**University of Utah**

May, 2014

**Capability Module for LCD**

**Flux Research Group**

**Muktesh Khole <muktesh.khole@utah.edu>**

Contents

[Introduction: 3](#_Toc386674332)

[Design: 3](#_Toc386674333)

[Data Structures: 4](#_Toc386674334)

[1. CSPACE 4](#_Toc386674335)

[2. Capability Table Entry 4](#_Toc386674336)

[3. Free Slot 4](#_Toc386674337)

[4. Capability Node 5](#_Toc386674338)

[5. Capability 5](#_Toc386674339)

[6. Capability Derivation Tree 6](#_Toc386674340)

[External Interface: 7](#_Toc386674341)

[1. Create Cspace: 7](#_Toc386674342)

[2. LCD Lookup Capability 7](#_Toc386674343)

[3. Create Capability 8](#_Toc386674344)

[4. Grant Capability 8](#_Toc386674345)

[5. Delete Capability 8](#_Toc386674346)

[6. Revoke Capability 9](#_Toc386674347)

[7. Destroy Cspace 9](#_Toc386674348)

[8. Get Capability Rights 9](#_Toc386674349)

[9. Mint Capability 10](#_Toc386674350)

[Locking Mechanism: 11](#_Toc386674351)

[1. Function: lcd\_cap\_create\_cspace 12](#_Toc386674352)

[2. Function: lcd\_cap\_lookup\_capability 12](#_Toc386674353)

[3. Function: lcd\_cap\_create\_capability 12](#_Toc386674354)

[4. Function: lcd\_cap\_grant\_capability 12](#_Toc386674355)

[5. Function: lcd\_cap\_delete\_capability 12](#_Toc386674356)

[6. Function: lcd\_cap\_revoke\_capability 13](#_Toc386674357)

[7. Function: lcd\_cap\_destroy\_cspace 13](#_Toc386674358)

[8. Function: lcd\_cap\_get\_rights 13](#_Toc386674359)

[9. Function: lcd\_cap\_mint\_capability 13](#_Toc386674360)

Low Level Design Document:   
Capability Module for LCD

# Introduction:

We intend to build a secured execution environment wherein untrusted code/applications will run in their own domain which will be known as LCD throughout this document. The main idea is to allow an LCD to have a minimum set of privileges required to perform its job. A precise and well-defined interface is defined for the LCD to exercise its privileges and also to request for additional privileges if required.  
The privileges are defined using capabilities. Capabilities allow flexible and fine grain access control over resources (memory, interrupt, ports etc.). Using capabilities allows us to create a secured environment wherein each individual LCD domain will have minimum privileges to perform its duties and the small and well-defined interface with the outside world (outside of domain) provides isolation.

# Design:

The design is borrowed from Sel4 and Barrlefish. The capabilities are stored in a data structure known as the cspace (capability space). For now the implementation provides each thread with its own cspace, going ahead this could be easily modified to make each LCD have a single cspace which will be shared by all the threads/processes executing within the LCD, as the space needed by this data structure is not trivial.

When the LCD is started it receives a few basic capabilities listed below (except for LCD\_CapInitThreadTCB others are not mandatory)

LCD\_CapInitThreadTCB - Capability to thread control block (must be provided).

LCD\_CapInitThreadPD - Capability to Page Directory

LCD\_CapIRQControl - Capability to get notified for interrupts

LCD\_CapInitThreadIPCBuffer - Capability for the IPC buffer

With these capabilities an LCD has only those privileges which are required to perform its duties and the interface defined by the IPC mechanism isolates LCD.

LCD cannot access any of the kernel resource directly. Each capability acts as a contract with the kernel, a method invoked on the capability allows LCD to access and use the resource managed by the capability. A capability identifier identifies each capability that the cspace has, each request to use or access the resource carries this capability identifier.

The kernel does a privilege/access rights check using this identifier and then a decision is made to whether allow the access.

