VERIFOXX

Verifoxx Morello Compartment Framework Software Architecture Design

Version 0.1 (Draft)

Verifoxx LTD,
128 City Road,
London,
EC1V 2NX
United Kingdom

Company Registration Number:
12107610

Change History

Date	Version	Changed By	Details
15th May 2024	0.1 (Draft)	Pete Dunne	Initial Version

Table of Contents

1 Introduction

This document describes the software architecture and design of the Compartment Framework for Morello and is intended as a guide to understanding how to the code works. Please note that this is still an experimental framework and a work in progress.

The framework was originally developed for the compartmentalisation of the Web Assembly Micro-runtime (WAMR) application for CHERI Morello purecap. WAMR is a complex codebase implemented as a shared object library, which is intended to be linked against a user application or loaded at runtime. It exposes a large API which enables the user application to perform WAMR operations.

The framework is designed to support compartmentalisation of all code loaded from a dynamic shared object library. A user application, known as the *capability manager,* runs in the Morello PE Executive state and calls API operations which are implemented in the compartment, running in the Morello PE Restricted state. The framework aims to make it easier to call API operations from the executable, via a local proxy, which under the hood transits into the compartment. It is intended for porting large codebases with many existing API functions to an executable / compartment library realisation.

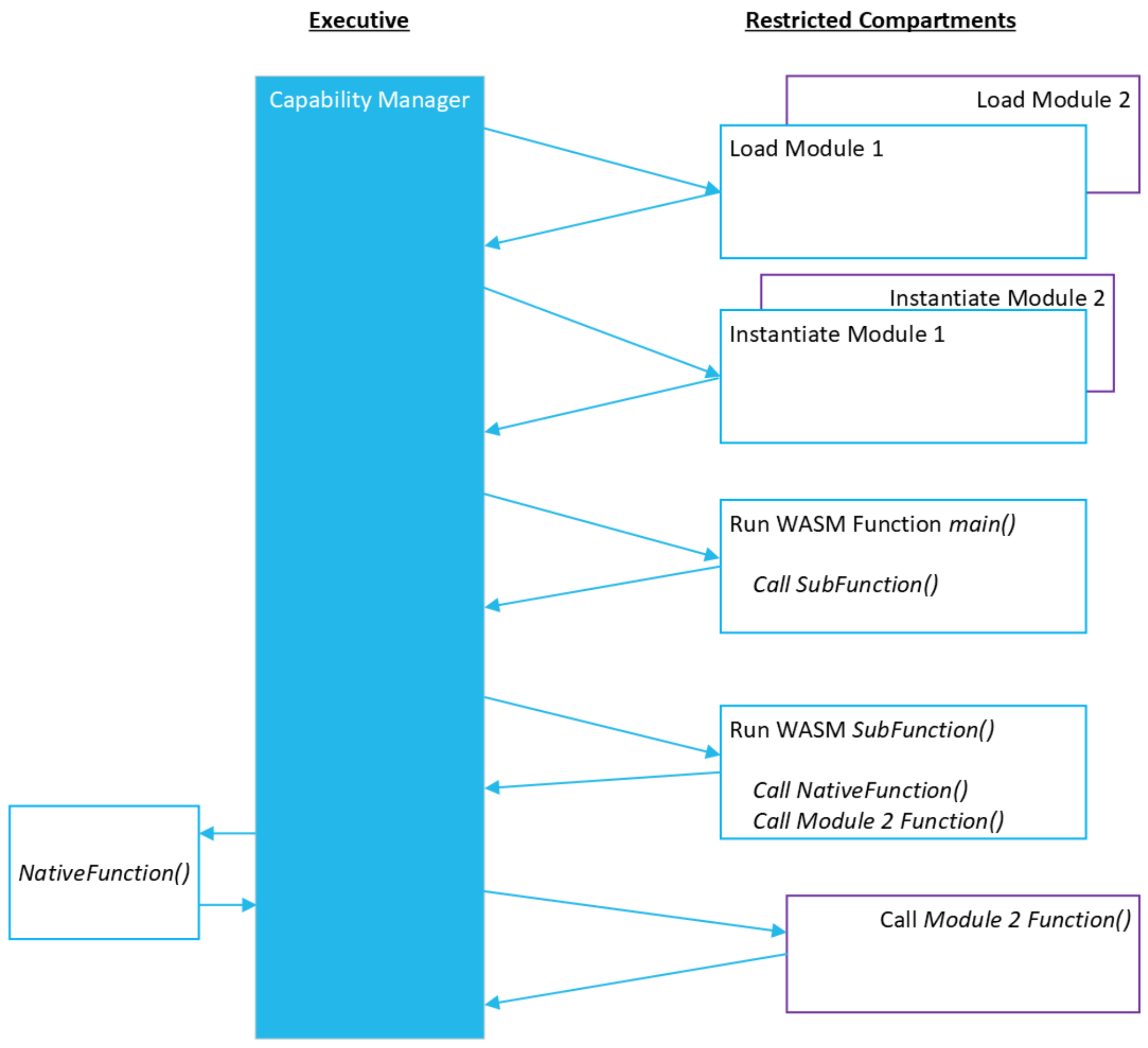
2 Concept

The concept is that the entry point application is the *Capability Manager*, running in the Executive state. This has overall control, full access, and is responsible for creating one of more Compartments to sandbox library functions. The capability manager should be the master for all compartments and store capabilities they need; should a compartment need to call a function in another compartment then this must be transited through, and marshalled by, the Capability Manager.

Compartment restricted code should not (in the final realisation of Morello Linux) be able to access any system functions or native code directly, and therefore all such calls must involve a transit through the capability manager which handles the call on behalf of the compartment.

This is ideally suited for Web-Assembly (WASM) which needs a sandboxed environment and such an environment is provided by a WASM runtime. We then have a double-sandboxed environment because no WASM runtime code has direct access to any native code outside of its compartment.

This concept is shown below, the example is shown for executing a WASM application:



In this diagram the system supports multi-modules, and two modules are loaded. The WASM entry point is *main()* and this function is called first. It subsequently calls another function, which then calls a native function and a function in a different WASM compartment. Conceptually, calls to other WASM functions happen seamlessly within the WASM application and WAMR but in practice these route through the capability manager.

Native calls are invoked directly by the capability manager; calls to other WASM function re-enter a compartment.

2.1 Same or Multiple Compartments

The compartment is an abstract concept. In practice it means that all code and data is isolated from other parts of the system, by means of suitable restricted capabilities.

It is therefore not realistic to say that we are using the "same" or "different" compartment as it depends on what code and data capabilities are set up ahead of transitioning to the compartment (and restricted state). We can then say that:

- The "same" compartment can be considered to be the case when the same native stack and/or native heap is applied for an operation that runs inside a compartment

The framework as currently implemented supports only a single compartment but it can be extended to support multiple compartments as a natural evolution of the codebase.

2.2 Calling into a Compartment

The general concept of capability manager calling a compartment is as follows:

Code:

1. There is a *compartment entry* executable capability which has
    - Bounds set to the range of compartmentalised code only
    - Executive permission cleared (i.e it is restricted)
    - A specific, sealed, entry point (likely to always be the same marshalling function)
2. There is an *operation to execute capability*; this is the actual API function to be run in the compartment
    - This is also bound & restricted accordingly
    - The marshalling code at the common entry point uses this to call the actual operation intended
3. There is an *exit point capability* which allows the compartment to return
    - This is a sealed entry (sentry) capability which allows transfer back to executive
    - It has the executive permission and is ideally tightly bound to a small part of the capability manager code, as it is used only in the transition from restricted -> executive

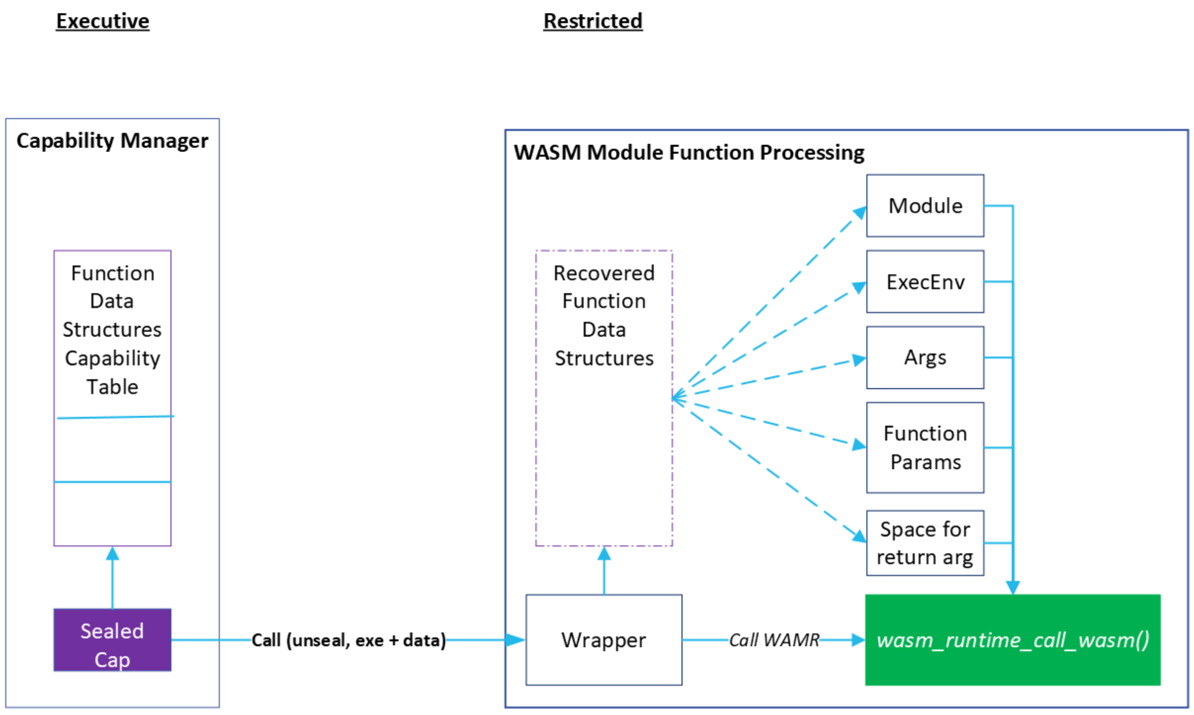
Data:

4. The compartment has a dedicated, memory mapped stack
    - This is switched in automatically by Morrello on an Executive -> Restricted transition, as the *RCSP* register is set up to point to this dedicated stack
5. There is *either* a dedicated capability for a native heap pool, *or* a *service* is provided in the capability manager to allow the compartment to e.g allocate heap memory (discussed further, below)
6. There is a *sealed* capability (restricted and bound for the compartment) which provides access to a capability table for the compartment that comprises individual capabilities for:
    - Function arguments to be passed to the operation
    - Data structures needed by the operation

NOTES:

- There is no capability for returning arguments; the API is assumed to follow the C calling standard so any return value is an argument to the function call that processes the compartment exit function
- Function argument capabilities may be reference types (pointers to pointers) allowing argument return by reference

An example of the compartment calling concept is shown below, the example shows calling a WAMR API with WAMR data structures. Note that a single entry point "unwraps" all needed arguments and data structures and then calls the actual API function that is the entry point for the needed operation. In this case the API operation is *wasm_runtime_call_wasm()* and it makes use of different data structures shown as input with the blue arrow:



2.3 Capability Manager Callback Services

The flow as described so far will allow an API operation to run in a compartment by calling a C (or *extern "C"* C++) function with provided arguments and which returns a value. This API operation function entry point may itself call many other functions within the compartment, which does not affect any part of the framework and is so not relevant for this discussion.

However it is necessary to extend this simple flow by providing *services* within the capability manager which a compartment function can use as part of its processing. A simple example of this would be to log information to *stdout*. This involves a *printf()* type call, which is a native function and which therefore cannot be made from the (PE restricted) compartment without a transfer back to the capability manager in the executive state. As has been discussed, even calling other user native functions should involve a call through the capability manager – again, this is considered a *capability manager callback service*.

NOTE: *At the time of writing, Arm have not yet implemented a security policy for this and so on Linux it ****is***** still currently possible to call system functions from the restricted PE state, although this is not final intended behaviour*.

In order to allow the compartment to make calls back into the capability manager to process a service we need to extend the model and provide additional capabilities to the compartment upon entry. This is because, without a dedicated capability, the compartment cannot "magically" just call into the capability manager as it likes. Therefore the following additional capabilities are needed:

Code:

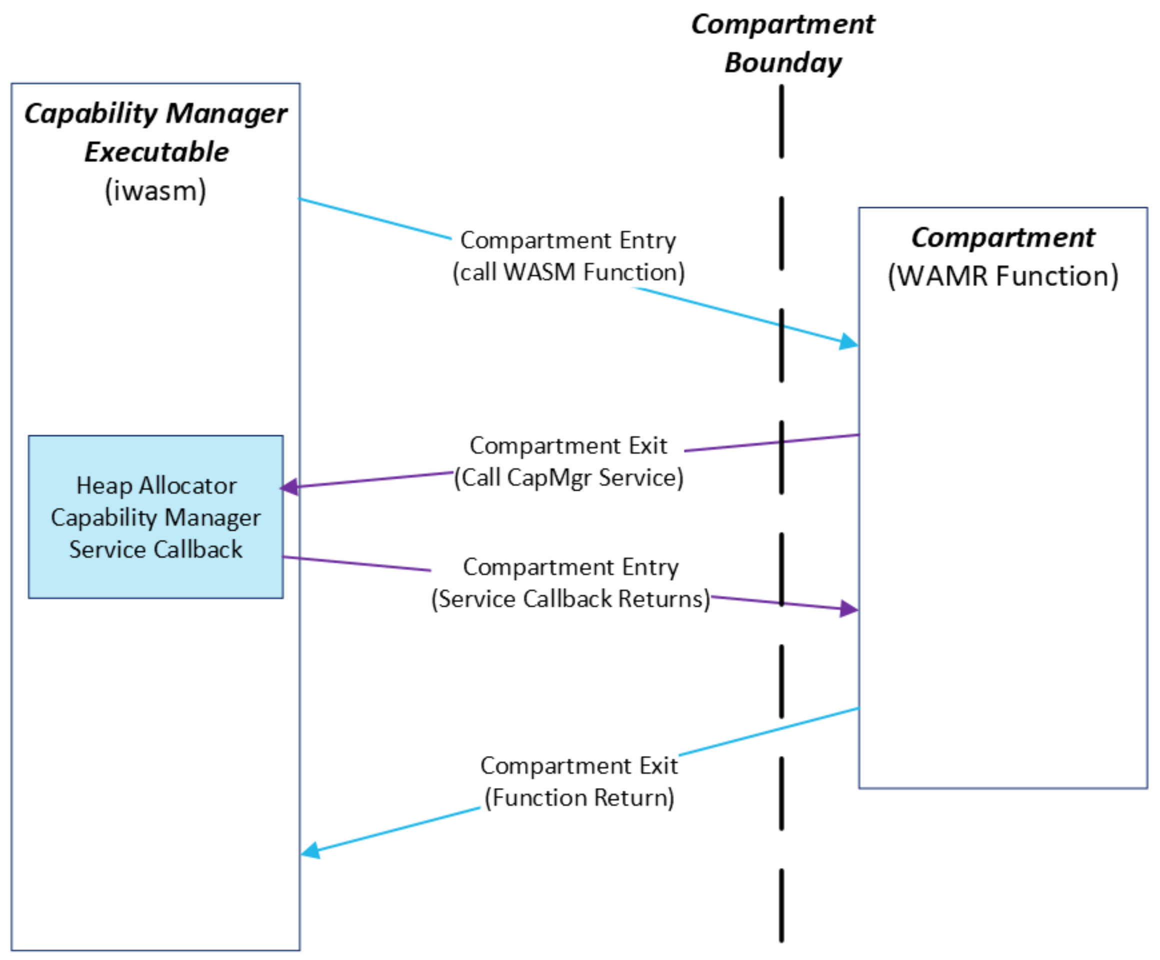
- Service callback entry point (generic entry point)
    - Executive permission, ideally tightly bound, for calling the service wrapper
- Table of available services
    - Allows compartment operation to choose the service
    - Can be table of capability function pointers or other secure mechanism to identify the service callback required

Data:

- Arguments needed by the service callback function

Note that as the service callback is handled in the capability manager then the native stack / heap etc. are not needed, as these will be the main executive stack and heap.

