LA-UR-23-28874

Approved for public release; distribution is unlimited.

Title: seL4 Notes

Author(s): Lara, Brandon Garrett

Intended for: Report

Issued: 2023-08-02 (Draft)

seL4 Notes
Brandon Lara

Helpful Links:

https://rbg.systems/

Questions:

#1
In the academic paper, klein writes this in relation to schedulers:

> "Here the solution was a redesign of scheduling, dubbed Benno scheduling [Blackham et al. 2012], which has the same average-case behavior while avoiding the pathological worst case of lazy scheduling: When a thread unblocks during IPC, it is not immediately entered into the ready queue (as it may soon block again). Like lazy scheduling, this avoids queue manipulations during IPC, but now the cleanup is simple and O(1): when the time slice expires, only the preempted thread may have to be moved (from an endpoint waiting queue to the ready queue)."

This should not work. If a thread is blocked and in the waiting queue, it cannot block again until some amount of execution time is devoted to it to move it to the next blocking instruction. I am very confused about this, and extremely skeptical that this works as written.

seL4 Overview:

seL4 is a microkernel operating system. Microkernels are operating systems with a different design philosophy than Linux (and I think Windows ?). Linux is a monolithic operating system, meaning that all of the Linux functionality runs in kernel mode. By contrast, seL4, and others like it (see Minix), choose to implement only the bare necessities in kernel mode. Other functionality is handled through "servers" which amount to userland processes. This means that a compromise in the file system is localized. The attacker is still trapped inside the virtual memory and userland constraints enforced by the kernel.

The seL4 kernel provides very few resources to the user. Most of the functionality associated with an operating system is actually in userland processes. What seL4 provides is an enforcement of userland boundaries (virtual memory), dynamic memory allocation (paging), and a message passing system to allow for IPC calls (system calls). These IPC calls provide the functionality users are expecting. In this light, it is probably more helpful to think of seL4 as a hypervisor for userland programs.

Sel4 kernel objects:

System calls to Sel4 are done through kernel objects that hold certain functionality.

1. CNodes - CNodes are objects that hold capability pointers. These pointers provide threads with the ability to call certain functionality. Capability pointers point to kernel objects. These objects export methods for the user to call. Capability pointers can also point to other CNodes. Traversing this linked list gives a DAG of CNodes. The thread has all access rights that it can reach by traversing the DAG. The DAG reachable by the CNode is referred to as the CNode's "CSpace".
2. Thread Control Block (TCB) - Represents the context for a thread. Usually an OS will include register values and scheduler information, along with a lot of OS specific values. The structure is defined here. The TCB's are arranged in a queue (I suspect this is the queue of processes given to the scheduler).
3. Scheduling Contexts - Represents the amount of time a thread is meant to receive from the scheduler. The information contained inside a scheduling context object can be found here. Thread priority is decided by two values: the Maximum Controlled Priority and the priority. The priority is the current scheduling priority. This number will be used to decide how to schedule the process at each tick. The Maximum Controlled Priority is the maximum number the thread can set its priority number to.
4. Endpoints - The capability that facilitates message passing between two threads. They also hold the "grant" right, which allows them to send capabilities between threads as well. Threads communicating on an endpoint will block and wait for a received signal.
5. Reply Objects -
6. Notifications -
7. Virtual Address Space -
8. Interrupts -
9. Untyped Memory -

These kernel objects are created by taking untyped memory and performing a retype command on them, seL4_Untyped_Retype(). The function is generated in python files at build time, found here. The arguments passed to the function determine the type of object created.

Sel4 CNodes:

CNodes are an array of capability pointers. These capability pointers provide functionality to the controlling thread, such as message passing rights. They do this by pointing to kernel objects that export methods for system calls. These pointers are both immutable and come attached with a signature that verifies the access rights. This means that the pointer cannot be corrupted to point to an unallowed object, and the access is also controlled by having the correct signature. CNodes can also hold capability pointers that point to another CNode. This can form a DAG where the thread has access to any capability it can find while traversing the DAG. This chain of CNodes forms the CSpace of any given CNode. Note that this is a DAG or a tree, meaning that CNodes can not traverse back up the DAG to get capabilities, only down.

Because capabilities are pointers, they provide no information about the object they point to. CAmkES ensures that any interaction with the pointer is done through method calls. This means that a curious user should not be able to dereference the pointer to examine the data and reveal the underlying object. When the object receives a system call, it can then determine and process how to interpret the request. This helps prevent invalid (malicious) system calls from getting through the system.

The basic capabilities are:

1. sel4_Yield() - This command yields the cpu back to the scheduler. All userspace programs can do this
2. sel4_Send() - Sends a message to some other object (Question: Can this be CNodes?). If that object is classified as an “endpoint” then that means it represents some other userland thread. All messages to non endpoints are processed by the kernel. Messages to endpoints are handled by the kernel after both threads reach a blocking state waiting for the message to pass.
3. sel4_Recv() - The mirror of sel4_Send(). This function blocks waiting for the sender to call sel4_Send(). Additionally, it can also be used on “notification” objects, which are kernel objects that give signals (Questions: Are these like interrupts?)

More capabilities are available, but they merely act as wrapper functions for some combination of the above three functions, with some niceties added for convenience (such as handling blocking or error cases).

Most definitions of capabilities are represented in .bf files. This is an inhouse file type. In order to translate these .bf definitions into valid C code, a python processor is invoked at build time. This is done with the intent of optimizing the bitfield compilation. The authors did not trust the gcc bitfield implementation to be sound, consistent, and fast, so keeping the structures out of the source files prevents gcc from compiling them.

As mentioned above, Capabilities are placed inside of a CNode array. The CNode array is always indexed and referenced via a raw pointer (seen here in `cptr_t cPtr`). This is almost certainly because the actual array is inside of a .bf file somewhere, and not defined in any .h or .c files (Question: The memory must still be allocated though. Where is that?). This would cause an error if attempting to use array syntax. The dereferencing of the pointer is handled by a supporting function. LookupSlot() is defined here, along with other useful functions to hunt down, but it is not implemented. The implementation is proving tricky to track down.

Sel4 Thread Control Block (TCB):

The thread control block contains all of the information required to schedule a thread for execution. This contains things like the register state.

The reference manual spends a long time talking about the specifics of the scheduling algorithm. The long story short is that threads get scheduled. This is very not important in the scope of this project.

Threads *can* but are not required to have IPC buffers to pass more information. This is important as users because we are responsible for the correct initialization of the TCB's for threads we intend to run. When using CAmkES, this process is taken care of for us, but it is important to understand what falls under the purview of CAmkES vs seL4. One is verified and one is not.

(Note: Finish)

Sel4 Scheduling Contexts Objects (SCO):

Scheduling decisions are primarily based on two values: the budget (b) and period (p). Periods are blocks of clock ticks. An example would be 10 clock ticks. The budget is how many of those clock ticks can be given to the current thread. If a period is 10 clock ticks and the budget is 3 clock ticks for a given thread, then that thread would get scheduled for 3/10 of the clock ticks. Once the period ends, and the next one begins, then the budget is reset for the given thread. This assumes that the thread is not competing with other threads for the CPU. If a higher priority thread is ready to be scheduled, then it will be scheduled until it runs out of budget, then our thread will be permitted to run.

Threads have their budget replenished by recording how long the program ran per scheduling, its scheduler runtime, and schedules that at the next period it will receive that much time added to its budget. This guarantees that the thread will only receive the budget that it uses.

If a program has a budget of 10 clock ticks, but only runs for 3 clock ticks, then its budget will be set to 7. At the next period, the process will receive +3 budget, changing its budget value from 7 back to 10. This means that the scheduler does not need to keep track of what the threads budget is supposed to be (10), but rather how much budget is owed to the thread (3).

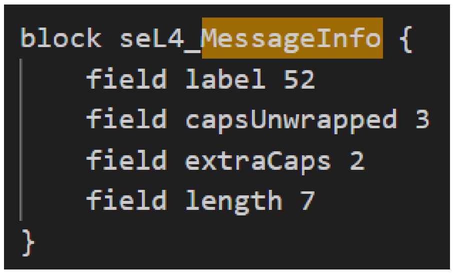
(Note: I think this is right, but I need source code proof)

The abstraction under the hood is that TCB's and Scheduling Context Objects must be made separately and then "binded". This can be found here. invokeSchedContext_Bind() makes a call to this function. Here you can see that the TCB and SCO are linked together here.

When threads expend their budget, they must get most clock ticks allocated to them via a refill. Under the hood, This process can be seen here.

Sel4 Endpoints:

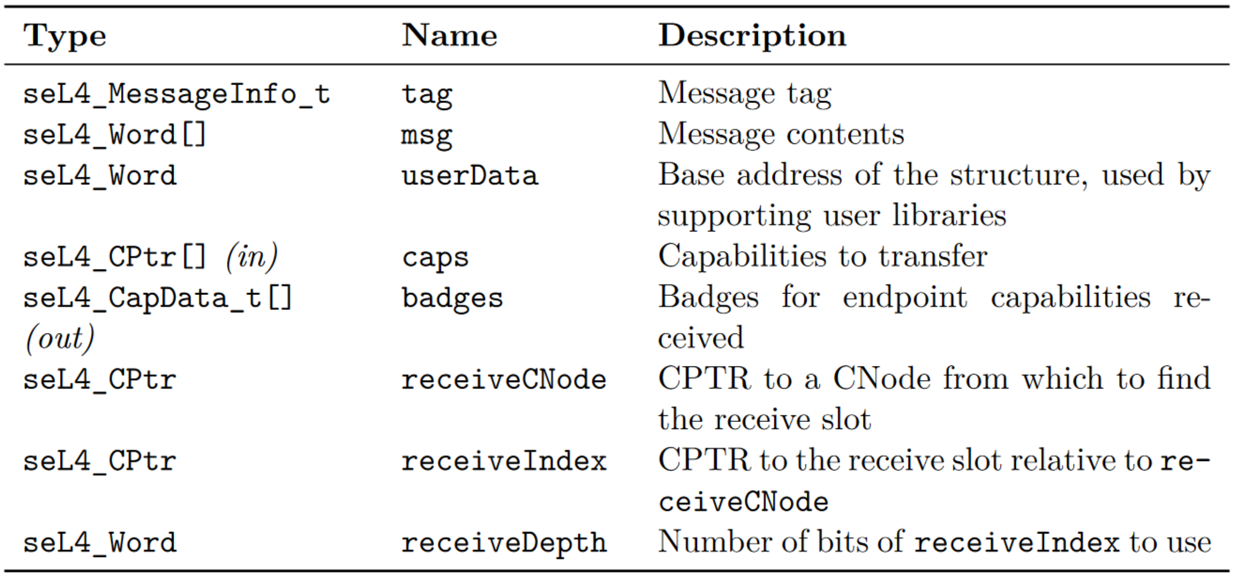
Message passing takes the form of physical registers and an IPC buffer. Each message is also accompanied by a seL4_MessageInfo_t struct. The struct is defined in the .bf files discussed earlier. Here is the definition:



```
block seL4_MessageInfo {
    field label 52
    field capsUnwrapped 3
    field extraCaps 2
    field length 7
}
```

Each field contains N number of bits where N is the number next to the field name. Label is 52 bits, as an example. The label contains part of the content of the message, extraCaps specifies if and how many capabilities are being sent over the connection. This is how one thread can distribute minted capabilities to new threads. Length refers to the number of registers required to facilitate the message passing, and capsUnwrapped indicates to the receiver how to interpret the capabilities being sent.

This MessageInfo structure is stored inside of the seL4_IPCBuffer structure. The reference manual defines the structure like this:



Type	Name	Description
`seL4_MessageInfo_t`	`tag`	Message tag
`seL4_Word[]`	`msg`	Message contents
`seL4_Word`	`userData`	Base address of the structure, used by supporting user libraries
`seL4_CPtr[]` *(in)*	`caps`	Capabilities to transfer
`seL4_CapData_t[]` *(out)*	`badges`	Badges for endpoint capabilities received
`seL4_CPtr`	`receiveCNode`	CPTR to a CNode from which to find the receive slot
`seL4_CPtr`	`receiveIndex`	CPTR to the receive slot relative to `receiveCNode`
`seL4_Word`	`receiveDepth`	Number of bits of `receiveIndex` to use

But the actual definition can be found here. We see that the first field is the MessageInfo_t struct just discussed. The key difference between the two representations is that caps and badges share an array. The array is either for caps or badges. This would normally be a union. I am assuming a union has been avoided to avoid proof complications. This is mentioned in the reference manual, but now we see it.

The Endpoint functionality forces threads to “rendezvou” at the Endpoint. This means that threads that arrive at the message passing call sooner than others will block to wait for the other thread. Previously, we looked at this file in the CNode section. We will now look closer at seL4_Send() to see how Endpoint calls function.

First, seL4_Send() is just a wrapper for an architecture dependent version of the function. This is true for all architectures, but we will be looking at 64-bit ARM. Here is the ARM version. Mr0 - mr3 are messages meant to be passed via the ARM registers. This can be seen here. Sys refers to a number that signifies the system call meant to be made. All of the parameters to the function are loaded into registers, and then svc is called. Scv stands for supervisor call. This means that the system call, sys, will be called in the supervisor mode of the ARM chip.

This is certainly interesting, but we are concerned with how the functions block to wait for each other. To answer that question, we need to see the sys functions that scv can call. From the ARM documentation page linked earlier, we can see in the description that the scv call jumps to an scv vector. This is likely identical to the x86 interrupt table. Looking at the x86 implementation, we see that the assembly uses the syscall instruction. This is a convenient way to write “int 0x80” added to 64-bit x86 chips. This confirms that we are looking for a vector of defined syscall functions.

I am going to show you where this scv vector is defined, and we are going to follow it together. This is going to be a very long chain of function calls until we get to where we are going. As a reminder, we are looking for where endpoints block to wait for one another. Here is where the scv vector is defined. We can see the actual function here. The function calls slowpath(). The difference between slowpath and fastpath syscalls is very important, but not required to be understood yet. We will cover it in the System Calls section below. The gist is that fastpath() is a streamlined operation for a select few syscalls. Send is not one of these syscalls, so we will be taking slowpath. Here is the slowpath() function. slowpath() either calls handleUnknownSyscall() or handleSyscall(). Unknown Syscalls refers to “not normal” syscalls. This includes things like debug system calls only present in debug builds. Send is a very normal system call so we will be looking at handleSyscall(). There is a switch statement that covers all of the possible system calls (you can see more on this in the System Calls section). We will skip straight to the send syscall. It calls this function. handleInvocation() does a lot of things. Most of it can be summed up as “checking permissions” by collecting together all of the capability pointers that are supposed to be used in this transaction. All of these capabilities, along with the messageInfo struct are passed to decodeInvocation().

We are going to move a little quicker:

1. decodeInvocation() calls performInvocation_Endpoint()
2. Which calls sendIPC()

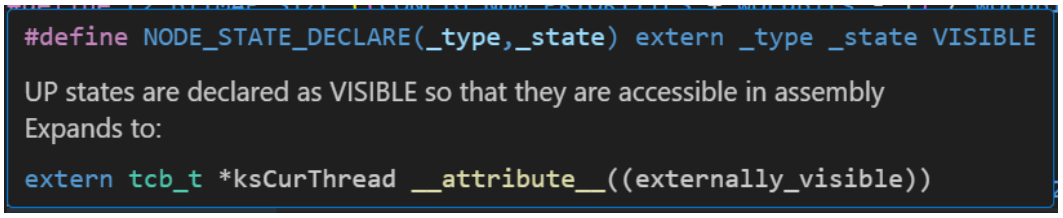
3. Which deschedules the thread and places it in a waiting queue

This is how endpoints block threads. They place them into a waiting queue which reactivates upon receiving an interrupt from the other thread reaching their endpoint. The excruciating details will be covered in the Systems Calls section.

seL4 Internals:

ks* Variables:

A close examination of the seL4 source code will reveal variables that do not seem to be defined. This comes from one of two sources: ks* Variables and Python generated types. The python generated types will be covered in a further section, once it is required, but we must examine ks* variables now as they play a central role in many functions in the kernel. Here is an example: this function is part thread scheduling process. Looking here in the function, we see several compile-time definitions and ksCurThread. This is an example of a very common type of variable, the ks* variable. These are global variables defined at compile time using compile-time definitions. They are accessible in assembly code, due to the fact that they are declared as external variables. This can be seen here. VSCode is kind enough to tell us what this expands to:



```
#define NODE_STATE_DECLARE(_type,_state) extern _type _state VISIBLE
```

UP states are declared as VISIBLE so that they are accessible in assembly
Expands to:

```
extern tcb_t *ksCurThread __attribute__((externally_visible))
```

So ksCurThread is a tcb_t pointer that is externally visible. What remains is when ksCurThread is assigned a tcb_t.

That can be seen here. The switchToThread() function performs a context switch using the ARM ASID and then once the context switch is complete, then the global ksCurThread variable is set to the new thread. This is a similar story for each of the ks* variables. They are global variables maintained as part of the operation of the system.

Secret Boot Process (ElfLoader):

ElfLoader is an seL4 utility that comes with a standard installation of seL4 but is not present inside the seL4 repo on github. Instead it can be found here. It is the defacto bootloader of seL4, and for this reason I mention it. It is not developed by the seL4 team (although version that comes with seL4 seems to be costumed designed for seL4), and therefore not considered a part of the seL4 kernel. ElfLoader loads both the kernel and the user image. In the case of the Hello World tutorial, the hello world program is the user image and seL4 is the kernel image. This answers a couple of mysteries about the process that is outlined in the next section. First, in the seL4 boot process, seL4 never loads the user image from disk. This is highly disturbing at first look because data does not magically appear in RAM. The answer is that ElfLoader loads both for us before giving control to seL4. ElfLoader also does some basic initialization of the CPU. Not much of the process is worth mentioning, except for the fact that the initial thread user image is loaded along with the kernel image.

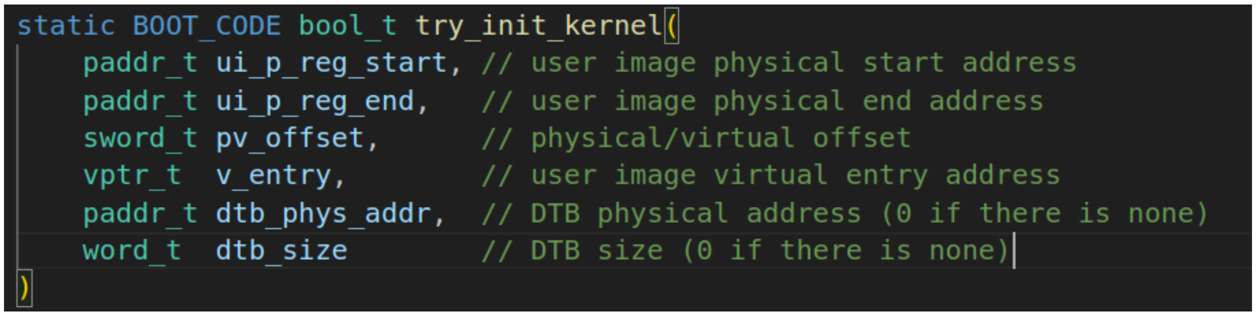
Booting:

The seL4 boot process starts in the architecture specific folders. For this document, I will assume ARM 64 bit. The first bit of code that will run on an ARM 64 bit machine can be found here. The first thing the kernel does upon taking control from the bootloader is initialize sctlr_el1 or sctlr_el2. These are special System Coprocessor Registers that provide access to hardware functionality. I suspect that this register controls things like the MMU, Page Tables, and TLB. The difference between the EL1 and EL2 version is whether or not seL4 will act like a hypervisor. EL(n) denotes the access control of the current execution environment. EL3 would be equivalent to ring 0. EL2 is then lower priority than EL3 and EL1 is lower priority than EL2 and so on. In the case that seL4 is acting as the hypervisor, EL2 denotes seL4, EL1 denotes VMs, and EL0 denotes userland programs.In the case that seL4 is a standard operating system, EL1 denotes the kernel and EL0 denotes the userland programs.

Next the system prepares to load the correct value for this boot into the SCLTR register. This is done here. Most of the instructions that follow are useless. The value of SCLTR is always the exact value of the CR_BITS_CLEAR variable.

The kernel continues some setup functionality, largely the same as before, either building the special requirements for SMP, multiprocessor support, or setting everything up in EL2 in the case of hypervisor support. Additionally, for hypervisor support on ARM, certain CPU functionalities must be interacted with and enabled (Note: Fill in what these are later).