Each capability entry that is made within the cspace radix tree also has a corresponding node within a capability derivation tree. Since, capabilities can be created, granted and copied we need to maintain the relations like parent-child and siblings. This is necessary because we support an operation called revoke on a capability (it actually acts on a capability derivation tree which corresponds to the capability, parent of the capability is not affected apart from its child pointer nothing else changes in parent) which will delete this capability from every cspace where this capability is mapped as a child or descendent of the capability that is being revoked.

# Data Structures:

1. CSPACE **struct cap\_space**

**{**

**struct cte root\_cnode;**

**struct semaphore sem\_cspace;**

**};**  
This defines the cspace which stores all the capabilities which a thread has in a radix tree like data structure. Each node in the radix tree is an array of struct cte. Each node can hold MAX\_SLOTS number of entries. Each entry could be a capability to a resource or a free slot or a capability node object which can point to another such array.

root\_cnode - this identifies the root of the cspace.

sem\_cspace - this is the semaphore which protects the cspace from concurrent access.

1. Capability Table Entry

**struct cte // capability table entry**

**{**

**union**

**{**

**struct cap\_node cnode;**

**struct capability\_internal cap;**

**struct free\_slot\_t slot;**

**};**

**lcd\_cap\_type ctetype;**

**uint16\_t index;**

**};**

This is union and represents the contents of the cspace radix tree node (or the entry within the array of struct cte). As each entry in our cspace radix tree node could be a free slot, a capability entry or another cnode which is a pointer to a child node in the radix tree this is defined to be a union.

cnode - Capability table entry is a cnode which will have a pointer to another node in the cspace radix tree.

cap - Capability table entry represents an actual capability.

slot - Capability table entry represents a free slot.

1. Free Slot

**struct free\_slot\_t**

**{**

**int next\_free\_cap\_slot;**

**int next\_free\_cnode\_slot;**

**struct cap\_space \*cspace;**

**};**

This represents that the entry within the cspace radix tree node is a free slot. The free slots are maintained on a linked list managed by index into the array of struct cte, the entries where capabilities can be added are placed on next\_free\_cap\_slot list and where a cap\_node can be added are placed on next\_free\_cnode\_slot.

next\_free\_cap\_slot - the next index within the array of struct cte which is free to hold a capability.

next\_free\_cnode\_slot - the next index within the array of struct cte which is free to hold a cnode.

cspace - A pointer to the cspace.

1. Capability Node

**struct cap\_node**

**{**

**cap\_id cnode\_id;**

**struct cte \*table;**

**uint16\_t table\_level;**

**};**

This is a capability node which can point to another cspace radix tree node, there are a few slots maintained at the end of each node (CNODE\_SLOTS\_PER\_CNODE) to store capability nodes their actual number decides the level/height of the radix tree.

cnode\_id - The identifier bits for the table, each slot index gets appended in front of this identifier to form  
 the identifier for the data structure stored at that slot.

table - Pointer to the array of struct cte.

table\_level - This determines the appropriate bit position at which the slot index bits will be appended to form the identifier.

The cnode contains a pointer to another node in the radix tree, which is the array of struct cte. It also has an identifier for the entire node and a level ID. The level ID determines the bit positions where the slot index bits are to be appended to form the final identifier for the entry.

1. Capability

**struct capability\_internal**

**{**

**void \*hobject; // a pointer to a kernel object**

**struct cap\_derivation\_tree \*cdt\_node; // list of domain ids to whom this capability is granted**

**lcd\_cap\_rights crights; // specifies the rights the domain has over this capability**

**};**

This is the actual representation of the capability within the cspace. An identifier identifies this capability uniquely.

Each capability has its own node on the capability derivation tree.

hObject - Pointer to the kernel object.

cdt\_node - pointer to the corresponding capability derivation tree node.

crights - The rights that the owner of this capability has over the capability.

1. Capability Derivation Tree

**struct cap\_derivation\_tree**

**{**

**struct cte \*cap;**

**struct semaphore \*sem\_cdt;**

**struct cap\_derivation\_tree \*next;**

**struct cap\_derivation\_tree \*prev;**

**struct cap\_derivation\_tree \*parent\_ptr;**

**struct cap\_derivation\_tree \*child\_ptr;**

**};**

Describes a node on the capability derivation tree.

cap - reverse mapping to the corresponding capability.

sem\_cdt - a semaphore which is common to the entire capability derivation tree. Locking this semaphore locks the entire tree.

next - pointer to the next sibling within the capability derivation tree.

prev - pointer to the previous sibling within the capability derivation tree.

parent\_ptr - pointer to the parent node within the capability derivation tree.

child\_ptr - pointer to the child node within the capability derivation tree.