The generalised flow for a service callback is shown in the below example. Again, this is a WAMR example so the Capability Manager program is called "iwasm" and the API operation is a "WAMR function", but the generic case should be obvious:


Compartment Bounday
Capability Manager Executable (iwasm)
Compartment (WAMR Function)
Compartment Entry (call WASM Function)
Heap Allocator Capability Manager Service Callback
Compartment Exit (Call CapMgr Service)
Compartment Entry (Service Callback Returns)
Compartment Exit (Function Return)

In the diagram we can see the call into the compartment operation (in this case to run a WASM function), itself involves a service in the capability manager before it finally returns.

3 Compartmentalising Library Code

The Capability Manager is a Linux executable which is either statically or dynamically linked, this is configured to be a PIE. The library code to be compartmentalised needs to be split so that any native code involves calling system functions via a *service callback API* and the API implementation to be compartmentalised needs to be built as a dynamic shared object library.

If this compartment library is dynamically loaded, at runtime, then its contents are placed into dedicated memory sections by the loader and we can maintain capabilities bound to the library (and hence compartment) code that do not include the capability manager virtual memory space. Furthermore, loading the library in this way allows it to be PIC which means we have full flexibility and portability that would not be achieved if we attempted to bind the library to a specific address.

3.1 Initialized Capability Problem

There is though still a problem with the library approach, which concerns initialized capabilities.

Within the library code there will be a number of execution code symbols, but also initialized global data and constant tables symbols (and in a complex codebase there will be a vast number of these). These symbols cannot be resolved at link time because the target destination address is unknown and in fact the symbol may be in a different shared object library. Instead, they are resolved at runtime by the loader.

On CHERI, these particular symbols will be *initialized capabilities* and a full capability is added by the loader.

The problem here is that the loader will use a *root capability* which is provided by the system to the executable program (i.e the capability manager) in the auxiliary vector – but that root will have Executive permission enabled. But in our case the compartment code must run in the Restricted state.

This problem applies to not only the dynamic library loaded for compartment code, but also any dependent dynamic libraries. At runtime, a symbol may need to be resolved to an actual piece of code in another library by means of a *Procedure Linkage Table*.

<u>Example</u>

Consider the example that the capability manager has obtained the symbol address of a particular function in library code. It will use a suitably restricted and bound capability to call this piece of code.

However this will actually invoke a function in a PLT which then looks up the actual symbol (which maybe in a child dynamic library). The symbol address is resolved using a full capability which is initialized at runtime to the actual destination address of the symbol. The problem as stated is this full capability is not restricted, and potentially not correctly bound either.

What then happens at runtime is we make an unexpected switch from restricted to executive state, and so when we then return (via a normal *RET* not a special state-switch return) this is an invalid operation as it switches PE state back to restricted, and there is a capability fault which then occurs.

3.1.1 Potential Solutions

Arm have produced two examples which offer potential solutions to this problem.

In the first case, code to run in a compartment is linked to an executable which is then loaded to an absolute address (i.e not relocatable) and the "relocation" actions that would normally be performed by the loader are performed manually using a modified root capability. This is made possible because the loading of the executable in memory is performed using bespoke code.

In the second case, individual functions are compartmentalised at runtime by mapping a dedicated memory area and then copying the code at runtime with manual set up of suitable capabilities.

Unfortunately neither of these approaches are practical for a large codebase designed as a dynamic library with a dedicated API providing entry points. It is unrealistic to statically link compartment code; it would be too inflexible not least if a non-CHERI "upstream" version of the library is constantly in development (as is the case for WAMR, which led to the development of this framework). Further, implementing a bespoke manual loader is too large an overhead which would then be a local solution to a potentially widespread problem.

Regarding the second approach, it is not feasible to manually copy and set up symbols to the hundreds of code and data symbols that would have to be dealt with.

3.1.2 Actual Solution

The chosen solution is to use the dynamic shared object library loading mechanism "as is", and then to manually patch all of the resolved symbols in order to remove the executive permissions. This is not ideal, as it is a time-consuming overhead that could in theory be solved by changes to C standard library functions to add Morello support (e.g modifying *dlopen()* system function). Operation of the solution is described in the next sub-section, below.

3.2 Relocation Symbol Patching

The dynamic shared object library is loaded via the POSIX system function *dlopen()*. This is a standard technique to load a library at runtime.

As part of this process, the loader library (*ld.so*) is used to resolve any link-time symbols whose address could only be known at runtime. The library is (on Linux) an ELF file, which is a data structure that includes a list of *relocation tables*. These inform:

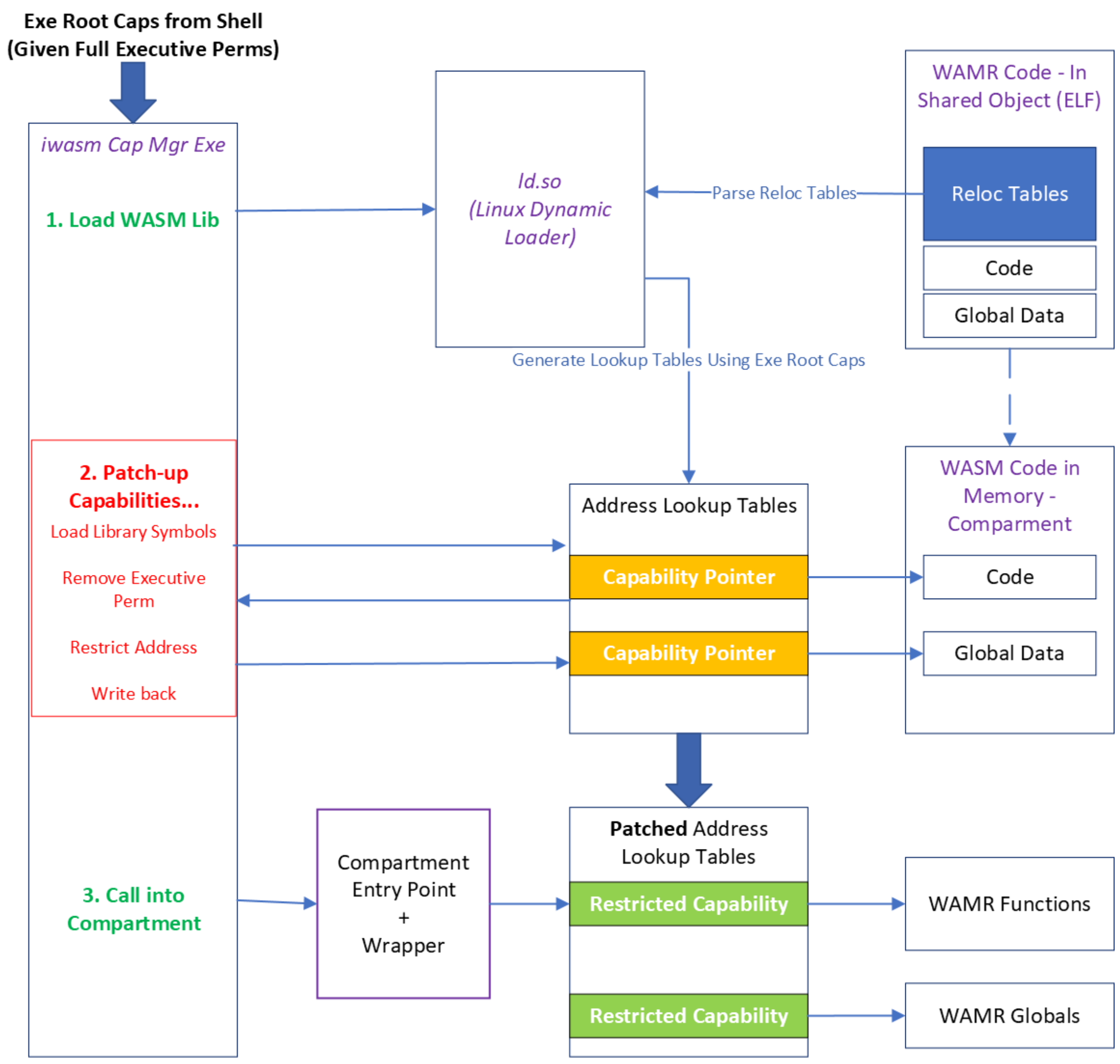
- Where is the symbol within the file (as an offset) which needs resolving
- What sort of symbol is it (how should the resolution be performed)
- What is the target address that it needs resolving to

Morello ISA adds a number of new resolution types to deal with the fact that any addresses are now capabilities and not pointers (or number of bytes from start of memory). Therefore if there is a target address to be used for the resolution, in Morello it will also have bounds and permissions for example.

When the loader performs the resolutions, using the now-known actual addresses, on Morello these will be a full capability and will have the executive permission. This is because the loader is running in the executive state and is using a root capability with an executive permission. However in our case the restricted state is required for the symbol.

We must therefore “patch” all of the symbols resolved by the loader, after it is finished the loading process. This is effectively undoing a lot of what the loader is doing – therefore it could be considered that the loader code should be changed instead. However this would require rebuilding the standard C library which would presumably mean a rebuild of the Linux kernel, which is outside the scope of this work.

The process of patching the symbols is shown below (a very simplified example). Again, this is applied to the WAMR use case so we show a capability manager called “iwasm” which loads WAMR code as a library, and example code and data applicable for the WAMR solution – but the generic flow should be clear:



It can be seen that the loader actually maps memory and copies the relevant blocks from the shared object ELF file into that memory. At this point it knows the actual target address of all symbols, so a symbol offset from top of ELF file can now be converted into an actual address (e.g somewhere within a block of executable code).

It will then resolve the required address value to write into that location (e.g the target of a branch instruction) which may well be the address of a PLT placeholder in a PLT for one of the other libraries. This step is repeated for all the symbol tables, and the loader completes (and *dlopen()* returns).

At this point, we must now patch all the symbols which use full capabilities. Once this is done, we can successfully call into the compartment with the knowledge that we will remain in restricted mode.

3.2.1 Symbol Patching Caveats

Before explaining the patch-up process, some issues must be explained.

Disable Lazy Binding

In order for the patch-up to work, all symbols must have been resolved by the loader that is performed from the *dlopen()* call. Normally, PLT symbol lookup and resolution would only occur upon the first access to the symbol (e.g the first branch instruction) which means initial loading is quicker. This is known as *lazy binding*, but we need to tell the loader to not do this in our case.

Namespaces

If the capability manager executable itself is dynamically linked and happens to use a symbol in a dependent dynamic shared object that is also used by the compartment library then there would only be one copy of the library loaded and the same bit of code would be used in both cases.

However in our case this would cause a problem because the capability manager executable requires a library loaded into the executive state but for the compartment library it must be restricted.

The solution in this case is to load the library into its own *namespace* which means that any shared libraries are loaded into memory twice – and the new namespace is isolated from any others.

This requires use of the *dlmopen()* instead of *dlopen()* function. Note that if the executable is linked statically, then there is no issue because all symbols are resolved at runtime. Currently, the capability manager is statically linked but in the future it will depend on libraries which do not have a static variant and therefore it will need to be dynamically linked.

To facilitate this the framework enables the code to be built for both statically and dynamically linked executable.

NOTE: MUSL LIBC, which is the target standard library for Arm's LLVM Toolchain port, does not support static builds of dlopen() and does not support dlmopen() at all. For those reasons, our example framework must currently be build with GCC and use GLibC.

Library Cleanup

This is only an issue when the executable is dynamically linked, i.e *dlmopen()* is used instead of *dlopen()*.

In this case, ahead of application exit any library termination functions are called by the system using hidden functions within the loader. The problem here is that this will invoke an exit function for the compartment library which was loaded (and its namespace) but this is a problem because we will then invoke restricted code directly from executive.

The solution taken to this is to, just ahead of application exit, repeat the symbol-patching process but this time reverse all of the patching and hence restore the executive permission to all symbols.

Due to the need to do this, a root capability is needed to be used for all symbol patching – we cannot use the symbol directly as we cannot "create" new permissions for a capability, only disable existing permissions.

Toolchain Differences

The LLVM and GNU toolchains before differently with regard to symbol relocation during dynamic library loading and parsing an ELF file. Although the LLVM toolchain can be used, as stated previously the MUSL LibC cannot work with the way we need to use *dlopen()* or *dlmopen()* and therefore currently only building with the Gnu Toolchain is supported.

3.2.2 Process

The symbol patching is complex. In essence it involves parsing the ELF structures which can be obtained from the *dl{m}open()* call and locating the relocation tables. We then iterate through each entry in each relocation table looking for types which would need fixing. Types which need fixing are those which involve a capability (i.e a Morello relocation type), which are in a section we care about and which are valid (which have the tag set).

Note that it is not necessary to resolve the address the symbol references – this was already done by the loader. We just need to modify the capability already set for that address.

In order to perform the relocation, a base capability is used which is itself derived from the executable's RW auxiliary vector capability (passed into the executable startup by the shell). As stated already, this is to deal with the need to revert capability patchups in some instances and because you cannot set permissions to a capability which does not have them already, so it is necessary to source the fixups from a root capability in this case.

Apart from anything we want to restrict, all other settings of the capability to fix-up are derived from its existing value. This includes making the capability a sealed entry capability if it is a function call.

3.2.2.1 Finding Relocation Tables

Relocation tables are parsed to identify symbols to be fixed up, but first we need to locate the relocation tables in the dynamic library structure.

