Dynamic Patching on OS X PowerPC, Intel, and Rosetta

Jim Dovey

October 25, 2010

Abstract

Patching (or *overriding*) functions at run-time on Mac OS X has already been described[1]; just recently that implementation has been ported to Intel[2]. However, so far there is no information on this process for the Rosetta dynamic translation environment on the Intel-based Macintosh. This paper will detail error-handling patching routines for PowerPC and Intel, and will describe injection processes for both these architectures. It will also provide details on how to inject/override functions within a PowerPC application running in the Rosetta translation environment.

Contents

1 Native Function Patching/Overriding

1.1 Branch Islands

The patching procedure itself works in the same general manner, except that in this implementation, we use registers 11 & 12 to setup the branch target, and we encode the address into memory, rather than directly into the instructions. We also add another address to use as an error handler. **Example 1** shows the code for this block.

Listing 1: Both branch islands on PowerPC are based on this template

```
static unsigned int branch_template[] = {
// block_addr:
    0x00000000,     // .long    branch_target_addr
    0x00000000,     // .long    error_branch_target
    0x3D600000,     // lis      r11,(msw of block_addr)
    0x616B0000,     // ori      r11,r11,(lsw of block_addr)
    0x818B0000,     // lwz      r12,0(r11)
    0x2C0C0000,     // cmpwi    r12,0
    0x7D8903A6,     // mtctr    r12
    0x60000000,     // nop
    0x60000000,     // nop
    0x4C820420,     // bnectr
    0x818B0004,     // lwz      r12,4(r11)
    0x7D8903A6,     // mtctr    r12
    0x4E800420      // bctr
};
```

The reason we use registers 11 and 12 is to match the behaviour of other dynamic branches such as C++ virtual function calls, and Objective-C message calls, both of which load the address into r12 before moving it to the count register. Since these are considered volatile during function calls (i.e. not likely to be preserved by the called function) their use is acceptable.

In the branch-to-patch island, the first address is that of the target patch function, and the second is the address of the reentry island. The idea being that the patch could be simply disabled by zeroing the target address. At this point, the error handler comes into play and the original function is called instead.

In the reentry-island, the first address is that of the original function's second instruction, and the second is left zero. This causes a noticeable error if the target is ever set to zero as well. Alternatively, the error handler could be the address of a clever function which would be able to unroll the stack, remove the patch, and return execution to the original function. Also in here we replace the two no-op instructions. The first one decrements the value in r12 by four bytes, so a called function can potentially use that address to access other relative functions (well, it *could* happen). The second contains the saved first instruction of the target function, which will execute just prior to our branch.

On Intel, the code works in a similar manner, although two different templates are used, due to the potentially large number of instructions to save in the reentry island (see **Examples 2 & 3**).

Listing 2: The IA-32 branch-to-patch island template.

```
static unsigned char patch_template[] = {
// .template:
    0x00,0x00,0x00,0x00,             // .long branch_target
    0x00,0x00,0x00,0x00,             // .long error_handler
    0xBA,0x00,0x00,0x00,0x00,        // movl .template, %edx
    0x8B,0x02,                       // movl (%edx), %eax
    0x85,0xC0,                       // test %eax, %eax
    0x0F,0x85,0x03,0x00,0x00,0x00,   // jne  .branch
    0x8B,0x42,0x04,                  // movl 4(%edx), %eax
// .branch:
    0xFF,0xE0                        // jmp  *%eax
};
```

Listing 3: The two components of the IA-32 reentry island template.

```
static unsigned char reentry_template_start[] = {
// .template:
    0x00,0x00,0x00,0x00,             // .long branch_target
    0x00,0x00,0x00,0x00,             // .long error_handler
    0xBA,0x00,0x00,0x00,0x00,        // movl L_Template, %edx
    0x8B,0x02,                       // movl (%edx), %eax
    0x85,0xC0,                       // test %eax, %eax
    0x0F,0x85,0x08,0x00,0x00,0x00,   // jne  .call_orig
    0x8B,0x42,0x04,                  // movl 4(%edx), %eax
    0xEB,0x01                        // jmp  .branch
// .call_orig:
};

// ... saved instruction goes in the middle ...

static unsigned char reentry_template_end[]  = {
// .branch:
    0xFF,0xE0                        // jmp  *%eax
};
```

