_t category_enum;
    unsigned int size;
    int new_targ_entry;
} instr_map_t;
```

```
instr_map_t *instr_map = NULL;
int num_of_instr_map_entries = 0;
int max_ins_count = 0;
```

Data structures – translated_rtn struct & list

```
// total number of routines in the main executable module:
int max_rtn_count = 0;
```

```
// Tables of all candidate routines to be translated:
typedef struct {
    ADDRINT rtn_addr;
    USIZE rtn_size;
    int instr_map_entry; // negative instr_map_entry means routine does not have a translation.
    bool isSafeForReplacedProbe;
} translated_rtn_t;
```

```
translated_rtn_t *translated_rtn;
int translated_rtn_num = 0;`
```

Data Structures - Translation Cache

```
// tc containing the new code:
char *tc;
int tc_cursor = 0;
```

allocate_and_init_memory()

```
// Calculate size of executable sections and allocate required mem:
for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
{
    if (!SEC_IsExecutable(sec) || SEC_IsWriteable(sec) ||
        !SEC_Address(sec))
      continue;

     if (!lowest_sec_addr || lowest_sec_addr > SEC_Address(sec))
         lowest_sec_addr = SEC_Address(sec);

     if (highest_sec_addr < SEC_Address(sec) + SEC_Size(sec))
         highest_sec_addr = SEC_Address(sec) + SEC_Size(sec);

      // need to avouid using RTN_Open as it is expensive...
      for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn);
             rtn = RTN_Next(rtn)) {
          if (rtn == RTN_Invalid())
             continue;
          max_ins_count += RTN_NumIns  (rtn);
          max_rtn_count++;
        }
}

max_ins_count *= 4; // estimating that the num of instrs of
the inlined functions will not exceed the total nunmber of the
entire code.

// Allocate memory for the instr map needed to fix all branch
targets in translated routines:
instr_map = (instr_map_t *)calloc(max_ins_count,
                                  sizeof(instr_map_t));
if (instr_map == NULL) {
   perror("calloc");
   return -1;
}
```

```
// Allocate memory for the array of candidate routines containing inlineable
function calls:
// Need to estimate size of inlined routines.. ???
translated_rtn = (translated_rtn_t *)calloc(max_rtn_count,
                                            sizeof(translated_rtn_t));
if (translated_rtn == NULL) {
   perror("calloc");
   return -1;
}

// get a page size in the system:
int pagesize = sysconf(_SC_PAGE_SIZE);
if (pagesize == -1) {
   perror("sysconf");
   return -1;
}

ADDRINT text_size = (highest_sec_addr - lowest_sec_addr) * 2 + pagesize * 4;

int tclen = 2 * text_size + pagesize * 4;   // estimated tc size

// Allocate the needed tc with RW+EXEC permissions and is not located in an
address that is more than 32bits afar:
char * addr = (char *) mmap(NULL, tclen,
                            PROT_READ | PROT_WRITE | PROT_EXEC,
                            MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
if ((ADDRINT) addr == 0xffffffffffffffff) {
    cerr << "failed to allocate tc" << endl;
    return -1;
}
tc = (char *)addr;
return 0;
```

find_candidate_rtns_for_translation()

```
int find_candidate_rtns_for_translation(IMG img)
{
    int rc;

   // go over routines and check if they are candidates for translation
and   mark them for translation:

  for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec)) {

     if (!SEC_IsExecutable(sec) || SEC_IsWriteable(sec) ||
        !SEC_Address(sec))
       continue;

     for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn))
     {

        if (rtn == RTN_Invalid()) {
           cerr << "Warning: invalid routine " << RTN_Name(rtn) << endl;
           continue;
        }

       translated_rtn[translated_rtn_num].rtn_addr = RTN_Address(rtn);
       translated_rtn[translated_rtn_num].rtn_size = RTN_Size(rtn);
       translated_rtn[translated_rtn_num].instr_map_entry =
                                  num_of_instr_map_entries;

       translated_rtn[translated_rtn_num].isSafeForReplacedProbe = true;