# External Interface:

1. Create Cspace:

**struct cap\_space \* lcd\_cap\_create\_cspace(void \*objects[], lcd\_cap\_rights rights[]);**

Creates a cspace for a thread and inserts a pointer to the cspace into the TCB of the thread. Should be called during LCD/thread creation, expects the following objects as input with the rights for those objects:

**Input:**

objects: array of pointers where each pointer is as follows:

|  |  |  |
| --- | --- | --- |
| **Index within array** | **Object** | **Comments** |
| 0 | LCD\_CapNull | First entry should be NULL |
| 1 | LCD\_CapInitThreadTCB | Pointer to task\_struct of thread, \*cannot be NULL. |
| 2 | LCD\_CapInitThreadPD | Pointer to the page directory of process. |
| 3 | LCD\_CapIRQControl | will be used later |
| 4 | LCD\_CapInitThreadIPCBuffer | Pointer to the IPC buffer which will be used. |

Total number of object pointers expected in array is equal to LCD\_CapFirstFreeSlot.

Put a NULL pointer for entries which the thread is not expected to have.

e.g. if the thread does not need an IPC buffer then objects[4] should be NULL.

**Output:**

Pointer to the newly created cspace.

1. LCD Lookup Capability

struct cte \* lcd\_cap\_lookup\_capability(struct task\_struct \*tcb, cap\_id cid, bool keep\_locked);

Will be used to access the object on which a method is to be invoked. The returned value is a pointer to the entry in the table where the capability maps. This contains the handle to the object which can be used by the invoked method.

**Input:**

tcb: pointer to the thread control block of the thread which invokes method.

cid: capability identifier of the object on which the method is to be invoked.

keep\_locked: whether or not to keep the semaphore protecting cdt locked.

**Output:**

pointer to the capability table entry if the cid is valid else NULL is returned.

1. Create Capability

**cap\_id lcd\_cap\_create\_capability(struct task\_struct \*tcb, void \* hobject, lcd\_cap\_rights crights);**

Creates a new capability and inserts it into cspace of caller and returns the capability identifier.

Currently, it is unclear who will have the right to perform this operation.

**Input:**

ptcb: pointer to the task\_struct of the thread which intends to create the capability.

hobject: pointer to the underlying kernel object for which capability is being created.

crights: the rights associated with the capability for the object.

**Output:**

capability identifier within the cspace of the thread which intended to create this capability.

0 = Failure

1. Grant Capability

**cap\_id lcd\_cap\_grant\_capability(struct task\_struct \*stcb, cap\_id src\_cid, struct task\_struct \*dtcb, lcd\_cap\_rights crights);**

Will be used to grant a capability to another lcd/thread. The owner of the capability who is granting it should have a right to grant it to others. If it has this right then the capability is granted to the other LCD/thread and inserted into his cspace. Returns the address of the capability within the cspace of the receiver thread. The rights the receiver has can be controlled using the input parameter crights, a logical AND of the input parameter crights and the rights on the capability being granted will decide the final rights the granted capability will have in receiver cspace.

**Input:**

src\_tcb: pointer to the task\_struct of the thread which internds to grant the capability.

src\_cid: capability identifier of the capability being granted.

dst\_tcb: pointer to the task\_struct of the receiver thread.

crights: the rights on the capability to be granted to the receiver.

**Output:**

Returns the capability identifier within the cspace of the receiver.

1. Delete Capability

**uint32\_t lcd\_cap\_delete\_capability(struct task\_struct \*tcb, cap\_id cid);**

Will be called to delete a particular capability in the calling thread's cspace. Threads have right to delete capabilities in their own cspace. This modifies the capability derivation tree by deleting the corresponding CDT node within the tree.