Here we use eax & edx to load the branch target. It's also worth noting that the last byte of `reentry_template_start` contains the number of bytes of saved instructions that follow. This is used in the `RemovePatch` function to copy those bytes back into the original function.

1.2 Atomic Operations

On PowerPC, we have an easy time with regard to rewriting instructions, as all instructions are 32-bits long. We can also implement a compare-and-swap routine to swap around two 32-bit values, as shown in **Example 4**.

Listing 4: PowerPC `CompareAndSwap()` routine.

```
.cas_retry:
    lwarx   r6, 0, r5           // locked load value
    cmpw    r6, r3              // compare with 'oldVal'
    bne-    .cas_fail           // end if not equal
    stwcx.  r4, 0, r5           // try to store 'newVal'
    bne-    .cas_retry          // if store failed, retry
    isync                       // sync instruction cache
    li      r3, 1               // return 1 on success
    blr
.cas_fail:
    li      r3, 0               // return 0 on failure
    blr
```

This will check if the value at the given address is the same as expected, and will write out the new value if this proves to be the case. In the event that the operation cannot be performed atomically, the store instruction will fail, and the code will loop and retry. It returns 1 if the value was written and 0 if the caller should re-read the 'old' value again.

On Intel, the same effect can be gained through the use of the `cmpxchgl` and `cmpxchg8b` instructions. When these are preceded by the lock meta-instruction, they are guaranteed to be completed before the processor switches contexts. The latter 8-byte variant is used on Intel since the branch-absolute instruction (jump to 32-bit inline address) we use is five bytes long. In order to atomically place this value, we pad it out to 8 bytes by reading more from the original function, and we then swap those 8 bytes directly. In the event that we need to copy more than 8 bytes, we have to trust to memcpy() and a little luck; however, I've yet to see an instance where this has ever happened.

1.3 Patch Removal

In the event that patches need to be removed, each architecture has its own removal function. These will start at the target function, follow its branch instruction to the branch-to-patch island, read the address of the re-entry island from there, and will read the saved instructions from that block of code. In the case of the PowerPC patches, this is just a 32-bit value read from a definite offset (see **Example 5**). On Intel, we have to read the one-byte size value first, then read that many bytes (**Example 6**).

Listing 5: PowerPC patch removal algorithm.

```
// see if this is a branch absolute
if ( (instr & 0xFE000003) == 0x4A000002)
```

```
{
    // pull out the address and
    // sign-extend it
    // clear bottom three bits, set top six bits
    pFrom = (unsigned int*) ( (instr & ~3) | 0xFC00000 );

    // read address of low table from here
    pFrom = (unsigned int *) pFrom[1];

    // instruction we're grabbing is at offset 8 (32 bytes)
    restore = pFrom[8];

    // atomic swap
    DPCompareAndSwap( instr, restore, pTo );

    DPCodeSync( fn_addr );
}
```

Listing 6: Intel patch removal algorithm.

```
// If instruction begins with 0xE9 it's a jump
if ( *((unsigned char *) fn_addr) == 0xE9 )
{
    size_t size = 0;
    void * addr = NULL;

    // Read four-byte jump target address
    addr = *((void **)(fn_addr + 1));
    // Deduct 4 bytes, read address of reentry code
    addr = *((void **)(addr - 4);
    // Advance 19 bytes, read one-byte length
    addr += 19;
    size = (size_t) *((unsigned char *)addr);
    // Advance one byte, read saved instruction(s)
    addr++;
    // Copy back into target function
    CopyInstruction( fn_addr, addr, size );

    DPCodeSync( fn_addr );
}
```

2 Code Injection

Injection is implemented here by using a pre-compiled chunk of standard code. The idea behind this implementation is that a number of pre-built 'patch bundles' are to be loaded into a number of applications, usually all that launch, in order to alter functionality on a system-wide level.