       // Open the RTN.
       RTN_Open( rtn );
```

```
  for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins)) {

     ADDRINT addr = INS_Address(ins);
     xed_decoded_inst_t xedd;
     xed_error_enum_t xed_code;

     xed_decoded_inst_zero_set_mode(&xedd,&dstate);

     xed_code = xed_decode(&xedd, reinterpret_cast<UINT8*>(addr), max_inst_len);
     if (xed_code != XED_ERROR_NONE) {
        cerr << "ERROR: xed decode failed for instr at: " << "0x" << hex << addr <<
endl;
        translated_rtn[translated_rtn_num].instr_map_entry = -1;
        break;
     }

     // add instr to map table:
     rc = add_new_instr_entry(&xedd, INS_Address(ins), INS_Size(ins));
     if (rc < 0) {
       cerr << "ERROR: failed during instructon translation." << endl;
       translated_rtn[translated_rtn_num].instr_map_entry = -1;
       break;
     }
    } // end for INS...

   // Close the RTN.
   RTN_Close( rtn );

    translated_rtn_num++;

  } // end for RTN..
} // end for SEC...

return 0;
```

add_new_instr_entry()

```
int add_new_instr_entry(xed_decoded_inst_t *xedd, ADDRINT pc, unsigned
int size)
{
  // copy orig instr to tc and fix rip or branch offsets when needed:
  ADDRINT orig_targ_addr = 0;

  if (xed_decoded_inst_get_length (xedd) != size) {
    cerr << "Invalid instruction decoding" << endl;
    return -1;
  }

  xed_uint_t disp_byts =
     xed_decoded_inst_get_branch_displacement_width(xedd);

  xed_int32_t disp;

  if (disp_byts > 0) { // there is a branch offset.
    disp = xed_decoded_inst_get_branch_displacement(xedd);
    orig_targ_addr = pc + xed_decoded_inst_get_length (xedd) + disp;
  }

  // Converts the decoder request to a valid encoder request:
  xed_encoder_request_init_from_decode (xedd);

  unsigned int new_size = 0;

  xed_error_enum_t xed_error = xed_encode (xedd,
reinterpret_cast<UINT8*>(instr_map[num_of_instr_map_entries].encoded_i
ns), max_inst_len , &new_size);
  if (xed_error != XED_ERROR_NONE) {
    cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) <<
endl;
   return -1;
  }
```

```
  // add a new entry in the instr_map:

  instr_map[num_of_instr_map_entries].orig_ins_addr = pc;
  instr_map[num_of_instr_map_entries].new_ins_addr =
              (ADDRINT)&tc[tc_cursor];
   instr_map[num_of_instr_map_entries].orig_targ_addr =
orig_targ_addr;
   instr_map[num_of_instr_map_entries].hasNewTargAddr = false;
   instr_map[num_of_instr_map_entries].new_targ_entry = -1;
   instr_map[num_of_instr_map_entries].size = new_size;
   instr_map[num_of_instr_map_entries].category_enum =
         xed_decoded_inst_get_category(xedd);

  num_of_instr_map_entries++;
  tc_cursor += new_size;

  if (num_of_instr_map_entries >= max_ins_count) {
     cerr << "out of memory for map_instr" << endl;
     return -1;
  }

  return new_size;
}
```

chain_all_direct_br_and_call_target_entries()

```
//fix direct branch and call instructions to point to corresponding target instr entries:

int chain_all_direct_br_and_call_target_entries()
{
  for (int i=0; i < num_of_instr_map_entries; i++) {

    if (instr_map[i].orig_targ_addr == 0)
      continue;

    if (instr_map[i].hasNewTargAddr)
       continue;

    for (int j = 0; j < num_of_instr_map_entries; j++) {

       if (j == i)
         continue;

       if (instr_map[j].orig_ins_addr == instr_map[i].orig_targ_addr) {
            instr_map[i].hasNewTargAddr = true;
            instr_map[i].new_targ_entry = j;
            break;
      }
    }
 }

 return 0;
}
```

fix_instructions_displacements()