The kernel is ready to jump to C code. This is done here. This function jumps to the code found here. init_kernel() is mostly a wrapper for try_init_kernel(), found here. Here are the comments I added to clarify what the parameters are:


```
static BOOT_CODE bool_t try_init_kernel(
    paddr_t ui_p_reg_start, // user image physical start address
    paddr_t ui_p_reg_end,   // user image physical end address
    sword_t pv_offset,      // physical/virtual offset
    vptr_t  v_entry,        // user image virtual entry address
    paddr_t dtb_phys_addr,  // DTB physical address (0 if there is none)
    word_t  dtb_size        // DTB size (0 if there is none)
)
```

The very first thing the kernel does once operating in C code is start creating pointers to capabilities.The IPC buffer is for message passing overflow and the root CNode is the root of the CSpace. The pd pointer will hold the capability for paging management, and the ad pointer will hold the capability for ASIC management. These pointers will be transferred to the “initial thread” which is when the userland processes begin running. The kernel then sets up variables to keep track of the physical memory size, and the size of the initial threads virtual memory. The physical memory variables are defined here, and the virtual memory variables are defined here. Note that the physical memory regions are hardware specific and so require an additional function to make the appropriate checks. Additionally, it is important to note that these memory regions are for the userland image that is to be loaded at boot time. The kernel adds two additional pages, here and here, after the virtual memory window it has created. These will store the bootinfo meta data. The final memory region, defined here, consists of the initial image binary, the IPC buffer region, and the bootinfo metadata.

Once the memory region is established, it is time to start populating the region with the appropriate data. First the initial thread is given some important capabilities for managing the rest of the system. create_domain_cap() provides a capability to the initial thread that allows it to manage the system domains. Think of domains as user groups for processes. A process can belong to one domain and only one domain can be active at a time. This allows the system to deschedule entire portions of the processes by switching domains.

Next, the system initializes the IRQ capability. This capability allows the initial thread to create IRQHandlers. IRQHandlers are a special capability that delegates to a certain thread the ability to handle execution following an interrupt. The bootinfo frame is then configured to allow only a certain number of maximum CNodes, here. Most of the code that follows is checking for special configurations that require adding additional information to the bootinfo section. This requires resizing the section and checking for conflicts, hence all the if statements that immediately follow.

Next, the kernel creates a virtual memory address space for the initial thread here. This leads to a function, found here. The function creates capabilities to the page directories and page tables, and returns them to the initial thread. This gives the initial thread management of the paging structure. The ASID capability, provided here, allows the initial thread to initialize new applications with allocated pages.

The kernel initializes the initial thread TCB with all of the data it has generated, and returns back to init_kernel(). The TCB is then added to the scheduler, and control is transferred.

This completes the boot process.

Scheduling:

We have already covered, loosely, the scheduler. Here we will examine the scheduler in full detail, uncovering the algorithms that decide which processes are scheduled. The main algorithm for scheduling can be found here. The first thing one might notice is that the scheduler's actions depend upon a global ks* variable. This variable is defined here. The variable, ksSchedulerAction, is a TCB pointer. This is a bit confusing. We expect ksSchedulerAction to hold values 1 or 0, based on the if statements. Why is it a TCB pointer?

To understand how ksSchedulerAction is being used, we can look here. The macro is not super important, but the comment is. If the value it points to is 1 or 0, then it should be treated as an enum value. If the value is anything else, then we expect it to be a schedulable thread. Knowing this, There are effectively three options for a scheduling task:

1. Resume the current thread (seen here)
2. Find a new thread (seen here)
3. Schedule the thread found inside ksSchedulerAction (found here)

Let's start with the first option, resuming the current thread. At this line, the scheduler is asking if the current scheduling task is to resume the current thread. If that is not true, then the scheduler asks if the current thread should be put back in the queue.

For the second option, finding a new thread, the scheduler calls the scheduleChooseNewThread() function. This function can be found here. This is effectively a wrapper around another function, chooseThread(). This wrapper first checks if the current domain is out of time. If it is then we move to the next domain. The domain business is not important. For now, let's assume that we are not making use of domains, meaning our domain time is infinite. In this case, we skip straight to chooseThread(), which can be found here. The function starts by checking if there are any runnable threads. If there are, then we go to the highest priority queue in our domain and retrieve the head of that list. This is done on this line. From this we can deduce the structure of the scheduler. ksReadyQueues is an array of queues, each containing a linked list of tcb_t structs. For every priority level, there is a queue. This means that if multiple threads are schedulable at the same priority level, then the first in line (IE the head of the linked list) is the next thread to be scheduled.

We can confirm this is actually the case by looking here. ksReadyQueues is indeed an array, but our assumption is that it is an array large enough to hold 1 index for each priority level for each domain. It should be the number of domains times the number of priority levels. Looking at the definition of NUM_READY_QUEUES, we see this.

Note that if the scheduler only has 5 schedulable threads, then this means that the majority of ksReadyQueues is spare. It contains an index for each priority level, regardless of if there are scheduled threads or not.

The chooseThread() function is not quite finished with its operations just yet. After a thread is selected out of this array of queues, then the function calls switchToThread(). This function can be found here. The function starts with several asserts to ensure everything is well formed. The first real action the function takes is calling Arch_switchToThread(). The function can be found here. It is, yet another, function that is just a wrapper for some other function. Let's go to setVMRoot().

(Note: Finish. setVMRoot() heads into hardware context switching territory. Very time consuming to parse)

System Calls:

System calls in seL4 go through capability pointers. Each pointer points to an "object" which can export methods. It is much like object oriented programming where an object holds methods. Holding a pointer to that object gives access to the methods. Capability pointers provide a similar functionality to object methods. The only real difference is that the capability pointers provide *granular* access, meaning not all capability pointers are created equal. Some have higher permissions than others, allowing access to more methods, or more functionality. Much of this granularity is defined in the CAmkES section below.

To better understand the system calls in seL4, we will be following a system call from userspace all the way through the kernel, until it returns to userspace. We will have to do this twice because the system calls can either take the slowpath() or the fastpath(). The fastpath is a different mechanism for servicing system calls that streamlines some of the process by skipping some of the functionality that the slowpath provides. We will examine system calls that go down each path.

Before we dive into source code, we should examine more closely the stated difference between seL4's fastpath and slowpath. (Note: Generalized formalization of a microkernel paper)

We will start with the slowpath. It is productive to think of the fastpath as a specialized version of the slowpath, so we will start with the "normal" operation and cover the specialized version second.

We have shown much of what follows before, particularly in the endpoints section. We cover all of this information again to make this section self contained. Feel free to skip if you remember the details, but we will be exploring new details not covered in the Endpoints section, and adding more detail to what was covered.

We will be examining the seL4_Send() system call. Its entry point is a library function that is linked into userspace. We can see here that sel4_Send() is a wrapper for an architecture dependent version of the system call. This is required because we are currently in user mode and we need to trigger an interrupt to send us into kernel mode. The way interrupts are triggered is architecture dependent. The 64-bit ARM version of seL4_Send() can be found here.

CAmkES Overview:

Since much of the functionality associated with seL4 is actually provided by IPC calls, the framework for which these calls are permitted, transmitted, and received must also be heavily and provably guarded. The CAmkES framework is essentially a modeling language that describes the possible interactions between userland components. CAmkES intends to enforce that only the interactions modeled will be permitted. The power of the modeling language is unclear. Can users model regex for invalid inputs?

When compiling a project to run on seL4, the CAmkES framework generates additional code to facilitate the boundaries modeled. This is the CapDL language and glue code. CapDL, Capability Distribution Language, acts as a boot time operation to generate the userland processes, the required capabilities (See below), and enforce access rights. Once CapDL is finished setting up the system, the system is now only able to follow the modeled CAmkES specification. Additionally, CAmkES also automatically generates “glue code”. This glue code contains the actual system calls made between processes. This means that a userland process never actually writes the C code that interacts with another userland process. Message passing and memory sharing code is generated by CAmkES, meaning that users can not alter the way the communication works (supposedly).

CAmkES ADL:

The core of the CAmkES framework and the Architecture Description Language it provides are components and interfaces. Components designate userland processes. Each component will receive its own address space. Here is an example of component definitions:

```
/* apps/helloworld/components/Hello/Hello.camkes */

import "../../interfaces/MyInterface.idl4";

component Hello {
  provides MyInterface inf;
}

/* apps/helloworld/components/Client/Client.camkes */

import "../../interfaces/MyInterface.idl4";

component Client {
  control;
  uses MyInterface iface;
}
```

Here two userland processes, client and hello, will be created at boot time. This boot time initialization is covered in the next section. For now, trust me that when seL4 boots, a client and hello process will be running.

In this example, the components are also making use of an interface: MyInterface. This is a bit of a silly name, but the idea here is that hello provides an interface of functionality that can be accessed via IPC. Here is what that looks like in C code;

```
#include <camkes.h>
#include <stdio.h>

void inf__init(void) {
}

void inf_print(const char *message) {
  printf("Client says: %s\n", message);
}
```

Here inf_print means that print() is a part of the inf interface. Any outside process that has access to the inf process will have access to this function through IPC. Client is defined as “uses MyInterface iface”. Iface is just a name for local scoping. This line says that the client should be given access to all of the inf functions. Client can call inf_print():

```
#include <camkes.h>

int run(void) {
  const char *s = "hello world";
  iface_print(s);
  return 0;
}
```

Client uses iface, its local name, but this function call refers to the “MyInterface” interface and will subsequently call inf_print() inside hello.

It should be noted that MyInterface cannot consist of arbitrary functions. The functions must be defined before hand in a procedure element:

```
procedure MyInterface {
  void print(in string message);
}
```

Defining all of these structures and functions is not enough. The actual boot process of seL4 (specifically capDL) will only initialize the things found in the assembly and composition tags:

```
/* apps/helloworld/helloworld.camkes */

import <std_connector.camkes>;
import "components/Hello/Hello.camkes";
import "components/Client/Client.camkes";

assembly {
  composition {
    component Hello h;
    component Client c;
    connection seL4RPCCall conn(from c.iface, to h.inf);
  }
}
```

The assembly tells capDL to initialize the compositions found below. The components we just defined are here, Hello and Client, but also a connection sel4RPCCall con() function. This is letting capDL know ahead of time that we will be calling functions from Client to Hello.

The communication between two processes does not necessarily need to be a function call. Notifications (such as a notification that a certain task has completed) can also be transferred and special functionality has been built into the CAmkES ADL to facilitate this type of communication. Events (notifications) can be created in the ADL by adding special syntax to components:

```
component Emitter {
  control;
  emits MyEvent e;
}
```

Emitter is a component, just like client and hello, but it has been given the “emits” trait. This is paired up with another component that “consumes” the event:

```
component Consumer {
  control;
  consumes MyEvent s;
}
```

To link these events up so that emitter can send a notification to consumer, we add a connection just like before:

```
/* apps/helloevent/helloevent.camkes */

import <std_connector.camkes>;
import "components/Emitter/Emitter.camkes";
import "components/Consumer/Consumer.camkes";

assembly {
  composition {
    component Emitter source;
    component Consumer sink;
    connection seL4Notification channel(from source.e, to sink.s);
  }
}
```

This time the connection is of type “sel4Notification”. This is the type of system call that is being used to facilitate this communication. Before it was seL4RPCCall, in this case it is seL4Notification.

This can be used for asynchronous programming. We can define a callback with the register_callback() function. When the notification is received, the running thread will be preempted and the callback will execute. Here is what that looks like in C:

```c
#include <camkes.h>
#include <stdio.h>

static void handler(void) {
  static int fired = 0;
  printf("Callback fired!\n");
  if (!fired) {
    fired = 1;
    s_reg_callback(&handler);
  }
}

int run(void) {
  printf("Registering callback...\n");
  s_reg_callback(&handler);

  printf("Polling...\n");
  if (s_poll()) {
    printf("We found an event!\n");
  } else {
    printf("We didn't find an event\n");
  }

  printf("Waiting...\n");
  s_wait();
  printf("Unblocked by an event!\n");

  return 0;
}
```

s_reg_callback() registers a callback for the event named s. Scrolling up, you can see that the component named s is Consumer's "Consume MyEvent s". This callback will be triggered when the MyEvent notification is received.

The final way CAmkES components can communicate is through dataports. Dataports are shared memory. Dataports share memory at page granularity, meaning that the shared memory is 4K in size. The reason for this is that the paging system assigns both processes the same page. Meaning both are writing to the same 4K section of RAM (frame). A full page is defined by a CAmkES type "Buf" that is specified in the ADL.

We can see an example of how this might work:

```c
typedef struct MyData {
  char data[10];
  bool ready;
} MyData_t;
```

Given this struct definition, we can define dataports in this way:

```
component Ping {
  include "porttype.h";
  control;
  dataport Buf d1;
  dataport MyData_t d2;
}
```

We have defined a component just like before. Previously components either used or exported an interface, consumed or used a signal, and in this case, have a dataport, which will be a page of memory mapped to Ping and another process. We still have to see how to link the Ping dataport to another dataport. Let's see how we can do that:

Given a second component, Pong:

```
component Pong {
  include "porttype.h";
  control;
  dataport Buf s1;
  dataport MyData_t s2;
}
```

We can link up the dataports in Ping and Pong in the Assembly and Composition:

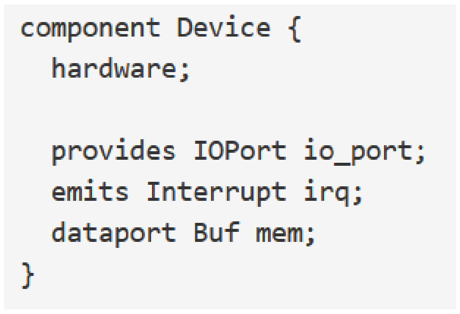
```
import <std_connector.camkes>;
import "components/Ping/Ping.camkes";
import "components/Pong/Pong.camkes";

assembly {
  composition {
    component Ping ping;
    component Pong pong;

    connection seL4SharedData channel1(from ping.d1, to pong.s1);
    connection seL4SharedData channel2(from ping.d2, to pong.s2);
  }
}
```

It should be noted that seL4 and CAmkES only facilitate the linking of these two memory regions. Helper functions are provided to ensure reading/writing blocks, but managing this memory correctly is the responsibility of the programmer.

CAmkES users can also set error handling functions. CAmkES code places glue code inside the userspace program and the ADL can configure how these operations function (for more on glue code, see the next few sections on CAmkES internals). camkes_register_error_handler() is called, passing the function and registering the function as the error handler.

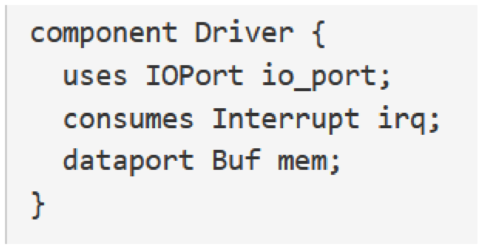
(Note: Demonstrate this in Ghidra)

Hardware components in the seL4 system can also be configured in the CAmkES ADL. Here is an example of a hardware component:



```
component Device {
  hardware;

  provides IOPort io_port;
  emits Interrupt irq;
  dataport Buf mem;
}
```

This component, Device, is declared as a hardware component. Here a dataport refers to memory mapped IO and Interrupt refers to an interrupt that the device can emit. Here is a device driver that interacts with the above component:



```
component Driver {
  uses IOPort io_port;
  consumes Interrupt irq;
  dataport Buf mem;
}
```

The way you should read the component Driver is that it can send signals along an IO_port, it responds to interrupts from the device, and it has access to the memory mapped IO. Here is an example assembly that links these components together:

```
assembly {
  composition {
    component Device dev;
    component Driver drv;
    ...
    connection seL4HardwareIOPort ioport_c(from drv.io_port, to dev.io_port);
    connection seL4HardwareInterrupt irq_c(from dev.irq, to drv.irq);
    connection seL4HardwareMMIO mmio_c(from drv.mem, to dev.mem);
  }
}
```

In order to use memory mapped IO, you must configure the CAmkES ADL to specify the region of physical memory. Here is what that might look like:

```
assembly {
  composition {
    component Device d;
    ...
  }
  configuration {
    d.mem_paddr = 0xE0000000;
    d.mem_size = 0x1000;
    ...
  }
}
```

Here the location (physical address) is set to 0xE0000000, and the size is 0x1000. As an example for why this is required, the raspberry pi has GPIO pins that are accessed with memory mapped IO. The memory mapping is set in stone (circuitry), and cannot be changed. seL4 does not have the power to reconfigure this memory mapping to somewhere else. This is why it is the responsibility of the user to ensure that the memory mapping is at the correct location.

Depending on the structure and design of the card, you may need to also specify an interrupt handler:

```
configuration {
  d.irq_irq_number = 2;
  ...
}
```

Note that not all cards require this. Modern x86 boards require a different configuration, but most ARM boards will require an interrupt number.

(Note: Discuss thread priority and demonstrate in Ghidra)

In CAmkES, a user can define co-located components. This means that the components are located in the same userspace. This presents a problem for heap allocation. Co-located processes are each given their own heap, and free() cannot distinguish between the heaps. It is the user's responsibility to make sure that the correct component calls free() on a memory object.

These examples and more information can be found here.

CAmkES Internals:

seL4 as a system enforces protection primarily of memory. The kernel itself is correctly configured in such a way that userland processes cannot corrupt memory beyond the memory mapped into its virtual memory space. This is meant to isolate exploitation. Processes that are exploited and under attacker control have a difficult time breaking outside the virtual memory barrier. This does not mean that an attacker cannot escape however. It is typical that processes will want to communicate with each other through one of three mechanisms: Message passing, Shared memory, Notifications. These messages would be a prime point of attack for any malicious actor. They allow the attacker to request functionality of other processes.

An Example:

Suppose that there is an exploitable website running on an seL4 system. Suppose that an attacker can gain full access to the website and change the functionality of the process in any arbitrary way they choose. seL4 ensures that this malicious behavior is locked into the virtual address space of the website; however, suppose also that the website manages a database that must periodically be flushed to disk. The website must have a communication line to the filesystem to request flushing to disk. If the attacker gains control of this communication line, and can send arbitrary commands to the file system, then the attacker can request horrible things like "Please delete your entire contents" or "Replace the seL4 kernel image with an insecure version with a backdoor". This is obviously very bad.

Enter CAmkES and capDL. These tools provide two basic guarantees. CAmkES allows systems designers to model what the communication lines between processes should look like. CapDL takes this specification and automatically generates an "Initial Thread" program that will spawn the specified programs, and give them the correct access credentials for the types of commands desired.

As another example, consider the website and file system again. Assume that the website no longer maintains any mutable data (the database in this case) and no longer needs to flush data to disk. If the website has no reason to communicate with the file system, then the communication line will not be created in the first place.

The way seL4 determines what communication lines need to exist and which ones do not need to exist is based on what the user supplies in the CAmkES modeling language. If the modeling language does not specify a connection, then a connection will not be created. We are concerned with instances where a connection is required, is created, and is hijacked by a malicious user.

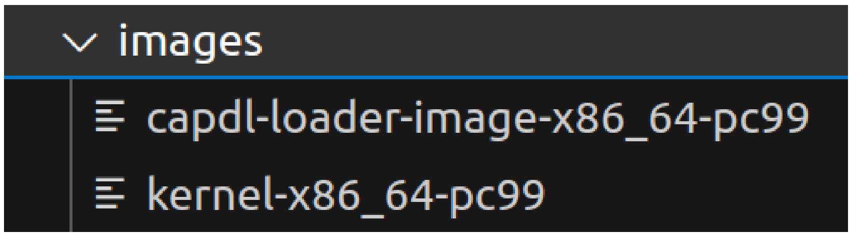
Initial Thread Internals:

CAmkES generates a large amount of code at build time to support the modeled functionality. I suspect that this code generation exists as a means to ensure that any behavior *not requested* is *not compiled*. This means that an attacker would theoretically only have access to the behavior that was modeled and validated by the system designer. This makes auditing the CAmkES code that does actually appear in the program difficult. In order to get access to CAmkES code, one must build a CAmkES project and compile it. This will generate the .c and .h files, which will be left behind, for review. Unfortunately, these .c files do not tell the whole story. We can read the code that will be compiled in the system, but it is difficult to know *who* has access to the code.

At the top level of the seL4 system is the kernel. The kernel works to spawn the initial thread. The initial thread is a program that either we write or CAmkES generates (CAmkES always generates this file, but you don't have to use CAmkES to use seL4). The initial thread finishes setting up the system (managing paging, semaphore access, capabilities) and spawns the programs we write.

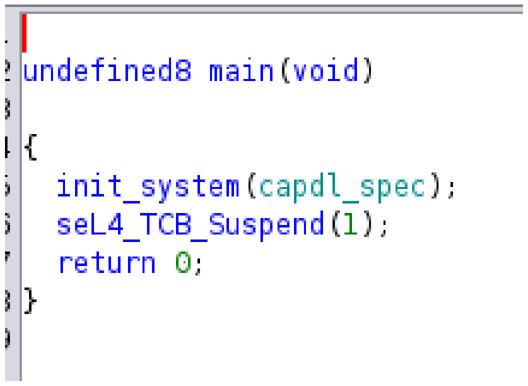
In the tutorial hello-camkes-2, two threads are spawned. One is called a client and the other an echo server. In reality, they are just two threads that send messages back and forth via shared memory (dataports). We will be examining this tutorial as a case study of the mechanisms behind CAmkES.