This is the same approach taken by the folks at Unsanity (http://www.unsanity.com/) with their Application Enhancer and Haxies.

2.1 Entry point and arguments

In this case, then, injection is merely the means to call code inside the framework, which handles the duties of enumerating and loading these bundles. The injected code reflects that. It includes two functions, an initial function designed to setup a mostly stable operating environment, and which then creates a proper pthread to run the other. These take a large structure as an argument, which contains function pointers and storage, illustrated in **Example 7**.

Listing 7: Injected code parameter block.

```
typedef struct __newthread_args
{
    ___pthread_set_self_fn      setSelfFn;
    ___pthread_create_int_fn    createFakeFn;
    __pthread_create_fn         createPthreadFn;
    __loadimage_fn_ptr          addImageFn;
    __lookup_fn_ptr             lookupFn;
    __lookup_sym_fn_ptr         lookupSymFn;
    __sym_addr_fn_ptr           symAddrFn;
    __thr_term_fn               terminateFn;
    __thr_me_fn                 selfFn;

    void *                      stack_base;

    char fn_name[ 32 ];

    char lib_name[ PATH_MAX ];
    char patch_name[ PATH_MAX ];

    struct _opaque_pthread_t    fakeThread;
    pthread_attr_t              fakeAttrs;

} newthread_args_t;
```

Firstly there are some function pointers. These are all initialized to the addresses of several function within the System library, which loads at a static address (0x90000000), and which loads directly after the program binary; as such, there is very little chance that it would be relocated.

The addresses include those for `__pthread_set_self`, `_pthread_create` (used to wrap a pthread structure around a kernel thread), `pthread_create`, `NSAddImage`, `dlopen`, `NSLookupAndBindSymbol`, `NSAddressOfSymbol`, `thread_terminate`, and `mach_thread_self`. There are slots for both the dlopen-style APIs and the NSLookup-APIs, although the latter are left in purely for compatibility on Mac OS X 10.2, where the former is not available. These are used to load the DynamicPatch framework itself, and to lookup the address of the injection start function within that framework.

Following this comes the stack base address (used by `_pthread_create`), the name of the startup function (there are two options: one which loads all bundles, one which takes a path to a single bundle), and the FQPNs of both the framework and (optionally) the specific patch bundle to load. Lastly are two variables used with `_pthread_create` to get a proper pthread-safe environment in the kernel thread.

The injecting process itself is less involved: it uses the mach kernel routines to allocate the stack in the remote task, copy in the code and the argument block, and create the thread. It creates the thread in a suspended condition, then sets up the thread state such that when it is resumed, it will begin executing the injected code, and will have the addresses of the pthread entry point and the argument block as parameters. This precompiled code then leads into the 'real' functions, which use the CoreFoundation APIs to load the patch bundles, which will then perform their own tasks.

2.2 Patch Bundles

The patch bundles themselves would implement a single function, as shown in **Example 8**.

Listing 8: Patch bundle native entry code.

```
1 int PatchMain( CFBundleRef myBundle )
2 {
3     // do stuff ...
4     return (1);
5 }
```

They can return 1 if they want to stay resident in memory, or optionally can return zero to be unloaded (if they decided not to patch anything, or if they encountered an error that prevented their continuation). Normally, the bundle would call `DPCreatePatch` to patch some functions, but that's not technically necessary: one of the included examples simply puts up an alert dialog.