This is done by parsing the program header structures obtained from the *dlopen()* load of the shared object. We find the dynamic data area (PT_DYNAMIC) which itself is a table of addresses and sizes of different dynamic data sections.

This enables us to build a map of *dynamic data IDs → [address | size]*, and cherry-pick those we are interested in.

From this we can determine the start address and end address where the relocation tables within the shared object have been loaded into memory, namely the *.rela.dyn*, *.rel.dyn* and *.plt.rel* sections.

3.2.2.2 *Parsing Relocation Tables*

Parsing the relocation tables is a relatively easy process given we don't need to actually calculate the fix up address and just need to modify the capability permissions and bounds as required.

Only certain relocation types need to be patched, which are defined in the Morello ISA spec and are as follows:

- R_MORELLO_CAPINIT
- R_MORELLO_GLOB_DAT
- R_MORELLO_JUMP_SLOT
- R_MORELLO_RELATIVE
- R_MORELLO_TLSDESC

Standard *Aarch64* types which do not involve a capability can safely be ignored.

Additionally, there are certain addresses which should not be patched. These are those which are called intentionally from the executive which are related to the loading and teardown of the library. We check for those addresses by loading information from the *PT_DYNAMIC* area of the shared object ELF structure.

3.2.2.3 *Allowing Address Writes*

In order to patch up an address capability then it needs to be writable. But if it is code or a constant value it will not be. To deal with this, before the patch up process we again consult the program headers for the loaded shared object and for any *PT_LOAD* regions we use *mprotect()* to allow write access (this is permitted as it runs from the executive capability manager and we use a root capability to access the memory).

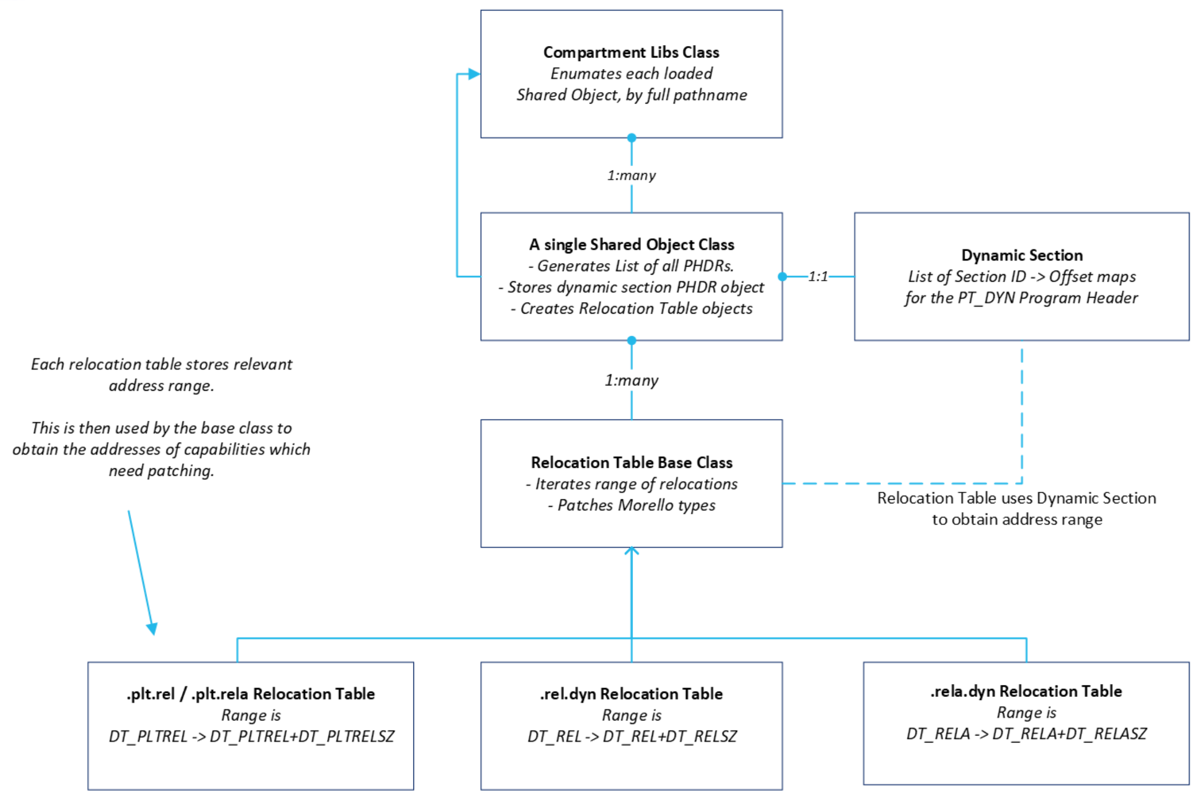
At the end of the patch up process we restore the *PT_LOAD* region to read and execute permissions only.

3.2.2.4 *Repeating for all Shared Objects*

As the compartment shared object library may depend on other shared objects then the fixup is needed for these, too.

This is handled by iterating all the shared objects in the namespace - apart from *ld.so*, the loader, which is never called from the compartment.

Overall, the approach yields a set of data structures, which is shown below:


Compartment Libs Class
Enumates each loaded Shared Object, by full pathname
1:many
A single Shared Object Class
- Generates List of all PHDRs.
- Stores dynamic section PHDR object
- Creates Relocation Table objects
1:1
Dynamic Section
List of Section ID -> Offset maps for the PT_DYN Program Header
Each relocation table stores relevant address range.
This is then used by the base class to obtain the addresses of capabilities which need patching.
1:many
Relocation Table Base Class
- Iterates range of relocations
- Patches Morello types
Relocation Table uses Dynamic Section to obtain address range
.plt.rel / .plt.rela Relocation Table
Range is DT_PLTREL -> DT_PLTREL+DT_PLTRELSZ
.rel.dyn Relocation Table
Range is DT_REL -> DT_REL+DT_RELSZ
.rela.dyn Relocation Table
Range is DT_RELA -> DT_RELA+DT_RELASZ

Internally, in the capability manager code, the above data structures are loaded into a class hierarchy using the *link map* obtained from *dl{m}open()* and *program headers* for each loaded shared object.

The single *CompartmentLibs* object enumerates all loaded shared objects, and for each of then a *Shared Object* class contains:

- List of *Program Headers:* this is a map, header ID -> header info (which includes start/end addresses)
- A *Dynamic Section object* which is a parsed *PT_DYNAMIC* program header
  - This object holds a map of all section symbol IDs -> address of symbol
- A list of *Relocation Tables* which are generated by parsing the dynamic section map
  - These are stored as a vector of shared pointers to the relocation table data

These structures allow the self-describing ELF headers to be built into ever more intricate structures, resulting in a relocation table structure for each of the relocation tables we need to know about.

The relocation structure comprises a start and end address, resolved from an ELF offset by converting the offset base to the base address of the library in memory. The start and end address themselves are obtained by looking up the symbols in the *Dynamic Section* map. For example, here is the code to get the *.rel.dyn* relocation table start address, end address and element size:

```
Range CDynamicSection::GetRelRel(size_t& elem_size) const
{
    uintptr_t addr = reinterpret_cast<uintptr_t>(m_base + GetEntry(DT_REL));
    size_t sz = reinterpret_cast<size_t>(GetEntry(DT_RELSZ));
    elem_size = reinterpret_cast<size_t>(GetEntry(DT_RELENT));

    return Range(addr, sz);
}
```

Here we lookup three symbols in the map, DT_REL which is the start of the relocation table, DT_RELSZ which is the length and DT_RELENT which is the size of each entry (so we know how many bytes to skip in passing each table entry). We return these as a Range, which is a utility class to represent an address block.

All the relocation tables can be handled in the same way, hence use of a base class. The only difference is how their addresses and size can be resolved, hence the use of a base class virtual function to call into the derived class for each relocation table type.

Patching

The actual capability fixup patching code performs a number of checks before then modifying the capability to using a base capability. The checks are:

1. Relocation type must be one of the Morello types which need patching
2. Target address must not be within a *skip range*
    - Skip ranges are read from the Dynamic section symbols
    - They are things like FINI code which is part of executive state library cleanup only
3. Capability tag must be valid

The fix-up value is derived by taking the bounds and permissions from the current capability and then using a base root to set or remove executive permissions and reseal *sentry* as needed. The code snippet below shows the derivation of a fix up value:

```
uintptr_t DeriveFixupValue(uintptr_t val_to_fixup, bool makeRestricted=true) const
{
    auto cap = Capability(m_fixup_cap)
        .DeriveFromCap(reinterpret_cast<void*>(val_to_fixup),
            makeRestricted ? 0 : ARM_CAP_PERMISSION_EXECUTIVE, // Perms to add
            makeRestricted ? ARM_CAP_PERMISSION_EXECUTIVE : 0  // Perms to remove
        );

    return cap;
}
```

The *DeriveFromCap* method of the *Capability* utility class loads the base *m_fixup_cap* which is a root and copies the *val_to_fixup* but adds in and removes permissions as shown. Effectively, if we switch from executive -> restricted we explicitly remove the Executive permission and if not we are reverting the fixups on exit so we explicitly add the Executive permission.

DeriveFromCap uses intrinsic methods from *cheriintrin.h*:

```
Capability& DeriveFromCap(void *p, size_t set_perms=0, size_t clear_perms=0)
{
    // If requested, make sure p is within range of existing
    if (cheri_base_get(p) >= cheri_base_get(m_cap) &&
          (cheri_length_get(p) + cheri_base_get(p))
              <= (cheri_length_get(m_cap) + cheri_base_get(m_cap)))
    {
       // Restrict lower bound to same as p
       m_cap = cheri_address_set(m_cap, cheri_base_get(p));

      // Restrict upper bound same as p
       m_cap = cheri_bounds_set(m_cap, cheri_length_get(p));

       // Current offset (like address) same as p
       m_cap = cheri_offset_set(m_cap, cheri_offset_get(p));
    }
    else
    {
       m_cap = cheri_address_set(m_cap, cheri_address_get(p));
    }

    size_t desired_perms = cheri_perms_get(p);
    m_cap = cheri_perms_and(m_cap, desired_perms | set_perms);
    m_cap = cheri_perms_clear(m_cap, clear_perms);

    if (cheri_is_sentry(p))
    {
        m_cap = cheri_sentry_create(m_cap);
    }
    return *this;
}
```

Note the application of *cheri_sentry_create()* if the source capability was a sealed entry one.

Writable Area for Patch

Before the patchup, the area containing the symbols which may be patched is switched to writable, and this is switched back after the patching completes.

The program headers for the library were previously created in a map, actually this is a *std::multimap* to permit duplicate keys because there may be multiple *PT_LOAD* blocks. For each block the program header structures will inform the start & end addresses, along with the flags (read | write | execute).

Ahead of the capability patch, the map of program headers is searched to find PT_LOAD entries and the address range for each is retrieved. *mprotect()* is then used to give the range writable permissions.

After the patch, the original values are written by checking the flags from the PT_LOAD entry in the program headers map.

The function *ProtectAllBlocks()* handles this in the code, here is the call to make the blocks writable:

```
ProtectAllBlocks(PT_LOAD, false, PROT_READ | PROT_WRITE);
```

To restore the settings according to flags (after all relocation table patchups done), the third argument defaults to true and flags are not needed:

```
ProtectAllBlocks(PT_LOAD)
```

ProtectAllBlocks() just iterates all matching entries in the multimap:

```
bool CSharedObject::ProtectAllBlocks(Elf64_Word type, bool restore_original
                                         int64_t prot_required) const
{
    // Find matches
    auto itrs = m_phdrs.equal_range(type);
    bool result = true;

    for (auto itr = itrs.first; itr != itrs.second; ++itr)
    {
        // Keep going on error
        result &= ProtectBlock(itr->second, restore_original, prot_required);
    }
    return result;
}
```

ProtectBlock is given an *Elf64_Phdr* structure which contains address and flags, and looks like this:

```
bool CSharedObject::ProtectBlock(const Elf64_Phdr& phdr, bool restore_original,
                                       int64_t prot_required) const
{
    // Get the address of the start of the block and the size
    void* base = m_base;
    uint8_t* block_start = &(reinterpret_cast<uint8_t*>(base))[phdr.p_vaddr];

    // Align so we are always working with values aligned to an OS page size
    uint8_t* block_aligned = cheri_align_down(block_start, m_page_size);

    size_t sz = phdr.p_memsz +
                       reinterpret_cast<ptrdiff_t>((block_start - block_aligned));

    // Figure out the perms we need
    if (restore_original)
    {
        // Derive from the supplied flags, otherwise is supplied
        // Need to map Program header constants to mprotect constants
        prot_required = 0;
        if (phdr.p_flags & PF_X)
            prot_required |= PROT_EXEC;

        if (phdr.p_flags & PF_W)
            prot_required |= PROT_WRITE;

        if (phdr.p_flags & PF_R)
            prot_required |= PROT_READ;
    }

    if (0 != mprotect(block_aligned, sz, prot_required))
    {
        return false;  // mprotect failed
    }
    return true;
}
```

4 Calling API Operations in a Compartment

The framework makes it easier to call API functions in the compartment; this has involved creating a lot of “boiler plate” code with the result that one can call a “proxy function”, and the framework will then transit this across the compartment boundary, unwrap, and call the actual API function implementation inside the compartment.

Note that a number of complexities were faced when trying to implement a modern-C++ STL framework to handle this, and in the end simpler constructs were used. The problems were down to attempting to move complex, virtual class instances between the executive and restricted states but this caused problems with the virtual function tables and other hidden aspects within library code. This prevented aspects of modern C++ such as *std::bind()* and RTTI from functioning properly. Instead, simple derived class structures were used which relied on static casting based on a passed in type identifier (similar to how Microsoft COM was implemented originally, without RTTI support).

4.1 Compartment Setup

The capability manager sets up a compartment by building a set of data structures for it. This process involves setting the compartment entry and exit capabilities and allocating the compartment stack by mapping a memory area. At the time of writing the compartment has a large stack size for development purposes only, but this should be reduced to a user-specified constraint, something which can be set at build time.

The compartment setup results in a number of capabilities being created to represent the compartment, which are as follows:

- Compartment Data Structure, itself comprising:
    - Compartment CSP: The compartment stack, which is *mmap’d()* to a block of available memory suitable for a stack
    - The permissions for this are set to *Restricted* and *Non-executable* (data only)
- Compartment DDC: Set to null capability
    - This is because DDC is only applicable for Hybrid-cap, and we only support purecap
- Compartment CTPIDR: Set to same as the capability manager CTPIDR
    - This is reserved for application specific usage and is not relevant to the framework
- A sealer capability, which is used to seal the API function argument data structure that will be created later
    - The sealer is created from the auxiliary vector *AT_CHERI_SEAL_CAP*
- A read+exec capability which points to the compartment exit function in the capability manager
    - This is the known, bound point which the compartment will use to return
- A capability which is the common compartment entry point, used to unwrap and route the actual function call
    - This is derived from the auxiliary vector *AT_CHERI_EXEC_RX_CAP*

- The address is determined by looking up the function in the loaded shared object namespace by a hard-coded given name, specifically this function is called *CompartmentUnwrap* and must be implemented in any compartment

The remaining information is constructed when a specific API function is called as at this point the data structures passed in function arguments can be sealed.

