Using capability to create protection domains and compartments for processes and also for memory bounds checking

Wuyang Chung, wy-chung@outlook.com

Abstract

This paper will describe how to use capability to protect processes from each other on a single address space. It will also describe how to protect compartments within a process from each other also using capability. Those protections are coarse-grained on segment level so we also need a fine-grained protection on object level. Again capability can be used for this fine-grained protection. The capability-based memory bounds checking provides both the spatial and temporal violation check. It has a performance similar to fat-pointer but with low-fat pointer. Finally I will also describe how capability can simplify the design of IOMMU.

Introduction

Traditionally OS uses virtual memory to create protection domains for processes. Each process has its own private virtual address space so there are multiple virtual address spaces in the system. This is a good fit at the time when the system has only 32-bit address space. With private address space for each process, the system as a whole can have more than 4G address space. Nowadays with 64-bit machines, it is no longer necessary for each process to have its own private virtual address space in order to break the 4G limitation. All the processes can share a single address space thus the system will need only one page table. In order to do that, we need another way to create protection domains on a single address space. There are already several methods [1, 2, 4] proposed for creating protection domains on a single address space and capability-based addressing is one of them.

There are two ways to implement a capability-base addressing system [6], the tagged approach and the partitioned approach. In tagged approach, e.g. CHERI [5], each memory word has a tag bit associated with it. If the tag bit is 1, the corresponding memory word is a capability. If it is 0, the corresponding memory word is a regular data. If a program tries to write a regular data to a memory location that previously has a capability stored in it, its tag bit will be cleared by the hardware. It is how the tagged approach prevents the capability from being forged. In partitioned approach, e.g. Plessey 250 [3], a segment can store either regular data or capabilities but not both. It means that regular data and capability cannot be mixed together in one segment. If a program tries to write a regular data into a capability segment, it will trigger a hardware exception. It is how the partitioned approach prevents the capability from being forged. In this paper I am using the partitioned approach to design a capability-based addressing system. Each process has a capability table which stores the list of segments that the process can access. Thus the capability table actually creates a protection domain for a process on a single address space.

Nowadays a process can contain many modules that may come from different sources, so we also need a way to protect modules within a process from each other. The idea of dividing a process into multiple compartments is then arising. Compartment is very important for system security. If each module runs in its own compartment, they can be protected from each other. In my design a thread cannot access any segment in the process' capability table. A thread can access a segment only if it has the segment handle for that segment. A segment handle is similar to a file descriptor. Unlike a file descriptor, segment handle has a tag number field. This tag number can restrict a thread to access only subset of the segments in the process' capability table. This creates a sub-protection domain within a process and each sub-protection domain is a compartment in a process.

In addition to creating protection domains for processes and compartments, capability can also be used for memory bounds checking. For spatial violation check, each pointer is followed by a capability that specifies which object the pointer can access. The pointer can only access memory ranges within this object. For temporal violation check this approach uses capability revocation. When an object is freed, all the capabilities that point to this object will be revoked. And a pointer followed by a revoked capability becomes an invalid pointer.

In the following sections I will provide more detailed information for each of the topics.

Capability-based addressing

In a pure capability system, capability is used to control the access of many different system objects. Those objects can be process objects, thread objects, file objects, memory objects, etc. And in capability-based addressing, capability is only used to control the access of memory object, that is a segment. There are two ways to implement capability-based addressing system. This paper will use the partitioned approach. In partitioned approach, a segment can store either capabilities or regular data but not both. Usually in partitioned approach, each process has a root capability segment that stores capabilities that can point to either regular segment or capability segment. So the protection domain of a process is a tree structure. A process can access those regular segments of which a capability in this tree points to. In my design, a process has only one capability segment, that is the root capability segment. All the capabilities in the root capability segment will only point to regular segment. The root capability segment is also called the capability table of a process. Since the capability table stores a list of segments a process can access, the capability table actually forms a protection domain for a process on a single address space.