**Input:**

ptcb: pointer to the task\_struct of the thread intending to delete a capability.

cid : capability identifier of the capability to be deleted.

**Output:**

0 = Success

Any other value indicates failure.

1. Revoke Capability

**uint32\_t lcd\_cap\_revoke\_capability(struct task\_struct \*tcb, cap\_id cid);**

Will be called to delete the capability and all its children from any cspace in which it is mapped. For the parent node only the child pointer will be affected. The children can be present in cspace belonging to different threads. As such the thread owning the parent capability has a right to delete a capability which is its child or was derived from the child.

**Input:**

ptcb: pointer to the task\_struct of the thread which is invoking this revoke operation.

cid : the capability identifier of the capability being revoked.

**Output:**

0 = Success

Any other value indicates failure.

1. Destroy Cspace

**void lcd\_cap\_destroy\_cspace(struct task\_struct \*tcb);**

Should be called when the thread exits, or whenever the cspace needs to destroyed. This is extremely heavy function which updates the CDT for all capabilities present in the cspace of the exiting thread.

**Input:**

ptcb: pointer to the task\_struct of the thread which is getting terminated.

1. Get Capability Rights

**uint32\_t lcd\_cap\_get\_rights(struct task\_struct \*tcb, cap\_id cid, lcd\_cap\_rights \*rights);**

Will be used to get the rights associated with the capability whose identifier is cid.

**Input:**

ptcb: pointer to the task\_struct of the thread in whose cspace the capability resides.

cid : capability identifier of the capability whose rights are being queried.

rights: The rights will be saved in this variable.

**Output:**

Return Value: 0 = Success, Any other value indicates failure.

The rights associated with the capability will be saved in the rights ouput paramter.

1. Mint Capability

**cap\_id lcd\_cap\_mint\_capability(struct task\_struct \*tcb, cap\_id cid, lcd\_cap\_rights rights);**

Will be used to craete a copy of the capability identified by cid within the same cspace, the rights of the copied capability could be modified using the rights parameter.

**Input:**

tcb: pointer to the thread control block whose cspace has the capability to be copied.

cid: the capability identifier of the capability to be copied.

rights: the new rights for the copied capability, the final rights the capability will be the Logical AND of the original rights and the parameter rights.

**Output:**

the capability identifier for the copied capability is returned.

0 indicates failure.

# Locking Mechanism:

Coming up with a locking mechanism which prevented deadlocks, race conditions and also does not hamper performance was extremely tricky given the nature of the data structures used. Main insights while designing the locking mechanism was that there are only two main data structures which need to be protected from concurrent updates, the Capability Derivation Tree and the CSpace Radix Tree. Instead of having a lock per entry within the cspace radix tree node, a lock was added per CDT. Locking the CDT node provided an exclusive access to the entry within the cspace radix tree node and also protected the CDT hierarchy. The design strives to allow concurrent access to the cspace if it is to different entries.

The cspace has a semaphore which must be acquired before making any changes to the cspace radix tree. The capability derivation tree also has a semaphore which is shared across the entire capability derivation tree and must be acquired before accessing the CDT nodes.

Different functions use different sequence for acquiring these two locks, which could result in deadlocks if not handled properly. It is important to read and understand this section before making any changes to the locking mechanism.

Since, delete, revoke, grant, mint and lookup will be working on a particular capability entry it is necessary to acquire exclusive access to that entry. Each capability entry has a corresponding CDT node, acquiring a lock on the CDT node locks down the entire CDT tree. This serves two purpose, we get exclusive access over the capability entry as we prohibit any other LCD/thread from performing a delete or revoke or destroy on any of the ancestors or descendents of the capability. Protecting the ancestors and descendents was important as it may also change the parent-child or sibling relation pointers of the current capability or may even delete the current capability when we are working on it.

So all updates to the cspace radix tree follow the sequence of locking the entry first and then the cspace.