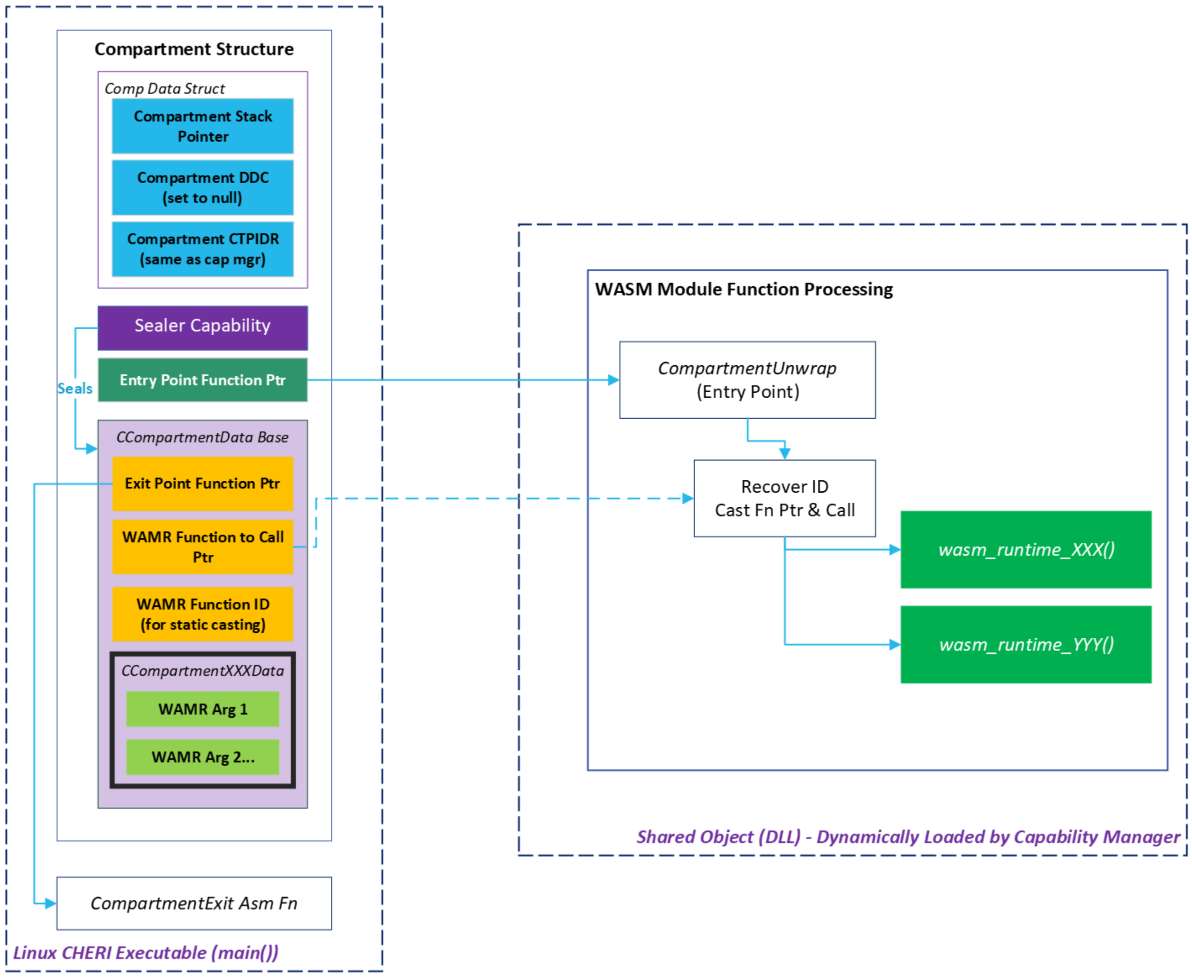
4.2 Compartment Call API Function

When an API function is called, the remaining data structures are built and then the compartment can be entered. All arguments for the function need to be passed to the compartment as a data structure accessed through a single capability.

Additionally, the API function to call within the compartment needs to be passed as a capability. This is tightly controlled from the capability manager, so only the intended top-level API function can be called. When the compartment is entered, the "unwrap" function will retrieve the data arguments and call the API function implementation itself, so as far as the API implementation is concerned it was called directly. The function in question is resolved, by name, by looking up the symbol in the library.

This whole process behaves like an IPC mechanism, whereby the appearance is given of calling the API function directly from the capability manager but in reality the call involves bundling up data, crossing the compartment boundary and then unpacking the data.

The final data structure is built at the point the API function to call is actually known. This is then sealed. The data structure is shown below, along with the behaviour of the unwrapping function in the compartment. The example is shown for generic WAMR functions (*wasm_runtime_*()*) and passing in suitable WAMR data structure arguments to these:


Compartment Structure
Comp Data Struct
Compartment Stack Pointer
Compartment DDC (set to null)
Compartment CTPIDR (same as cap mgr)
Sealer Capability
Seals
Entry Point Function Ptr
CCompartmentData Base
Exit Point Function Ptr
WAMR Function to Call Ptr
WAMR Function ID (for static casting)
CCompartmentXXXData
WAMR Arg 1
WAMR Arg 2...
CompartmentExit Asm Fn
Linux CHERI Executable (main())
WASM Module Function Processing
CompartmentUnwrap (Entry Point)
Recover ID Cast Fn Ptr & Call
wasm_runtime_XXX()
wasm_runtime_YYY()
Shared Object (DLL) - Dynamically Loaded by Capability Manager

With reference to the above, we have:

- Compartment CSP, DDC and CTPIDR as mentioned previously
- Sealer capability, as mentioned previously
- Compartment entry point, as mentioned previously
- Sealed entry Capability which is the compartment exit function in the capability manager (as set previously in compartment setup)
- Sealed entry Capability which is the actual function to call inside the compartment
  - This is obtained by looking up the function name in the loaded shared object library, and building a capability with restricted permissions and bounds using the auxiliary vector *AT_CHERI_EXEC_RX_CAP* as a base
- An identifier of the API function being called, this is needed by the “unwrap” function in the compartment
  - This is required to be able to resolve the API function pointer and argument data to the correct type
  - This is necessary because RTTI is not really possible as discussed previously, hence we perform a static lookup to access the derived class

- All data arguments, of varying type, for the specific function in question

A capability is taken to the above structure, which is itself then sealed. This can then be passed to the assembly code which switches into the compartment.

4.2.1 API Function Arguments

Each API operation takes a different number of functional arguments, of varying types. API entry functions must be in C, not C++.

To pass function arguments across the compartment boundary a class instance is used. There is a different class for each API function, with the function arguments being stored as member data within the class.

Each of these classes derives from a base class (named *CCompartmentData*) which itself has attributes that are common to all API functions, namely the capability for the actual API function to call, the capability for the exit function, and an identifier to know which function is being called.

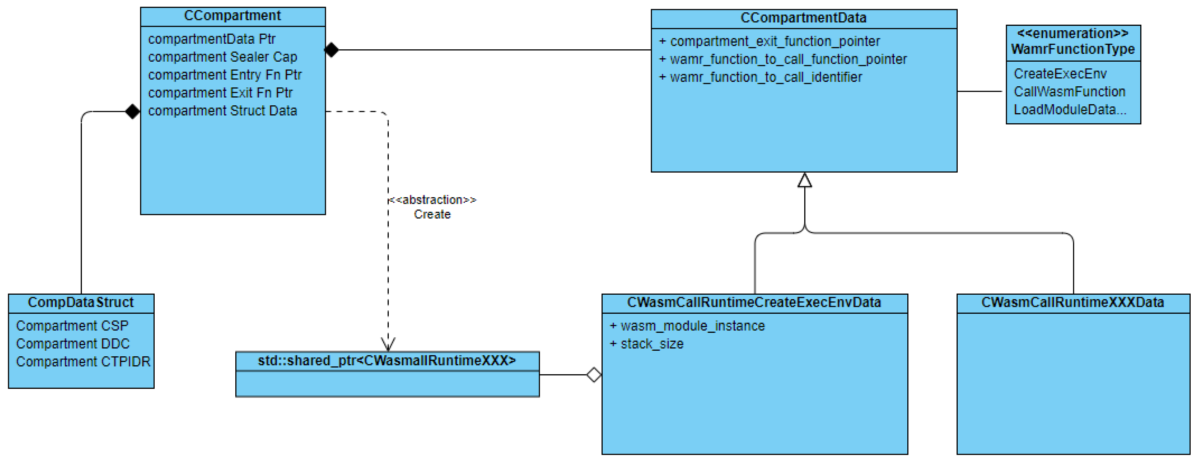
When a function proxy is called in the capability manager it causes an instance of the correct class to be constructed and a capability pointer (actually a *std::shared_ptr*) to be created for it. The entry function is set based on resolving the API function to its name and looking up the address of the function in the loaded shared object library.

Inside the compartment unwrap function, the exit function pointer capability and the "which-API-function-to-call" capability are extracted from passed in data. The identifier reveals how the API function pointer should be cast, and which arguments are available in the passed-in data. This then enables the function to be called directly using passed in arguments.

The return value from the function is always passed back as a *uintptr_t* in order to be large enough for any type (including any capability type). This can then be cast in the exit function in the capability manager.

The exit function is called directly, which is actually again in assembler to marshal the return back from the proxy function.

The compartment data structure is internally handled as a class object called *CCompartment.* The class structures for *CCompartment, CCompartmentData* and derived classes are shown below. The example API function being called is *wasm_call_runtime_create_exec_env_data()*:



4.3 Crossing the Compartment Boundary

The capability manager calls functions in the compartment, and receives a return value, across a well-defined boundary. This boundary switches from the PE *Executive* to *Restricted* state and involves a call into a function implemented in the dynamically loaded library. This function is called the *compartment entry point* and its job is to marshal the call into the specific API function that is being called.

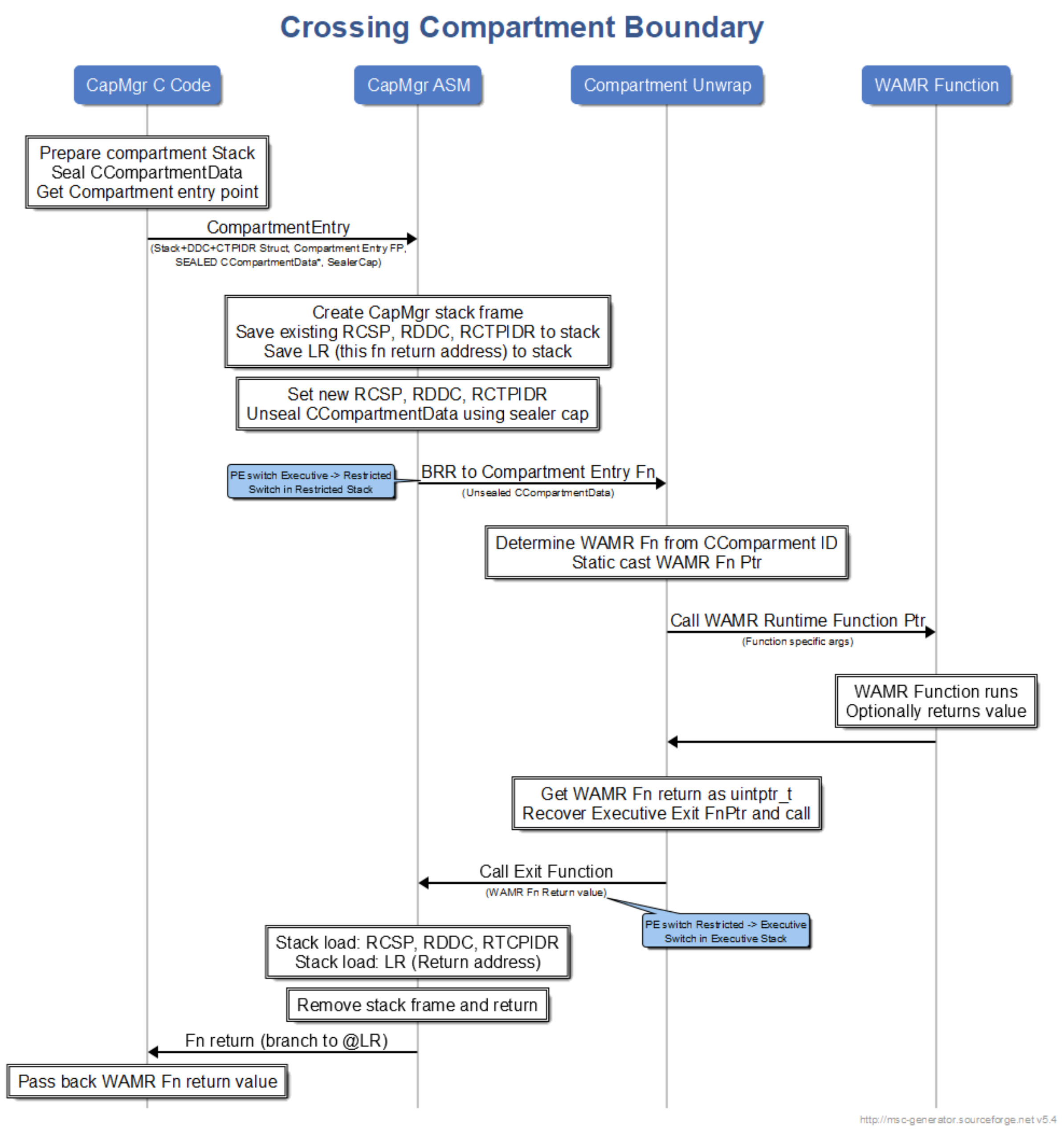
"Crossing the boundary" must be done in Morello assembler and involves using the *BRR* instruction to branch to a function with a switch to restricted state.