3 Rosetta Injection/Patching

From the point of view of dynamic overriding and code injection, the Rosetta environment has a couple of interesting properties:

1. It is actually an Intel process. That bears repeating: the threads running via the Mach kernel have i386 thread states.
2. It runs by launching the *translate* application, which loads into high memory and replicates a lot of the system library. It also provides shims for certain of the items within that library, which are (I believe) used to handle system calls and Mach calls from the PowerPC 'threads'.
3. The PowerPC application sees itself as a normal PowerPC application, although its thread state is maintained entirely by the *translate* application for the purposes of 'fooling' the PowerPC code.

4. Overwriting code in a Rosetta process, even one which has already been called and therefore translated, appears to work just as it would natively.
5. The libraries loaded are the PowerPC ones – no Intel-based code is loaded, except that handled directly by the (private) functions within the *translate* application.
6. Upon investigation, if a translated application has n threads, then there will be n+1 Intel threads running. One is the main (translator) app thread, the others each appear to correspond to a real PowerPC thread.

3.1 Problems

This gives us a specific set of problems to investigate:

1. If we inject PowerPC code, how can we set up a thread to run it?
2. If we inject Intel code, how can we affect the PowerPC environment?
3. If we copy in or otherwise attempt to load a bundle or other binary file, what will dyld do with that, and will it be treated as translated or native?
4. Will an injected Mach-O binary file be able to link against system libraries properly?

3.1.1 Injecting PowerPC Code

We can create a thread easily, through the use of the `thread_create` Mach call. We can then use `thread_get_state` and `thread_set_state` to setup the registers for that thread. However, upon closer inspection it becomes clear that the kernel is only creating i386 threads internally, and so handing it a PowerPC thread state results in invalid data, causing the new thread to crash the target application once it is resumed. This is really only to be expected: although it would have been nice if the kernel could have recognised the target as a translated application and forwarded on the call to that process to handle internally, it would make it somewhat difficult for the *translate* application to create its own Intel-native threads. So, we can inject PowerPC code (we could inject anything, after all, it's just data), but we can't use it as the entry point of a remotely-created thread. To create PowerPC threads, we need to call `thread_create` from the context of a PowerPC thread running within the target application already.

3.1.2 Injecting Intel Code to Start a Translated Thread

Given the conditions laid out above, this would be the next-best thing: create a new translated thread by initialising an i386 thread structure at the same entry point used by the other translation threads. This entry point isn't entirely impossible: we can detect where the new thread is starting (essentially the *translate* application's implementation of `_pthread_body`) and point our own thread at that. We can even deduce the caller-supplied pthread start function and perhaps even build something like we have in the native injector, where our Mach thread actually uses `pthread_create` to set up a complete environment for the target. The downside is that the parameter block appears to be quite complex, and takes significant reverse-engineering; and if it is

reverse-engineered, there's nothing to stop it being changed in the next revision, thus breaking our software. Also, the same applies to the thread entry point: it's not a public symbol, nor even a private extern symbol. So again, there's little that can be done without writing a backtracer with some very clever code introspection (counting backwards through variable-length instructions, too—there's a reason why GDB shows the address of the next instruction when it gives a stack backtrace, after all) to automatically work out where this entry point really is in a reliable way.

3.1.3 Copying In a Mach-O File

This in itself seems promising. It's the method used by mach_star, and although I'm generally inclined against copying in code which would include calls to unbound functions and dyld stubs from a module which hasn't been handled by dyld (and therefore isn't in the dyld image table), I'd be willing to try this out here. However, we can't inject PowerPC code without being able to launch a PowerPC translator thread, which means we're confined to the i386 execution environment, which is fairly minimal, and is mostly statically linked to functions implemented within the *translate* application's binary. So, reliance on dyld might not help at all, because it's actually fairly likely that the translation environment uses its own private dyld implementation.

3.1.4 Linking Injected Binaries

Again the transation environment gets in the way: Since we can only inject an i386 thread, we'd need to link against an i386 system. Using dyld stub binding for external functions won't help, since the i386 versions of those functions aren't loaded. In fact, some of the few external symbols in the *translate* application are related to looking up & binding addresses from its shim libraries, indicating that the dyld implementation used by the i386 contexts is actually internal to the *translate* application. So, we can only copy in some entirely self-contained code.