We expect, when running ./simulate, that the kernel will spawn the initial thread (generated by CAmkES) and the initial thread will spawn the client and echo processes. When running ninja, two images are created seen here:



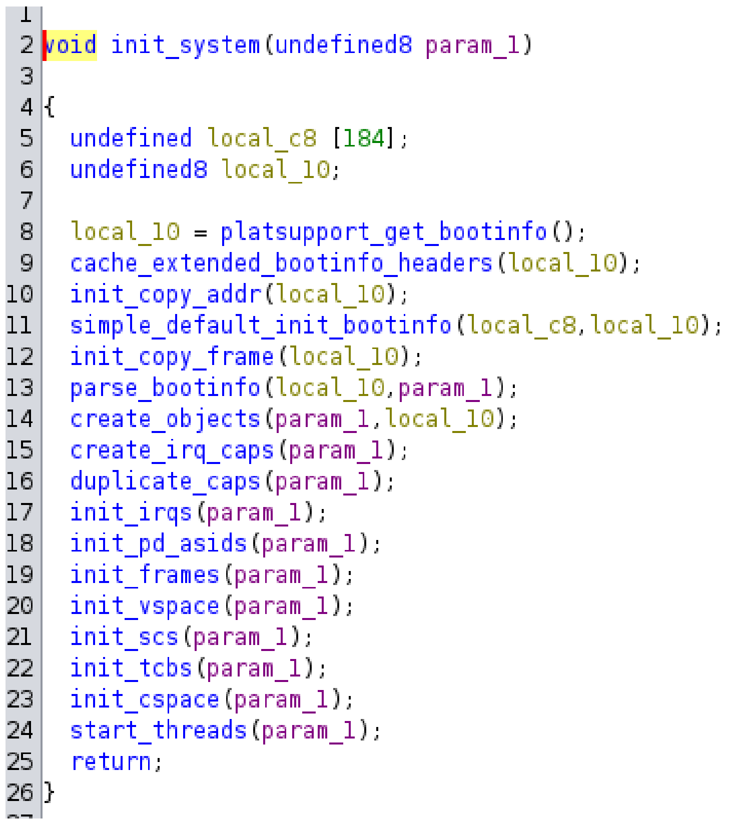
We know what the kernel does. It will spawn the initial thread. As a small note to clear up any confusion, CAmkES generates an initial thread by passing the model to the capDL tool. The capDL tool then generates the actual initial thread code. This is why the second image is called capdl-loader-image. CapDL and CAmkES are intimately tied together and can be used interchangeably without fear of confusion; just remember that CAmkES is the modeling system and CapDL generates the initial thread.

We want to interrogate what exactly CapDL (or CAmkES) has put into the capdl-loader-image. Luckily the image is just a standard elf binary. Loading it into Ghidra, we find this as the main() function:



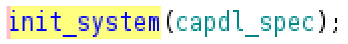
```

undefined8 main(void)

{
  init_system(capdl_spec);
  seL4_TCB_Suspend(1);
  return 0;
}

```

Here is the first function called from main, init_system():



```
 1
 2 void init_system(undefined8 param_1)
 3
 4 {
 5   undefined local_c8 [184];
 6   undefined8 local_10;
 7
 8   local_10 = platsupport_get_bootinfo();
 9   cache_extended_bootinfo_headers(local_10);
10   init_copy_addr(local_10);
11   simple_default_init_bootinfo(local_c8,local_10);
12   init_copy_frame(local_10);
13   parse_bootinfo(local_10,param_1);
14   create_objects(param_1,local_10);
15   create_irq_caps(param_1);
16   duplicate_caps(param_1);
17   init_irqs(param_1);
18   init_pd_asids(param_1);
19   init_frames(param_1);
20   init_vspace(param_1);
21   init_scs(param_1);
22   init_tcbs(param_1);
23   init_cspace(param_1);
24   start_threads(param_1);
25   return;
26 }
```

We can see from line 14 - 23, the initial thread is performing much of the functionality discussed in previous sections. It is the responsibility of the initial thread to create the virtual memory, capabilities, TCBs, etc. It must also start the client and echo program. This is what happens on line 24. Before jumping into any of these functions, we need to know what param_1, local_c8, and local_10 are. Examining the main function again, we see that init_system() is called with capdl_spec as the parameter:

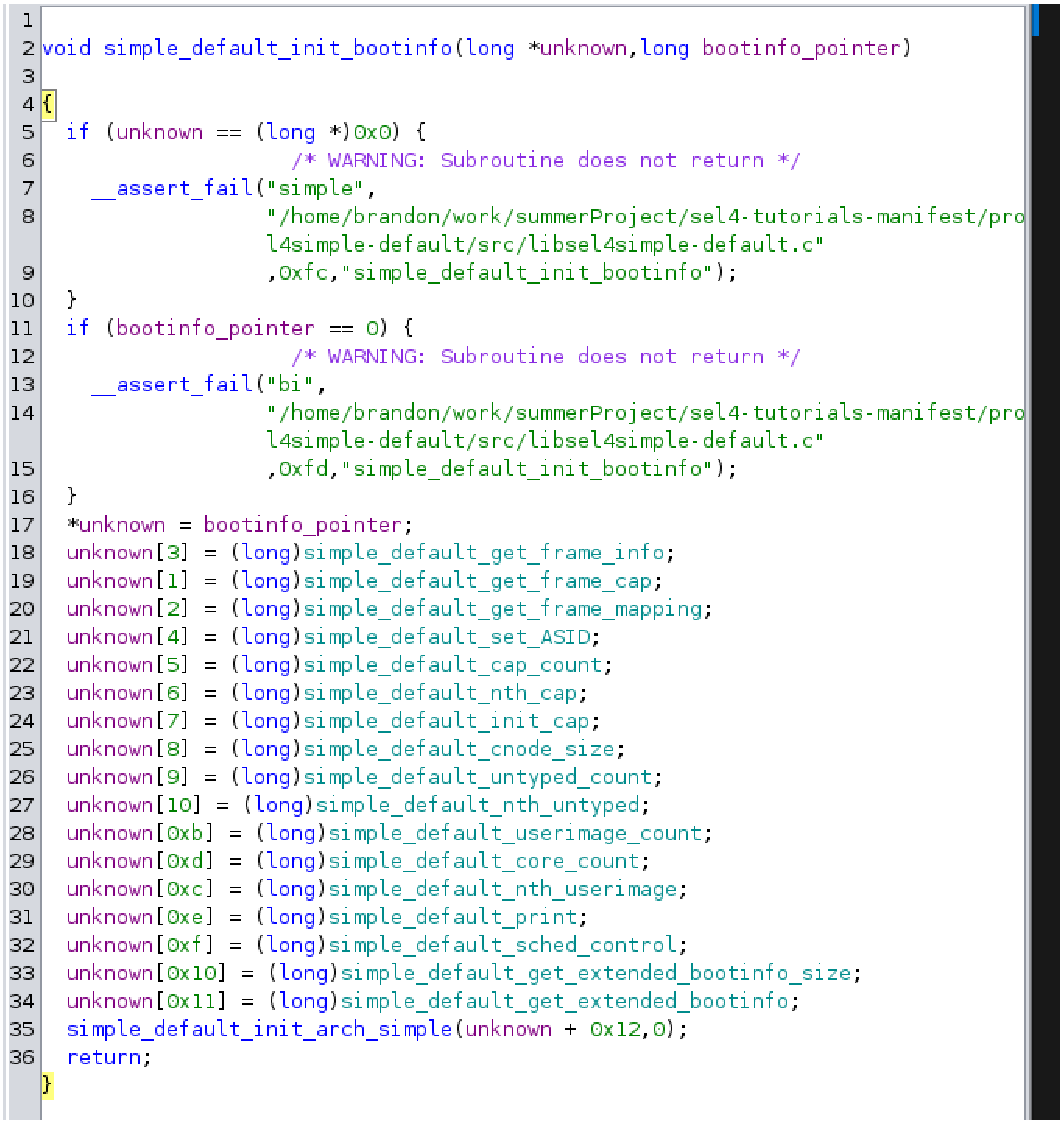


```
init_system(capdl_spec);
```

This settles param_1. It is the capdl_spec. We will investigate what exactly this is later.

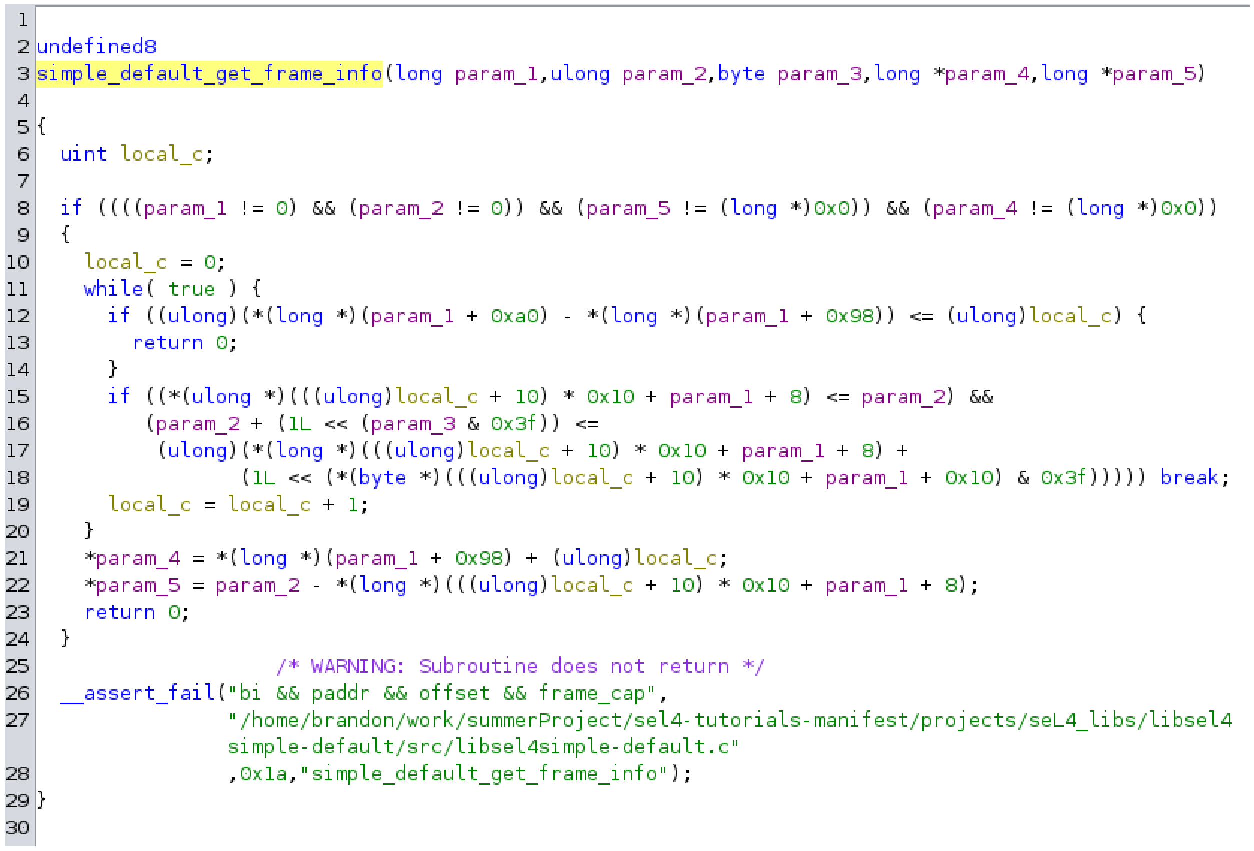
We see that local_10 is set to the return value of platsupport_get_bootinfo(). This function just returns a pointer to an arbitrary space in memory that holds the boot info. This memory is not loaded in this elf, which means it is linked at runtime. This settles local_10. It is a pointer to the bootinfo structure.

Lastly, we must find out what the array local_c8 is. The first place local_c8 is used is in the function simple_default_init_bootinfo(). Here is that function:



```

void simple_default_init_bootinfo(long *unknown,long bootinfo_pointer)

{
  if (unknown == (long *)0x0) {
                    /* WARNING: Subroutine does not return */
    __assert_fail("simple",
                  "/home/brandon/work/summerProject/sel4-tutorials-manifest/pro
                  l4simple-default/src/libsel4simple-default.c"
                  ,0xfc,"simple_default_init_bootinfo");
  }
  if (bootinfo_pointer == 0) {
                    /* WARNING: Subroutine does not return */
    __assert_fail("bi",
                  "/home/brandon/work/summerProject/sel4-tutorials-manifest/pro
                  l4simple-default/src/libsel4simple-default.c"
                  ,0xfd,"simple_default_init_bootinfo");
  }
  *unknown = bootinfo_pointer;
  unknown[3] = (long)simple_default_get_frame_info;
  unknown[1] = (long)simple_default_get_frame_cap;
  unknown[2] = (long)simple_default_get_frame_mapping;
  unknown[4] = (long)simple_default_set_ASID;
  unknown[5] = (long)simple_default_cap_count;
  unknown[6] = (long)simple_default_nth_cap;
  unknown[7] = (long)simple_default_init_cap;
  unknown[8] = (long)simple_default_cnode_size;
  unknown[9] = (long)simple_default_untyped_count;
  unknown[10] = (long)simple_default_nth_untyped;
  unknown[0xb] = (long)simple_default_userimage_count;
  unknown[0xd] = (long)simple_default_core_count;
  unknown[0xc] = (long)simple_default_nth_userimage;
  unknown[0xe] = (long)simple_default_print;
  unknown[0xf] = (long)simple_default_sched_control;
  unknown[0x10] = (long)simple_default_get_extended_bootinfo_size;
  unknown[0x11] = (long)simple_default_get_extended_bootinfo;
  simple_default_init_arch_simple(unknown + 0x12,0);
  return;
}
```

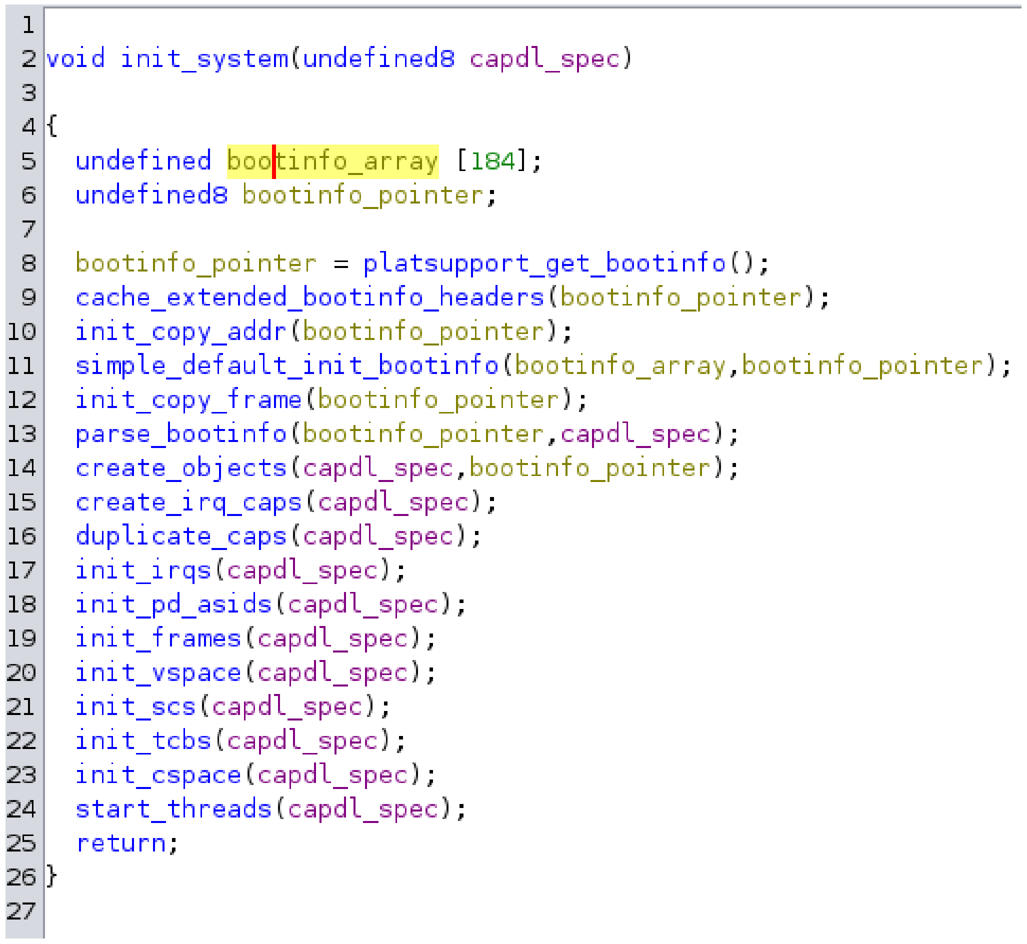
The function starts by pointing our unknown array (local_c8) to the bootinfo section. The function then writes to different indices in the bootinfo section. It is important to understand that the array and the bootinfo_pointer are pointing to the same location. The array is *writing* to the bootinfo section. Double clicking one of the values being written (we will be looking at simple_default_get_frame_info) brings us to a function:



```
undefined8
simple_default_get_frame_info(long param_1,ulong param_2,byte param_3,long *param_4,long *param_5)

{
  uint local_c;

  if ((((param_1 != 0) && (param_2 != 0)) && (param_5 != (long *)0x0)) && (param_4 != (long *)0x0))
  {
    local_c = 0;
    while( true ) {
      if ((ulong)(*(long *)(param_1 + 0xa0) - *(long *)(param_1 + 0x98)) <= (ulong)local_c) {
        return 0;
      }
      if ((*(ulong *)(((ulong)local_c + 10) * 0x10 + param_1 + 8) <= param_2) &&
         (param_2 + (1L << (param_3 & 0x3f)) <=
          (ulong)(*(long *)(((ulong)local_c + 10) * 0x10 + param_1 + 8) +
                 (1L << (*(byte *)(((ulong)local_c + 10) * 0x10 + param_1 + 0x10) & 0x3f))))) break;
      local_c = local_c + 1;
    }
    *param_4 = *(long *)(param_1 + 0x98) + (ulong)local_c;
    *param_5 = param_2 - *(long *)(((ulong)local_c + 10) * 0x10 + param_1 + 8);
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __assert_fail("bi && paddr && offset && frame_cap",
                "/home/brandon/work/summerProject/sel4-tutorials-manifest/projects/seL4_libs/libsel4
                simple-default/src/libsel4simple-default.c"
                ,0x1a,"simple_default_get_frame_info");
}
```

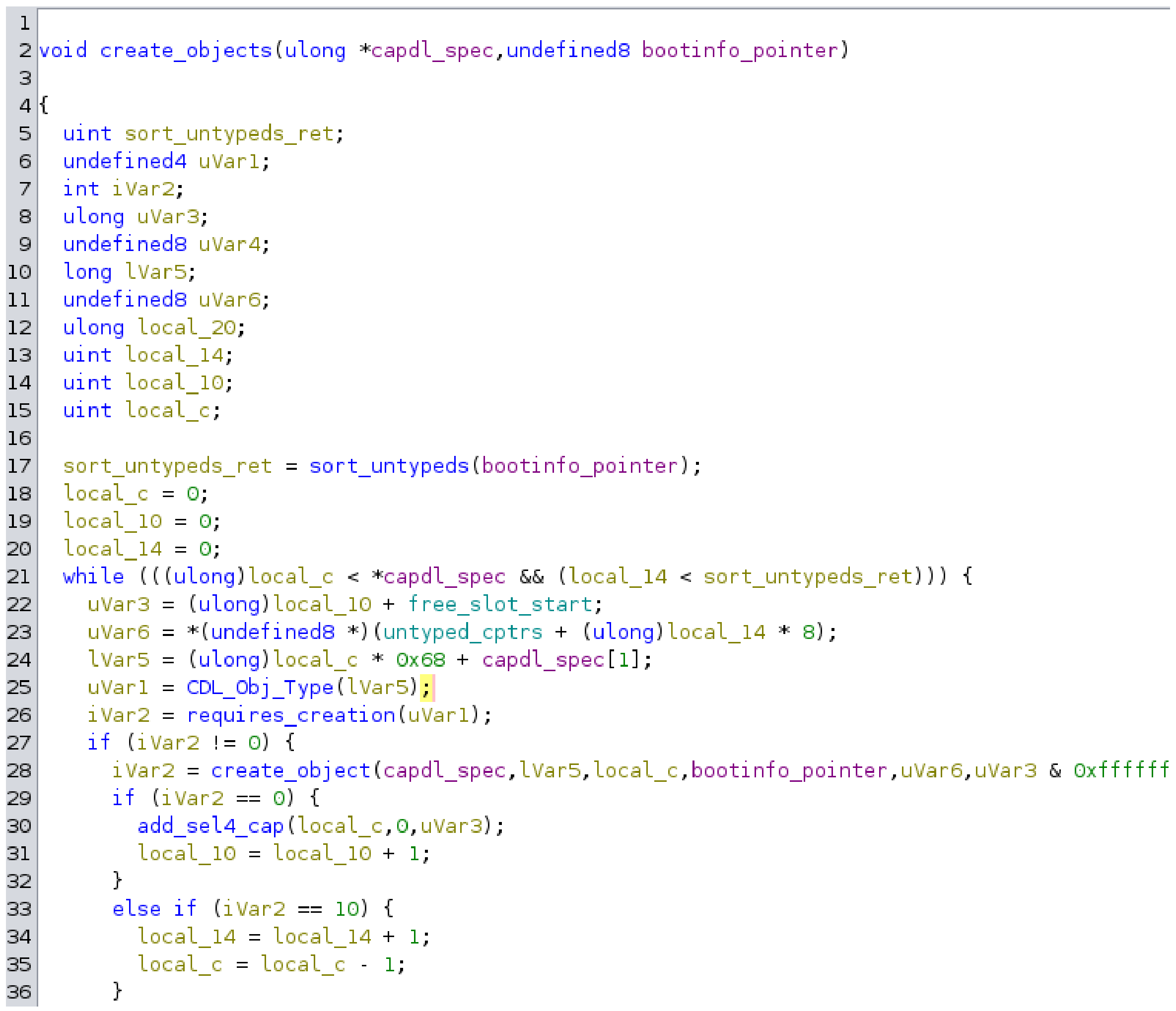
What this function does is unimportant at this stage in the process. What this tells us is that the bootinfo_pointer is pointing to an array of pointers. This settles what local_c8 is. It is an array used to write to the bootinfo section. It is a temporary variable made for convenience.