The process is as follows:

1. The compartment builds the final data structures and calls a plain C function, *CompartmentCaller(),* passing the following arguments:
    a. The assembly function to call to actually switch to the compartment
    b. The data structure containing:
        i. Compartment CSP
        ii. Compartment DDC
        iii. Compartment CTPIDR
    c. The sealed entry capability which is the *CompartmentUnwrap* entry point function inside the compartment
    d. The sealed capability which points to the class instance that holds:
        i. The API function pointer capability, and an identifier for the function
        ii. The capability for the compartment exit function in the capability manager
        iii. The API function argument data
    e. The sealer capability used to unseal the above
2. The assembly function passed in is then called from C, passing the remaining arguments

3. The assembly code saves the current values of the restricted CSP, CTPIDR and DDC along with the link register (return address) onto the stack and then prepares the compartment stack using the values passed in for compartment CSP etc.
4. The sealed capability is then unsealed and set as *C0* i.e the first argument to a C function a direct branch to the compartment entry point (the "unwrap" function is performed) which switches to the restricted state
5. We are now in the compartment. The Unwrap function is passed a capability which points to the class instance containing needed arguments. This is first treated as the base class *CCompartmentData*, in order to retrieve the API function ID and the underlying API function pointer capability.
6. As the function ID is known, we can now static cast to the actual derived class instance, and directly call the API function and provide all function data arguments correctly. The API operation runs, and may return a value which is cast to a *uintptr_t*.
7. It is now time to return from the compartment. This is done by calling the compartment exit function via the passed in capability and giving it the API function return value as an argument.
8. This calls us back into the capability manager in the executive state. The exit function is implemented in assembler. As we are in executive state:
    a. We are now switched back to the main stack - no longer using compartment's
    b. We can therefore directly retrieve the link register from before, i.e the return address within the capability manager to go back to
9. We finally then return, ensuring *C0* register correctly contains the return value from the API function. This is then made available as a return value from the proxy, giving the appearance the API function was called and returned directly.

The diagram below illustrates the process, using calling a WAMR runtime function as an example:


Crossing Compartment Boundary
CapMgr C Code
CapMgr ASM
Compartment Unwrap
WAMR Function
Prepare compartment Stack
Seal CCompartmentData
Get Compartment entry point
CompartmentEntry
(Stack+DDC+CTPIDR Struct, Compartment Entry FP, SEALED CCompartmentData*, SealerCap)
Create CapMgr stack frame
Save existing RCSP, RDDC, RCTPIDR to stack
Save LR (this fn return address) to stack
Set new RCSP, RDDC, RCTPIDR
Unseal CCompartmentData using sealer cap
PE switch Executive -> Restricted
Switch in Restricted Stack
BRR to Compartment Entry Fn
(Unsealed CCompartmentData)
Determine WAMR Fn from CCompartment ID
Static cast WAMR Fn Ptr
Call WAMR Runtime Function Ptr
(Function specific args)
WAMR Function runs
Optionally returns value
Get WAMR Fn return as uintptr_t
Recover Executive Exit FnPtr and call
Call Exit Function
(WAMR Fn Return value)
PE switch Restricted -> Executive
Switch in Executive Stack
Stack load: RCSP, RDDC, RTCPIDR
Stack load: LR (Return address)
Remove stack frame and return
Fn return (branch to @LR)
Pass back WAMR Fn return value
http://msc-generator.sourceforge.net v5.4

4.3.1 Compartment Entry Point

The compartment entry point as explained previously is resolved from looking for a hard-coded symbol name in the loaded shared object library. This entry function is key, as it is the only function which must always exist in the library and which provides a hard link between the capability manager (executable) and compartment (dynamic library). The compartment exit point is handled by capabilities passed to the compartment, and individually named API functions are dynamically resolved via a proxy mechanism (so it is up to the using class to know the desired name) – but the entry point is effectively a hard coded constant.

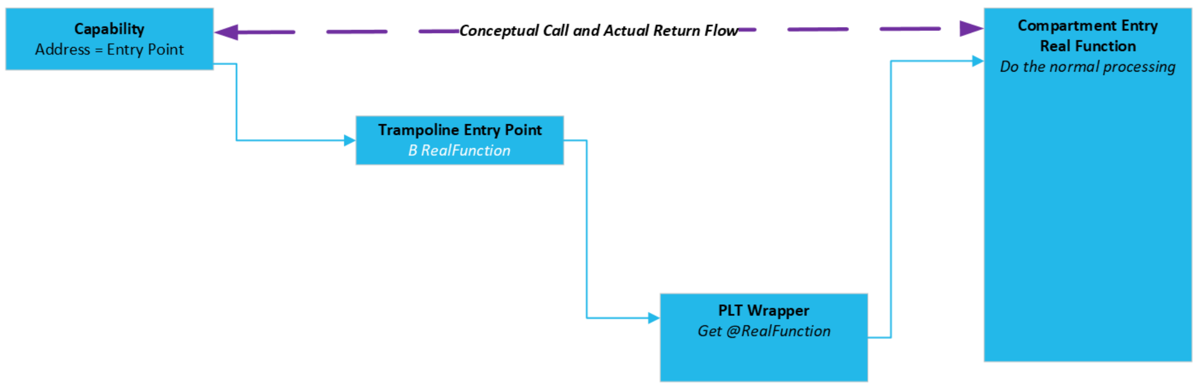
It is therefore necessary that it never changes, but this could be problematic if the library needs to change or new functionality is added in the future (and we don't want to break backwards compatibility). It is therefore convenient to provide a *trampoline* function which simply branches to the real entry point – this branch code is implemented as an ASM branch. This has the additional benefit of the address of the capability (which does not have write permissions) being elsewhere to a function which actually does anything useful. Therefore it is a slight security improvement.

A further improvement is to restrict the capability to be bound only to the address range of the library itself. If the PCC was used, this would include the range of the entire application which would mean the compartment could in theory access anywhere in the capability manager. By restricting to only the address range of the compartment's loaded shared object library in memory it means that there is this added security benefit.

NOTE: *It is not possible to set the bounds to only the trampoline function. This is because the next step in the calling process is to resolve a capability from the PLT of the loaded shared object. The accessor for this PLT capability uses the ADRP instruction which derives an address from PCC. If PCC at that point did not include the PLT in its addressing range then a fault would occur due to an invalid capability tag.*

In theory it could be possible to resolve a capability bounds to be limited to the library's PLT, but this is a future work item that would be non-trivial to achieve.

The concept of the trampoline is shown below (PLT redirect shown also):



4.4 Proxy Functions

Where it is necessary to write user-API specific code to support the framework, this should be largely "boiler-plate" for each API function. Such code would lend itself to code-generation tools in the future.

The objective is then that a proxy API function will be called in the capability manager, with the same function declaration as the real API function implemented in the compartment, which the capability manager SW appears to invoke directly. However under the hood it will actually transit to the compartment and then call the API implementation for real. The framework should then handle everything in the middle layers, and it should be made as simple as possible to support a new function which may be added to the API in the future.

In our WAMR case there were in excess of 40 functions in the exported API, and many of these are needed for a basic program, so clearly it was required to be minimal effort to support the full API. Additionally it is quite feasible that functions will change and more will be added to the API in the future – therefore the framework should be flexible enough to support this.

To this end, a proxy class is provided in the capability manager. In the code example provided it is called ***CApiProxy.*** Any API function can then be called by calling a member of this proxy, for example for the WAMR function *wasm_runtime_create_exec_env()* we may have:

```
exec_env = m_proxy.wasm_runtime_create_exec_env(m_module_inst, STACK_SIZE)
```

which will end up, in the compartment, calling the actual implementation:

```
wasm_exec_env_t wasm_runtime_create_exec_env(wasm_module_inst_t module_inst,
                                              uint32_t stack_size);
```

The names are the same, and the signatures are the same.

In a high-level programming language, this would of course be easy to do and it is no different in modern C++. Unfortunately when we cross the boundary between capability manager and compartment, and change PE state, things become much more complicated and so a hybrid solution is created which needs a little additional work to add any new API function to the mechanism.

4.4.1 Implementation of Solution

The *CCompartmentData* class was mentioned previously, along with the need to have a derived class which provides the specific argument data for the API function being called. This allows the handling of capabilities and functionality needed for the general compartment framework that unwraps arguments and calls the underlying API function to be in the base class, whilst the actual arguments (of varying type and quantity) is I the derived *CCompartmentData* class.

To add support for a new WAMR function, a derived *CCompartmentData* class is added which simply stores arguments passed to its constructor in the object instance. For example, here is a possible implementation of a class for the *wasm_runtime_create_exec_env()* function shown above:

```
class CWasmCallRuntimeCreateExecEnvData : public CCompartmentData
{
public:
    wasm_module_inst_t module_inst;
    uint32_t stack_size;

public:
    CWasmCallRuntimeCreateExecEnvData(
        wasm_module_inst_t module_inst_,
        uint32_t stack_size_
    ) : CCompartmentData(WAMRCall_callCreateExecEnv),
        module_inst(module_inst_),
        stack_size(stack_size_)
    {}
};
```

Note that due to the static casting in the compartment it is necessary to pass some ID to define the underlying data type and WAMR function being called (in this case *WAMRCall_callCreateExecEnv*, which is just an enum (not a class enum) value).

This class is used by creating a new instance of it, passing needed arguments, and obtaining a *std::shared_ptr* to this. This is then passed to a function in the capability manager which expects a *CCompartmentData* pointer, i.e a base class pointer. This is the function which is responsible for finishing the setup of the data structure and transferring the object to the compartment.

Inside the compartment the standard unwrapping is done, and we cast the base class pointer to get the actual derived class type, and then extract arguments to call into the actual underlying API function implementation.

The final step is to add a little more boilerplate in the capability manager in order to avoid lots of copy-pasting of code, as the proxy function will essentially just need to forward arguments to the *CCompartmentData* class function. Therefore we use a *variadic template function* for the proxy function. Again using the example from above, the code looks like this:

```
template <typename T, typename... Args>
uintptr_t CallWamrFn(const std::string& fn_name, Args&&... args)
{
  return m_compartment.CallCompartmentFunction(fn_name,
              std::make_shared<T>(std::forward<Args>(args)...)
  );
}

template <typename... Args>
wasm_exec_env_t wasm_runtime_create_exec_env(Args&&... args)
{
  return (wasm_exec_env_t)CallWamrFn<CWasmCallRuntimeCreateExecEnvData>
        (__func__, std::forward<Args>(args)...);
}
```

Here, the *wasm_runtime_create_exec_env()* passes its function name (i.e the string "wasm_runtime_create_exec_env", resolved from __func__) and passed in arguments to *CallWamrFn()*. *CallWamrFn()* creates an instance of a *CCompartmentData* child class (in this case, *CWasmCallRuntimeCreateExecEnvData*) and calls a function called *CallCompartmentFunction()* to finish building the data structures and transfer control to the compartment.

As part of this step, we will resolve the symbol named "wasm_runtime_create_exec_env" inside the shared object library, and then pass a capability to this symbol along with all argument data to the compartment, where casting will give the correct *CCompartmentData* derived object pointer from which the actual arguments can be obtained.

When creating the proxy function it is only necessary to ensure the function name matches that of the plain C API function implementation, and that the correct *CCompartmentData* class has been used.

Note also that a return argument is always passed back as *uintptr_t* to ensure any size of value can be returned. Only at the top-level is the cast performed to the actual desired API function return type.

An additional point to note, demonstrated by the above example, is the creation of data structures within the compartment which are then passed back to the capability manager. In this case the data structure is only needed by the compartment, so is essentially an opaque handle within the capability manager – it will be passed as an input argument to other functions that run in the compartment. This suggests the data could be sealed, however as the capability manager is the "overseer" there is not really an added benefit in doing this.

In future, these data structures will be needed in the capability manager as well.

5 Compartment Callback to Capability Manager Services

The compartment needs to be able to call services in the capability manager as part of its normal flow, for example to allocate memory on its heap or print a message to *stdout*. Anything which involves a system call or native code cannot be performed in the compartment, as it runs in *Restricted* state.

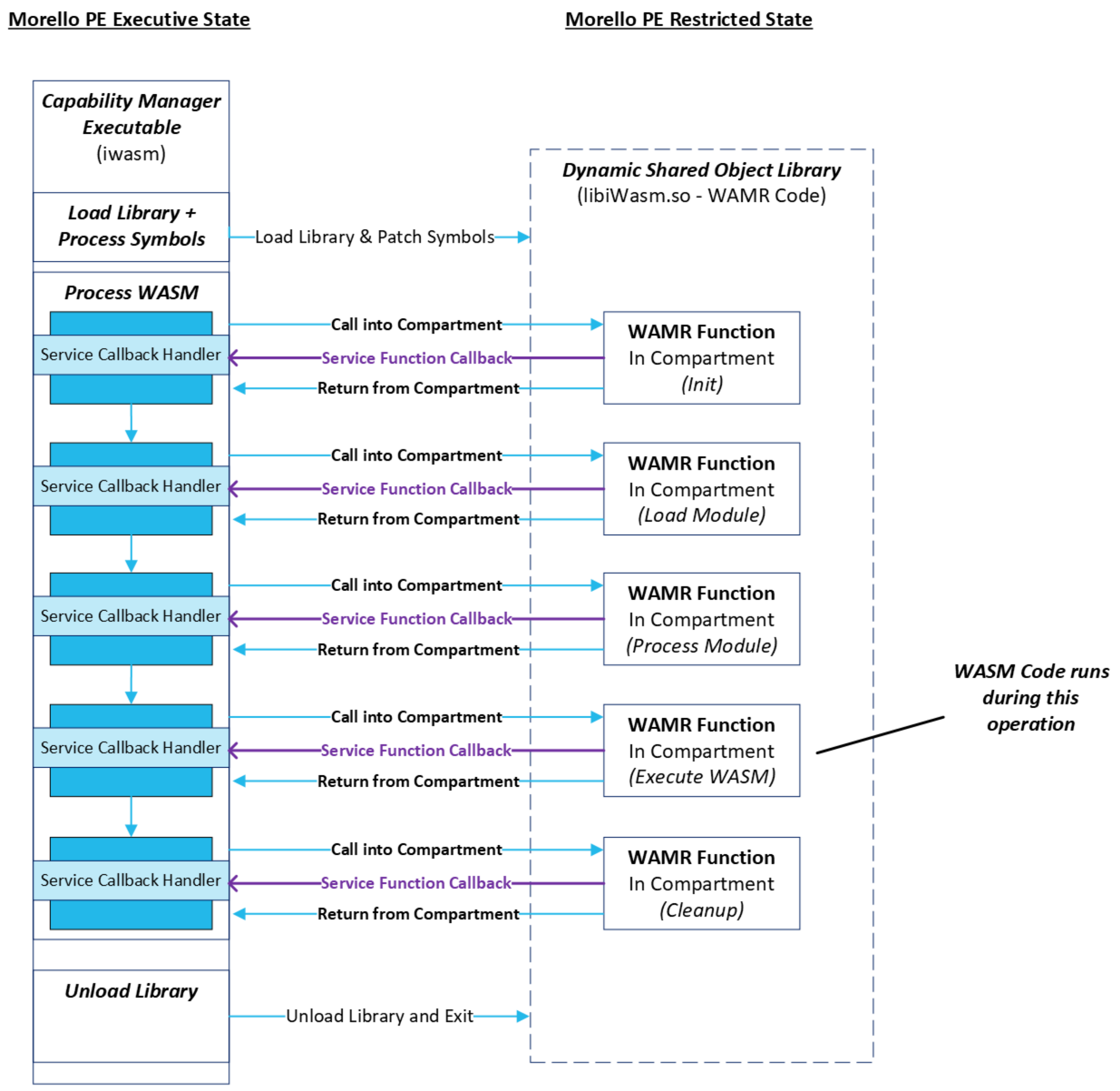
A generalised mechanism is therefore required to make this possible.