3.2 Solutions

So, it looks like there's no particularly simple way of injecting code that'll do all the work we need to do. However, we can make one important assumption: we inject code so that we can override the target application. And what does patching do? It transfers program control to our code. Aha!

So, the idea now would be that the loading & binding be handled by the target of a branch island. The island itself can be installed easily enough from outside the target process using `vm_write`, and since we can inject Intel code an atomic write of the branch-absolute instruction probably wouldn't be too difficult. So now all we need is something not entirely different from the dyld stub binding helper function, which would be the initial branch target, and which would then load the things that need loading, bind the patch function properly, and pass re-rentry island addresses back into the patch bundle when it loads.

This gives us a rough idea of the overall process:

1. The injector loads the bundle/bundles, and gets a list of patches to install.

2. It adds patches to a list, which contains the location of the patch bundle, the relative address of the patch function, and the reentry island address.
3. It builds a similar list containing the branch absolute instructions and the addresses at which they should be written.
4. It then copies these lists wholesale into the target process, along with the branch islands and a stub helper function.
5. Lastly, it injects a small i386 routine, along with the table from step 3. This will then atomically install the branch instructions.
6. The first time a patch function is called, it goes to the stub, which links everything it needs and changes the branch target address to point to the real patch function.

This can then be broken down into bite-sized pieces.

3.2.1 New Patch Bundle Entry Points

We need to determine what patches need to be installed prior to injecting any code, and we have to decide this from a separate application, quite possibly a native IA-32 one. Therefore, we will need some more entry points into the bundle code to handle the Rosetta case:

- `WillPatch`: A function to query whether any functions in the target app will be patched at all.
- `GetPatches`: A function to request information on all prospective patches.
- `GiveReentry`: A function to provide reentry information back to the bundle once it's loaded in the target process.

Rather than mess about handing linked lists or other such structures into the patch bundle, we will supply a callback routine in the second function, so that the patch bundle can tell us about each patch individually. Also, for better handling of future architectural differences, we should tell the bundle which architecture the target application is using; this way, it can pass that value when calling our cross-architecture symbol lookup routines.

Here, we end up with the prototypes shown in **Example 9**.

Listing 9: Rosetta entry point prototypes.

```
1 typedef void (*__patch_details_cb)(void * target_addr,
2             const char * patch_fn_name, void * info);
3 int WillPatchApplication( CFBundleRef myBundle,
4             pid_t app_pid, const char * app_name );
5 int GetPatchDetails( __patch_details_cb cb,
6              int target_arch, void * info );
```

The last function, to get the reentry addresses, will also use a callback. We don't want to restrict the patch bundle to only having the reentry address for each patch applied as corresponding patch is called: they should all be set up before the patch bundle receives any patch calls whatsoever.

As such, we use a callback function, which will look through the data tables for a target function address, and will return the corresponding reentry address. Also, since this function is to be called when the bundle first loads into the target process, and can therefore be considered a good place for initialisation of other things, we pass the bundle its own executable path, from which it can (if it so chooses) infer its bundle path, and recreate the CFBundleRef it would normally receive in `PatchMain`. These are shown in **Example 10**.

Listing 10: Rosetta linkage functions

```
typedef void * (*patch_lookup_fn_t)(void * patched_fn_addr);
void LinkPatches( patch_lookup_fn_t cb,
            const char * exec_path );
```

3.2.2 Patch Info Data Table

The details necessary to load the patch bundle and both link to the patch handler function within, and to give reentry code back out, involves three things: the reentry address, the path to the bundle executable, and the address of the patch function (relative to the base address of the bundle).