```
int fix_instructions_displacements()
{
   // fix displacemnets of direct branch or call instructions:

   int size_diff = 0;

   do {

      size_diff = 0;

      if (KnobVerbose) {
       cerr << "starting a pass of fixing instructions displacements: "
       << endl;
      }

     for (int i=0; i < num_of_instr_map_entries; i++) {

         instr_map[i].new_ins_addr += size_diff;

         int rc = 0;

         // fix rip displacement:
         rc = fix_rip_displacement(i);
         if (rc < 0)
           return -1;

        if (rc > 0) { // this was a rip-based instruction which was fixed.

           if (instr_map[i].size != (unsigned int)rc) {
             size_diff += (rc - instr_map[i].size);
             instr_map[i].size = (unsigned int)rc;
           }

           continue;
        }

      // check if it is a direct branch or a direct call instr:
      if (instr_map[i].orig_targ_addr == 0) {
         continue;  // not a direct branch or a direct call instr.
      }


      // fix instr displacement:
      rc = fix_direct_br_call_displacement(i);
      if (rc < 0)
         return -1;

      if (instr_map[i].size != (unsigned int)rc) {
         size_diff += (rc - instr_map[i].size);
         instr_map[i].size = (unsigned int)rc;
      }

    }  // end int i=0; i ..

   } while (size_diff != 0);

    return 0;
}
```

copy_instrs_to_tc()

```
int copy_instrs_to_tc()
{
 int cursor = 0;

 for (int i=0; i < num_of_instr_map_entries; i++) {

    if ((ADDRINT)&tc[cursor] != instr_map[i].new_ins_addr) {
        cerr << "ERROR: Non-matching instruction addresses: " << hex <<
                (ADDRINT)&tc[cursor] << " vs. " << instr_map[i].new_ins_addr << endl;
        return -1;
    }

    memcpy(&tc[cursor], &instr_map[i].encoded_ins, instr_map[i].size);

    cursor += instr_map[i].size;
  }

  return 0;
}
```

commit_translated_routines()

```
inline void commit_translated_routines()
{
// Commit the translated functions:
// Go over the candidate functions and replace the original ones by their new successfully translated ones:

for (int i=0; i < translated_rtn_num; i++) {

  //replace function by new function in tc

  if (translated_rtn[i].instr_map_entry >= 0) {

    if (translated_rtn[i].rtn_size > MAX_PROBE_JUMP_INSTR_BYTES && translated_rtn[i].isSafeForReplacedProbe) {

      RTN rtn = RTN_FindByAddress(translated_rtn[i].rtn_addr);;

      if (RTN_IsSafeForProbedReplacement(rtn)) {

        AFUNPTR origFptr = RTN_ReplaceProbed(rtn,(AFUNPTR)instr_map[translated_rtn[i].instr_map_entry].new_ins_addr);

        if (origFptr == NULL) {
           cerr << "RTN_ReplaceProbed failed.";
        } else {
          cerr << "RTN_ReplaceProbed succeeded. ";
        }
      }
    }
  }
 }
}
```

Fixing displacement routines

fix_rip_displacement()

```
int fix_rip_displacement(int instr_map_entry)
{

  xed_decoded_inst_t xedd;
  xed_decoded_inst_zero_set_mode(&xedd,&dstate);

  xed_error_enum_t xed_code = xed_decode(&xedd,
     reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins),
     max_inst_len);
  if (xed_code != XED_ERROR_NONE) {
     cerr << "ERROR: xed decode failed for instr at: " << "0x" << hex <<
             instr_map[instr_map_entry].new_ins_addr << endl;
     return -1;
  }

  unsigned int memops = xed_decoded_inst_number_of_memory_operands(&xedd);

  if (instr_map[instr_map_entry].orig_targ_addr != 0)  // a direct jmp or call
instruction.
  return 0;