After relabeling the variables, our function now looks like this:



```

void init_system(undefined8 capdl_spec)

{
  undefined bootinfo_array [184];
  undefined8 bootinfo_pointer;

  bootinfo_pointer = platsupport_get_bootinfo();
  cache_extended_bootinfo_headers(bootinfo_pointer);
  init_copy_addr(bootinfo_pointer);
  simple_default_init_bootinfo(bootinfo_array,bootinfo_pointer);
  init_copy_frame(bootinfo_pointer);
  parse_bootinfo(bootinfo_pointer,capdl_spec);
  create_objects(capdl_spec,bootinfo_pointer);
  create_irq_caps(capdl_spec);
  duplicate_caps(capdl_spec);
  init_irqs(capdl_spec);
  init_pd_asids(capdl_spec);
  init_frames(capdl_spec);
  init_vspace(capdl_spec);
  init_scs(capdl_spec);
  init_tcbs(capdl_spec);
  init_cspace(capdl_spec);
  start_threads(capdl_spec);
  return;
}

```

Much better. We have already examined platsupport_get_bootinfo and simple_default_init_bootinfo. The next interesting function is create_objects(). The function can be seen, in part, here:

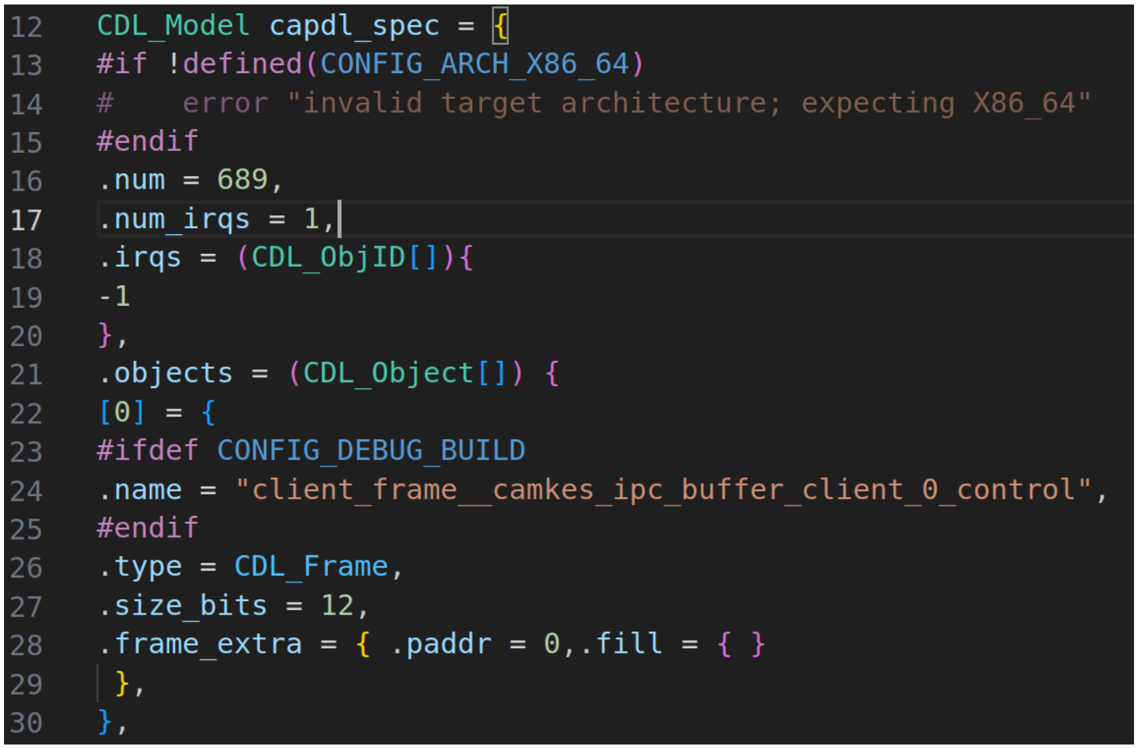


```
 1
 2 void create_objects(ulong *capdl_spec,undefined8 bootinfo_pointer)
 3
 4 {
 5   uint sort_untypeds_ret;
 6   undefined4 uVar1;
 7   int iVar2;
 8   ulong uVar3;
 9   undefined8 uVar4;
10   long lVar5;
11   undefined8 uVar6;
12   ulong local_20;
13   uint local_14;
14   uint local_10;
15   uint local_c;
16
17   sort_untypeds_ret = sort_untypeds(bootinfo_pointer);
18   local_c = 0;
19   local_10 = 0;
20   local_14 = 0;
21   while (((ulong)local_c < *capdl_spec && (local_14 < sort_untypeds_ret))) {
22     uVar3 = (ulong)local_10 + free_slot_start;
23     uVar6 = *(undefined8 *)(untyped_cptrs + (ulong)local_14 * 8);
24     lVar5 = (ulong)local_c * 0x68 + capdl_spec[1];
25     uVar1 = CDL_Obj_Type(lVar5);
26     iVar2 = requires_creation(uVar1);
27     if (iVar2 != 0) {
28       iVar2 = create_object(capdl_spec,lVar5,local_c,bootinfo_pointer,uVar6,uVar3 & 0xffffff
29       if (iVar2 == 0) {
30         add_sel4_cap(local_c,0,uVar3);
31         local_10 = local_10 + 1;
32       }
33       else if (iVar2 == 10) {
34         local_14 = local_14 + 1;
35         local_c = local_c - 1;
36       }
```

There is quite a bit to unpack in the above photo. This is a long journey. Maybe take a break before reading further.

Alright, welcome back. Let’s unpack the create_objects() function. The first thing we see is that there are alot of local variables defined at the top of the function. We will need to parse what each of these variables are before we are done. The first real statement of the function is line 17. This sorts the objects in bootinfo_pointer. It is unclear why the objects are not loaded into memory sorted already. I suspect that the return value is the number of left over, open slots in the object array, but the function is too messy to give a sure answer.

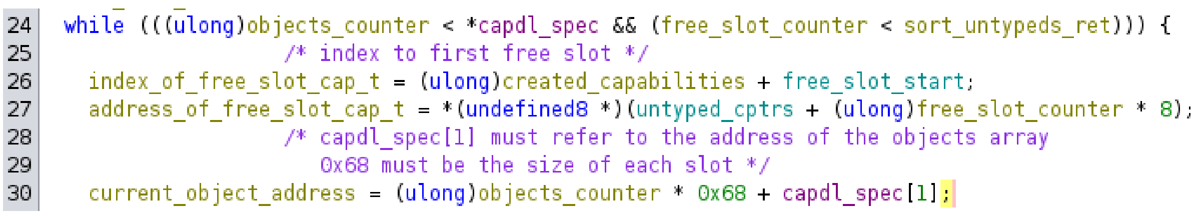
Now is a good time to bring up that the capdl_spec structure can be found in the build folder of your project. In the top of the build directory, you should find a file called capdl_spec.c. Opening it, we see this:



```
CDL_Model capdl_spec = {
#if !defined(CONFIG_ARCH_X86_64)
#    error "invalid target architecture; expecting X86_64"
#endif
.num = 689,
.num_irqs = 1,
.irqs = (CDL_ObjID[]){
-1
},
.objects = (CDL_Object[]) {
[0] = {
#ifdef CONFIG_DEBUG_BUILD
.name = "client_frame__camkes_ipc_buffer_client_0_control",
#endif
.type = CDL_Frame,
.size_bits = 12,
.frame_extra = { .paddr = 0,.fill = { }
 },
},
```

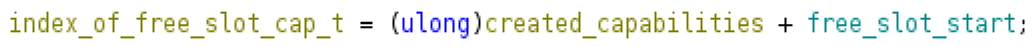
This is the structure that capdl_spec is pointing to. Given this, we can deduce that the first condition of the while loop in create_objects() is counting up from 0 to .num (689). This number refers to the number of capability objects that need to be created and distributed. We can also see in the C representation of capdl_spec that there is an object array starting with [0]. This array is very long, much of it is omitted, but these are the objects that we will be creating in our create_objects() function.

Inside the while loop of create_objects() we see this:



```
24  while (((ulong)objects_counter < *capdl_spec && (free_slot_counter < sort_untypeds_ret))) {
25                      /* index to first free slot */
26    index_of_free_slot_cap_t = (ulong)created_capabilities + free_slot_start;
27    address_of_free_slot_cap_t = *(undefined8 *)(untyped_cptrs + (ulong)free_slot_counter * 8);
28                      /* capdl_spec[1] must refer to the address of the objects array
29                         0x68 must be the size of each slot */
30    current_object_address = (ulong)objects_counter * 0x68 + capdl_spec[1];
```

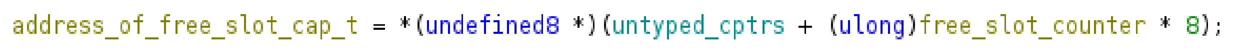
I have already named some of the variables. Going through them one by one. Inside the while loop, the two counters used are counting the total objects, and the free slots left in memory. We know this from the capdl_spec.c structure, and the sort_untypeds() function. On line 26:



```
index_of_free_slot_cap_t = (ulong)created_capabilities + free_slot_start;
```

We see that we are adding a variable, I have named created_capabilities, to the index of the first free slot. We learn later that this variable is the number of created capabilities. Each time a capability is created later, created_capabilities is incremented. All of these capabilities are placed in an array, which means that this is the index into the next free slot in that array.

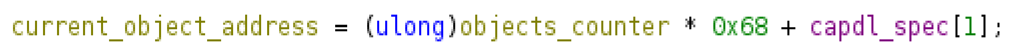
On line 27:



```
address_of_free_slot_cap_t = *(undefined8 *)(untyped_cptrs + (ulong)free_slot_counter * 8);
```

I marked the variable as "current_free_slot_index". Free slots refer to capability pointers, so each index is only 8 bytes. As the counter goes up, this calculates the memory address of the free slot.

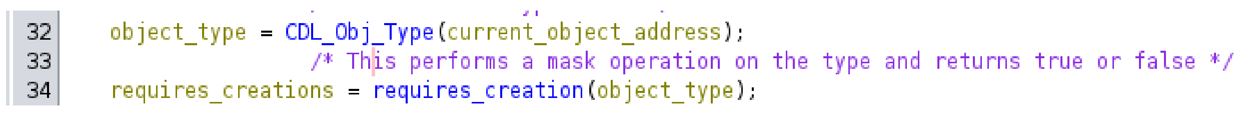
On line 30:



```
current_object_address = (ulong)objects_counter * 0x68 + capdl_spec[1];
```

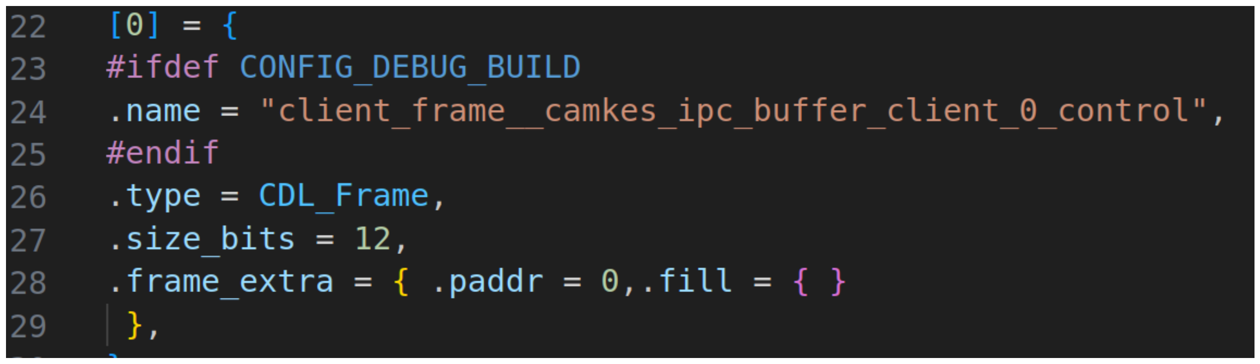
We see that the program multiples the number of objects by the max size of the objects (0x68). This is added to the address of the array start. This gives an index into the array.

We are very nearly reversing all of the variables in this function. Once this is done, parsing the function becomes much easier. We have two more variables to reverse:



```
  object_type = CDL_Obj_Type(current_object_address);
                    /* This performs a mask operation on the type and returns true or false */
  requires_creations = requires_creation(object_type);
```

CDL_Obj_Type simply just returns the value of the .type value in the capdl_spec.c file. You can see that field here:

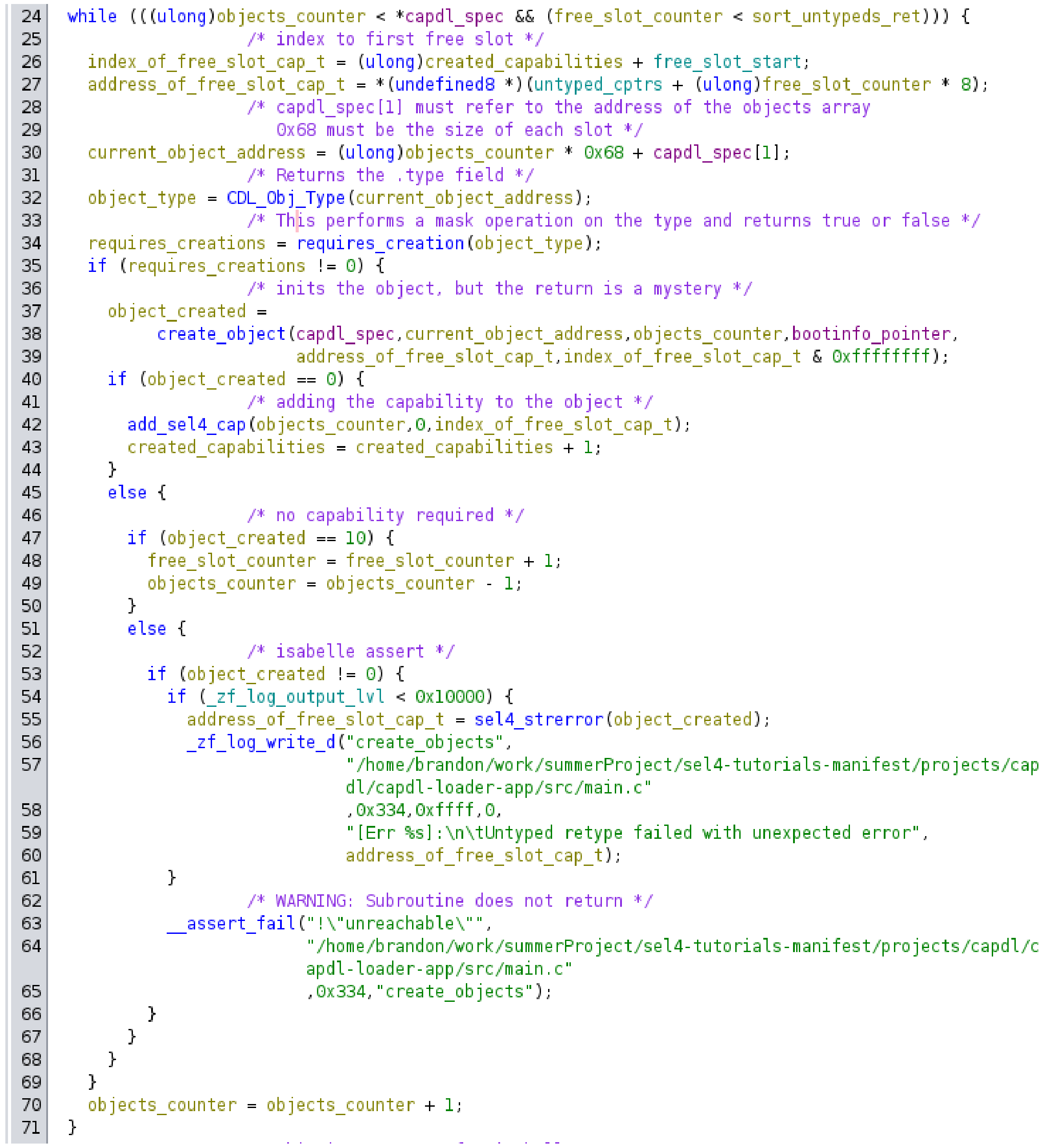


```
    [0] = {
    #ifdef CONFIG_DEBUG_BUILD
    .name = "client_frame__camkes_ipc_buffer_client_0_control",
    #endif
    .type = CDL_Frame,
    .size_bits = 12,
    .frame_extra = { .paddr = 0,.fill = { }
     },
```

Each object has a .type field.

Requires_creation probes the .type field of the object and determines if the object needs to be created or not. It returns true or false. If the object is created then we will increment our counters, if not we will move on to the next object.