This can be achieved using the same ASM marshalling code that is used to switch from the capability manager executable into a compartment, however a modification is needed because there is no compartment CSP/DDC/CTPIDR that needs to be switched in - there is only one capability manager executable, and so only one main stack etc.

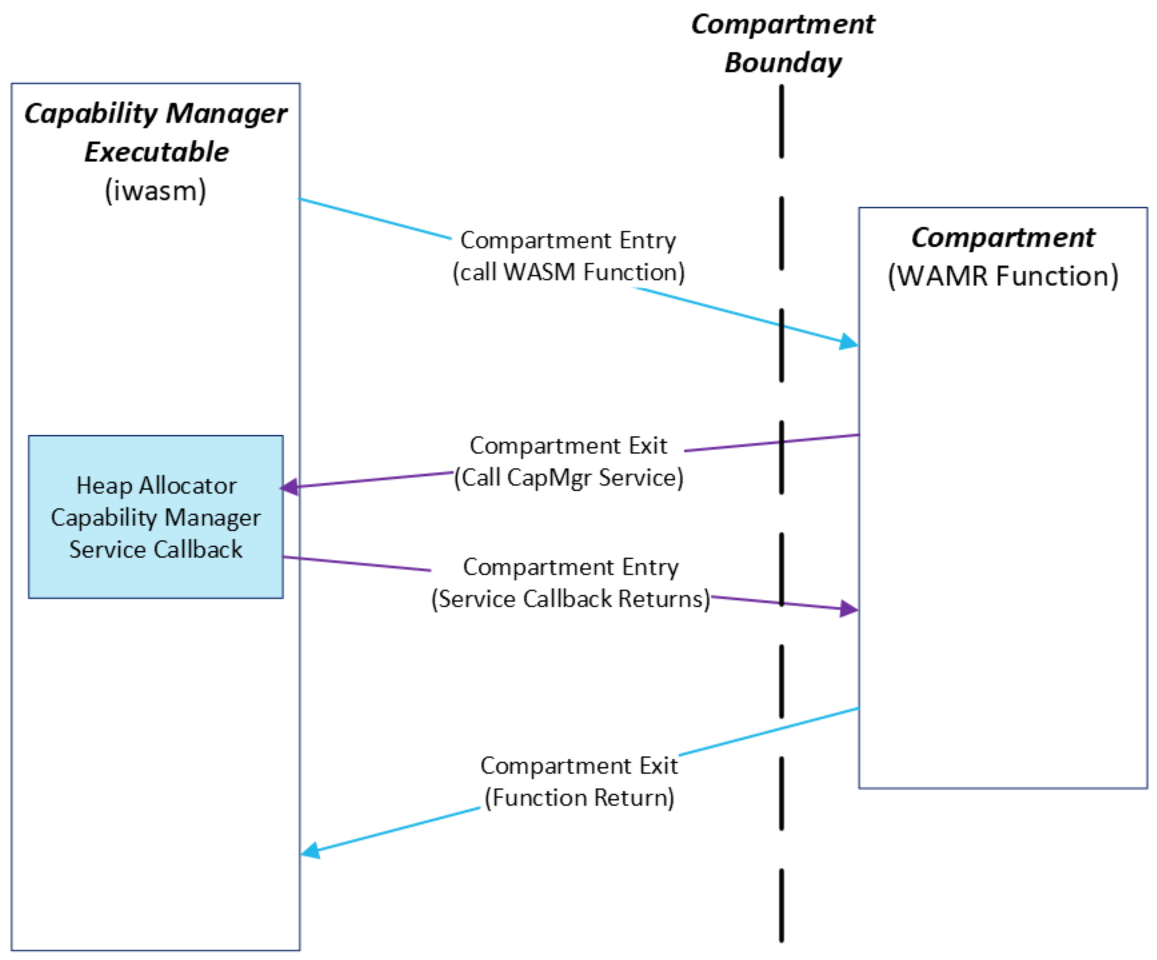
NOTE: *The same mechanism *could* allow a transit from one compartment to another, however it is expected that all calls are "overseen" by the Capability Manager, and therefore the Capability Manager is always directly involved in switching between compartments.*

5.1 Service Callback Flow

The overall flow of how an operation in a compartment can then callback to services in the capability manager, as part of its processing before it completes, is illustrated below. Here is a WAMR example which involves the capability manager executable (called "iwasm") loading the WAMR code inside a shared object library and then for each WAMR API operation the compartment is entered. The compartment operation itself makes use of system resources, and so calls back to run a capability manager service:


Morello PE Executive State
Morello PE Restricted State
Capability Manager Executable
(iwasm)
Load Library + Process Symbols
Load Library & Patch Symbols
Dynamic Shared Object Library
(libiWasm.so - WAMR Code)
Process WASM
Service Callback Handler
Call into Compartment
Service Function Callback
Return from Compartment
WAMR Function In Compartment (Init)
WAMR Function In Compartment (Load Module)
WAMR Function In Compartment (Process Module)
WAMR Function In Compartment (Execute WASM)
WAMR Function In Compartment (Cleanup)
WASM Code runs during this operation
Unload Library
Unload Library and Exit

If we examine the an individual operation in more detail, the sequence occurs like this (shown for the example of allocating heap memory):


Compartment Bounday
Capability Manager Executable (iwasm)
Compartment (WAMR Function)
Compartment Entry (call WASM Function)
Compartment Exit (Call CapMgr Service)
Heap Allocator Capability Manager Service Callback
Compartment Entry (Service Callback Returns)
Compartment Exit (Function Return)

The compartment entry and exit points are shown, which would be abstracted to a function call and then return. But *within* this processing we have an exit back to the capability manager to call a specific identified service, which itself then returns. This indicates that two more capabilities are needed to be passed to the Compartment during the initial entry:

- Exit point to callback a service
- Specific function to call which implements a service in the capability manager

The service callback does not interfere with the original compartment call because the calling mechanism in the capability manager saves stack states, so we return to the original compartment processing as if any other function were being called.

The mechanism implements a *proxy* within the compartment that will actually call into the capability manager to carry out the real implementation of the function.

This is the same mechanism used to call functions inside the compartment, and the classes and data structures for service callbacks are analogous to the compartment calls themselves.

5.2 Service Callback Processing

The same assembly function inside the Capability Manager that is called when entering the compartment is used for calling a service callback function – this entry point is called *CompartmentSwitchEntry()*. This is entered via the same C calling mechanism - a C Compartment Caller function is implemented in the compartment in the same way as it is implemented in the capability manager.

To this end, the same data structures are used. However, the call from the Compartment to the Capability Manager does not need to switch in restricted parameters for stack etc. (RCSP/RDDC/ RCTPIDR) because the capability manager must use the executive (main) stack. Therefore, these are set to Null by the compartment's calling code as they will be unused.

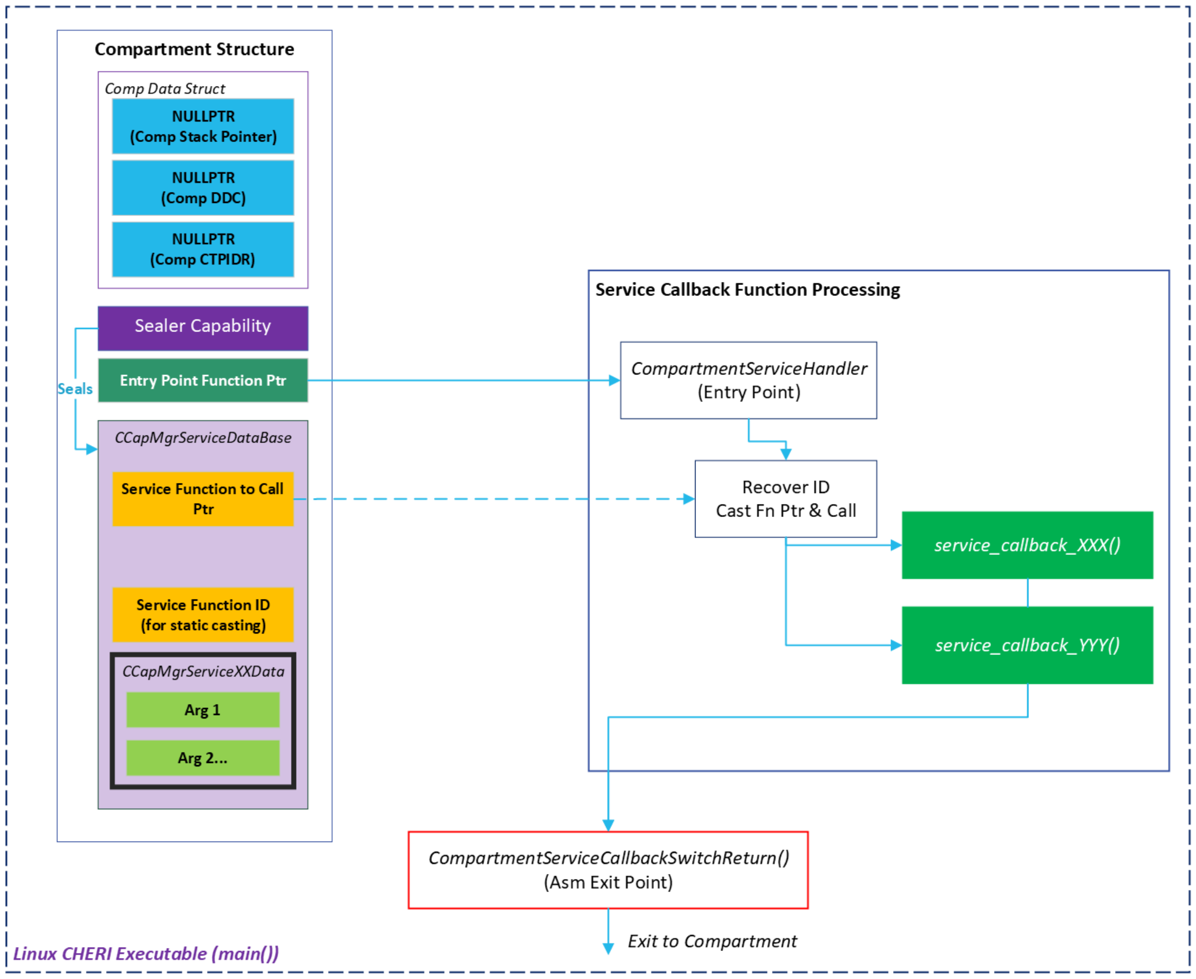
When the capability manager switches to a compartment, as we have seen already a *CCompartmentData* derived data structure object is used to provide function-specific arguments along with various capabilities needed such as the compartment exit point back to capability manager. The same concepts are used for the service callback, however the base data class is slightly different and so we instead have *CCapMgrServiceData* in lieu of *CCompartmentData*. (In theory there could have been a common base class for both of these, and a code improvement should make it so).

The differences for the service callback vs the compartment call are as follows:

- No exit function pointer for the service callback exit function is required. This is because it is itself in the capability manager, and therefore known to the capability manager
- This exit function is not the same as used for the compartment → capability manager returning, the reason for this is explained further below
- The "function to call" is a function within the capability manager, instead of a function within the compartment
    - This is still supplied by the compartment, the reasons for which are explained further in the remainder of this section
- The ID for static function pointer casting is that of a service callback function, and is analogous to the function ID used in calling capability manager → compartment function call

We can now present the data structures used in service callback processing. Previously when calling the compartment was described earlier in this document, some of the additional capabilities used to enable service callbacks were omitted, to avoid overcomplicating the explanation.

Structures are shown below:


Compartment Structure
Comp Data Struct
NULLPTR (Comp Stack Pointer)
NULLPTR (Comp DDC)
NULLPTR (Comp CTPIDR)
Sealer Capability
Seals
Entry Point Function Ptr
CCapMgrServiceDataBase
Service Function to Call Ptr
Service Function ID (for static casting)
CCapMgrServiceXXData
Arg 1
Arg 2...
Service Callback Function Processing
CompartmentServiceHandler (Entry Point)
Recover ID Cast Fn Ptr & Call
service_callback_XXX()
service_callback_YYY()
CompartmentServiceCallbackSwitchReturn() (Asm Exit Point)
Exit to Compartment
Linux CHERI Executable (main())

The compartment structure is that passed *from* the compartment *to* the capability manager assembly code. We have the values to be used as the new CSP, DDC and CTPIDR but note these are all Null. This is because they are not needed as there is no restricted state to switch in – the result is to set restricted versions of the register to Null. This is a good security enhancement as if for some reason we did accidentally attempt to switch to the restricted state then a fault would occur.

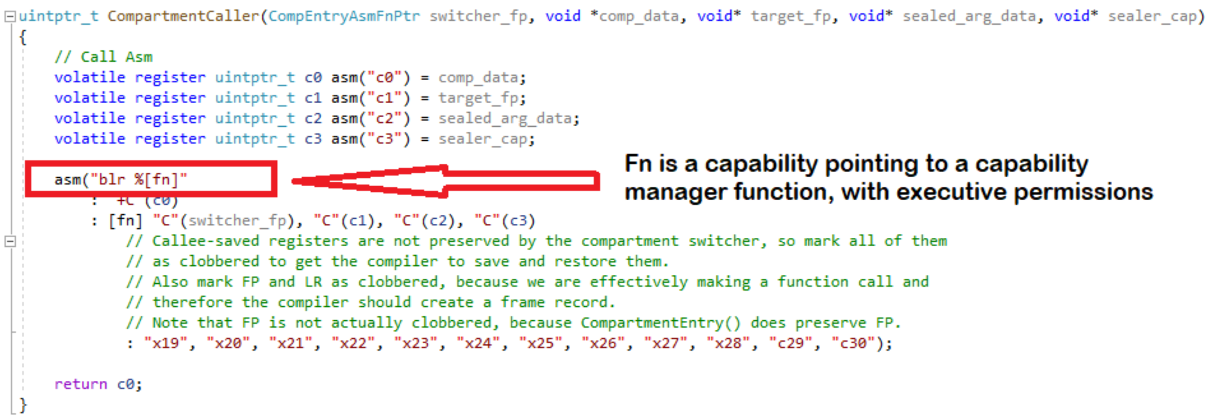
The sealer capability is the same one used when calling into the compartment – it seals the callback function data arguments, static function ID and the individual function's entry point capability. The main entry point is the unwrapper for the service callback mechanism, in this case *CompartmentServiceHandler*. This *CompartmentServiceHandler* function determines which underlying "do something" function needs to actually be called and uses the passed in capability to call it.