  //cerr << "Memory Operands" << endl;
  bool isRipBase = false;
  xed_reg_enum_t base_reg = XED_REG_INVALID;
  xed_int64_t disp = 0;
  for(unsigned int i=0; i < memops ; i++)   {

    base_reg = xed_decoded_inst_get_base_reg(&xedd,i);
    disp = xed_decoded_inst_get_memory_displacement(&xedd,i);

    if (base_reg == XED_REG_RIP) {
       isRipBase = true;
       break;
    }
  }

  if (!isRipBase)
    return 0;
```

```
  xed_int64_t new_disp = 0;
  xed_uint_t new_disp_byts = 4;   // set maximal num of bytes for now.

  unsigned int orig_size = xed_decoded_inst_get_length (&xedd);

  // modify rip displacement. Use direct/immediate address without a base reg:
  new_disp = instr_map[instr_map_entry].orig_ins_addr + disp + orig_size;
  xed_encoder_request_set_base0 (&xedd, XED_REG_INVALID);

  //Set the memory displacement using a bit length
  xed_encoder_request_set_memory_displacement (&xedd, new_disp, new_disp_byts);

  unsigned int size = XED_MAX_INSTRUCTION_BYTES;
  unsigned int new_size = 0;

  // Converts the decoder request to a valid encoder request:
  xed_encoder_request_init_from_decode (&xedd);

  xed_error_enum_t xed_error = xed_encode (&xedd,
         reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins),
         size,
         &new_size);
   if (xed_error != XED_ERROR_NONE) {
     cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) << endl;
     dump_instr_map_entry(instr_map_entry);
     return -1;
   }

  return new_size;
}
```

X86 Direct Branch instructions

x86 instruction set:
http://www.felixcloutier.com/x86/

JMP - Direct Unconditional Jump:

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump short, **RIP = RIP + 8-bit displacement** sign extended to **64-bits (RIP is instruction following JMP instruction) displavement) targ addr = orig addr + size of instruction + 8-bit displacement**
E9 *cd*	JMP *rel32*	Jump near, relative, **RIP = RIP + 32-bit displacement** sign extended to 64-bits. **64-bits (RIP is instruction following JMP instruction)**

CALL - Direct (Unconditional) Routine Call:

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, relative**, displacement relative to next instruction.**
E8 *cd*	CALL *rel32*	Call near, relative, **displacement relative to next instruction**. 32-bit displacement sign extended to 64-bits in 64-bit mode.

JCC - Direct Conditional Jump:

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short to **RIP = RIP + 8-bit displacement** if above (CF=0 and ZF=0). **64-bits (RIP is instruction following JA instruction)**
73 *cb*	JAE *rel8*	Jump short to **RIP = RIP + 8-bit displacement** if above or equal (CF=0). **64-bits (RIP is instruction following JAE instruction)**
72 *cb*	JB *rel8*	Jump short to **RIP = RIP + 8-bit displacement** if below (CF=1). **64-bits (RIP is instruction following JB instruction)**
...more,,,		

Fixing unconditional JMP/CALL displacement to branch to *orig_targ_addr*

```
unsigned int ilen = XED_MAX_INSTRUCTION_BYTES;
unsigned int olen = 0;


ADDRINT new_disp = (ADDRINT)&instr_map[instr_map_entry].orig_targ_addr -
                    instr_map[instr_map_entry].new_ins_addr -
                   xed_decoded_inst_get_length (&xedd);

Step 1. xed_error = xed_encode(&enc_req,
                       reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins),
                       ilen, &olen);

Step 2. if (olen != xed_decoded_inst_get_length (&xedd)) {
          xed_error = xed_encode (&enc_req,
                                  reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins),
                                  ilen , &olen);
        }
```

fix_direct_br_call_to_orig_addr()

```
int fix_direct_br_call_to_orig_addr(int instr_map_entry)
{

  xed_decoded_inst_t xedd;
  xed_decoded_inst_zero_set_mode(&xedd,&dstate);

  xed_error_enum_t xed_code = xed_decode(&xedd,
       reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins), max_inst_len);
  if (xed_code != XED_ERROR_NONE) {
     cerr << "ERROR: xed decode failed for instr at: " << "0x" << hex <<
             instr_map[instr_map_entry].new_ins_addr << endl;
     return -1;
  }

  xed_category_enum_t category_enum = xed_decoded_inst_get_category(&xedd);

  if (category_enum != XED_CATEGORY_CALL && category_enum != XED_CATEGORY_UNCOND_BR) {
     cerr << "ERROR: Invalid direct jump from translated code to original code in
rotuine: "
      << RTN_Name(RTN_FindByAddress(instr_map[instr_map_entry].orig_ins_addr)) << endl;
      return -1;
  }