Now we can view the full function:



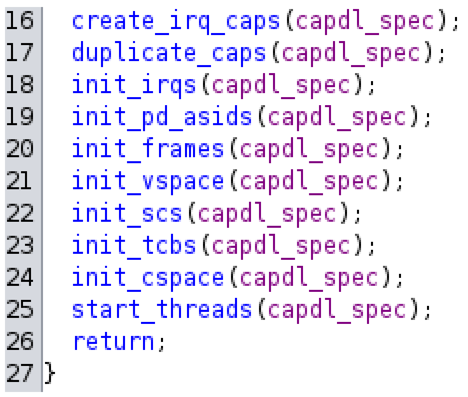
```
24  while (((ulong)objects_counter < *capdl_spec && (free_slot_counter < sort_untypeds_ret))) {
25                   /* index to first free slot */
26    index_of_free_slot_cap_t = (ulong)created_capabilities + free_slot_start;
27    address_of_free_slot_cap_t = *(undefined8 *)(untyped_cptrs + (ulong)free_slot_counter * 8);
28                   /* capdl_spec[1] must refer to the address of the objects array
29                      0x68 must be the size of each slot */
30    current_object_address = (ulong)objects_counter * 0x68 + capdl_spec[1];
31                   /* Returns the .type field */
32    object_type = CDL_Obj_Type(current_object_address);
33                   /* This performs a mask operation on the type and returns true or false */
34    requires_creations = requires_creation(object_type);
35    if (requires_creations != 0) {
36                   /* inits the object, but the return is a mystery */
37      object_created =
38           create_object(capdl_spec,current_object_address,objects_counter,bootinfo_pointer,
39                         address_of_free_slot_cap_t,index_of_free_slot_cap_t & 0xffffffff);
40      if (object_created == 0) {
41                   /* adding the capability to the object */
42        add_sel4_cap(objects_counter,0,index_of_free_slot_cap_t);
43        created_capabilities = created_capabilities + 1;
44      }
45      else {
46                   /* no capability required */
47        if (object_created == 10) {
48          free_slot_counter = free_slot_counter + 1;
49          objects_counter = objects_counter - 1;
50        }
51        else {
52                   /* isabelle assert */
53          if (object_created != 0) {
54            if (_zf_log_output_lvl < 0x10000) {
55              address_of_free_slot_cap_t = sel4_strerror(object_created);
56              _zf_log_write_d("create_objects",
57                              "/home/brandon/work/summerProject/sel4-tutorials-manifest/projects/cap
                                dl/capdl-loader-app/src/main.c"
58                              ,0x334,0xffff,0,
59                              "[Err %s]:\n\tUntyped retype failed with unexpected error",
60                              address_of_free_slot_cap_t);
61            }
62                   /* WARNING: Subroutine does not return */
63            __assert_fail("!\"unreachable\"",
64                          "/home/brandon/work/summerProject/sel4-tutorials-manifest/projects/capdl/c
                            apdl-loader-app/src/main.c"
65                          ,0x334,"create_objects");
66          }
67        }
68      }
69    }
70    objects_counter = objects_counter + 1;
71  }
```

We can see that this loop:

1. Checks if the object needs to be made (line 34)
2. Creates the object (line 37)
3. Either creates the capability or not (line 40, line 47)
4. Increments the object counter to move on to the next object (line 70)

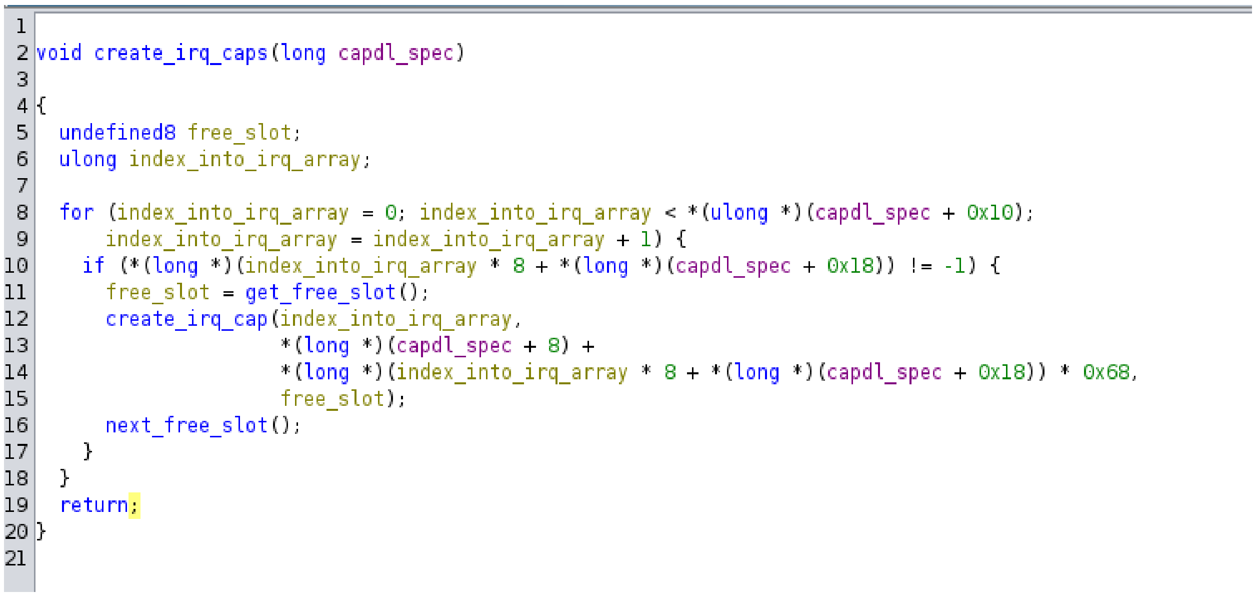
Astute observers will notice that we are ignoring a second while loop in the function. This function is not important at this moment. We may come back to it. We can conclude that kernel objects have been registered and capabilities pointing to them have been created.

Unfortunately, the pain is just beginning. We still have 9 functions to reverse:



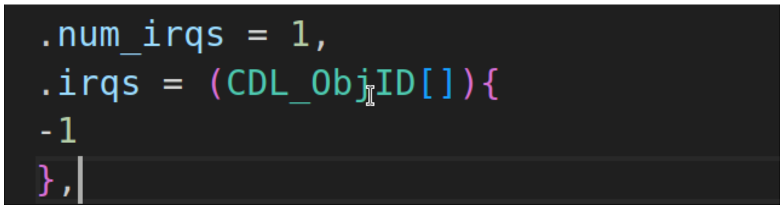
```
  create_irq_caps(capdl_spec);
  duplicate_caps(capdl_spec);
  init_irqs(capdl_spec);
  init_pd_asids(capdl_spec);
  init_frames(capdl_spec);
  init_vspace(capdl_spec);
  init_scs(capdl_spec);
  init_tcbs(capdl_spec);
  init_cspace(capdl_spec);
  start_threads(capdl_spec);
  return;
}
```

Here is the full create_irq_caps() function, with the variables already filled in:



```

void create_irq_caps(long capdl_spec)

{
  undefined8 free_slot;
  ulong index_into_irq_array;

  for (index_into_irq_array = 0; index_into_irq_array < *(ulong *)(capdl_spec + 0x10);
      index_into_irq_array = index_into_irq_array + 1) {
    if (*(long *)(index_into_irq_array * 8 + *(long *)(capdl_spec + 0x18)) != -1) {
      free_slot = get_free_slot();
      create_irq_cap(index_into_irq_array,
                     *(long *)(capdl_spec + 8) +
                     *(long *)(index_into_irq_array * 8 + *(long *)(capdl_spec + 0x18)) * 0x68,
                     free_slot);
      next_free_slot();
    }
  }
  return;
}

```

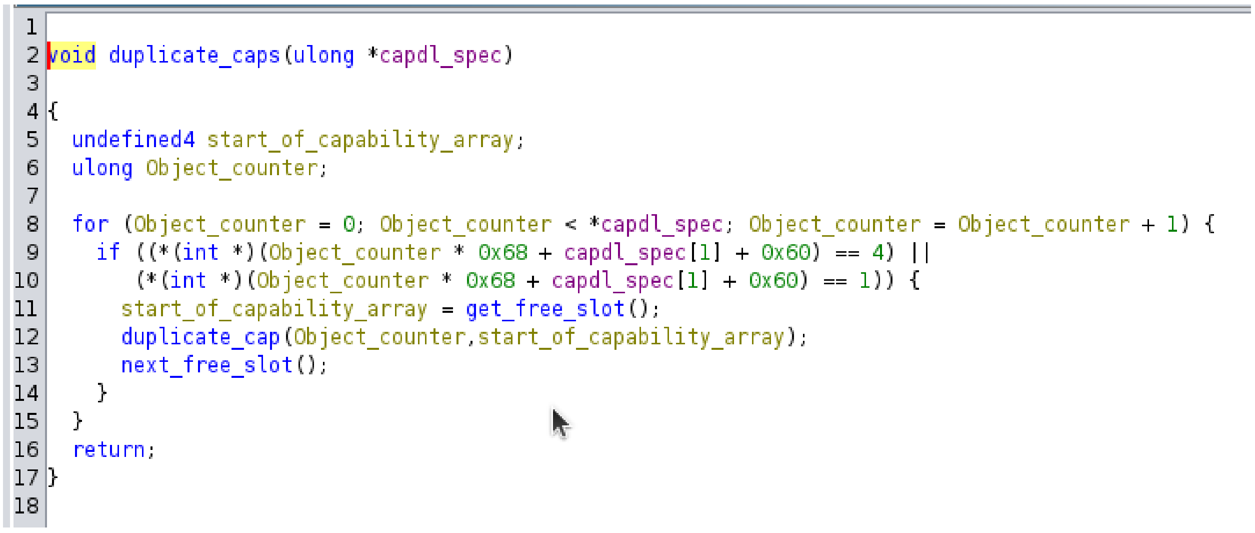
The indices you see refer to this structure in capdl_spec.c:



```
.num_irqs = 1,
.irqs = (CDL_ObjID[]){
-1
},
```

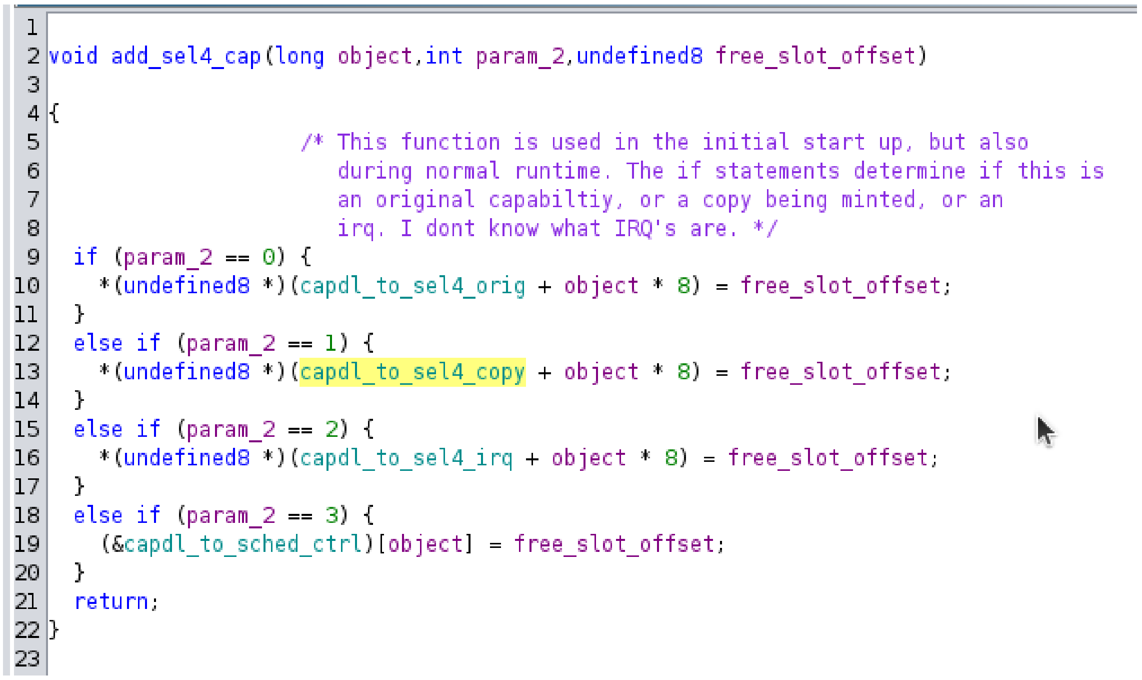
There isn't much to see here. The function iterates through the .irqs array and checks for irqs that need to be created. Our echo and client programs do not contain irq objects, so we can ignore this function for the most part.

Here is the next function, variables all renamed:



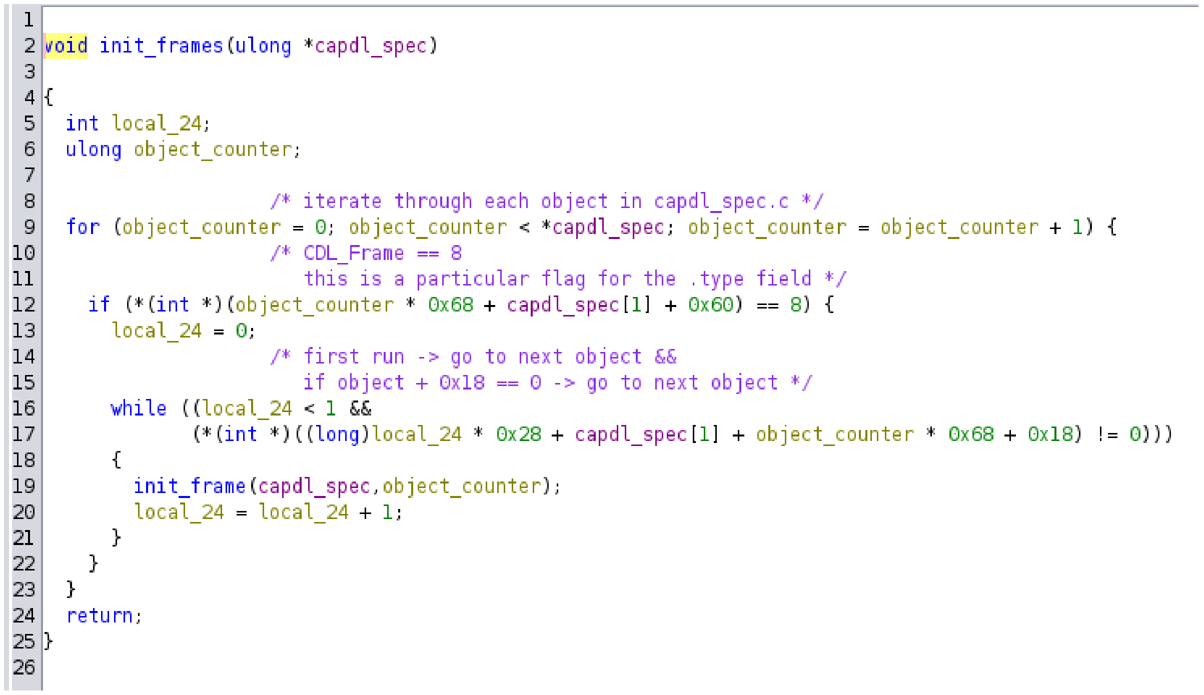
```

void duplicate_caps(ulong *capdl_spec)

{
  undefined4 start_of_capability_array;
  ulong Object_counter;

  for (Object_counter = 0; Object_counter < *capdl_spec; Object_counter = Object_counter + 1) {
    if ((*(int *)(Object_counter * 0x68 + capdl_spec[1] + 0x60) == 4) ||
       (*(int *)(Object_counter * 0x68 + capdl_spec[1] + 0x60) == 1)) {
      start_of_capability_array = get_free_slot();
      duplicate_cap(Object_counter,start_of_capability_array);
      next_free_slot();
    }
  }
  return;
}

```

This creates a second copy of the capabilities. The function iterates through the capabilities we just created with get_free_slot() and next_free_slot(). These names are a bit misleading. The memory location these refer to are occupied by the created capabilities. The duplicated capabilities get dumped into this global offset inside the duplicate_cap() function:



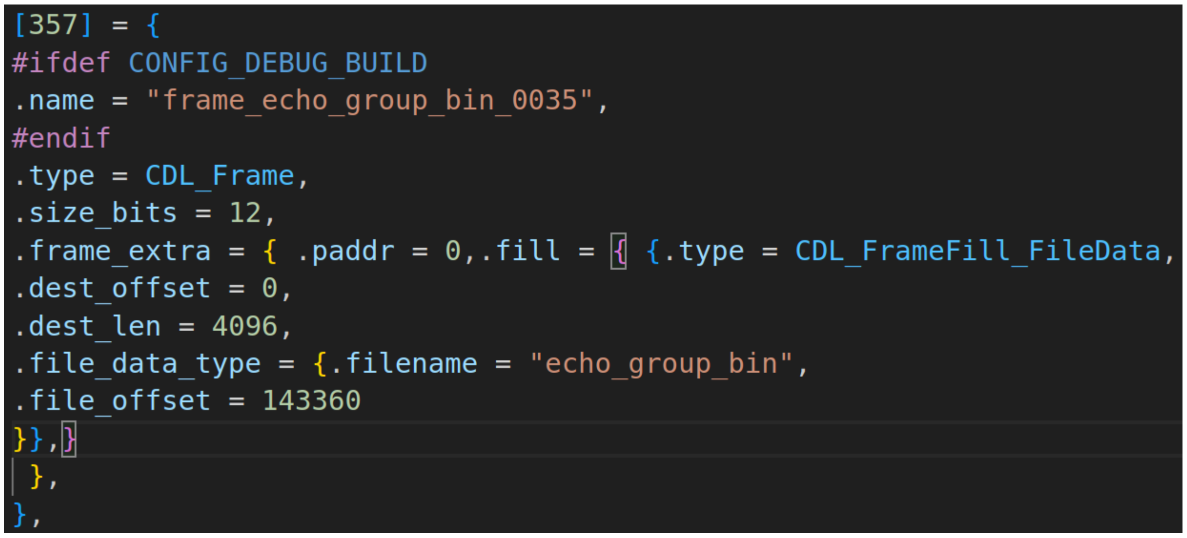
```

void add_sel4_cap(long object,int param_2,undefined8 free_slot_offset)

{
                    /* This function is used in the initial start up, but also
                       during normal runtime. The if statements determine if this is
                       an original capabiltiy, or a copy being minted, or an
                       irq. I dont know what IRQ's are. */
  if (param_2 == 0) {
    *(undefined8 *)(capdl_to_sel4_orig + object * 8) = free_slot_offset;
  }
  else if (param_2 == 1) {
    *(undefined8 *)(capdl_to_sel4_copy + object * 8) = free_slot_offset;
  }
  else if (param_2 == 2) {
    *(undefined8 *)(capdl_to_sel4_irq + object * 8) = free_slot_offset;
  }
  else if (param_2 == 3) {
    (&capdl_to_sched_ctrl)[object] = free_slot_offset;
  }
  return;
}

```

Next, we are going to look at init_frames(). This function is a bit difficult to parse. We will go over the high level idea. What init_frames() aims to do is initialize and load into memory the data required to start echo and client. These are frames, so this memory is being loaded directly into 4K blocks of RAM. Here is the function:



```
 1
 2 void init_frames(ulong *capdl_spec)
 3
 4 {
 5   int local_24;
 6   ulong object_counter;
 7
 8                     /* iterate through each object in capdl_spec.c */
 9   for (object_counter = 0; object_counter < *capdl_spec; object_counter = object_counter + 1) {
10                     /* CDL_Frame == 8
11                        this is a particular flag for the .type field */
12     if (*(int *)(object_counter * 0x68 + capdl_spec[1] + 0x60) == 8) {
13       local_24 = 0;
14                     /* first run -> go to next object &&
15                        if object + 0x18 == 0 -> go to next object */
16       while ((local_24 < 1 &&
17              (*(int *)((long)local_24 * 0x28 + capdl_spec[1] + object_counter * 0x68 + 0x18) != 0)))
18       {
19         init_frame(capdl_spec,object_counter);
20         local_24 = local_24 + 1;
21       }
22     }
23   }
24   return;
25 }
26
```

This is iterating through the capdl_spec.c file and checking for objects that require frame creation. Objects requiring frame creation look like this:

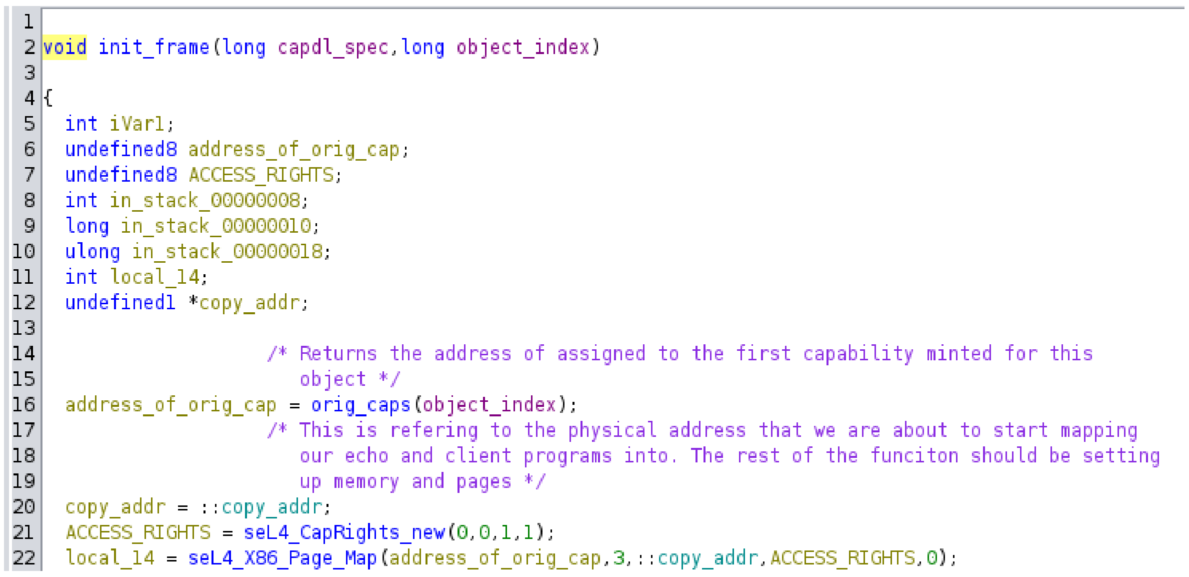


```
[357] = {
#ifdef CONFIG_DEBUG_BUILD
.name = "frame_echo_group_bin_0035",
#endif
.type = CDL_Frame,
.size_bits = 12,
.frame_extra = { .paddr = 0,.fill = { {.type = CDL_FrameFill_FileData,
.dest_offset = 0,
.dest_len = 4096,
.file_data_type = {.filename = "echo_group_bin",
.file_offset = 143360
}},}
 },
},
```

Here the .type is CDL_Frame. This is a constant that evaluates to 8. This is the check that is happening on line 12:

