Umbra – A Tutorial from client_share_detect.c

Qin Zhao, Syed Raza

June 25, 2010

1 Introduction

Umbra is an efficient and scalable memory shadowing tool built on top of DynamoRIO, which is a state-of-the-art runtime code manipulation system. Using the APIs provided by DynamoRIO, Umbra inserts code into the applications runtime instruction stream to perform memory address translation from application memory to shadow memory. Umbra also provides a simple interface that enables developers to create custom shadow memory clients without requiring them to understand Umbra's memory translation details.

This document illustrates how Umbra can be used to develop shadow memory tools via a sample client. The client detects shared data accesses to memory, and reports interesting memory access statistics for a target application.

2 Main Ingredients of an Umbra Client

Umbra clients outline how the shadow memory corresponding to application memory must be updated whenever the latter is accessed/updated. Because of the way Umbra is designed, clients do not need to worry about the details of how application memory is mapped and translated to shadow memory.

Umbra exposes certain event callback hooks, which allow developers to specify how Umbra clients should behave when certain important"events" occur during program execution. Thus, an Umbra client typically implements most or all of the following functions, which are listed in the order in which they are likely to first occur during program execution:

- *umbra_client_init(...)*

  This function is called when the target application starts executing, and can be conceptualized as the *main()* method of an Umbra client. Apart from performing internal initialization of any global (per-process) data it will need, the Umbra client must register its own callback functions corresponding to any later events it is interested in.

- *shadow_memory_create(...)*

  This function is called when shadow memory for the target application is about to be created. This event can be useful for initializing the shadow memory, and for updating any client state.

- *client_thread_init(...)*

  This function is called when a thread in the target application is about to be created. This event can be useful for dynamically allocating and initializing any thread-local storage needed by the client, and for updating any (global) client state.

- *instrument_update(...)*

  This function is called for an instruction that accesses memory (before its execution), so that the client can insert any code to update shadow memory corresponding to the memory location being accessed. The shadow memory will be updated using the inserted code every time the application instruction is executed.

- *client_thread_exit(...)*

  This function is called when a thread in the target application is about to terminate. This event can be useful for dynamically freeing any thread-local storage previously used by the client, and for updating or reporting any client state.

- *shadow_memory_delete(...)*

  This function is called when shadow memory in the target application is about to be freed. This event can be useful for updating or reporting any client state.

- *client_exit(...)*

  This function is called when the target application is about to terminate. It can be used by the client to update and report any information it collected during the program execution.

For each of these functions, Umbra provides information specific to the events via the callback arguments. Details of the implementations of each function, the Umbra-specific data-structures used by client_share_detect.c, along with some stylistic guidelines for Umbra clients in general follow in the next few sections.

3 Client Initialization Event

As mentioned before, each Umbra client *must* define the *umbra_client_init()* function.

Listings 1-2 contain code excerpts relevant to the client initialization in the sample client. In general, the code does the following things: it initializes the client's internal state, it registers callbacks for the events of interest to the client, and it specifies the amount of shadow memory required by the client.

The client's internal state is encapsulated by the *client_proc_data_t* struct defined in Listing 1, lines 1-6. The client tracks the number of threads in the target application's execution, statistics about the memory accesses (shared or not), and has a pointer to a mutex which will be used to synchronize updates to its internal state via concurrent threads. A single instance of this structure is initialized as a global variable after its definition (Listing 1, line 8). This is a nice stylistic feature, which allows a reader to immediately identify the global state maintained by the client.

Listing 1: Client Internal Global State

```
1 typedef struct _client_proc_data_t {
2     int    num_threads;
3     void *lock;
4     reg_t total_mem;
5     ...
6 } client_proc_data_t;
7
8 client_proc_data_t client_proc_data;
```

In the *umbra_client_init()* function, the first thing the client does is to initialize its internal state (Listing 2, line 6-8), which includes creating a mutex using a DynamoRIO function.

Event callbacks of interest to clients and the amount of shadow memory required by the client are specified using the *umbra_client_t* struct. An instance of this structure is available for Umbra clients through a global variable (*proc_info*) defined in Umbra (Listing 2, line 10), and its fields for the respective callbacks can be set to function pointers of client-defined functions (Listing 2, lines 12-16). Other callbacks that are not of interest can be set to NULL function pointers (Listing 2, line 17).

In Umbra the size of the shadow-memory mapping is specified by setting the appropriate fields of the *umbra_client_t* struct. Umbra allows only powers-of-two sizes for shadow-memory sizes (for implementation convenience). In this case, the client specifies that 4 bytes of the application's memory must be mapped to 4 bytes of shadow memory (List 2, lines 19-20).