The second item here is problematic; from a pure-structure point of view, we want to be able to iterate over an array of these structures, and use indices to reach each item in the array. Strings complicate the matter, since they are of variable length; the idea of storing `PATH_MAX` bytes of string data (most of which would likely be unused) in the data table is not very useful, especially when we want to keep our memory usage to a minimum. However, here we have an example to follow already: the symbol tables of binary files. So, just like those, we will split our data table into statically-sized and variable-sized segments— namely the patch info table and the string table. The string table will just contain strings, one after another, each with a zero byte as a terminator. Any other structures that would contain variable-length strings will simply contain a four-byte offset into the string table.

This gives us the structure shown in **Example 11**.

Listing 11: Rosetta info table entry structure.

```
struct rosetta_info_table_entry
{
    // address of branch-to-original block
    // (what the normal patch functions would return)
    void * branch_original_code;

    // offset in the string table to the path of the bundle
    // containing code for this patch
    unsigned patch_bundle_path_offset;

    // offset of the patch function within its file image
    unsigned patch_fn_offset;
};
```

3.2.3 Branch Installation Table

The branch installation table is very simple, since each element just contains two 32-bit values, an address and an instruction (see **Example 12**). The instruction needs to be stored in big-endian format, however, while the address needs to remain little-endian.

Listing 12: Branch table element structure.

```
// this holds everything used by the injected routine in
// a rosetta application: what to write, and where.
// Everything else has been done already
// by the time this is used.
typedef struct _patch_entry_struct
{
    vm_address_t    fn_addr;
    natural_t       ba_instr;
} patch_entry_t;
```

3.2.4 Copying Into Target Process

The copy part comes once everything else has been done. Once we've enumerated all the patch bundles, we will have built the jump tables and the data tables locally, and we will also have allocated the space for these inside the target process (the branch islands need to know one another's addresses, as do the patch info table entries). At this point, we can simply copy the blocks, one by one, into the target process. We then make them executable using `vm_protect`.

However, we need a little more information than this alone. We need the assembly stub function itself, and we also need book-keeping information, such as the locations of the various tables, which we will need to deallocate later. Because of this, the patch data table begins with a header structure, which contains not only these addresses and their corresponding sizes, but also the offsets used to reach the patch info table and the string table, storage for the C-style name of the stub binding function proper (passed to `dlsym`), and the stub helper code itself. This is illustrated in **Example 13**.

Listing 13: Rosetta data table header structure.

```
struct rosetta_data_table_header
{
    // length of data table (will be multiple of page-size)
    vm_size_t data_table_size;

    // addresses of jump tables, so they can be deallocated
    // note that this *still* won't happen if no patched functions are
    // called
    vm_address_t low_jump_table;
    vm_size_t low_jump_table_size;
    vm_address_t high_jump_table;
```

```
12      vm_size_t high_jump_table_size;
13
14      // patch info table offset/count:
15      unsigned info_table_offset;     // offset from data table start
16      unsigned info_table_count;      // number of items in info table
17
18      // string table - a block of data; items in info table contain
19      // offsets from the start of the string table. Here we have an
20      // offset to the start of the string table, relative to the start
21      // of the data table
22      unsigned string_table_offset;
23
24      // one string gets coded in explicitly: its address is needed by
25      // the stub helper code, and must be 'compiled' in
26      // not that the size is a multiple of 4 to keep alignment. Padding
27      // bytes don't matter, so long as the string itself is
28      // null-terminated.
29      char bind_fn_sym[24];           // "__rosetta_bind_helper"
30
31      // at this point, we place the rosetta stub helper code, which
32      // includes some static variables:
33      // unsigned fmwk_ok;
34      // const char * fmwk_path;      // zero, filled at runtime
35      // const char * bind_fn_sym;    // addr of var in this header
36      // void * load_fn;              // address of NSAddImage
37      // void * sym_fn;               // address of dlsym
38      // void * bind_fn;              // zero, filled at runtime
39      // void * table_addr;           // address of this header
40      unsigned char stub_helper_interface[1]; // actually larger
41  };
```

At the start of the stub helper interface code there is another block of variables, including precomputed function addresses. These are both implemented in `libSystem`, which should load at the same address in memory within every application. These variables are all used directly by the stub helper code, which is why they're part of the assembly block there.