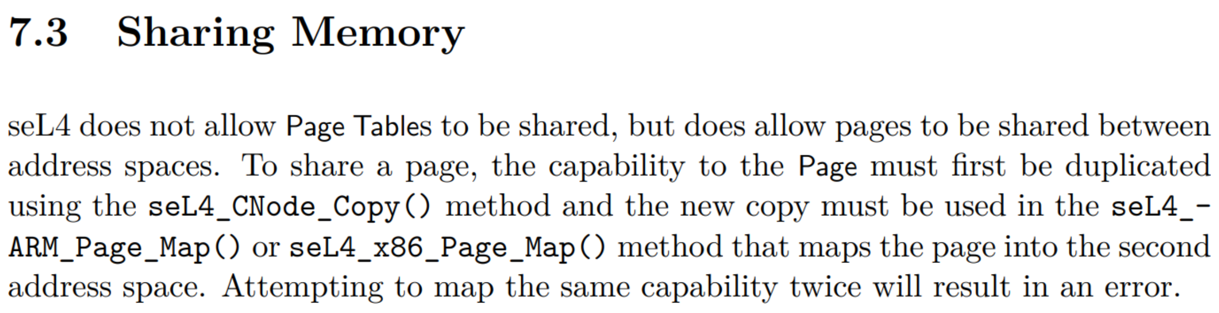
```
if (*(int *)(object_counter * 0x68 + capdl_spec[1] + 0x60) == 8) {
```

The rest of the fields will be used momentarily. The function loops through all of the objects in capdl_spec.c and checks if they need to be mapped into memory at initialization. For each frame that does need to be created, it calls init_frame(). Here is the beginning of that function:



```
 1
 2 void init_frame(long capdl_spec,long object_index)
 3
 4 {
 5   int iVar1;
 6   undefined8 address_of_orig_cap;
 7   undefined8 ACCESS_RIGHTS;
 8   int in_stack_00000008;
 9   long in_stack_00000010;
10   ulong in_stack_00000018;
11   int local_14;
12   undefined1 *copy_addr;
13
14                     /* Returns the address of assigned to the first capability minted for this
15                        object */
16   address_of_orig_cap = orig_caps(object_index);
17                     /* This is refering to the physical address that we are about to start mapping
18                        our echo and client programs into. The rest of the funciton should be setting
19                        up memory and pages */
20   copy_addr = ::copy_addr;
21   ACCESS_RIGHTS = seL4_CapRights_new(0,0,1,1);
22   local_14 = seL4_X86_Page_Map(address_of_orig_cap,3,::copy_addr,ACCESS_RIGHTS,0);
```

On line 16, the function gets the address of the first capability minted for this object. This is the one created during the create_objects() function. Copy_addr is the location in memory that we are going to map in echo or client. Most of the magic happens on line 22. Here is an excerpt from the reference manual regarding seL4_x86_Page_Map():

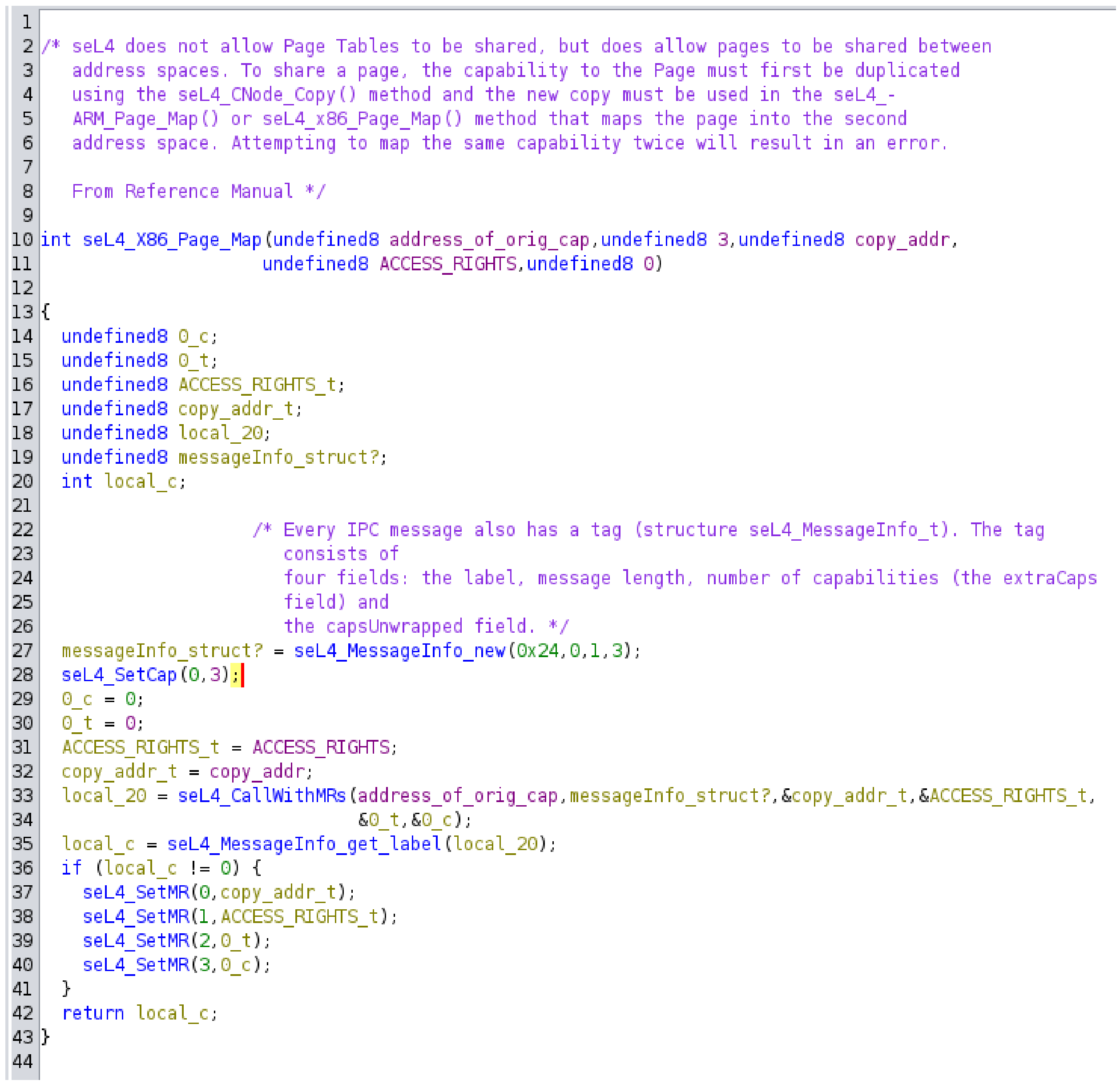


7.3 Sharing Memory

seL4 does not allow Page Tables to be shared, but does allow pages to be shared between address spaces. To share a page, the capability to the Page must first be duplicated using the `seL4_CNode_Copy()` method and the new copy must be used in the `seL4_ARM_Page_Map()` or `seL4_x86_Page_Map()` method that maps the page into the second address space. Attempting to map the same capability twice will result in an error.

Before we go into sel4_x86_Page_Map(), we must understand at a high level what it is hoping to accomplish. sel4_x86_Page_Map() is supposed to map the full contents of client/echo into their respective virtual memory spaces.

Here is the function:



```

/* seL4 does not allow Page Tables to be shared, but does allow pages to be shared between
   address spaces. To share a page, the capability to the Page must first be duplicated
   using the seL4_CNode_Copy() method and the new copy must be used in the seL4_-
   ARM_Page_Map() or seL4_x86_Page_Map() method that maps the page into the second
   address space. Attempting to map the same capability twice will result in an error.

   From Reference Manual */

int seL4_X86_Page_Map(undefined8 address_of_orig_cap,undefined8 3,undefined8 copy_addr,
                     undefined8 ACCESS_RIGHTS,undefined8 0)

{
  undefined8 0_c;
  undefined8 0_t;
  undefined8 ACCESS_RIGHTS_t;
  undefined8 copy_addr_t;
  undefined8 local_20;
  undefined8 messageInfo_struct?;
  int local_c;

                    /* Every IPC message also has a tag (structure seL4_MessageInfo_t). The tag
                       consists of
                       four fields: the label, message length, number of capabilities (the extraCaps
                       field) and
                       the capsUnwrapped field. */
  messageInfo_struct? = seL4_MessageInfo_new(0x24,0,1,3);
  seL4_SetCap(0,3);
  0_c = 0;
  0_t = 0;
  ACCESS_RIGHTS_t = ACCESS_RIGHTS;
  copy_addr_t = copy_addr;
  local_20 = seL4_CallWithMRs(address_of_orig_cap,messageInfo_struct?,&copy_addr_t,&ACCESS_RIGHTS_t,
                              &0_t,&0_c);
  local_c = seL4_MessageInfo_get_label(local_20);
  if (local_c != 0) {
    seL4_SetMR(0,copy_addr_t);
    seL4_SetMR(1,ACCESS_RIGHTS_t);
    seL4_SetMR(2,0_t);
    seL4_SetMR(3,0_c);
  }
  return local_c;
}

```

(Note: Finish)
(Hint: do the mask math, and look up syscall identifiers.)

A really incredible description of the above system can be found in the paper “Formally Verified System Initialisation”. Reference that paper if any of the above seems confusing or incomplete.

CAmkES Glue Code:

We have covered how CapDL and CAmkES create the initial thread, and what that initial thread does, but it is still unclear what effect CAmkES has once the userland programs have started.

The CAmkES documentation claims that CAmkES makes “glue code” that facilitates the IPC communication between processes modeled CAmkES language. We want to find that code.

Let’s look at Ghidra again :)

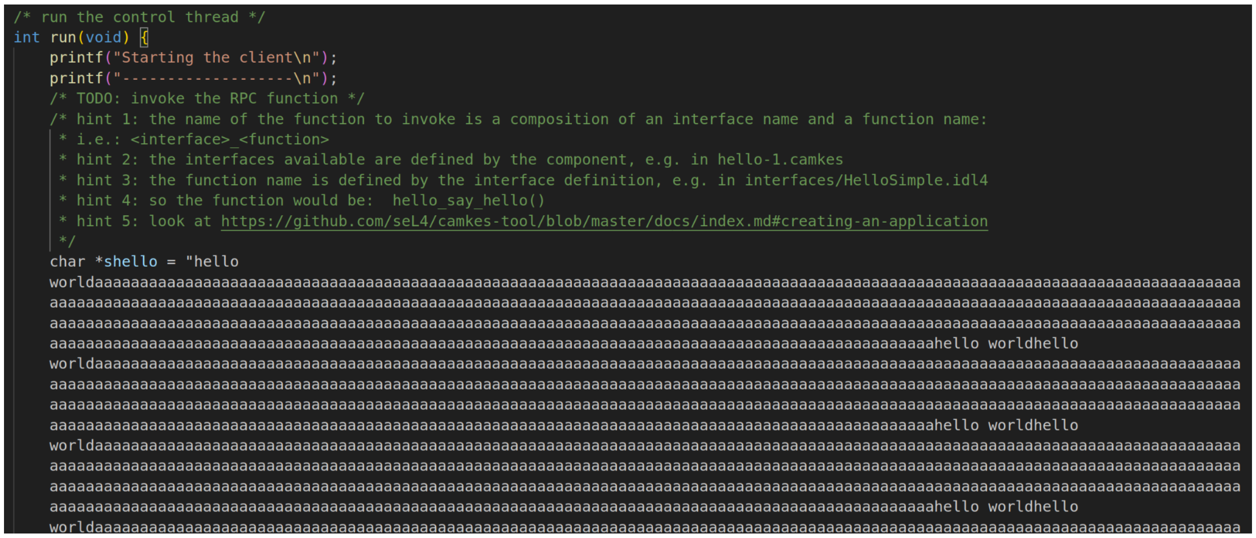
Inside of the build directory, at the top level, you will find a file called client_group_bin. Loading this into Ghidra, this is the main function:

(Note: Finish)

Glue Code Protection:

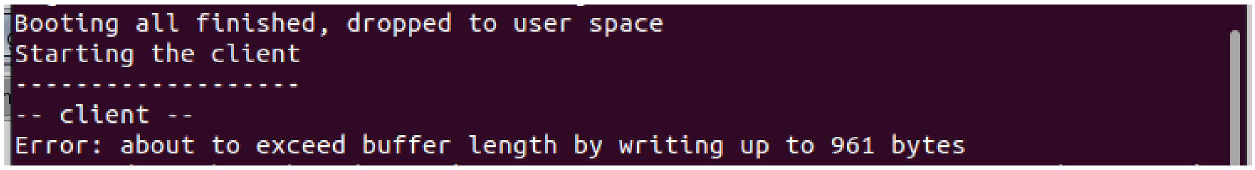
Glue code refers to code added to CAmkES IPC calls that are supposed to apply the protections to those calls. In this section, we will hunt down where this glue code is located, and what it looks like.

I have a modified version of hello-camkes-1. You can find this in the tutorials. This is my modified client program:



```
/* run the control thread */
int run(void) {
    printf("Starting the client\n");
    printf("-------------------\n");
    /* TODO: invoke the RPC function */
    /* hint 1: the name of the function to invoke is a composition of an interface name and a function name:
     * i.e.: <interface>_<function>
     * hint 2: the interfaces available are defined by the component, e.g. in hello-1.camkes
     * hint 3: the function name is defined by the interface definition, e.g. in interfaces/HelloSimple.idl4
     * hint 4: so the function would be:  hello_say_hello()
     * hint 5: look at https://github.com/seL4/camkes-tool/blob/master/docs/index.md#creating-an-application
     */
    char *shello = "hello
    worldaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaahello worldhello
    worldaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaahello worldhello
    worldaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaahello worldhello
    worldaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

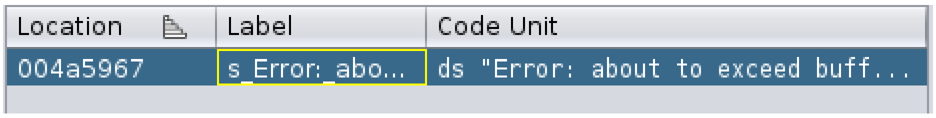
This shello string is meant to be sent via message passing. If you compile and run seL4 with the above client, you get this error message:



```
Booting all finished, dropped to user space
Starting the client
-------------------
-- client --
Error: about to exceed buffer length by writing up to 961 bytes
```

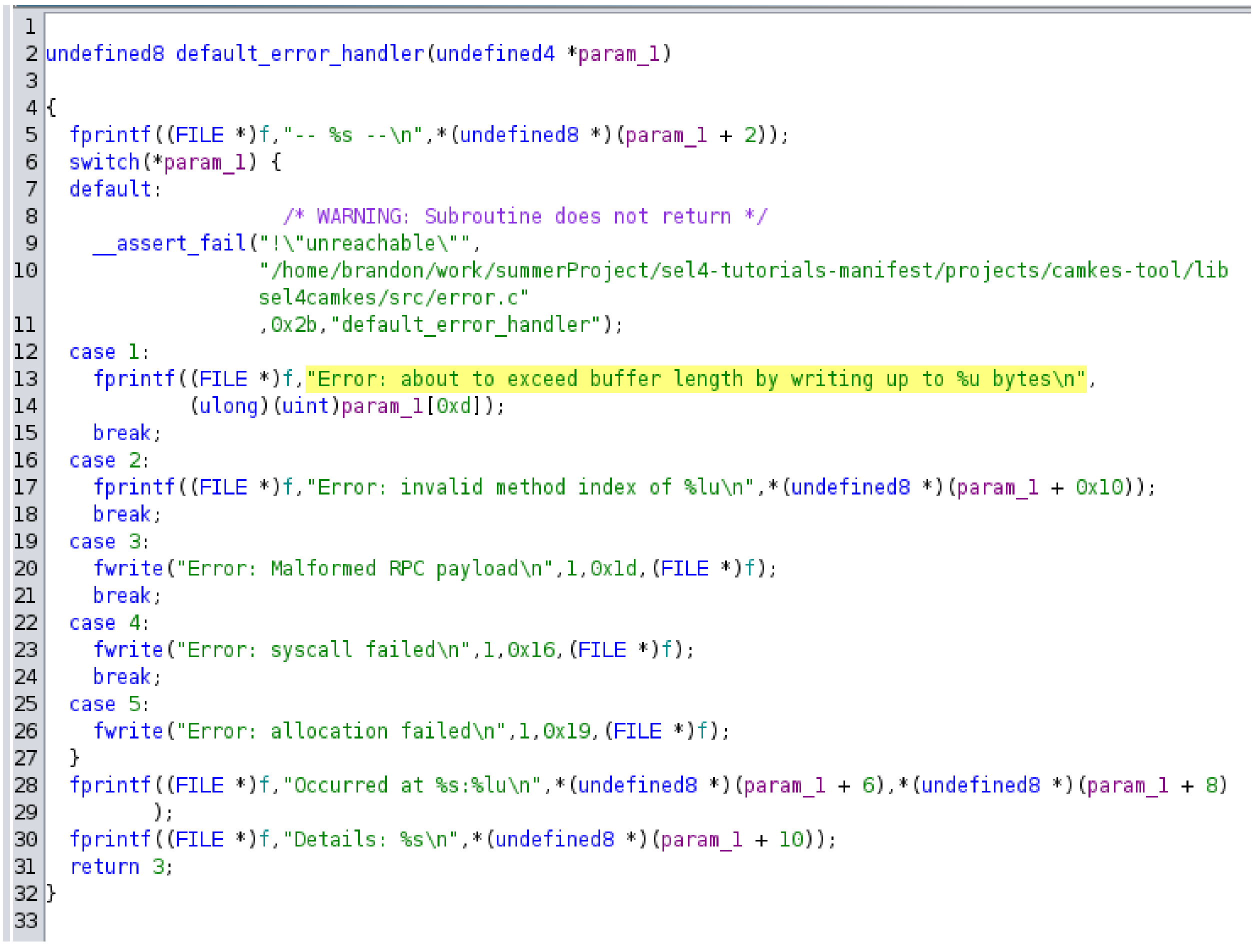
This message is telling us that we can't write more than 960 bytes. This is inline with what we would expect from a formally verified system. seL4 is not allowing us to write arbitrarily large buffers.

We would like to know how this error message is getting printed. Where is seL4 checking to make sure the buffer about to be written is of an acceptable length. If we look inside the client_group_bin, we can search for the error message:



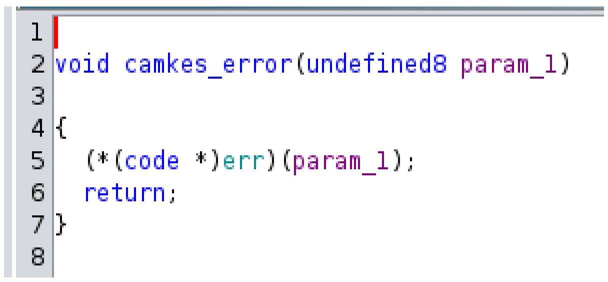
Location	Label	Code Unit
004a5967	s_Error:_abo...	ds "Error: about to exceed buff...

We find only one instance of this error message in memory. By checking which memory references this address, we find this:



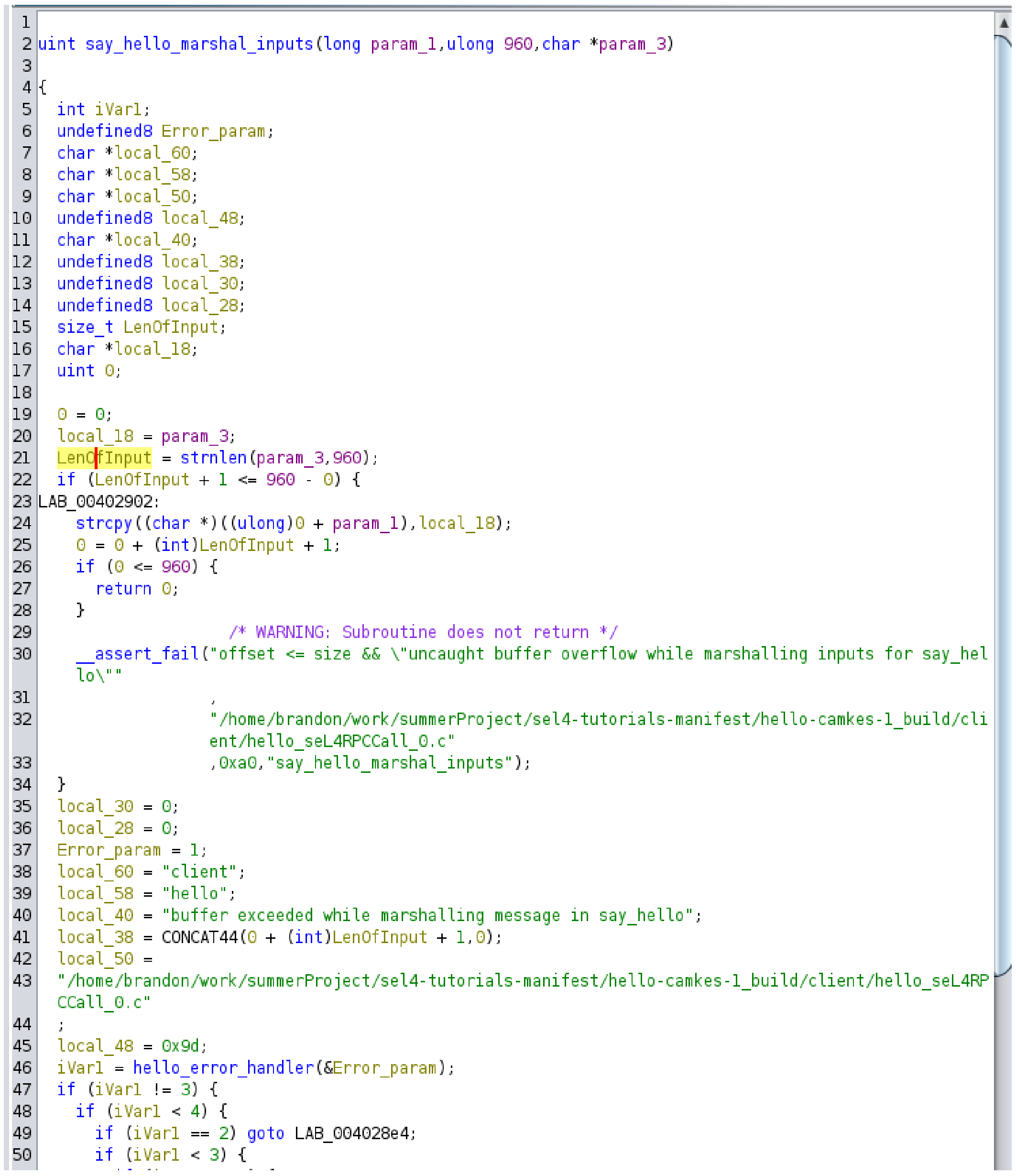
```

undefined8 default_error_handler(undefined4 *param_1)