Finally, the client specifies that Umbra must steal one register during program execution, which is the minimum because one register is used to communicate offsets between shadow and application memory units, as explained later.

Listing 2: Client Initialization

```
1  void
2  umbra_client_init()
3  {
4      umbra_client_t *client;
5
6      client_proc_data.lock  = dr_mutex_create();
7      client_proc_data.total_mem = 0;
8      client_proc_data.num_threads = 0;
9      ...
10     client = &proc_info.client;
11
12     client->thread_init = umbra_client_thread_init;
13     client->thread_exit = umbra_client_thread_exit;
14     client->client_exit        = umbra_client_exit;
15     client->instrument_update = instrument_update;
16     client->shadow_memory_module_destroy = shadow_memory_remove;
17     client->shadow_memory_module_create  = NULL;
18     ...
19     client->app_unit_bits  = 2;
20     client->shd_unit_bits  = 2;    /* 4-byte-2-4-byte mapping */
21     ...
22     client->num_steal_regs = 1;
23 }
```

4 Thread Initialization and Exit Events

As mentioned before, the function *umbra_client_thread_init()* is called whenever a new thread is created in a target application.The sample client creates a unique thread identifier, stores it in some newly allocated thread-local storage, and and updates the global client state's field (*num_threads*) atomically using the global mutex (Listing 4, lines 1-12).

Similarly, whenever a thread is about to terminate in the target application, the function *umbra_client_thread_exit* is called in which the previously allocated thread-local storage is freed (Listing 4, lines 14-20).

Note that the struct *umbra_info_t* is passed as an argument to both of the functions, and its field *client_tls_data* stores a pointer to the thread-local storage for each thread, which is how any other function can access per-thread allocated data. As was the case with the global client state, the thread-local storage is defined in the struct *client_tls_data_t*, which simply stores an unsigned integer corresponding to the thread id of the thread (Listing 3). This again improves the readability of code, and is therefore good stylistic practice.

Listing 3: Thread-Local Storage

```
1  typedef struct _client_tls_data_t {
2      unsigned int tid_map;
3  } client_tls_data_t;
```

Listing 4: Thread Initialization and Exit

```
1  static void
2  umbra_client_thread_init(void *drcontext, umbra_info_t *umbra_info)
3  {
4      client_tls_data_t *tls_data;
5      tls_data = dr_thread_alloc(drcontext, sizeof(client_tls_data_t));
6      umbra_info->client_tls_data = tls_data;
7
8      dr_mutex_lock(client_proc_data.lock);
9      tls_data->tid_map =
10         (1 << ((client_proc_data.num_threads++) % MAX_NUM_THREADS));
11     dr_mutex_unlock(client_proc_data.lock);
12 }
13
14 static void
15 umbra_client_thread_exit(void *drcontext, umbra_info_t *umbra_info)
16 {
17     dr_thread_free(drcontext, umbra_info->client_tls_data,
18                    sizeof(client_tls_data_t));
19     return;
20 }
```

5 Instrumentation for Updating Shadow Memory

The most important callback function in Umbra is *instrument_update(...)*. The arguments to this function include an *umbra_info_t* struct that allows access to thread-local storage, the memory reference for the instruction (type: *mem_ref_t*), the list of instructions in the basic block (type: *instrlist_t*), and the exact instruction of interest that references memory (type: *instr_t*).

The types *instr_t* and *mem_ref_t* are internal types in DynamoRIO and Umbra respectively; their internal implementation details are not available to clients. However, **sizeof** is defined for these structures, so that they can be allocated on the stack by clients (Listing 5, lines 8-10). The same is true for *opnd_t*. All of these types are self-descriptive, and correspond to assembly-level instructions and their operands/memory references.

The *instrument_update()* function in the sample client essentially does the following: it updates the shadow memory corresponding to the application memory accessed to flag that the current thread has also accessed this memory location. This is done through an atomic *or* instruction, which sets a bit corresponding to the current thread in the shadow memory for the memory location. As a performance optimization, before writing to memory, the instrumentation first checks to see if the current thread has already accessed this memory location, in which case an update can be avoided.

Note that because this sample application uses 4-bytes of shadow memory for every 4-bytes of application memory, it can track 32 threads at the same time (thread i has the i^{th} bit set in its thread_id).

Specifically, the client first adds a label before the application-level instruction which is being instrumented (Listing 5, lines 15-16). Then, the client adds an instruction that computes the bit-wise AND of the current thread's id with the shadow memory bitmap of threads that have accessed the same memory location. This instruction (*test [reg], thread_id*) sets conditional flags to store the result; these flags will be read by the next inserted instruction (described later), and then discarded.