Note that no capability needs to be passed to provide the exit point back to the compartment. This is already known to the capability manager, and it is more secure to let the capability manager decide this for itself.

5.2.1 PE State Switching

The compartment runs in restricted state, therefore there is a switch to the executive state to execute the service callback function, and a switch back to restricted during the return.

Switching from restricted -> executive state in Morello does not require a special instruction, unlike executive -> restricted. It actually occurs during common C code inside the compartment, when a direct branch (not a function call) is made to the call back entry point (*CompartmentSwitchEntry*). The snippet of code which does this is shown below:



```
uintptr_t CompartmentCaller(CompEntryAsmFnPtr switcher_fp, void *comp_data, void* target_fp, void* sealed_arg_data, void* sealer_cap)
{
    // Call Asm
    volatile register uintptr_t c0 asm("c0") = comp_data;
    volatile register uintptr_t c1 asm("c1") = target_fp;
    volatile register uintptr_t c2 asm("c2") = sealed_arg_data;
    volatile register uintptr_t c3 asm("c3") = sealer_cap;

    asm("blr %[fn]"
        : "+C" (c0)
        : [fn] "C"(switcher_fp), "C"(c1), "C"(c2), "C"(c3)
            // Callee-saved registers are not preserved by the compartment switcher, so mark all of them
            // as clobbered to get the compiler to save and restore them.
            // Also mark FP and LR as clobbered, because we are effectively making a function call and
            // therefore the compiler should create a frame record.
            // Note that FP is not actually clobbered, because CompartmentEntry() does preserve FP.
            : "x19", "x20", "x21", "x22", "x23", "x24", "x25", "x26", "x27", "x28", "c29", "c30");

    return c0;
}
```

Fn is a capability pointing to a capability manager function, with executive permissions

This same code is compiled in both the capability manager and the compartment. In the compartment, the branch will perform a PE state switch since the target capability has executive permissions and source does not; in the capability manager the state switch is not necessary.

5.3 Service Callback Return

When the compartment returns back to the capability manager it calls an (executive) assembly code function in the capability manager which returns back to the point at which the compartment was called.

The same method is used to return from a capability manager service function; an assembly function is called. However as execution is already in the capability manager, the function can be referenced directly. That function in assembly code is called *CompartmentServiceCallbackSwitchReturn()* and the argument to that function is the return value, which is always a full sized capability and passed as a *uintptr_t.*

It should be noted that here there is a slight difference in the assembly code between returning *from* a compartment and returning *to* a compartment from a service callback. The reason relates to the way stack frames work in C and the fact that service callback functions utilise the main stack within the capability manager.

In the case of the compartment -> capability manager return (not related to service callbacks), there is a function call to an exit function from within an existing C function that itself never actually returns. This means the stack frame for this function is never unwound.

This though is not a problem, because at this point we are using the *restricted* stack. Calling into the capability manager immediately switches to the executive PE state which then switches back to the main executive stack - this main stack was not used while we processed the compartment and therefore the main (executive) stack pointer is set up correctly to return at this point. Before the next compartment entry the original top-of-stack is restored and so the correct stack frame hierarchy is restored automatically. Essentially, after we are finished with the compartment its stack can be left dangling.

5.3.1 Service Callback Stack Frame

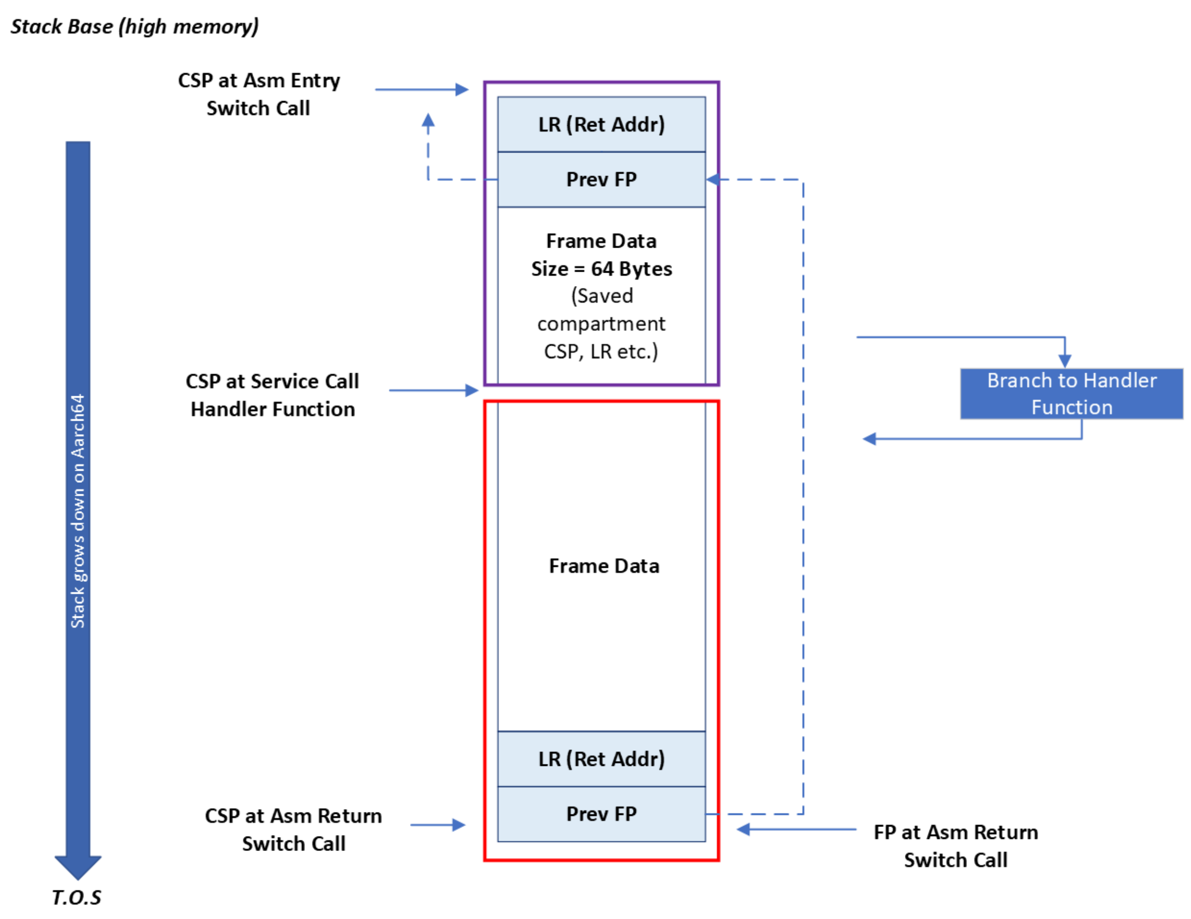
For the capability manager service function things are different because the main (executive) stack is being used throughout. Therefore we cannot leave this "dangling" when we switch back to the compartment, as we'll need it again when the overall compartment operation returns (in fact we need to retrieve the return address from the main stack). We therefore need to fix this stack when we make the service callback exit call, otherwise it will be corrupted.

One approach to this problem would be to use a separate, bespoke, stack for the capability manager service functions which would then isolate the main stack. However this is difficult to achieve because unlike the restricted stack we would not be able to cleanly switch between the two when the PE state changes, and also we would have to manage the stack pointer for our "extra" stack.

A simpler solution is to manually unwind the stack frames during the capability manager service function exit processing. We mandate that the assembly code *CompartmentServiceCallbackSwitchReturn()* function is only ever called from the entry point (the *CompartmentServiceHandler()* function) and therefore the unwinding comprises only and always:

1. Precisely one C code frame
2. A "pseudo-frame" saved by the Assembly code *CompartmentSwitchEntry()* function when we first invoked the service callback

The unwinding needed is shown below:


Stack Base (high memory)
CSP at Asm Entry Switch Call
LR (Ret Addr)
Prev FP
Frame Data Size = 64 Bytes (Saved compartment CSP, LR etc.)
CSP at Service Call Handler Function
Branch to Handler Function
Frame Data
Stack grows down on Aarch64
LR (Ret Addr)
Prev FP
CSP at Asm Return Switch Call
FP at Asm Return Switch Call
T.O.S

At the point when *CompartmentServiceCallbackSwitchReturn()* is called the stack is shown as above. The first frame to be unwound is that placed by the compiler when *CompartmentServiceHandler()* was called. It is unwound in the standard way, which on Morello Aarch64 is to retrieve the last frame's pointer from the location pointed to by the current frame pointer (CFP on Morello).

However, we cannot then do *new CSP = previous CFP* because the frame structure used in our assembly code is different. We need to retrieve the stack pointer at the point we called *CompartmentServiceHandler()*. From the diagram above, it can be seen that unlike the compiler, we have placed the frame data *after* the location of the FP on the stack. Fortunately the size of that data is known since we implemented it ourselves and therefore we simply need subtract this size from the location of the frame pointer to get correct address. (It is a subtract, not an add, because on Aarch64 - like many architectures - the stack grows downwards in memory).

Having now got the correct address, we have restored the stack state to how it was when we called the capability manager service handler function. From then on, the rest of the processing is exactly identical to the way we exit the compartment operation back to the capability manager and so we already have some code to do that.

The excerpt from the assembly code shows this additional processing needed in *CompartmentServiceCallbackSwitchReturn()* before the "exit from compartment to capability manager" common bit in *CompartmentSwitchReturn()* is run:

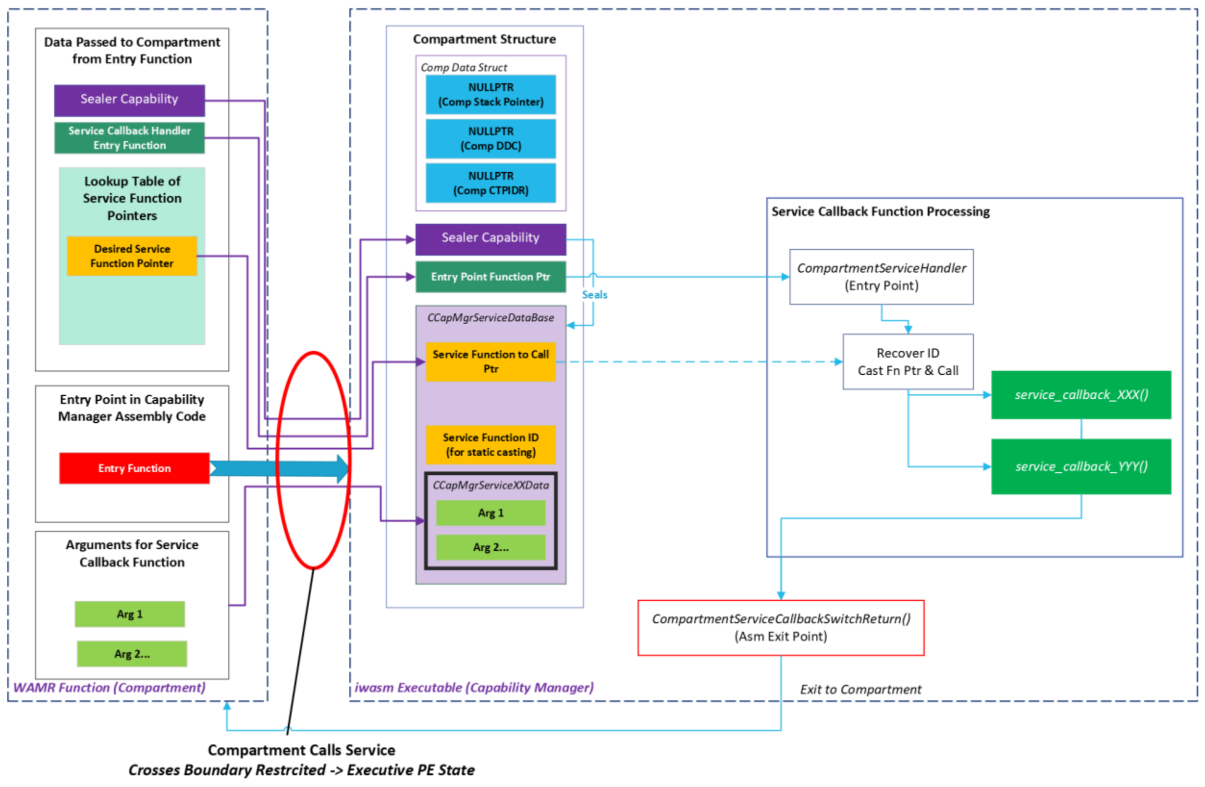
```
       .globl CompartmentServiceCallbackSwitchReturn
CompartmentServiceCallbackSwitchReturn:
       // Capability manager service return to compartment
       // Remove the last stack Frame
       // Note our "frame" is arranged differently to the compiler's
       ldr frame_ptr, [frame_ptr]
       sub stack_ptr, frame_ptr, #COMPDATA_STRUCT_SIZE

       // Everything else is same as CompartmentExit
       b CompartmentSwitchReturn
```

Note: *"frame_ptr" is an alias for CFP, i.e C29 register, and "stack_ptr" is an alias for the CSP register.*

5.4 Additional Compartment Entry Capabilities

Clearly some capabilities needed to manage the service callback mechanism need to be passed to the compartment from the capability manager in the first place. This is done at compartment entry time.

Previously we omitted these from architecture diagrams for reasons of brevity, but the below diagram shows in more detail what is needed – this is the same WAMR example diagram shown before, but with our new capabilities added:



Data Passed to Compartment from Entry Function
Sealer Capability
Service Callback Handler Entry Function
Lookup Table of Service Function Pointers
Desired Service Function Pointer
Entry Point in Capability Manager Assembly Code
Entry Function
Arguments for Service Callback Function
Arg 1
Arg 2...
WAMR Function (Compartment)
Compartment Structure
Comp Data Struct
NULLPTR (Comp Stack Pointer)
NULLPTR (Comp DDC)
NULLPTR (Comp CTPIDR)
Sealer Capability
Entry Point Function Ptr
Seals
CCapMgrServiceDataBase
Service Function to Call Ptr
Service Function ID (for static casting)
CCapMgrServiceXXData
Arg 1
Arg 2...
iwasm Executable (Capability Manager)
Service Callback Function Processing
CompartmentServiceHandler (Entry Point)
Recover ID Cast Fn Ptr & Call
service_callback_XXX()
service_callback_YYY()
CompartmentServiceCallbackSwitchReturn() (Asm Exit Point)
Exit to Compartment
Compartment Calls Service
Crosses Boundary Restrcited -> Executive PE State

On the left hand side we see the compartment making a service callback, with the red ellipse showing the crossing of the boundary back to the capability manager and the transfer of the "compartment structure" block.

The following fields are sourced from the capability manager, passed in the call to process the compartmentalised operation in the first place:

1. Sealer capability
2. Capability for Callback entry function in the capability manager (*CompartmentSwitchEntry())*
3. Capability for Service handler function to call *(CompartmentServiceHandler())*
4. Lookup table of capabilities for individual service call functions (e.g *service_callback_XXXX()* in the diagram)
    - This allows the compartment operation to select the service callback function it needs to make, and grab the capability for it from the table

More detail on each of these is now provided.