{
  fprintf((FILE *)f,"-- %s --\n",*(undefined8 *)(param_1 + 2));
  switch(*param_1) {
  default:
                    /* WARNING: Subroutine does not return */
    __assert_fail("!\"unreachable\"",
                  "/home/brandon/work/summerProject/sel4-tutorials-manifest/projects/camkes-tool/lib
                  sel4camkes/src/error.c"
                  ,0x2b,"default_error_handler");
  case 1:
    fprintf((FILE *)f,"Error: about to exceed buffer length by writing up to %u bytes\n",
            (ulong)(uint)param_1[0xd]);
    break;
  case 2:
    fprintf((FILE *)f,"Error: invalid method index of %lu\n",*(undefined8 *)(param_1 + 0x10));
    break;
  case 3:
    fwrite("Error: Malformed RPC payload\n",1,0x1d,(FILE *)f);
    break;
  case 4:
    fwrite("Error: syscall failed\n",1,0x16,(FILE *)f);
    break;
  case 5:
    fwrite("Error: allocation failed\n",1,0x19,(FILE *)f);
  }
  fprintf((FILE *)f,"Occurred at %s:%lu\n",*(undefined8 *)(param_1 + 6),*(undefined8 *)(param_1 + 8)
         );
  fprintf((FILE *)f,"Details: %s\n",*(undefined8 *)(param_1 + 10));
  return 3;
}

```

We should note that if the input to this function is 1, then we will get the error message we are looking for. Let's try to find out how this function might get called with input 1. If we check for references of default_error_handler, we find this:



```
void camkes_error(undefined8 param_1)

{
  (*(code *)err)(param_1);
  return;
}
```

And checking references to camkes_error, we find more functions. If we continue to follow this chain of references, we eventually find this:



```
uint say_hello_marshal_inputs(long param_1,ulong 960,char *param_3)

{
  int iVar1;
  undefined8 Error_param;
  char *local_60;
  char *local_58;
  char *local_50;
  undefined8 local_48;
  char *local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  size_t LenOfInput;
  char *local_18;
  uint 0;

  0 = 0;
  local_18 = param_3;
  LenOfInput = strnlen(param_3,960);
  if (LenOfInput + 1 <= 960 - 0) {
LAB_00402902:
    strcpy((char *)((ulong)0 + param_1),local_18);
    0 = 0 + (int)LenOfInput + 1;
    if (0 <= 960) {
      return 0;
    }
                    /* WARNING: Subroutine does not return */
    __assert_fail("offset <= size && \"uncaught buffer overflow while marshalling inputs for say_hel
    lo\""
                  ,
                  "/home/brandon/work/summerProject/sel4-tutorials-manifest/hello-camkes-1_build/cli
                  ent/hello_seL4RPCCall_0.c"
                  ,0xa0,"say_hello_marshal_inputs");
  }
  local_30 = 0;
  local_28 = 0;
  Error_param = 1;
  local_60 = "client";
  local_58 = "hello";
  local_40 = "buffer exceeded while marshalling message in say_hello";
  local_38 = CONCAT44(0 + (int)LenOfInput + 1,0);
  local_50 =
  "/home/brandon/work/summerProject/sel4-tutorials-manifest/hello-camkes-1_build/client/hello_seL4RP
  CCall_0.c"
  ;
  local_48 = 0x9d;
  iVar1 = hello_error_handler(&Error_param);
  if (iVar1 != 3) {
    if (iVar1 < 4) {
      if (iVar1 == 2) goto LAB_004028e4;
      if (iVar1 < 3) {
```

We do not need to parse this entire function. The important lines are line 21, line 37, and line 46. One line 21 and 22, we see the check for buffer size. On line 37, the error parameter is set to 1, and on line 46, the 1 is passed into the error function. This will chain down the references shown above and eventually reach the switch statement in default_error_handler(). If the if statement on line 22 fails, then we will get the original error message.

This code is automatically generated and injected by the CAmkES framework, and the code injected should reflect the scope provided by the CAmkES ADL. The code seen above is designed for error detection when communicating with outside processes, via IPC. Bypassing this check is as simple as setting the instruction pointer past the if check, but this does not necessarily bypass the security of the framework. The kernel still has to copy the contents of the buffer, and the kernel very likely performs its own bounds checks (more on this later).

Booting seL4 on Hardware:

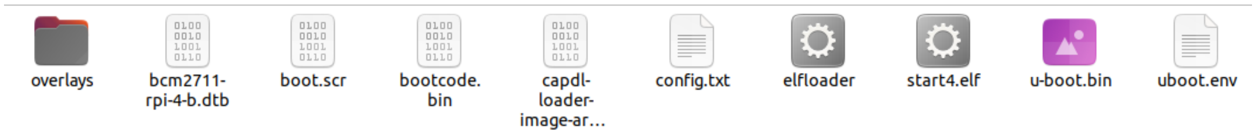
We will be booting seL4 on the Raspberry Pi 4. It has an ARM Cortex-A72 contained within a SoC onboard. All of the seL4 build directions encourage users to use the u-boot tool as a final stage bootloader. We will cover this tool in detail shortly. The boot process using u-boot looks like this:

1. Stage 1 is an on board ROM bootloader. This loads bootcode.bin.
2. Bootcode.bin inits the GPU and loads the next stage boot loader, start4.elf.
3. Start4.elf then loads our u-boot image. This is when the seL4 boot process begins and the raspberry pi generic boot process ends.
4. U-boot loads in Elfloader, which loads both images into memory
5. The kernel takes over from Elfloader and initializes the initial thread
6. The initial thread runs, schedules userland programs, and yields control.

There is a good reason for all of these bootloaders. The first ROM bootloader is required to be on board. Something must read the other bootloaders off of the SD card. Once bootcode.bin is in memory, it initializes the GPU. Start4.elf then initializes the CPU and loads u-boot into the cpu. We can expect that after the raspberry pi boot process has finished, the CPU and GPU are initialized (to a basic extent) and u-boot is the next thing that will execute. We should expect that u-boot will need to perform additional initialization and hardware configuration, and then it will load seL4's Elfloader. Elfloader will then load both images, drop to the kernel, and the process described in the sections above will take place.

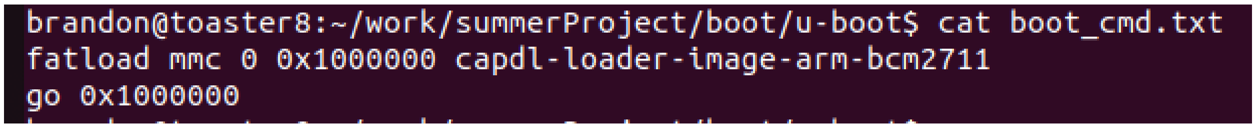
Additionally, seL4 will spit all of its output out over serial IO. **If you have an HDMI cord connected to a monitor, it will *NOT* display the output.** In order to see the output, you will need to download minicom. Connecting a serial cable from the Raspberry Pi 4 to your computer will allow minicom to collect the output and display it on your terminal. As we will be using u-boot, you will also need to download and build u-boot. You can find instructions for building u-boot here. When attempting the git revert, git might throw any number of errors. Be careful to examine the output to make sure the revert actually went through. The errors are all easy to handle, just run the commands git recommends.

Once you have a Pi, serial cable, minicom, SDcard, and a way to connect the SDcard to your computer, then you are ready to boot the raspberry pi. Here is a picture of the contents of my SDcard:



Going one-by-one from left to right:

1. Overlays are the files for the correct configuration of device trees on the pi (I have no idea what a device tree is, but we must include it). It can be found here. Note: You want the entire folder. All files.
2. Bcm2711-rpi-4-b.dtb is another device tree thing. Again, I have no idea what this does, but it is required. You can find it here.
3. Boot.scr is a bit more involved to get our hands on. It is a binary file containing the u-boot commands that will run automatically at start up. First, create a boot_cmd.txt file inside the u-boot folder. Here are the contents of my boot_cmd.txt:



```
brandon@toaster8:~/work/summerProject/boot/u-boot$ cat boot_cmd.txt
fatload mmc 0 0x1000000 capdl-loader-image-arm-bcm2711
go 0x1000000
```

The fatload command will load the capdl-* file from device 0 (the sd card) to the RAM address 0x1000000. If you try to use the address 0x10000000, like the seL4 guide recommends, then the boot process will fail. The start address is temperamental in general, many addresses did not work. 0x1000000 works, but in case you run into issues, you should be aware of a few things selecting a start address. Somewhere in memory is the u-boot binary. Loading the kernel into the u-boot section can cause weird undefined behavior. U-boot is close to 0x0 (I think). Additionally, the raspberry pi 4 has addresses that are reserved for hardware functionality. Things like ARM chip configuration, GPIO pins, and onboard circuitry. Writing to these fields will also cause undefined behavior. If the machine is doing something unexplainable and cursed, then just try moving the load address to somewhere else. Alright, with that out of the way, let's continue. Boot_cmd.txt is just a list of u-boot terminal commands. These two commands load our program into memory and jump to it. To "compile" this boot_cmd.txt into the binary version (boot.scr) we must run this command:

./u-boot/tools/mkimage -A arm64 -T script -C none -d ./u-boot/boot_cmd.txt ./u-boot/boot.scr;

4. Bootcode.bin is part of the raspberry pi boot chain, already discussed above. It can be found here.
5. Capdl-loader-image is also a bit involved to get our hands on. This is an image containing elfloader, the initial thread, and the kernel. The first address is the reset vector, which should start the setup process and eventually run the CAmkES programs. (Note: This is a little confusing given our current understanding of elfloader. The elfloader section might need to be revised after a deeper dive. For clarification, we expect to have 3 binary files, one for capDL, one for elfloader, and one for the kernel. All three of these get wrapped into a single image, which is not in line with what we expect). Inorder to compile seL4 and a CAmkES project, we need a CAmkES project. Rather than roll our own, there are a myriad of examples. The one we will be using can be compiled using the instructions found here. We need to make a slight adjustment. The instructions generate a binary for a different board, meant to run on qemu in simulation. We want a binary meant to run on a real raspberry pi. Change the line:

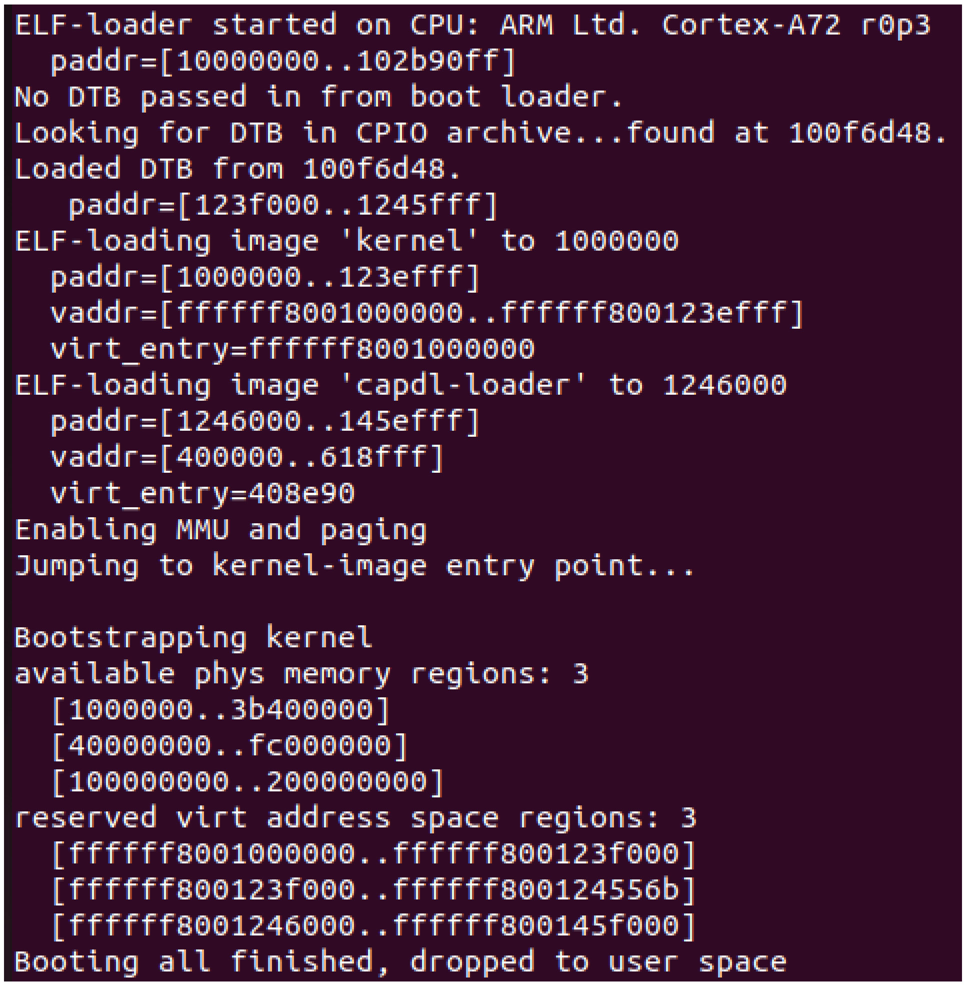
   ```
   ../init-build.sh -DPLATFORM=sabre -DAARCH32=1 -DCAMKES_APP=adder -DSIMULATION=1
   ```

   To:

   ```
   ../init-build.sh -DPLATFORM=rpi4 -DAARCH64=1 -DCAMKES_APP=adder
   ```

   Inside the git repository that you clone, camkes-project, you will find a folder called “kernel”. Unsurprisingly, this folder contains the seL4 kernel code that will be compiled into our image. **Do Not Forget:** You must edit the bcm2711 platform code to use a different serial number. The instructions can be found here. Additionally, if you follow the seL4 build instructions, you should have a cross-compiler installed. For some reason, my cross compiler was not recognized by the above command. If this happens, you can manually install the cross compiler here or here. Just add it to your PATH, .local folder, or whatever. Just make sure it is installed.
6. I had to borrow the config.txt from Sandy. I don’t know where this file comes from. (Note: Find out where to find this file and update)
7. This copy of elfloader is not used. I expected elfloader to be a part of the boot process, so I keep it around in case it comes up. It can be found in the elfloader folder after seL4 is built (Check your build directory).
8. Start4.elf is a raspberry pi boot file, and can be found here.
9. U-boot.bin is created as part of the u-boot build process. It can be found in the top level of the u-boot repo after following the build instructions.
10. It is unclear if uboot.env is required, but I keep it just in case. After the build process for u-boot, in the top level directory, there will be a file called .config. This is a txt file containing environment variables for u-boot. I renamed .config to uboot.env as that is the naming convention u-boot expects, and I copied the file to the SDcard.

With this, you have exactly the same files that I had when running my demos (even the unnecessary files). Plug the Pi into your computer, run minicom, and then supply power to the raspberry pi. You should see output like so:



```
ELF-loader started on CPU: ARM Ltd. Cortex-A72 r0p3
  paddr=[10000000..102b90ff]
No DTB passed in from boot loader.
Looking for DTB in CPIO archive...found at 100f6d48.
Loaded DTB from 100f6d48.
   paddr=[123f000..1245fff]
ELF-loading image 'kernel' to 1000000
  paddr=[1000000..123efff]
  vaddr=[ffffff8001000000..ffffff800123efff]
  virt_entry=ffffff8001000000
ELF-loading image 'capdl-loader' to 1246000
  paddr=[1246000..145efff]
  vaddr=[400000..618fff]
  virt_entry=408e90
Enabling MMU and paging
Jumping to kernel-image entry point...

