FlexAlloc: Dynamic Memory Partitioning between the Corevisor and Hostvisor in SeKVM

Matthew Nelson
Columbia University

Ryan Wee
Columbia University

Zeheng Yang
Columbia University

Abstract

SeKVM is a hypervisor for ARM processors that protects the memory of guest virtual machines (VMs) from an untrusted host machine. It comprises a corevisor that runs at EL2, and a hostvisor that runs at EL1 to leverage the functionality of the untrusted host kernel. However, SeKVM statically partitions memory between the corevisor and hostvisor at compile time. This leads to memory wastage when the number of VMs is low, and an inability to support the desired number of VMs when the number of VMs is high. In this paper, we present FlexAlloc, an extension of SeKVM that allows dynamic runtime partitioning of memory between the corevisor and the hostvisor. In particular, the memory needed for the stage-2 page tables of a guest VM is only allocated to the corevisor when a VM is created. This memory is then returned to the hostvisor when a VM is destroyed. Existing implementions of *alloc_pages* do not support allocating a contiguous region of memory equal in size to the memory needed for a VM's stage-2 page tables. Hence, FlexAlloc also modifies the iterators used to accesss a VM's stage-2 page tables, so that the memory used for a VM's stage-2 page tables can be composed of noncontiguous regions. Our work ensures that a machine running SeKVM uses memory more efficiently, and also allows such machines to support a larger number of guest VMs. This allows public cloud providers and organizations running their own VMs to scale up their operations at a lower cost.

1 Introduction and Background

Virtual machines (VMs) have become increasingly popular over the past few decades. In particular, they improve utilization of existing hardware by allowing users to run multiple unmodified operating systems on a single machine. A key concern when it comes to VMs is isolation. A single machine may be used to run multiple VMs created by different users who generally want their data to be protected from one another. As a result, hypervisors generally maintain some sort of secure boundary between guest VMs. However, another important aspect of isolation is protecting guest VMs from the host machine. A hypervisor that fully trusts its host machine is vulnerable to malicious actors who gain control of the host operating system itself. This is dangerous because the host machine could be used to steal confidential corporate data or sensitive personal information.

SeKVM is a formally verified hypervisor for ARM that isolates KVM's trusted computing base into a small core [6]. It builds on the previous work of HypSec, which is a hypervisor that protects VMs from the host machine [5]. HypSec consists of two layers. The first layer is an untrusted *hostvisor*, which operates at EL1 together with the host operating system. The hostvisor leverages many of the data structures and functions built into the host operating system for VM management. The second layer is a trusted *corevisor*, which operates at the more privileged EL2 level. This corevisor is responsible for isolating VMs from each other and from the host machine. SeKVM builds on HypSec's design, with the main difference being that its trusted computing base is formally verified. In SeKVM, the two layers are termed *KServ* and *KCore* respectively. However, in this paper, we use the terms hostvisor and corevisor to remain consistent with the underlying HypSec kernel source code.

SeKVM protects guest VMs in three main areas: CPU, memory, and I/O. In the area of memory, SeKVM leverages ARM hardware support for virtualization. ARM allows hypervisors to control memory access using stage-2 translation [3]. In particular, the guest operating systems and host operating system each think they are using page tables to map a virtual address space to the physical address space. However, in reality, they only control the mapping of virtual addresses to intermediate physical addresses. When translating virtual addresses, the MMU starts by using these page tables to translate virtual addresses into intermediate physical addresses. However, it then uses a second set of page tables to translate intermediate phyiscal addresses into the actual machine physical addresses. This second set of page tables, called stage-2 page tables, are controlled by the hypervisor running at EL2.

In the context of SeKVM, the stage-2 page tables are managed by the corevisor. When the host operating system tries to access some physical memory, the corevisor first checks that this physical memory does not belong to a guest VM.