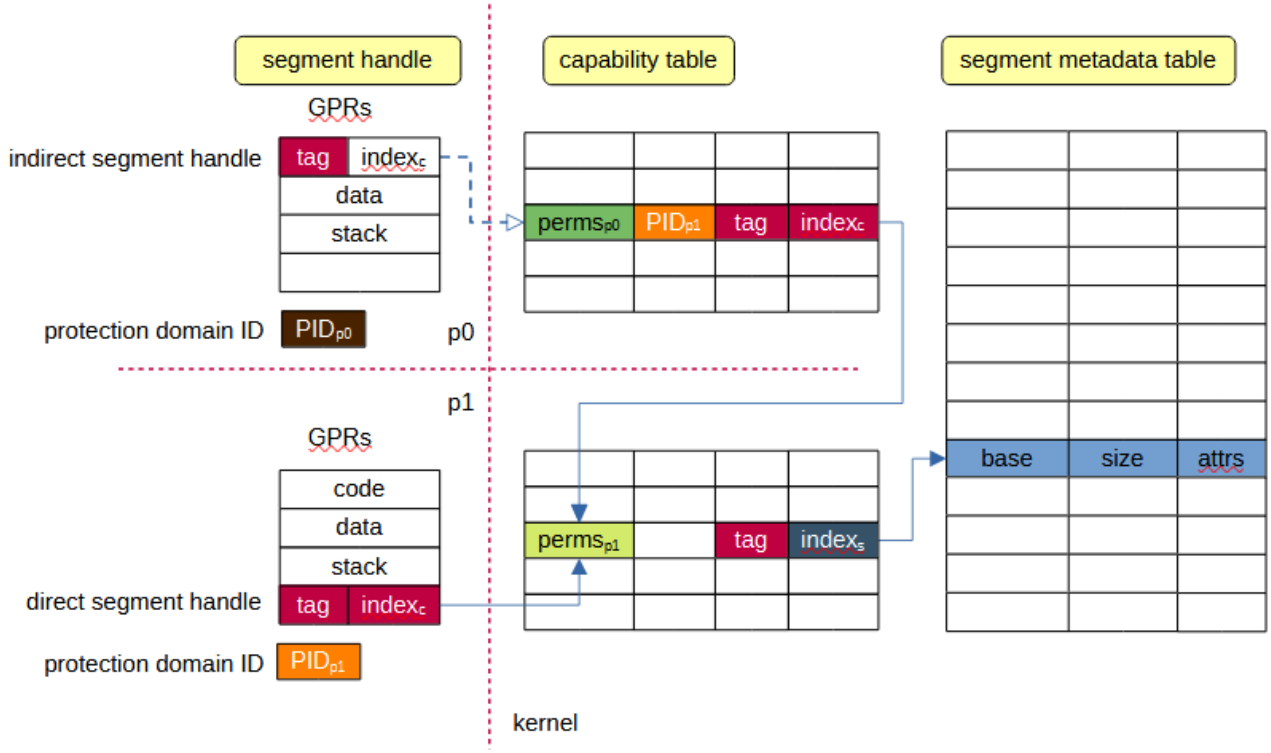

Figure 1: Segment metadata table, capability tables and segment handles

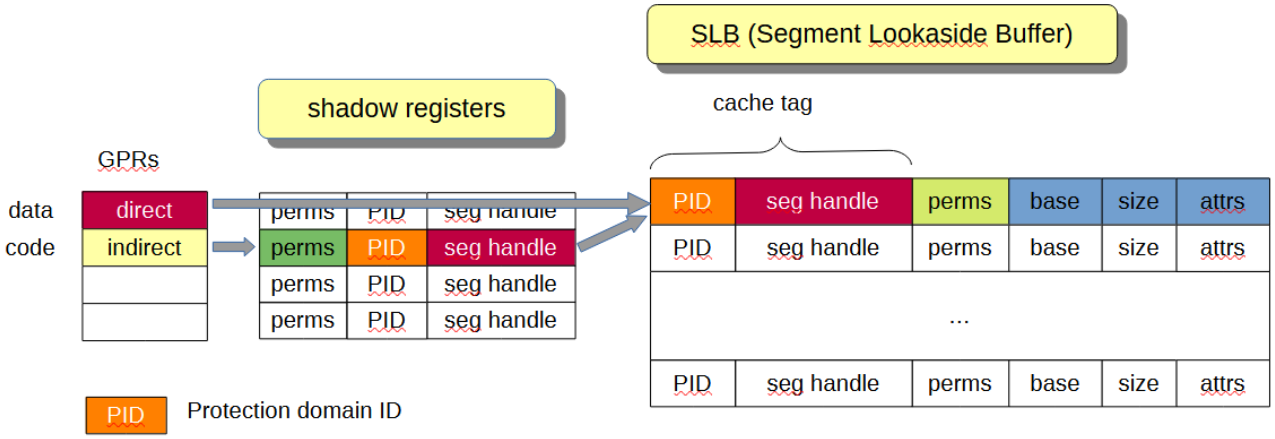

Figure 2: SLB (Segment translation Lookaside Buffer)

The hardware

Figure 1 shows the design of the hardware. The system has one segment metadata table. It stores the metadata of all the segments in the system. The metadata contains the base address and size of a segment. Each process has a capability table that stores a list of capabilities or direct segment handles that the process owns. Capability by definition contains a permission and a unique segment ID (for capability-based addressing). Since each segment in the system will have an entry in the system's segment metadata table, the index to this table can be seen as the unique segment ID. So capability actually contains a permission and an index to the segment metadata table. It also contains a tag number for segment handle verification. For security reason, both segment metadata table and process' capability tables are stored in the kernel and only kernel can modify it.

Segment handle is stored in segment register. It contains an index to the process' capability table and a tag number. There is also a tag number in the capability that the handle points to. Segment handle is valid only when the tag number in the handle and the corresponding capability are equal. There are two types of segment handle, direct segment handle and indirect segment handle. For direct segment handle, it points to a capability in the capability table. For indirect segment handle, it points to direct segment handle in the capability table. So the capability table actually contains either a capability or a direct segment handle. The system only supports one level of indirection, that is the indirect segment handle can only point to a direct segment handle. In figure 1 you can see that p1 has a direct segment handle that points to a capability and p0 has an indirect segment handle that points to p1's segment handle. A process that has a direct handle to a segment is the owner of that segment and the owner has more permissions on its segment such as freeing the segment.