Listing 5: Instrumentation Code

```
1  static void
2  instrument_update(void          *drcontext,
3                    umbra_info_t  *umbra_info,
4                    mem_ref_t     *ref,
5                    instrlist_t   *ilist,
6                    instr_t       *where)
7  {
8      instr_t *instr, *label;
9      opnd_t   opnd1, opnd2;
10     reg_id_t reg = umbra_info->steal_regs[0];
11     client_tls_data_t *tls_data;
12
13     tls_data = umbra_info->client_tls_data;
14
15     label = INSTR_CREATE_label(drcontext);
16     instrlist_meta_preinsert(ilist, where, label);
17     ...
18     /* test [%reg].tid_map, tid_map*/
19     opnd1 = opnd_create_base_disp(reg, REG_NULL, 0,
20                                   offsetof(shadow_data_t, tid_map),
21                                   OPSZ_4);
22     opnd2 = OPND_CREATE_INT32(tls_data->tid_map);
23     instr = INSTR_CREATE_test(drcontext, opnd1, opnd2);
24     instrlist_meta_preinsert(ilist, label, instr);
25     instr_set_ok_to_mangle(instr, true);
26     instr_set_translation(instr, ref->pc);
27     instr_set_meta_may_fault(instr, true);
28
29     /* jnz label */
30     opnd1 = opnd_create_instr(label);
31     instr = INSTR_CREATE_jcc(drcontext, OP_jnz, opnd1);
32     instrlist_meta_preinsert(ilist, label, instr);
33
34     /* or */
35     opnd1 = opnd_create_base_disp(reg, REG_NULL, 0,
36                                   offsetof(shadow_data_t, tid_map),
37                                   OPSZ_4);
38     opnd2 = OPND_CREATE_INT32(tls_data->tid_map | 1);
39     instr = INSTR_CREATE_or(drcontext, opnd1, opnd2);
40     LOCK(instr);
41     instrlist_meta_preinsert(ilist, label, instr);
42     instr_set_ok_to_mangle(instr, true);
43     instr_set_translation(instr, ref->pc);
44     instr_set_meta_may_fault(instr, true);
45
46     if (opnd_size_in_bytes(opnd_get_size(ref->opnd)) > 4) {
47         /* Add instrumentation for the next 4 bytes */
48         ...
49     }
50 }
```

To create the *test* instruction and insert it in the application code, the client first creates the operand representing the shadow memory bitmap for the memory location accessed by the current instruction (Listing 5, lines 19-21). The client needs two pieces of information for this operand. First, it needs the address of the shadow memory corresponding to application memory being accessed, which is provided by the one register that the client told Umbra to steal (Listing 2, line 22). The name of this register can be extracted from *umbra_info* struct that was passed in as the argument (Listing 5, line 10). The client also needs a offset into the shadow memory that is being accessed, which is derived from the **offsetof** function defined in ⟨*string.h*⟩. This illustrates another useful stylistic feature, whereby the client first defines a *shadow_data_t* structure with a size equal to the shadow memory size Umbra has been told to allocate for each memory unit (Listing 6). This way, the various sub-fields of the shadow memory for each application memory location can be indexed using the *offsetof(...)* function (Listing 5, line 20).

Listing 6: Shadow Memory Struct (Good Style)

```
1 typedef struct _shadow_data_t {
2     unsigned int tid_map;
3 } shadow_data_t;
```

The client then creates the second operand (Listing 5, line 22), which contains the thread_id of the current thread; this is simply a constant integer derived from the thread-local storage passed in as an argument to the function (Listing 5, line 13). The instruction is then created and inserted into the application code, (Listing 5, lines 23-27). The inserted instruction is flagged to be treated as an application-instruction (Listing 5, line 25) by DynamoRIO for purposes of fault translation, but to be otherwise treated as instrumentation-instruction that is not changed or mangled by DynamoRIO (Listing 5, line 27). This requires its translation pointer to be set (Listing 5, line 26). The translation pointer is used to recreate the application address corresponding to this instruction by DynamoRIO. A good reference for these DynamoRIO instructions is provided in `http://dynamorio.org/docs/`, especially in the "Globals" section.

After inserting the *test* instruction, the client creates a *jnz* instruction that jumps to the original application-level instruction and skips updating the shadow memory if the current thread bit was set to be true in the shadow memory already. The *jnz* instruction reads the flags set by the *test* instruction, so it is inserted after it in the code. (Listing 5, line 29-32).