The hostvisor cannot be allowed to access the memory used to store the stage-2 page tables of guest VMs. Otherwise, it could simply modify these stage-2 page tables so that guest VMs write their data to memory that can be accessed and manipulated by the hostvisor. In other words, the memory used for these stage-2 page tables must 'belong' to the corevisor. In SeKVM, the corevisor is statically allocated this memory on bootup. This is done by a linker script, which allocates some contiguous region of memory for the stage-2 page tables and demarcates it using the labels *stage2_pgs_start* and *stage2_pgs_end* [12]. The size of this memory region is determined by the macro *STAGE2_PAGES_SIZE* [11]. In particular, this memory region is large enough to store the stage-2 page table for the corevisor, the stage-2 page table for the hostvisor, and the stage-2 page tables for sixteen guest VMs. The corevisor and hostvisor are each allocated sixteen 2M pages for their stage-2 page tables, while each guest VM is allocated four 2M pages. Altogether, the corevisor is given 96 2M pages for all of these stage-2 page tables. During bootup, the hostvisor allocates the beginning of this memory region to the corevisor and itself. It then initializes the corevisor's VM-management data structure such that the remainder of this stage-2 page table memory region is evenly divided between each of the sixteen possible guest VMs. In particular, the hostvisor sets *el2_data->vm_info[i].page_pool_start = pool_start + (STAGE2_VM_POOL_SIZE * (i - 1))* for each possible VMID [8]. Here, *vm_info[i].page_pool_start* marks the start of the stage-2 page table for the *i*th VM.

The key observation here is that this partitioning of memory is done statically at compile time. In other words, the hostvisor cannot reclaim the memory it has allocated to the corevisor. Similarly, the corevisor cannot ask for more memory from the hostvisor. This is in contrast to the dynamic partitioning of memory between the hostvisor and guest VMs. Guest VMs can request memory from the hostvisor via a page fault, and the hostvisor can reclaim memory from guest VMs using ballooning [5]. The static partitioning of memory between the hostvisor and the corevisor is undesirable for two reasons:

- First, unused memory cannot be reclaimed from the corevisor. Say the number of VMs being run is n, such that $n < 16$. Then there will be $(16-n)*4$ 2M pages reserved for the stage-2 page tables of guest VMs that do not exist. Dynamic partitioning of memory would allow this memory to be reclaimed by the hostvisor, to alleviate memory pressure in the host. This memory could also be given to guest VMs, to alleviate memory pressure in these VMs. Giving VMs more memory would improve VM performance, because less data would need to be moved to the swap partition.
- Second, there is a hard upper limit on the number of VMs that can be supported by SeKVM. Users cannot add VMs above the limit of sixteen, because there will be no memory in the corevisor to support the stage-2 page tables of these VMs. Users could always choose to statically allocate even more memory to the corevisor upon bootup, but this would exacerbate the problem of unused memory mentioned above.

Static allocation of memory to the corevisor is not limited to memory for the stage-2 page tables of each VM. In addition to this, SeKVM also statically allocates memory to the corevisor to store the metadata of each VM. The same linker script mentioned above reserves another contiguous region of memory equal in size to 32 2M pages, and demarcates it using *el2_data_start* and *el2_data_end* [12]. This memory is given to the corevisor on bootup. Part of this memory is used to store an array of *struct el2_vm_info* objects, where each object is used to store the metadata of each guest VM. SeKVM could potentially also benefit from dynamic allocation of the memory used to store this VM metadata, where this memory is only given to the corevisor upon VM creation and returned to the hostvisor upon VM deletion. However, we choose not to focus on this because the size of each *struct el2_vm_info* object is 768 bytes. The total size of all of the *struct el2_vm_info* objects is 12288 bytes, which is less than four 4K pages in total. As a result, the problem of unused memory is not as significant, since we waste at most four 4K pages of memory. In any case, memory can only be allocated at the page granularity. Hence, in the case where we want to create a new VM and the new *struct el2_vm_info* object cannot be placed into the pages that have already been dynamically allocated to the corevisor, we will inevitably end up having to allocate an entirely new 4K page. The majority of this new 4K page would then remain unused, until another VM is created. This is not the case with VM stage-2 page tables, since the size of the stage-2 page table for each VM is nicely page-aligned. With the *struct el2_vm_info* objects, the problem of a hard upper limit is also not as significant, since we could easily make the array larger without incurring much memory overhead.

Although the *struct el2_vm_info* objects have a much smaller memory footprint than the stage-2 page tables, we designed and partially implemented a fully-generalized dynamic allocation interface that could be used to support dynamic allocation for all metadata structs. We call this the Dynamic Page Manager (DPM). DPM handles all of the work required to receive memory from and return memory to the hostvisor, allowing other "services" in EL2 to exchange memory with DPM using a very similiar interface to the *alloc_pages* and *free_pages* interface available in EL1. We describe this interface in detail in Section 4. However, the next section focuses exclusively on the changes made to support dynamic allocation of the memory needed for VM stage-2 page tables.

2 Design and Implementation

All of the source code described below can be found at `https://github.com/nelsonm2991/host-sekvm-project`. Our implementation of FlexAlloc is based on SeKVM, which is in turn based on Linux v5.4.55.