  // check for cases of direct jumps/calls back to the orginal target address:
  if (instr_map[instr_map_entry].new_targ_entry >= 0) {
    cerr << "ERROR: Invalid jump or call instruction" << endl;
    return -1;
  }

  unsigned int ilen = XED_MAX_INSTRUCTION_BYTES;
  unsigned int olen = 0;
  xed_encoder_instruction_t  enc_instr;

   ADDRINT new_disp = (ADDRINT)&instr_map[instr_map_entry].orig_targ_addr -
              instr_map[instr_map_entry].new_ins_addr -
    xed_decoded_inst_get_length (&xedd);

  if (category_enum == XED_CATEGORY_CALL)
    xed_inst1(&enc_instr, dstate, XED_ICLASS_CALL_NEAR, 64,
              xed_mem_bd (XED_REG_RIP, xed_disp(new_disp, 32), 64));

  if (category_enum == XED_CATEGORY_UNCOND_BR)
     xed_inst1(&enc_instr, dstate, XED_ICLASS_JMP, 64,
               xed_mem_bd (XED_REG_RIP, xed_disp(new_disp, 32), 64));
```

```
  xed_encoder_request_t enc_req;

  xed_encoder_request_zero_set_mode(&enc_req, &dstate);
  xed_bool_t convert_ok = xed_convert_to_encoder_request(&enc_req, &enc_instr);
  if (!convert_ok) {
    cerr << "conversion to encode request failed" << endl;
    return -1;
  }

  xed_error_enum_t xed_error = xed_encode(&enc_req,
       reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins), ilen, &olen);
  if (xed_error != XED_ERROR_NONE) {
     cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) << endl;
     return -1;
   }

  // handle the case where the original instr size is different from new encoded instr:
  if (olen != xed_decoded_inst_get_length (&xedd)) {

    new_disp = (ADDRINT)&instr_map[instr_map_entry].orig_targ_addr -
                instr_map[instr_map_entry].new_ins_addr - olen;

    if (category_enum == XED_CATEGORY_CALL)
       xed_inst1(&enc_instr, dstate, XED_ICLASS_CALL_NEAR, 64,
                 xed_mem_bd (XED_REG_RIP, xed_disp(new_disp, 32), 64));

    if (category_enum == XED_CATEGORY_UNCOND_BR)
       xed_inst1(&enc_instr, dstate, XED_ICLASS_JMP, 64,
                 xed_mem_bd (XED_REG_RIP, xed_disp(new_disp, 32), 64));

    xed_encoder_request_zero_set_mode(&enc_req, &dstate);
    xed_bool_t convert_ok = xed_convert_to_encoder_request(&enc_req, &enc_instr);
    if (!convert_ok) {
      cerr << "conversion to encode request failed" << endl;
      return -1;
    }

    xed_error = xed_encode (&enc_req,
          reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins), ilen , &olen);
    if (xed_error != XED_ERROR_NONE) {
       cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) << endl;
       return -1;
    }
}
```

```
instr_map[instr_map_entry].hasNewTargAddr = true;
```

Fixing conditional + unconditional displacement to branch to *new_targ_addr*

```
  xed_int32_t  new_disp = 0;
  unsigned int size = XED_MAX_INSTRUCTION_BYTES;
  unsigned int new_size = 0;

  new_disp = (new_targ_addr - instr_map[instr_map_entry].new_ins_addr) -
             instr_map[instr_map_entry].size; // orig_size;

  xed_uint_t   new_disp_byts = 4;

Step 1.  xed_error_enum_t xed_error = xed_encode (&xedd, enc_buf, max_size , &new_size);

  new_targ_addr = instr_map[instr_map[instr_map_entry].new_targ_entry].new_ins_addr;
  new_disp = new_targ_addr - (instr_map[instr_map_entry].new_ins_addr + new_size);  // this is the correct
displacement.