Finally, the client inserts an instruction that updates the shadow memory corresponding the application memory location being accessed, by setting the bit of the current thread to be true in shadow memory. This is done by storing the logical OR of the previous bitmap in the shadow memory with the current thread_id. This is done in the same way as before, but by creating an *or* instruction (Listing 5, lines 35-44). The operands are created much like before, though the least-significant-bit of the shadow memory is set to 1 by default. Signficantly, note that the instruction is made atomic (Listing 5, line 40), so that multiple threads don't try to update shadow memory concurrently.

Finally, note that an instruction may reference more than 4 bytes of memory, so the client also checks for this case and adds instrumentation to perform the same metadata updates as before for the next 4 bytes (Listing 5, lines 46-49).

Listing 7 shows how the inserted instrumentation instructions are added before the actual application instruction, and how the transformed assembly code may look like.

Listing 7: Inserted Instrumentation Instructions

```
1 test [stolen_reg], thread_id
2 jnz skip_write
3 or [stolen_reg], thread_id => [stolen_reg]
4 skip_write:
5 ;Original Application Instruction Follows
```

6 Shadow Memory Remove Events

Whenever a chunk of application memory is freed, the corresponding shadow memory is removed after a call to the *shadow_memory_remove(...)* function. The argument passed by Umbra to the callback is of the type *memory_map_t* which contains fields that store pointers to the start and end locations of the application block (*app_base* and *app_end* respectively), and pointers to start and end locations of the corresponding shadow memory block (*shd_base* and *shd_end*) respectively.

In the *shadow_memory_remove(...)* function, the sample client simply iterates over the shadow memory being freed, and counts the number of threads that accessed each 4-byte chunk (Listing 8, lines 10-19). It tracks memory that was accessed by only one thread (Listing 8, line 15), and memory that was accessed by multiple threads (Listing 8, line 17), and then reports these statistics for the application memory block (Listing 8, line 24). Finally, it updates memory statistics stored globally in the client's state atomically by first acquiring the DynamoRIO mutex (Listing 8, lines 27-30).

Listing 8: Shadow Memory Remove

```
1  static void
2  shadow_memory_remove(memory_map_t *map)
3  {
4      int i, j;
5      uint *addr;
6
7      i = 0;
8      j = 0;
9
10     for (addr = (uint *)map->shd_base;
11          addr < (uint *)map->shd_end;
12          addr++) {
13         if (*addr != 0) {
14             if (num_of_threads(*addr) == 1)
15                 i++; /* only one thread accessed this location*/
16             else
17                 j++; /* multiple threads accessed this location */
18         }
19     }
20
21     if (i != 0 || j != 0) {
22         i = i * 4;
23         j = j * 4;
24         dr_fprintf(proc_info.log, ...)
25     }
26
27     dr_mutex_lock(client_proc_data.lock);
28     client_proc_data.total_ref += i;
29     ...
30     dr_mutex_unlock(client_proc_data.lock);
31 }
```

7 Program Termination

When the client terminates, it simply reports the statistics it gathered from the previous events in a single print statement and destroys the DynamoRIO mutex it used. The code is omitted for brevity.

8 Miscellaneous

Umbra also uses two other callbacks, *bb_is_interested(...)* and *ref_is_interested(...)* which are used as performance optimizations, allowing a client to skip instrumentation/analysis for basic blocks or memory references that are not of interest to it. The sample client implements these two functions, and in *bb_is_interested(...)*, it returns **false** if a basic block does not access memory, to reduce instrumentation overhead. The *ref_is_interested(...)* callback simply returns **true** for all valid memory references.

Also, the sample client uses a clever, optimized way to count the number of threads that accessed each 4-byte chunk of shadow memory. It avoids run-time overhead of dynamic computation of the number of set bits in each byte, by using a static array with pre-computed results for each possible byte (Listing 9, lines 1-5). As a result the *num_of_threads* function simply indexes into the array to get the precomputed result from each byte , and just adds them together (Listing 9, lines 11-14).

Listing 9: Fast-Count of Number of Threads

```
1  int bit_count[256] = { 0, 1, 1, 2,
2                         1, 2, 2, 3,
3                         1, 2, 2, 3,
4                         2, 3, 3, 4,
5                         ... };
6
7  static int
8  num_of_threads(uint tid_map)
9  {
10     byte *map = (byte *)&tid_map;
11     return (bit_count[map[0]] +
12             bit_count[map[1]] +
13             bit_count[map[2]] +
14             bit_count[map[3]]);
15 }
```