2.1 Modifying the page table iterators

As described above, the original implementation of SeKVM statically allocates sixteen contiguous 8M regions to the corevisor (Figure 1). When a VM is created, the corevisor uses one of these contiguous 8M regions for the VM's stage-2 page table. To implement a system of dynamic allocation, we want the hostvisor to dynamically reserve an 8M region at runtime when a VM is created, and then pass this memory region to the corevisor. The kernel macro *alloc_pages* is responsible for allocating a large number of contiguous physical pages [4]. Examining the implementation of this macro reveals that it eventually calls *__alloc_pages_nodemask*, which checks that the order requested is strictly less than *MAX_ORDER* [2]. *MAX_ORDER* is in turn defined to be 11 [1]. This means that we cannot allocate a contiguous region of memory larger than $2^{10} * 4K = 4M$. This limit is set by the kernel because *alloc_pages* uses the buddy allocator, which recursively splits available memory into half to find the best possible position for a contiguous region of memory [4]. However, this also makes it undesirable to continuously allocate huge contiguous regions of memory.

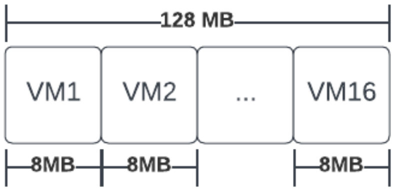

Figure 1: SeKVM: Layout of Statically-Allocated Region for VM Stage-2 Page Tables

To get around this limitation, there are two possible approaches. The first approach is to halve the size of the VM's stage-2 page table, so that it can fit within a single contiguous 4M region of memory. Alternatively, we can fragment the VM's stage-2 page table such that it can be composed of multiple non-contiguous regions of memory, whose sizes in total add up to 8M. We choose to implement the latter approach to avoid degrading VM performance, and to avoid breaking the current implementation of the buddy allocator. In particular, we fragment the VM's stage-2 page table so that it can be composed of eight distinct 1M regions of memory. Each of these 1M regions must be contiguous, but each region is distinct and need not be allocated contiguously with the other 1M regions. As a result, the hostvisor can easily use *alloc_pages* to allocate the memory needed for a guest VM's stage-2 page tables.

We implement this fragmentation by manipulating the iterators that provide the next useable page for entries at a given level of the page table. In the original SeKVM implementation, the pages containing entries from some level of the stage-2 page table are kept contiguous to one another, and are never intermixed with pages containing entries from other levels. Hence, there are distinct iterators for each level. Each iterator starts at a particular offset into the VM's statically-allocated 8M stage-2 page table region, and increments each time we need a new page for a new entry at that level. In particular, the PGD iterator operates over a *16 * PAGE_SIZE* pool, the PUD iterator operates over a *SZ_2M - (16 * PAGE_SIZE)* pool, and the PMD iterator operates over a *3 * SZ_2M* pool. Each of these pools must be contiguous, and the three pools must be contiguous relative to one another.

The naming semantics of the iterators at each level can be confusing, so we feel they are worth clarifying here. The iterator for level n allocates pages for entries in level n to point to. Hence, the iterator for level n points to the pool used to allocate entries belonging to level $n-1$. For instance, the PGD iterator allocates pages for PGD entries to point to. In other words, the PGD iterator actually allocates pages used to hold PUD entries. Hence, the PGD iterator is initialized to point to the start of the PUD pool.

Our design eliminates the need for all of these pools to be contiguous to one another, and enables fragmentation within each pool itself. To achieve this goal, we experiment with two distinct implementations. As mentioned above, the SeKVM implementation structures iterators such that a page with PUD entries will never be sandwiched between a page with PMD entries. In our first implementation, we ignore this semantic. In particular, we create a single, generalized iterator shared across all levels of the stage-2 page table. This iterator traverses all eight 1M regions. When it exhausts the space in a 1M region, it moves to the start of the next 1M region. However, we eventually decided to stop development of this version before it was fully functional, in favor of version two which more closely resembles recent changes made to SeKVM's stage-2 page table allocation scheme.