Step 2. xed_error = xed_encode (&xedd,
                      reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins), size, &new_size);
```

fix_direct_br_call_displacement()

```
int fix_direct_br_call_displacement(int instr_map_entry)
{

  xed_decoded_inst_t xedd;
  xed_decoded_inst_zero_set_mode(&xedd,&dstate);

  xed_error_enum_t xed_code = xed_decode(&xedd,
          reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins),
          max_inst_len);
  if (xed_code != XED_ERROR_NONE) {
     cerr << "ERROR: xed decode failed for instr at: " << "0x" << hex <<
             instr_map[instr_map_entry].new_ins_addr << endl;
     return -1;
  }

  xed_int32_t  new_disp = 0;
  unsigned int size = XED_MAX_INSTRUCTION_BYTES;
  unsigned int new_size = 0;

  xed_category_enum_t category_enum = xed_decoded_inst_get_category(&xedd);

  if (category_enum != XED_CATEGORY_CALL && category_enum != XED_CATEGORY_COND_BR &&
      category_enum != XED_CATEGORY_UNCOND_BR) {
     cerr << "ERROR: unrecognized branch displacement" << endl;
     return -1;
  }

  // fix branches/calls to original targ addresses:
  if (instr_map[instr_map_entry].new_targ_entry < 0) {
     int rc = fix_direct_br_call_to_orig_addr(instr_map_entry);
     return rc;
  }

  ADDRINT new_targ_addr =
        instr_map[instr_map[instr_map_entry].new_targ_entry].new_ins_addr;

  new_disp = (new_targ_addr - instr_map[instr_map_entry].new_ins_addr) -
             instr_map[instr_map_entry].size; // orig_size;

  xed_uint_t   new_disp_byts = 4; // num_of_bytes(new_disp);  ???

  // the max displacement size of loop instructions is 1 byte:
  xed_iclass_enum_t iclass_enum = xed_decoded_inst_get_iclass(&xedd);
  if (iclass_enum == XED_ICLASS_LOOP ||  iclass_enum == XED_ICLASS_LOOPE ||
      iclass_enum == XED_ICLASS_LOOPNE) {
     new_disp_byts = 1;
  }

  // the max displacement size of jecxz instructions is ???:
  xed_iform_enum_t iform_enum = xed_decoded_inst_get_iform_enum (&xedd);
  if (iform_enum == XED_IFORM_JRCXZ_RELBRb){
    new_disp_byts = 1;
  }

  // Converts the decoder request to a valid encoder request:
  xed_encoder_request_init_from_decode (&xedd);

  //Set the branch displacement:
  xed_encoder_request_set_branch_displacement (&xedd, new_disp, new_disp_byts);

  xed_uint8_t enc_buf[XED_MAX_INSTRUCTION_BYTES];
  unsigned int max_size = XED_MAX_INSTRUCTION_BYTES;

  xed_error_enum_t xed_error = xed_encode (&xedd, enc_buf, max_size , &new_size);
  if (xed_error != XED_ERROR_NONE) {
      cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) <<  endl;
      return -1;
   }

  new_targ_addr = instr_map[instr_map[instr_map_entry].new_targ_entry].new_ins_addr;
  new_disp = new_targ_addr - (instr_map[instr_map_entry].new_ins_addr + new_size);  // this
is the correct displacement.

  //Set the branch displacement:
  xed_encoder_request_set_branch_displacement (&xedd, new_disp, new_disp_byts);

  xed_error = xed_encode (&xedd,
         reinterpret_cast<UINT8*>(instr_map[instr_map_entry].encoded_ins), size ,
         &new_size); // &instr_map[i].size
  if (xed_error != XED_ERROR_NONE) {
    cerr << "ENCODE ERROR: " << xed_error_enum_t2str(xed_error) << endl;
    dump_instr_map_entry(instr_map_entry);
    return -1;
  }

  return new_size;
}
```