Now, let's look at the function lcd\_cap\_destroy\_cspace, this function will try to destroy the entire cspace, thus the first thing that it needs to do is prevent others from making any changes to the cspace. So we first acquire the cspace lock. Later as we start deleting each entry from the cspace radix tree nodes we have to update the capability derivation tree, as some of its nodes will be deleted. So now we acquire exclusive access to the CDT node. Note that this sequence is exactly opposite to what delete, revoke, mint, etc. are using and could possibly lead to deadlock. To avoid deadlock a mechanism similar to dining philosophers problem is applied. Destroying the capability gets the least preference, so if some other operation like delete, revoke, grant etc. is in progress destroy cspace will release the lock on cspace and will try again. Note that functions such as revoke are not going to release the lock on CDT node till they complete, hence it is necessary that functions which follow a different sequence of acquiring locks give up their locks to avoid deadlocks. We have only one function which goes against the normal sequence of first acquiring the CDT lock and then the cspace lock and hence, if it fails to acquire even a single lock it has to release all the locks which it has acquired so far.

Note that changing this logic can cause deadlocks.

Following is a list of functions, the sequence in which they acquire locks and their lock release strategy:

1. Function: lcd\_cap\_create\_cspace

**Lock Acquire Sequence:**

1. sem\_cspace - Lock cspace semaphore

**Release Strategy:**

Lock not released until operation completes.

1. Function: lcd\_cap\_lookup\_capability

**Lock Acquire Sequence:**

1. sem\_cdt - Lock the CDT semaphore

**Release Strategy:**

Lock could be released if the keep\_locked input argument is not set to true. If this is set to true then the lock is not released even after function exit, it is the responsibility of the caller who sets this input argument to true to release the lock.

1. Function: lcd\_cap\_create\_capability

**Lock Acquire Sequence:**

1. sem\_cspace - Lock the cspace semaphore

**Release Strategy:**

Released after completing its operation.

1. Function: lcd\_cap\_grant\_capability

**Lock Acquire Sequence:**

1. sem\_cdt - Lock the CDT semaphore belonging to the source
2. sem\_cpace - Lock the Cspace semaphore belonging to the destination/receiver.

**Release Strategy:**

CDT semaphore for the source is locked first and then the cspace semaphore for the receiver. Since, these are two separate LCDs/threads we cannot hold on to any lock if the other is not acquired. Hence, we try to acquire these locks in the order given, but if we are not able to acquire both the locks then we do not hold on to any lock, we release any lock which we are holding and then try again to acquire both the locks.

1. Function: lcd\_cap\_delete\_capability

**Lock Acquire Sequence:**

1. sem\_cdt - Lock the CDT semaphore.
2. sem\_cpace - Lock the Cspace semaphore.

**Release Strategy:**

CDT semaphore is locked first and then the cspace semaphore. The CDT semaphore is never released till the function completes and it will wait on the cspace semaphore to be acquired. This is the normal wait logic.

1. Function: lcd\_cap\_revoke\_capability

**Lock Acquire Sequence:**

1. sem\_cdt - lock the cdt semaphore of the CDT node belonging to the capability being revoked.

**Release Strategy:**

The semaphore is locked till the entire CDT tree is updated. The capability and its descendents are deleted and only after that the semaphore is released. This does not need to lock the cspace as the capability we are working on cannot be modified by anyone till we own the CDT lock on the CDT node corresponding to the capability. Note that not holding the cspace lock allows concurrent access to other parts of the radix tree, it is only the concerned capability hierarchy which is locked.

1. Function: lcd\_cap\_destroy\_cspace

**Lock Acquire Sequence:**

1. sem\_cspace - lock the cspace being destroyed.
2. sem\_cdt - lock the cdt nodes of each of the corresponding entries in cspace radix tree.

**Release Strategy:**

First note that the sequence of acquiring locks is exactly opposite to that of normal wait sequence. This can cause a deadlock and therefore, this function is not allowed to hold any locks and wait. Either it acquired both the locks and performs its operations or it releases all the locks it holds and tries again.

1. Function: lcd\_cap\_get\_rights

Does not hold any locks.

1. Function: lcd\_cap\_mint\_capability

**Lock Acquire Sequence:**

1. sem\_cdt
2. sem\_cspace

**Release Strategy:**

This is the normal wait logic and so when we are done minting the capability we will release the cspace lock and then the cdt lock.