On the hardware level, a program accesses memory by specifying a segment register (which contains a segment handle) and an offset within that segment to CPU. A segment handle is similar to a file descriptor. A program can access a segment only if it has the segment handle of that segment. If a program does not specify a segment register when accessing memory, the default segment register is used. Since segment register stores segment handle and segment handle is just a regular data, the system can reserve some general purpose registers as segment register. When a thread runs, it will access at least 5 segments (code, data, stack, TLS and heap). We also need two segment registers (fs1 and fs2) for accessing none-default segments. So at least 7 segment registers are needed for a running thread.

In order to speed up the translation from segment handle to segment metadata and segment permissions, CPU uses a segment translation lookaside buffer (SLB). Figure 2 shows that for direct segment handle CPU uses the protection domain ID of the process and the segment handle to check if a segment translation is cached in SLB. For indirect segment handle, CPU uses the cached direct segment handle in the shadow register and check to see if its translation is cached in SLB.

Compartmentalization

Now I will describe how the system divides a process into multiple compartments. Both the segment handle and the capability it points to have a tag number. A segment handle is valid only when its tag number is equal to the corresponding tag number in the capability. So a program can only access a segment if it knows the valid segment handle for that segment. It is very hard for a program to guess the tag number in a capability so it is very hard for a program to forge a segment handle. With tag number, the system can divide a process into multiple compartments and each compartment can only access a subset of segments in its process' capability table. When a program is loaded into the system, the kernel will first load the *user monitor* module to the process and it runs in its own compartment. User monitor has higher privilege than normal program but lower privilege than kernel. It is part of the trusted computing base (TCB). User monitor has the privilege to load default segment registers and it knows all the segment handles for all the compartments in a process. Each module runs in its own compartment. All compartment switch must go through user monitor and user monitor will limit the compartment to access only those segments that compose the module by loading segment registers with valid segment handles for that module.

	privilege	usage scenario
p = 0, user mode	normal user privilege	user program
p = 0, kernel mode	normal user privilege	device driver
p = 1, user mode	super user privilege	user monitor
p = 1, kernel mode	kernel privilege	kernel

Table 1: There are 3 privileges in the system.