The stub helper code does have a couple of instructions into which we have to place an address, however. This is the address of the `stub_helper_interface` variable above, the start of its own block. It uses this to access those local variables.

The layout of this whole block in memory is shown in **Table 1**.

Page One			Page Two
HeaderStruct	Stub Code	Patch Into Table	String Table

Table 1: Rosetta data table memory block

3.2.5 Injection and Overriding

The next part uses a standard i386 thread injection process to apply a small block of code which will loop through its parameters: the array from step 3, which are also copied in. Remember that the branch instructions were stored in that array in big-endian format, so this thread needs only to copy things.

The C version of the code is shown in **Example 14**.

Listing 14: Injected code C implementation

```
1  struct _r_args
2  {
3      unsigned int *  addr;
4      unsigned int    valu;
5  };
6
7  void RosettaPatchInstaller( struct _r_args *args,
8                              unsigned int count,
9                              void * flag_byte_addr )
10 {
11     unsigned int i;
12     for ( i = 0; i < count; i++ )
13     {
14         // value is big-endian already, addr is native
15         *(args[i].addr) = args[i].valu;
16         _mm_clflush( (void *)(args[i].addr) );
17     }
18
19     *((unsigned char *) flag_byte_addr) = 0xFF;
20     while ( 1 );
21 }
```

It takes three arguments, the last of which is used to signal completion to the injector process. The assembler code puts a one byte of zero at the start of the code block, and it is this that gets set to 0xFF at the end of the function. The reason for this is that the Intel implementation of the thread_terminate() function within *translate* is not listed in the symbol table, so we point this code towards it to stop itself. As such, the injector will watch the code block, and when the first byte is set nonzero it will terminate the thread remotely.

3.2.6 Patch Binding at First Call

As with the Intel patch code, there are two different templates for the branch islands under the Rosetta injection process. The reentry island is functionally identical to the standard one, with the exception that the modification of r12 is now explicitly coded, since it will always be used. The patch island is a little different since it needs to both store an extra variable at its head and needs to pull data into a couple more registers, ready to pass onto the stub handler interface (see

Example 15). Note that these arrays are implemented as byte-arrays, since they need to be identical on both big- and little-endian processors.

Listing 15: Rosetta branch-to-patch island template.

```
static unsigned char rosetta_patch_template[] =
{
    0x00,0x00,0x00,0x00,    // .long    branch_target_addr
    0x00,0x00,0x00,0x00,    // .long    error_branch_target
    0x00,0x00,0x00,0x00,    // .long    patch_table_index
    0x3D,0x60,0x00,0x00,    // lis      r11,(msw of this_entry_addr)
    0x61,0x6B,0x00,0x00,    // ori      r11,r11,(lsw of this_entry_addr)
    0x81,0x8B,0x00,0x00,    // lwz      r12,0(r11)
    0x7D,0x6D,0x5B,0x78,    // mr       r13,r11
    0x81,0xCB,0x00,0x08,    // lwz      r14,8(r11)
    0x2C,0x0C,0x00,0x00,    // cmpwi    r12,0
    0x7D,0x89,0x03,0xA6,    // mtctr    r12
    0x4C,0x82,0x04,0x20,    // bnectr
    0x81,0x8B,0x00,0x04,    // lwz      r12,4(r11)
    0x7D,0x89,0x03,0xA6,    // mtctr    r12
    0x4E,0x80,0x04,0x20     // bctr
};
```

The first time a patch function is called, it will find its way to the assembly stub function. This is split into two parts: the binder interface and the linker function. The code above leads into the binder interface, which first of all stores all parameter registers on a new stack frame. It then loads the address of its globals table (described at the end of the data table header section above) and checks the value of the `fmwk_ok` value. If this is zero, it calls the linker function. This one stores r13 and r14 on the stack, since it'll make function calls itself, and uses the data in the other members of the globals table to call `NSAddImage` and `dlsym`, each time passing a string whose address is also gleaned from the globals table. The address returned by `dlsym` is written into the globals table, and the `fmwk_ok` value is set to 1 to indicate that the framework has been loaded. This function restores r13 & r14 before returning to the stub interface function.