5.4.1 Sealing Capability

The sealing capability is the same as the one used by the capability manager to seal data to the compartment. It is provided as an attribute to the compartment, so it can be used for data being sent to a service callback.

The concept is that each compartment will be given its own sealer capability, which is unique to each compartment, for maximum security.

NOTE: Future work should remove passing this sealer capability back from the compartment during a service callback; instead the capability manager should resolve it by looking up compartment data when given the ID of a compartment. This requires additional implementation to add all compartment data to a <compartment_ID> keyed map, which can then be examined when the service callback is made. This work has not yet been implemented due to project timescale constraints.

5.4.2 CompartmentSwitchEntry Calling Point

This is the top-level entry point into the capability manager's assembly code involved in switching PE state. It is exactly the same as used when calling the compartment.

5.4.3 Service Handler Calling Point

This *CompartmentServiceHandler* function is the unwrapping function that marshals any service callback. It is called from the assembly code in the compartment manager which handled the state switch.

This function could be resolved directly by the capability manager as it is known to the capability manager. However passing it across the boundary and back again:

1. Permits the same assembly code to be used as calling into a compartment

2. Gives the framework maximum flexibility (e.g in future a different entry point could be used for different compartment types)
3. Makes it easier to support compartment -> compartment transitions in future (should the need arise, although this is not expected to be required)

5.4.4 Individual Service Function to Call

The actual service function to be called in the capability manager is determined by the compartment: the compartment calls whatever service it needs from the available services. For example, currently implemented are service functions to allocate general memory, free memory and allocate a linear memory buffer.

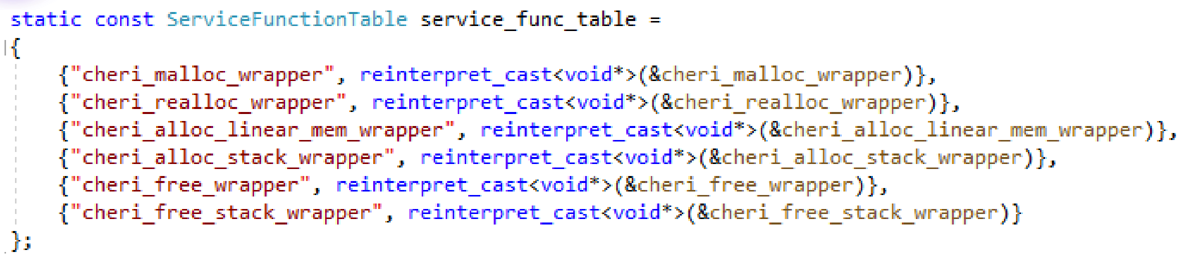
The actual service function is implemented in the capability manager and therefore the compartment either needs a capability for the function or a mechanism to identify the function. In the latter case, the resolution must then be done in the capability manager.

The implementation as written provides the compartment with the capabilities themselves so that resolution to which capability function pointer is used is done within the compartment. Resolution is done automatically using the name of the function as a key to a map to derive the correct function pointer capability. This is more automated, because the name of the function can be derived from the function call at compile time and then used to lookup the actual function pointer (this is a level of reflection that is not readily available due to the fact we are calling a function in a different PE state).

The mechanism is handled as follows:

1. Along with the rest of the *CompartmentData* attributes, the capability manager passes the compartment a reference to a map of string -> capability function pointers, for each service callback function
2. When a service callback function proxy is called in the compartment then the name of the function is looked up in the map to resolve the function pointer capability
3. This is passed to the capability manager and during the *CompartmentServiceHandler* unwrapping operation it is then called directly
    a. Note: A corresponding identifier for the function is also resolved by the proxy because static casting is used for service callback unwrapping as well (RTTI will not easily work due to the switch of PE state)

A snapshot of an example function pointer map is shown below, as defined in the capability manager. This shows a number of different functions related to allocating and freeing memory and other data structures:



```
static const ServiceFunctionTable service_func_table =
{
    {"cheri_malloc_wrapper", reinterpret_cast<void*>(&cheri_malloc_wrapper)},
    {"cheri_realloc_wrapper", reinterpret_cast<void*>(&cheri_realloc_wrapper)},
    {"cheri_alloc_linear_mem_wrapper", reinterpret_cast<void*>(&cheri_alloc_linear_mem_wrapper)},
    {"cheri_alloc_stack_wrapper", reinterpret_cast<void*>(&cheri_alloc_stack_wrapper)},
    {"cheri_free_wrapper", reinterpret_cast<void*>(&cheri_free_wrapper)},
    {"cheri_free_stack_wrapper", reinterpret_cast<void*>(&cheri_free_stack_wrapper)}
};
```

5.4.4.1 Justification for Implementation

It should be explained why the compartment is provided with a capability function pointer map of service callbacks – allowing the compartment to provide the capability function pointer itself – instead of an identifier to keep all capabilities inside the capability manager. It may seem a security risk to do it this way, but because the capabilities are sealed entry (sentry) with restricted bounds the risk is minimal, and the benefits outweigh the risk. They are as follows:

Localisation of Error Handling

If the function by name cannot be resolved to a capability, then this is handled in the compartment *using* information sourced from the capability manager. This avoids an unnecessary transition to the capability manager and a bespoke error code, which would anyway need handling in the compartment

Greater Flexibility for potential Future Code Improvements

Ideally, a mechanism such as *std::bind()* would be implemented to bind all service function callback arguments and a function pointer during the compartment itself. This would then require all the capability function pointers to be provided to the compartment. The framework has, therefore, been implemented in a way that makes it easier to extend in the future.

NOTE: *At this time, it has not been possible to implement std::bind() because the transfer across the compartment boundary causes a problem with the internal class object data and the virtual function tables. However with more study and support in the standard C++ library then this may be achievable in future.*

5.5 Invoking a Service Callback Function

Like calling compartmentalised API operations, service callbacks are invoked by calling a proxy function in a compartment service callback proxy class. However in our WAMR use case the codebase is largely written in C, and so although the proxy is implemented in C++ we should support a C interface to it.

This is achieved by instantiating a single instance of a proxy function class *CServiceCallProxy* when the compartment is first entered via starting a compartmentalised API operation. This instance remains active until the compartment finally exits and so its lifetime is the time the compartment function is running for. All service callbacks then use this proxy instance.

To provide access to the single proxy instance from any function, a static *getter* is provided, the code snippet is shown below:

```
// Global CServiceCallProxy ptr which is set to the single CServiceCallProxy on
each compartment call
static std::unique_ptr<CServiceCallProxy> g_service_call_proxy;

// Accessor for user classes
CServiceCallProxy *CServiceCallProxy::GetInstance()
{
    return g_service_call_proxy.get();
}
```

Using the service function can then be done as follows (example shown) - and note this is callable from C:

```
extern "C" void* cheri_malloc_wrapper(size_t size)
{
    return CServiceCallProxy::GetInstance()->cheri_malloc_wrapper(size);
}
```

5.5.1 Variadic Templates

Like the API operation proxies for the compartment, the service callback proxy functions also utilise C++ variadic templates to provide a level of boilerplate code.

Considering a *cheri_malloc_wrapper()* function as an example, the following proxy is implemented in the Compartment's proxy function (*CServiceCallProxy::cheri_malloc_wrapper()*):

```
template <typename... Args>
void *cheri_malloc_wrapper(Args&&... args)
{
  return (void*)CallServiceFn<CCheriMallocCapMgrServiceData>
              (__func__, std::forward<Args>(args)...);
}
```

This variadic template resolves arguments as supplied to the proxy, and calls a common function supplying the string "cheri_malloc_wrapper" as the first argument:

```
template <typename T, typename... Args>
uintptr_t CallServiceFn(const std::string& fn_name, Args&&... args)
{
  return CompartmentServiceCallback(fn_name,
       std::make_shared<T>(std::forward<Args>(args)...)
  );
}
```

This function creates the service callback data structure (in this case an instance of *CCheriMallocCapMgrServiceData*) and calls *CompartmentServiceCallback()* to actually transition into the capability manager. Before doing that, *CompartmentServiceCallback()*:

- Resolves the string "cheri_malloc_wrapper" to the corresponding capability function pointer (assuming it could be found in the map)

- Completes the *CCheriMallocCapMgrServiceData* object instance by adding the service handler capability function pointer and providing the other needed arguments to the assembly function call, namely:
    - The sealer capability
    - The map-looked-up service function pointer to call, from above
    - The RCSP/RDDC/RCTPIDR structure (all fields set to NULL as not required)

The *CCheriMallocCapMgrServiceData* object instance pointer is itself sealed and passed as a function argument.

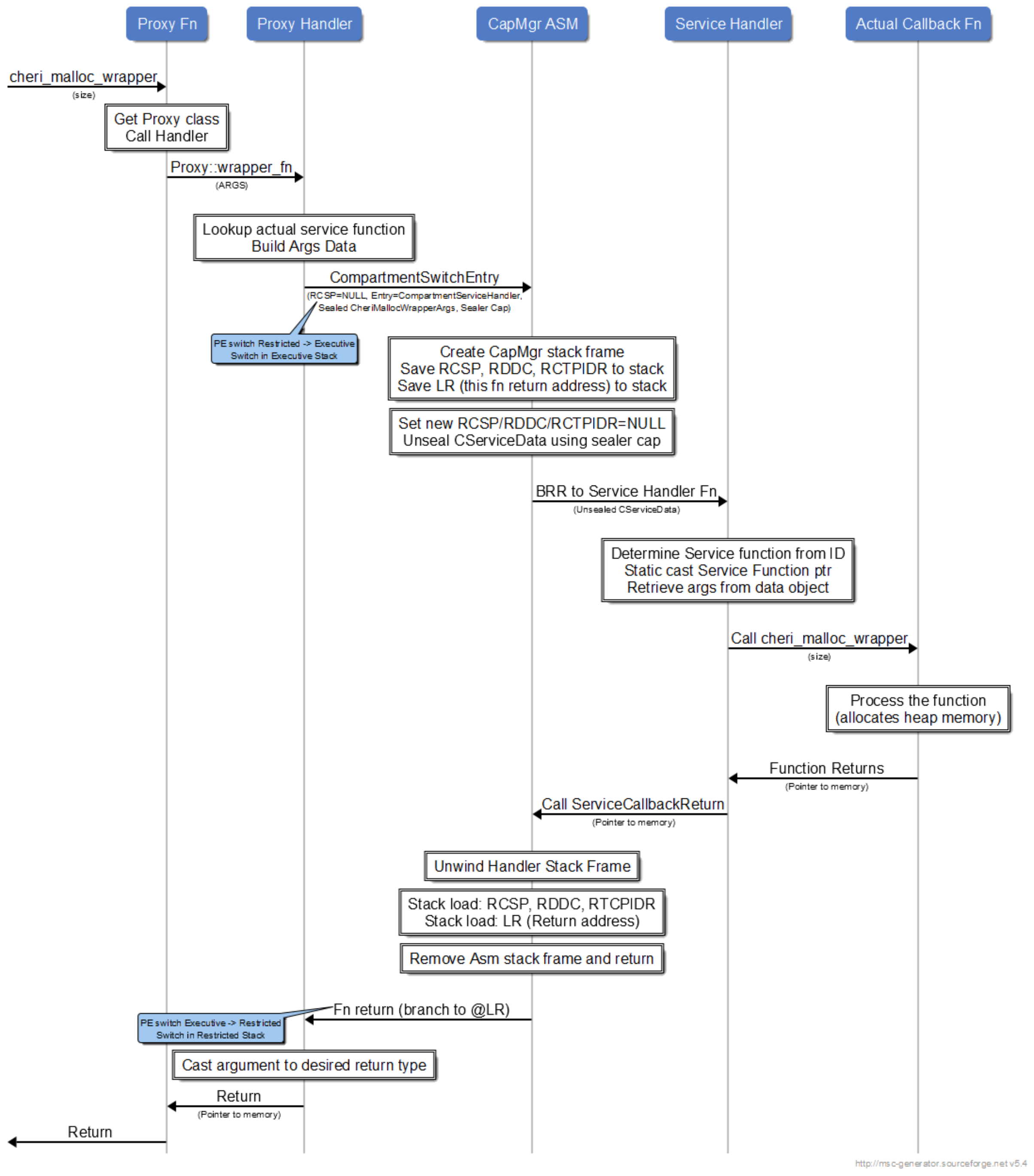
The final call which actually switches into the capability manager's assembly function (and the executive state) is *CompartmentCaller()*, a C function implemented in all compartments and capability manager. The arguments to this *CompartmentCaller()* is then as follows:

```
auto ret = CompartmentCaller(
    m_compartment_data->service_callback_entry_fp, /* CapMgr entry fn in Asm */
    &comp_data_nulls,                              /* Stack etc. to use (n/a) */
    reinterpret_cast<void*>(
       m_compartment_data->capmgr_service_fp
       ),      /* CapMgr service handler called from Asm */
    sealed,    /* Sealed argument data including the underlying function to call */
    m_compartment_data->sealer_cap                 /* Capability used to seal */
);
```

5.5.2 Service Callback Functional Flow

The MSC below shows the flow through the system for a *cheri_malloc_wrapper()* example. This demonstrates the flow through the service callback framework and notes the points where the PE state switches:

Service Callback Function Flow



Proxy Fn
Proxy Handler
CapMgr ASM
Service Handler
Actual Callback Fn
cheri_malloc_wrapper
(size)
Get Proxy class
Call Handler
Proxy::wrapper_fn
(ARGS)
Lookup actual service function
Build Args Data
CompartmentSwitchEntry
(RCSP=NULL, Entry=CompartmentServiceHandler,
Sealed CheriMallocWrapperArgs, Sealer Cap)
PE switch Restricted -> Executive
Switch in Executive Stack
Create CapMgr stack frame
Save RCSP, RDDC, RCTPIDR to stack
Save LR (this fn return address) to stack
Set new RCSP/RDDC/RCTPIDR=NULL
Unseal CServiceData using sealer cap
BRR to Service Handler Fn
(Unsealed CServiceData)
Determine Service function from ID
Static cast Service Function ptr
Retrieve args from data object
Call cheri_malloc_wrapper
(size)
Process the function
(allocates heap memory)
Function Returns
(Pointer to memory)
Call ServiceCallbackReturn
(Pointer to memory)
Unwind Handler Stack Frame
Stack load: RCSP, RDDC, RTCPIDR
Stack load: LR (Return address)
Remove Asm stack frame and return
Fn return (branch to @LR)
PE switch Executive -> Restricted
Switch in Restricted Stack
Cast argument to desired return type
Return
(Pointer to memory)
Return
http://msc-generator.sourceforge.net v5.4