Usually a system has kernel privilege and user privilege but in this design it has a new privilege called super user privilege. Table 1 shows that user monitor runs in this

privilege. This new privilege is achieved by having a p (privileged) bit in capability's permission field. When a program runs in user mode and the p bit is 1, the program will run in super user privilege. On the other hand when a program runs in kernel mode and the p bit is 0, it will run in normal user privilege. This is very useful for a module that runs in the kernel and we want to protect kernel from this module. By setting the p bit of the compartment where this module runs to 0, it will render its privilege to normal user privilege.

New CPU instructions

To access none default segments, we need new instructions. The far load and far store instructions are used to access none default data segment, also called far segment. In this instruction one of the source registers will store the segment handle of the far segment. The CPU will also need to support far call and far return instructions to call into and return from a function in a far code segment.

IOMMU

Capability can also make the design of IOMMU simple if the system supports physical segment, that is the logical address (segment handle and offset within this segment) is translated directly to physical address. Since it is a physical segment, no virtual to physical address translation is needed. Figure 3 shows that IOMMU will also have a SLB. It caches the recently used segment translation for hardware devices. When a device driver wants its device to do DMA, it first needs to allocate a physical segment from the OS. OS will then return a segment handle to the device driver. Device driver will then send this segment handle to the hardware device. Later when the hardware device wants to do DMA, it will send its device id and the segment handle to IOMMU. IOMMU will then check to see if it is cached in its SLB. If not, the system will use the device ID to get the device driver's capability table and use the segment handle to get the permissions and physical base address and size of the DMA buffer. If the segment handle is invalid, that is the tag number in the segment handle is not equal to the tag number in the corresponding capability, the hardware device is denied for the DMA operation. So the hardware device can access system memory only if it has a valid segment handle from the device driver.

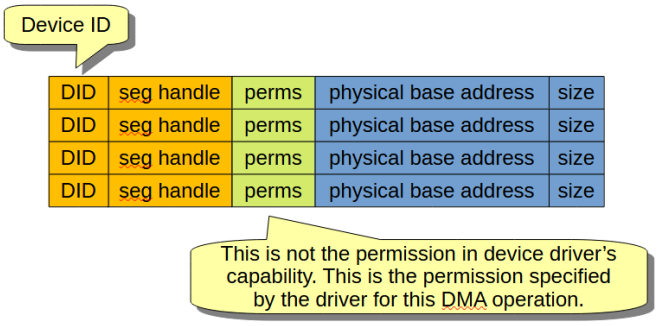

Figure 3: IOMMU's SLB (Segment translation Lookaside Buffer)

Memory bounds checking

In addition to creating protection domains for processes and compartments, capability can also be used for memory bounds checking. Memory bounds checking is used to prevent spatial security violation in a program. In order to do bounds checking, the compiler will need to store the metadata of all the objects in the program. The metadata contains primarily the start and end address of the object. Generally speaking, there are two ways to do memory bounds checking, the object-base approach and the pointer-based approach. In object-based approach, the metadata is associated with the object. And in pointer-based approach, the metadata is associated with the pointer. Usually in pointer-based approach, the metadata is stored alongside with the pointer. Since the pointer gets bigger, the pointer is also called fat pointer. Both approaches have its advantages and disadvantages. The biggest advantage of object-based approach is compatibility since the pointer format is not changed. But the performance is bad in object-based approach. This is the opposite for the pointer-based approach. The pointer-based approach is not compatible with old programs but its performance is much better.

In this paper I propose a third way to do memory bounds checking, that is using capability. Figure 4 shows that each pointer is followed by a capability. The capability specifies which object the pointer can access. It contains an index to the object metadata table. Each object metadata in the table contains the start and end address of the object. For each pointer memory access, the compiler will check if the pointer is within the range of the object to prevent buffer overflow or underflow. The capability and object metadata also contain a tag number. The capability is valid only when the tag number in the capability is equal to the tag number in the corresponding object metadata. With the tag number it is easy to revoke all the capabilities to an object by simply changing the tag number in the corresponding object metadata. When a memory buffer is freed, the library will

change the tag number in the corresponding object metadata. This will revoke all the capabilities that point to this object. When the capability of a pointer is revoked, it becomes invalid pointer. So the capability revocation can be used to detect the temporal security violations such as use-after-free and free-after-free bugs.