The stub interface then moves r13 & r14 into r3 & r4 to act as standard parameters, and also reads the address of the data table header from the globals table, passing that as a third parameter. It then loads and branches to the just-loaded bind function proper.

The bind function has a few functions to perform, itself. Firstly, if it hasn't been called before, it sets up a few global variables from the data table header: the addresses and sizes of the patch tables themselves, and pointers to a couple of internal data tables. It also initialises a pointer used to array-index the patch info table, and calls `atexit` to get things deallocated when the application quits.

Its standard function then kicks in: It looks up the supplied index in the patch info table, and loads the patch bundle it references. It then looks up the address of the patch binding function, and it calls that, passing in the callback which will be used to retrieve the reentry island addresses. This callback follows the branch absolute at the given target function address, and reads the info table index from that island; it then looks up the reentry value from the patch info table.

Once this is complete, the only remaining step is to compute the `vmaddr_slide` of the newly loaded bundle, and to offset the patch function address using that, to compute the absolute patch function address. This is then written back to the patch island through the supplied pointer, such that future calls to that particular patch will go straight through, rather than coming to the bind function again.

It also returns this address to the stub helper interface, which loads it into the counter register, restores the parameter registers, removes its stack frame, and branches to it directly.

3.3 The Complete Overview

This, then is the full process we will use on the injection side:

1. The injector loads the the bundle/bundles itself, and uses a couple of new entry points to determine what it wants to do:
   (a) It asks the bundle if it wants to patch the target, giving it a name and a process ID.
   (b) If the bundle wants to install patches, it asks for details, providing a callback.
   (c) The bundle calls the callback with the name of each patch handler function and the PowerPC address of its target.
      i. The injector takes the name and looks up the PowerPC address of this handler function.
      ii. It stores that address, along with the URL for the bundle and the target address, in a data table.
      iii. It installs a slightly different branch island, which will call a stub binding function, passing in the patch's location in the data table.
2. If any bundles said they wanted to patch anything, it injects some PowerPC code & data:
   (a) First of all, it copies across the (locally-created) blocks containing the branch islands.
   (b) Secondly it copies across a large (two pages) data table, whose contents include thing such as:
      - A table containing patch-info structures, which contain the info from 1.c.ii above.
      - A table containing string data (similar to string data in a binary object file).
      - Various pointers, used to access these tables.
      - The path to the patch framework and the symbol name of the stub binding function proper (written in C).
      - The FQPN of the override/injection library, to be passed to `dlopen`.
      - The PowerPC addresses of `dlopen` and `dlsym`.
      - The stub helper interface code, written in PowerPC assembler.

3. Lastly it injects some Intel code and a list of address/branch-absolute-instruction pairs as an argument:
    - The Intel code is a small loop which runs through the 2D array it's been provided, atomically copying the instructions to their targets.
4. Since the Intel implementation of `thread_terminate` isn't an external symbol in the *translate* app, the injector watches the first byte of the injected i386 code, and when it's set non-zero (when it's done copying stuff) it will terminate it from there.

References

[1] Jonathan 'Wolf' Rentzsch, *Dynamically Overriding Mac OS X*, 2003. http://www.rentzsch.com/papers/overridingMacOSX

[2] Bertrand Guihéneuf, *A port of mach_inject and mach_override to intel*, 2006. http://guiheneuf.org/Site/mach%20inject%20for%20intel.html