Bootstrapping kernel
available phys memory regions: 3
  [1000000..3b400000]
  [40000000..fc000000]
  [100000000..200000000]
reserved virt address space regions: 3
  [ffffff8001000000..ffffff800123f000]
  [ffffff800123f000..ffffff800124556b]
  [ffffff8001246000..ffffff800145f000]
Booting all finished, dropped to user space
```

Congratulations. You have booted the most temperamental OS in all of existence.

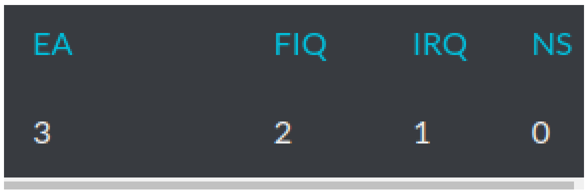
Examining the output, we can see that ElfLoader is in fact running, but we never loaded the elfloader file into memory. This means that elfloader, the kernel, and the initial thread are all compiled into the same memory image. I suspect that elfloader’s primary function is to relocate the kernel and the initial thread if u-boot (IE us) placed them in a wonky or unworkable position. I suspect that I misinterpreted this relocation feature as loading from disk during my initial analysis of elfloader. Additionally, elfloader inits certain hardware features that the kernel expects. This is still required even if relocation is not.

U-Boot Internals:

(Note: Revise section. The errors have been resolved and instead of talking about them, we can just directly talk about the correct configuration. See above).

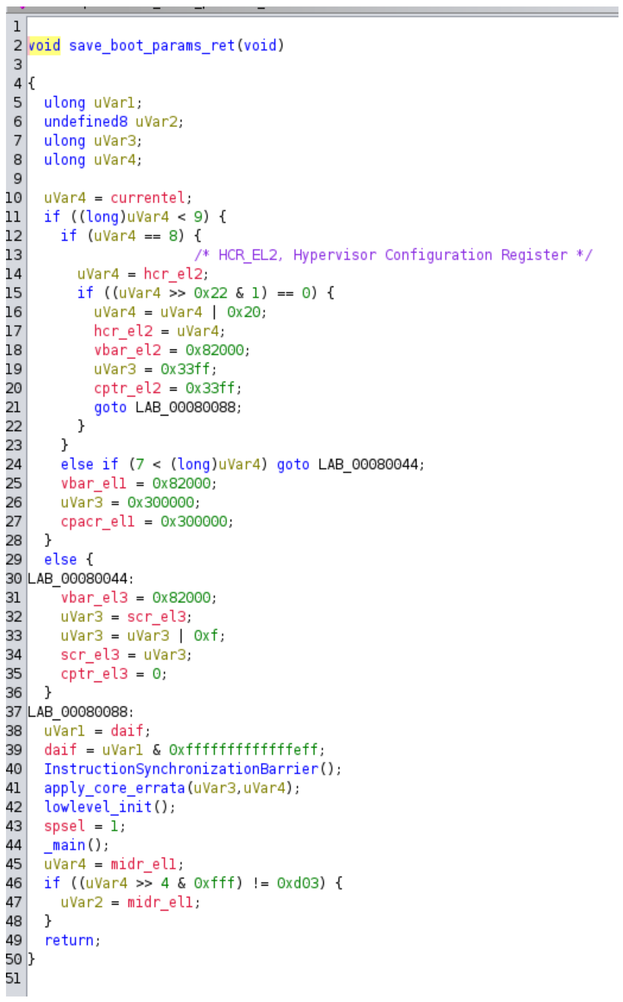
To understand why this error might be occurring, we are going to jump straight into the code of u-boot. We have an idea of the state of the raspberry pi once u-boot takes control. U-boot is loaded, seL4 and initial thread are not, the GPU and CPU are capable of running code and U-boot is inside the CPU. Looking inside the u-boot ARMv8 folder we find start.S. This file is the very first thing that runs. We can confirm this by examining the u-boot binary file that is generated by building u-boot with the raspberry pi configuration. You can find instructions on how to do this here.

To understand start.S, it helps to understand the structure of the ARM chip that runs on the raspberry pi. The chip has many configuration registers. These are called "system registers". The chip also has general purpose registers, some of which are hijacked for specific purposes. The general registers are typically referred to as r0 - r30 and the system registers are referred to by name. Configuration of the board is done by writing values to the system registers. The system registers are mostly bit flags, where each bit turns on or off a feature. For example, this instruction is reading the value of the scr_el3 register. The program then ORs it with new flags, and adds it back to the system register. The comment reads: /* SCR_EL3.NS|IRQ|FIQ|EA */. We would expect that this instruction set is setting the NS IRQ FIQ and EA flags in this system register. 0xf = 1111 in binary, so this effectively writes 1's to the first 4 bits. Looking at the ARM developer link for scr_el3, we see that the first for bits of the register are as follows:



EA	FIQ	IRQ	NS
3	2	1	0

Exactly what we expected. The majority of start.S is system register configuration. We could reverse the meaning of each of these configurations, but we must stay focused on why we are doing this. U-boot is not working and we want to know why. Because U-boot has reached a point where it is printing errors to the screen, we can assume that u-boot successfully gets past the configuration stage. Later on in the _start() function we find this. This call to main leaves _start(). Looking in Ghidra we can see this as well:

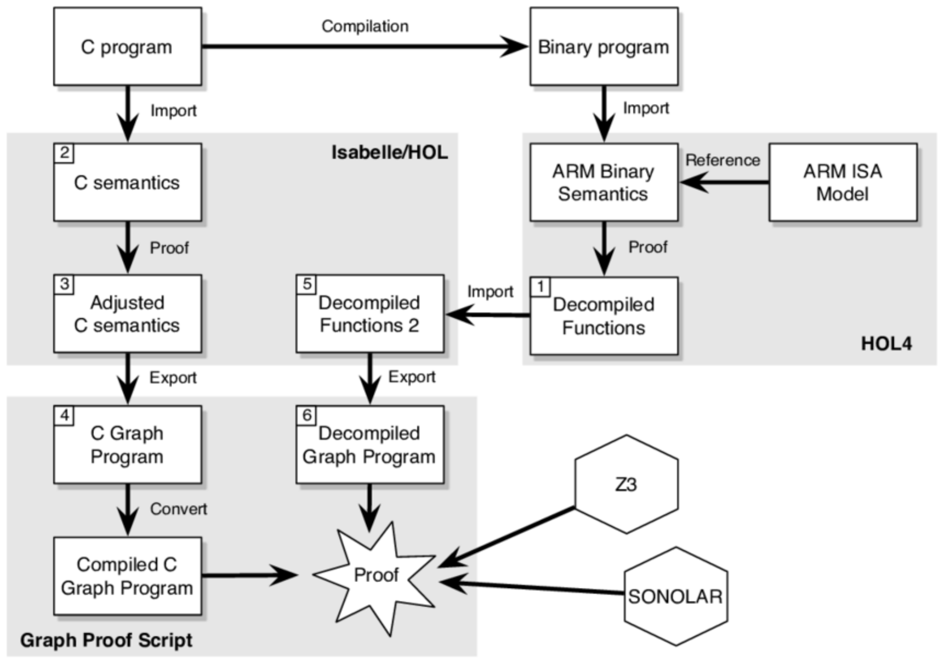


```
1
2  void save_boot_params_ret(void)
3
4  {
5    ulong uVar1;
6    undefined8 uVar2;
7    ulong uVar3;
8    ulong uVar4;
9
10   uVar4 = currentel;
11   if ((long)uVar4 < 9) {
12     if (uVar4 == 8) {
13                     /* HCR_EL2, Hypervisor Configuration Register */
14       uVar4 = hcr_el2;
15       if ((uVar4 >> 0x22 & 1) == 0) {
16         uVar4 = uVar4 | 0x20;
17         hcr_el2 = uVar4;
18         vbar_el2 = 0x82000;
19         uVar3 = 0x33ff;
20         cptr_el2 = 0x33ff;
21         goto LAB_00080088;
22       }
23     }
24     else if (7 < (long)uVar4) goto LAB_00080044;
25     vbar_el1 = 0x82000;
26     uVar3 = 0x300000;
27     cpacr_el1 = 0x300000;
28   }
29   else {
30 LAB_00080044:
31     vbar_el3 = 0x82000;
32     uVar3 = scr_el3;
33     uVar3 = uVar3 | 0xf;
34     scr_el3 = uVar3;
35     cptr_el3 = 0;
36   }
37 LAB_00080088:
38   uVar1 = daif;
39   daif = uVar1 & 0xfffffffffffffeff;
40   InstructionSynchronizationBarrier();
41   apply_core_errata(uVar3,uVar4);
42   lowlevel_init();
43   spsel = 1;
44   _main();
45   uVar4 = midr_el1;
46   if ((uVar4 >> 4 & 0xfff) != 0xd03) {
47     uVar2 = midr_el1;
48   }
49   return;
50 }
51
```

Line 44 contains a call to _main().

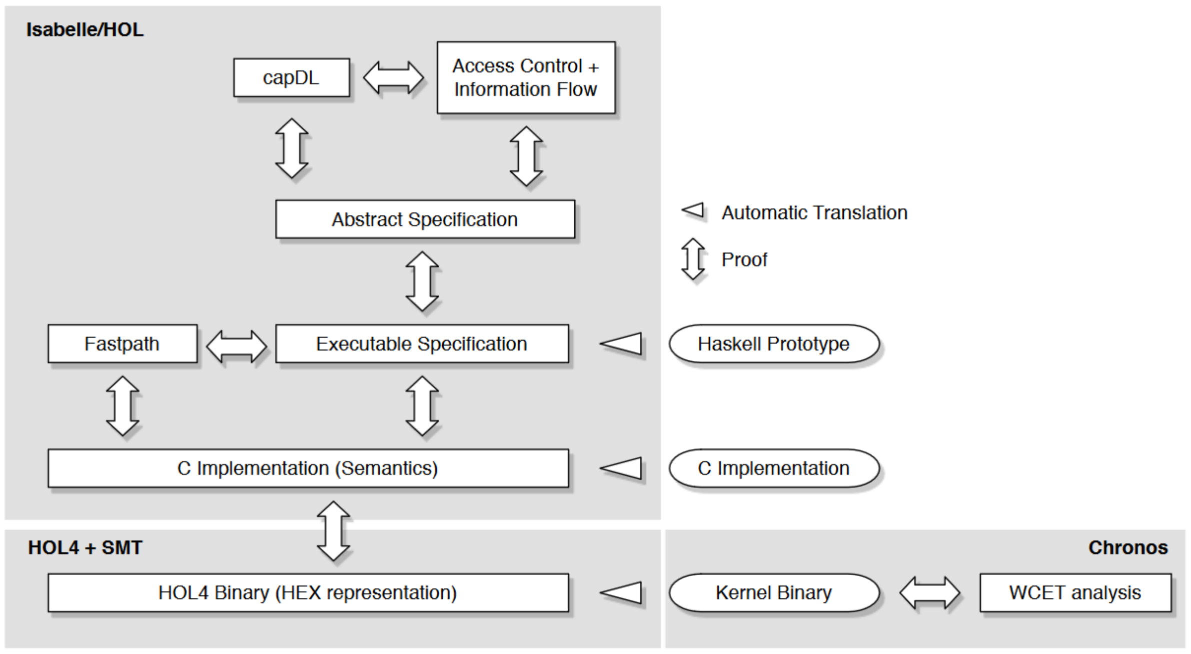
seL4 Proof Overview:

The seL4 proof pipeline is complicated. First the C source and Binary code are proven to correspond to each other 1 to 1. This is translation validation. It validates that the compiler maintains the expected semantics of the written C code. This pipeline is illustrated as such:



The C source code is fed into Isabelle via a programmatic translation. This is then proved to have certain security properties. A call graph is created and fed into a series of SMT solvers (symbolic execution). Parallel to this process, the binary is also used to generate a call graph. The SMT solvers are used on both the binary and the C source. If the outputs correspond then the semantics of the program were maintained during compilation.

This process glosses over the “proving” portion of the Isabelle implementation. It focuses on describing the verification of the compiler (that the binary and source match semantics). The proof process of the security properties are modeled at a high level here:



Isabelle is fed a C implementation and it is proved to match the specifications above it. The system continues to prove correspondence with more abstract representations until it reaches the CapDL and Access Control + Information Flow proofs. This is where the seL4 proof implementation proves the security properties expected of the system.

Note: In an interview, Klein stated that the Haskell Prototype was no longer used as part of the proof chain.

CapDL is a system which proves the initial state of the system. Users can use it to specify the init state of the system, and the proof shows that this state will be reached without additional side effects. Access Control + Information Flow is proof of the security properties of seL4.

Designing for Proof:

Several design decisions have been made in the C implementation of seL4 to ease the proof process. The ultimate goal was to reduce the amount of state each operation interacted with. With this in mind, the seL4 team has implemented a small global state footprint.

Many of seL4's API calls are split into a checking phase and an executing phase. The checking phase validates all the prerequisites required to run the actual API call. This means that the execution phase can assume things like correct access parameters. This makes proving each component of the call, the checker and the executor , much simpler. The proof of the checker can be directly fed into the proof of the executor as a way to validate the assumptions.

The kernel memory management does very little actual memory management. The allocation of memory (think ptmalloc2) is given to the userland programs. The userland programs must dynamically allocate data in the memory they own. This greatly simplifies the verification of the system, but also leaves unsuspecting users vulnerable as they have to perform the very complicated process of directly managing dynamically allocated memory. When memory must be reclaimed by the kernel, the kernel must invalidate all references to the particular memory region (page). Because all access rights are in a tree-like structure, the kernel can traverse the full tree to find all outstanding capability pointers to the revoked memory region.

seL4 is largely a uniprocessor system. The design decision is required because the scale of the proving process would grow exponentially over the current work. Asynchronous IO and interrupts still present concurrency problems for the proofs. Device drivers are moved to userland, which means that their behavior need not be proved.

Abstract Interpretation:

The Abstract interpretation is the highest level of the proof chain. While abstractly, this layer still models the full functionality of seL4. It does not reveal any implementation details. Rather, it models what the lower level implementation details should achieve. Further proofs are essentially proofs that the implementation achieves the goals set out by the abstract interpretation.

Executable Specification:

At this layer, the isabelle proof begins to define how actions should be carried out. High level Isabelle recursive types are now being replaced with pointer based variants of the types. Trees are no longer a recursive definition of Nodes and leaves; they are nodes with pointers attached.

C Implementation:

The C implementation layer aims to formalize the C semantics. This is aided by using a modified version of C with certain features banned from seL4. Namely, the “&” operator is banned. This has far reaching effects in kernel programming. Large local structures cannot be passed by reference. To get around this restriction, the seL4 team implemented structs that are referenced by multiple subroutines as global. This means that both the caller and the callee must reference the structure through a pointer, rather than the & operator. Other notable features banned to make the proofs easier include: Function calls through pointers, goto statements, and minimized side effects. This has a sizable effect on the way the kernel was written, and might make the C code look strange to someone familiar with Linux (or other open source operating systems).

Machine Model:

Some features, such as booting, setting up the C runtime environment, and handling certain CPU features requires assembly. Proofs of these machine level operations are not modeled or proved. Instead the seL4 team aimed to test these features for correctness. What is modeled is a machine state. This includes

Functional Correctness Refinement Proof:

The Refinement proofs work by modeling an interpretation of the C source in increasing levels of abstraction. The refinements are then all checked against a state machine:

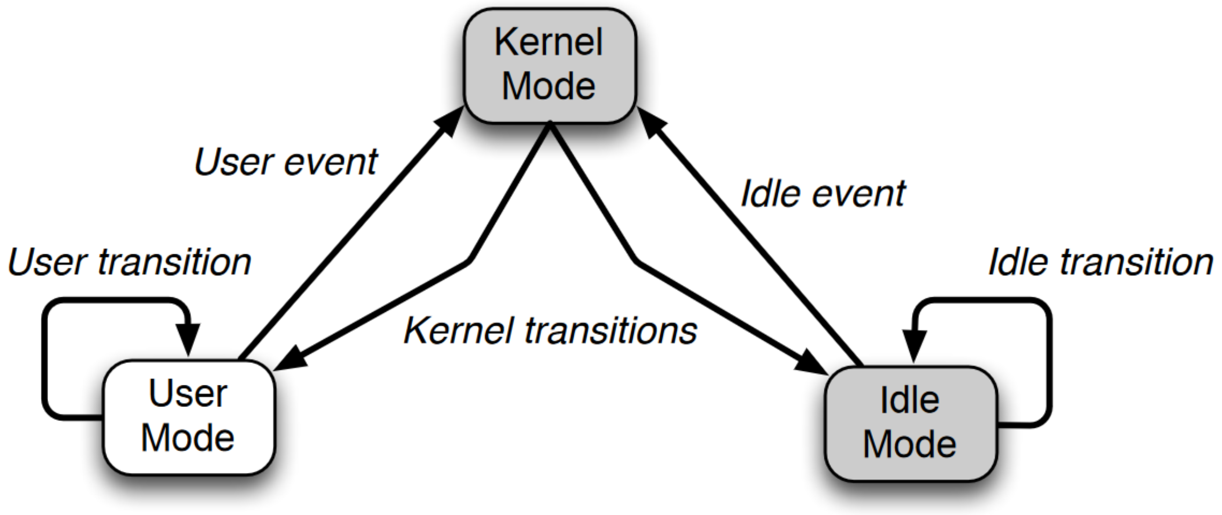


Fig. 10. Types of transitions.

Every event inside an seL4 system corresponds to some change in this state space diagram. Each model is checked against each other to ensure that the state of the state machine is correctly changed for each type of event shown. This shows the refinement property and thus proves the correspondence between the different layers of abstraction. An invariant, then, can be seen as a property of the system that is preserved under all possible state space changes in the above diagram.

Kernel transitions are complete actions that a kernel might take. In the case of a system call, a kernel transition would be from the time the kernel takes control from the userland program to the time the kernel returns control back to the userland program. This is a complete kernel action, from start to finish. *User transitions* are all state changes that take place within the confines of the virtual memory space allocated to that specific userland program. A *user event* is any event that causes the user program to yield control to the kernel. *Idle events* and *idle transitions* model the idle behavior of the system. This is important to be modeled because not every CPU cycle will have an action ready to take place. The idle portion of the state machine ensures that there is a valid state for “nothing happening”.

Correspondence:

I have mentioned correspondence several times in this document. This is correspondence in the English sense. Little would I know that correspondence would become a term defined inside of the seL4 proof documentation. From here on out, I will refer to correspondence in the sense of the seL4 documentation, namely:

Correspondence is the process of proving a code fragment by proving a direct relationship between its initial state and the abstract representation’s initial state followed by proving a direct relationship between the possible advances in state in both representations. This is the meat of the refinement proof inside of seL4.

Forward simulation is the name of the proof technique, and was used to prove the total state of the system. The seL4 proofs also apply this technique to code snippets and that is Correspondence.

Invariant Proofs:

Invariant proofs take up 80% of all the proofs in the seL4 proof code base. In seL4, an invariant proof is a proof that something stays the same. For example, when executing a system call the invariant proof of that system call is a proof that no part of the system was inadvertently changed after the call was executed. The only side effect is the expected side effect. This means that things like buffer overflows are ruled out by these proofs. If a system call is made, the invariant proof states that only the memory that was meant to be edited was actually edited.

Invariant proofs come in 4 categories inside seL4:

1. Low-level memory Invariants
2. Typing invariants
3. Data structure invariants
4. Algorithm invariants

Low-level memory invariants include proofs that memory is allocated correctly, that two processes do not accidentally get mapped the same frame, and that alignment is correct.

Typing invariants include proofs that types are well formed and do not become mangled. Pointers are proven to point to the object of their casted type and array access is inside the bounds of the array.

Data structure invariants are additional constraints to the structures beyond the well formed type proofs above. They include that doubly linked lists must be correctly back linked and lists are terminated with NULL. These invariants are usually violated during update operations. Imagine adding a node to the end of a doubly linked list. First the node must replace the NULL node. This violates the property that the list is ended by a NULL node. The algorithm will then add a NULL node after the new entry, but this means that the “invariant” was not truly invariant. The key idea is that in a Hoare triple of a precondition, statement, and post condition:

<P> S <Q>

The invariant is proven if the property holds in P and in Q. It can be violated in S so long as it is restored before Q.

Algorithm invariants state that each algorithm does not violate other invariants. As an example, during the creation of objects, objects must always have mutual references to one another. If one object has access via a pointer to another object, then the relationship must go both ways. An algorithm invariant would ensure that object creation, like minting new capabilities, maintains this property.

Access Control Model:

The access control model is an abstraction of the access control present in the real system. This abstraction exists to make the proving process easier than working with raw C semantics. A correspondence is present between the C semantics and the model, which makes all the proofs valid for the C semantics as well.

Fundamentally, the access control model is an enforcement of which threads can perform which actions. Recall that actions are dictated by having access to a capability pointer to a kernel object. The kernel objects export methods to the thread, and these methods correspond to what would traditionally be system calls. The access control model then captures the state of the system's ability to give out capability pointers to threads.

seL4 Proof Internals:

CapDL Proof Internals:

Before diving into the CapDL proofs, we need a primer on separation logic. I will summarize the core ideas, and the required judgements.

Separation logic is an extension of Hoare logic to encompass axioms, syntax, and judgements for heap, pointers, and memory.

```
V := [E] -- read the value at address E into V
[E] := E' -- write the value resulting from evaluating E' into address evaluated by E
V := cons(E1,...,En) -- create n new consecutive memory cells, storing values from E1,...,En
                     -- note this means V is a pointer l, but l+1, l+2, ..., l+n-1 are pointers too
dispose(E) -- evaluate E to pointer, and free memory associated. fails if pointer is not in heap
```

These additions allow Hoare triples to contextualize reading (V:=[E]) and writing ([E]:=E`) to memory. V:=cons(E1,...,En) represents a block of memory and dispose(E) deletes memory. This is going to be very useful for semantically quantifying memory management, which CapDL is responsible for. The judgments can look a bit confusing at first glance. Here is the assignment axiom:

```
 ------------------------------ [assign2]
⊢ {V = v} V := E {V = E[v/V]}
```

Underneath the bar is a Hoare triple. The first term, {V=v}, is the precondition. The middle term, V:=E, is the statement, and {V=E[v/V]} is the post condition. One should note that := refers to assignment, = refers to equality, and [v/V] means replace all instances of V with v. It is standard for Hoare statements to come in triples: a precondition, statement, and postcondition. This is commonly referred to as a Hoare Triple. The way you should read it is:

> If {V = v} is true and (V := E) is executed => {V = E[v/V]} is the result.

Those familiar with Hoare Logic will find this assignment axiom strange. The problem is that the [v\V] is usually in the precondition as seen here:

The Assignment Axiom

$$\vdash \{Q[E/V]\}\ V := E\ \{Q\}$$

The above is the standard assignment axiom for variables. The separation logic assignment axiom is for memory management. With that in mind, let’s break down the axiom.

Assume you have two pointers, V and v. These pointers point to the same memory location:

> V -> 0x20
> v -> 0x20

Now assume that V is being pointed to a new object E at location 0x100:

```
V -> 0x100
v -> 0x20
```

Now to understand the replacement part, [v/V] , assume that E is an object like this:

```
Struct example = {
        Vpointer -> V -> 0x20
}
```

It is just a struct that holds a pointer to V, but we are changing V. We expect this struct to point to 0x20, but after V changes it will look like this:

```
Struct example = {
        Vpointer -> V -> 0x100
}
```

To prevent this unwanted change, we replace V with v which still points to 0x20:

```
Struct example = {
        Vpointer -> v -> 0x20
}
```

This understanding will help greatly when examining the rest of the judgements:

Reading from the heap:

```
--------------------------------------------------------- [fetch]
⊢ {(V = v1) ∧ E → v2} V :=[E] {(V = v2) ∧ E[v1/V ] → v2}
```

Writing to the heap:

```
---------------------------- [heap-assign]
⊢ {E → _} [E]:=F {E → F}
```

Allocating memory:

```
----------------------- [alloc]
⊢ {V = v} V :=cons(E1, . . . , En) {V → E1[v/V ], . . . , En[v/V ]}
```

Freeing memory:

```
------------------------ [dispose]
⊢ {E → _} dispose(E) {emp}
```