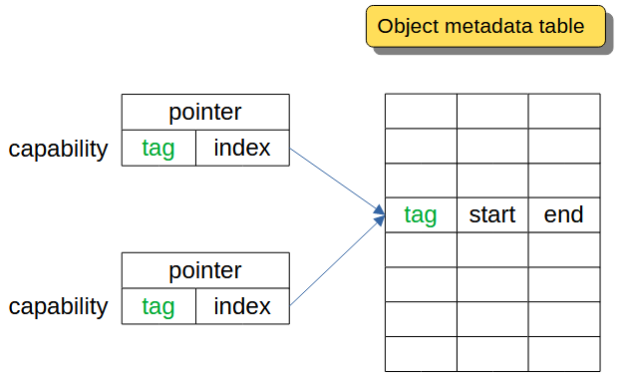

Figure 4: Memory bounds checking using capability. Each pointer is followed by a capability.

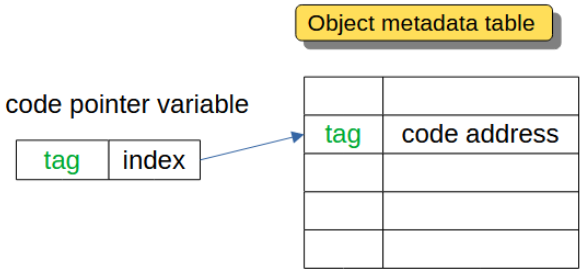

Figure 5: Code pointer verification using capability.

Capability can also be used for code pointer verification. Figure 5 shows that when a program wants to store a code address to a pointer variable, compiler can instead store a capability to this code pointer variable. Here is the steps that the compiler does. Compiler will allocate an entry from object metadata table and store the code address and a tag number here. The index to the object metadata table combined with the tag number is a capability to this code address. This capability is then stored to the code pointer variable. During code pointer dereference time, the compiler will use the capability stored in the variable to get the code address. It uses the index in the capability to read the object metadata and compare the tag numbers in the capability and in object metadata. If the tags are equal, it can get the code address from the object metadata. If the capability has been tampered, the tag numbers will be different and the capability is invalid.

Related work

There is another way to do capability-based addressing, that is the tagged approach. CHERI [5] is using this approach. In CHERI capability is used to replace pointer. It is protected by an out of band tag bit and it contains not only a pointer but also an object memory bounds and permissions. The advantage of CHERI is that capability creation is very fast since it is supported by the CPU instructions. But this is also its disadvantage. As capabilities can be duplicated everywhere in the memory, this makes it difficult to do capability revocation. The second disadvantage is with memory copy related operations. The memory copy operation will need to differentiate between data copy and capability copy which makes it slower. In this paper I am using the partitioned approach to design the system. The first advantage of this approach is that segment can be moved without affecting any pointers within it. On systems that support virtual memory, the segment move operation is very fast. Only the virtual to physical mappings are moved. Second, the segment metadata are only stored in system's segment metadata table. There is no duplication of the metadata and that makes it easy to do segment handle and capability revocation.

References

[1] Eric J. Koldinger, Jeffrey S. Chase, Susan J. Eggers. Architectural support for single address space operating systems. In ASPLOS-V, 1992.

[2] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, Jochen Liedtke. The mungi single-address-space operating system. Software: Practice and Experience, 1998.

[3] Henry M. Levy. Capability-based Computer Systems. Digital Press, 1983.

[4] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, Edward D. Lazowska. Sharing and protection in a single-address-space operating system. ACM Transactions on Computer Systems, 1994.

[5] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), June 2014.

[6] R.S. Fabry, Capability-based addressing, Communications of the ACM, 1974.

[7] Wuyang Chung, Using segmentation to build a capability-based single address space operating system, BSDCan 2023. paper.pdf, slides.pdf.