In our second implementation, we enforce the semantic that pages at each specific level are allocated from seperate pools, and thus are never intermixed with one another. To accomplish this while still maintaining the same pool of non-contiguous 1MB regions, we split the single, generalized iterator into nine distinct iterators. One PGD iterator is used to allocate pages from the PUD pool, two PUD iterators are used to allocate pages from the PMD pool, and six PMD iterators are used to allocate pages from the PTE pool. There are two key things to note here. First, we retain the existing scheme partitioning the stage-2 page table into three distinct pools: a PUD pool, a PMD pool, and a PTE pool. For instance, the PUD pool is still sized at *16 * PAGE_SIZE*. The second key thing to note is that our iterators never cross the boundary between two distinct 1M regions. Instead, if a pool needs to be split across two or more 1M regions, there is one distinct iterator for each 1M region covered by the pool. As a result, each of these iterators covers a contiguous subregion of the pool. Allocations from that pool begin by using the first iterator. When the first subregion of that pool is exhausted, allocations move on to use the next iterator. Essentially, when a page is allocated from a given pool, the correct iterator is chosen based on the pool in question and whether or not the preceding iterators for that pool have been exhausted.

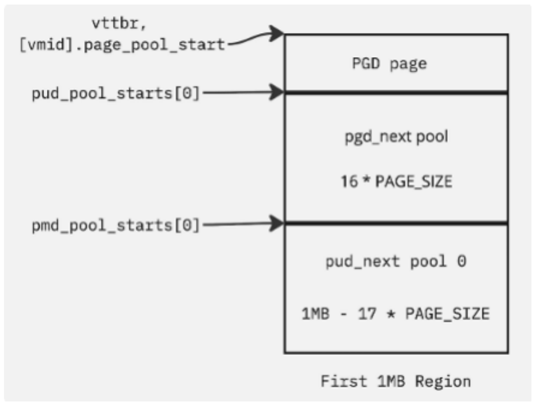

Figure 2: FlexAlloc: Metadata Setup for First 1MB Region

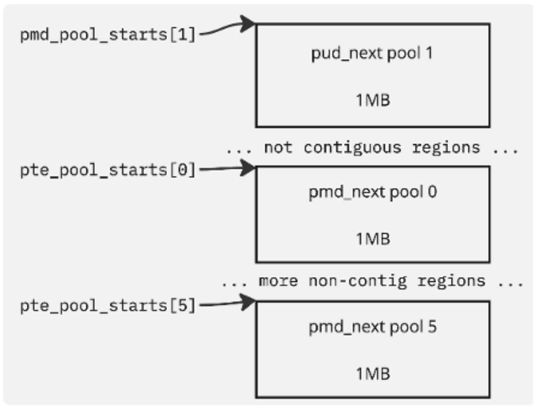

Figure 3: FlexAlloc: Metadata Setup for Second to Eighth 1MB Regions

In particular, the PUD pool is in the middle of the first 1M region, and there is a single PGD iterator pointing to the start of that pool (Figure 2). The PMD pool consumes the end of the first 1M region and the entire second 1M region. Hence, the first PUD iterator points to an address in the first 1M region, and the second PUD Iterator points to the start of the second 1M region (Figures 2, 3). Finally, the PTE pool consumes the third to eighth 1M regions. As a result, there is a distinct PMD iterator for each of these six regions (Figure 3). The code below illustrates how allocations from a given pool are made using the first possible iterator whose pool subregion has not been exhausted:

```
static u64 inline get_pmd_next(u32 vmid) {
    ...
    // pool_index: available iterator
    // we are using
    pool_index =
      el2_data->vm_info[vmid].pte_pool_index;
    // pool_start: beginning of the region
    // the pool_index corresponds to
    pool_start =
      el2_data->vm_info[vmid].
      pte_pool_starts[pool_index];
    // used_pages: number of pages in this
    // region we have exhausted
    used_pages = el2_data->vm_info[vmid]
      .pte_used_pages_vm[pool_index];

    // If we've used up all the pages and we
    // have another region, then use the new
    // region.
    if (used_pages == PTE_ITER_MAX_PAGES &&
        pool_index + 1 <
        PTE_USED_ITER_COUNT) {
        pool_index += 1;
        el2_data->vm_info[vmid].
          pte_pool_index = pool_index;

        // Set the pool_start to the next
        // available PTE iterator.
        pool_start = el2_data->vm_info[vmid].
          pte_pool_starts[pool_index];
        used_pages = el2_data->vm_info[vmid].
          pte_used_pages_vm[pool_index];
    }

    // Make sure we have not exhausted all
    // pages across all regions.
    if (used_pages == PTE_ITER_MAX_PAGES &&
            pool_index ==
            PTE_USED_ITER_COUNT - 1) {
        __hyp_panic();
    }

    // Return the physical address of the
    // next page that this iterator can
    // allocate.
    return pool_start +
        (used_pages * PAGE_SIZE);
}
```

2.2 Allocating memory to the corevisor

FlexAlloc dynamically allocates eight 1M memory regions to the corevisor when a VM is created. It does so by passing the base addresses of these eight 1M regions as arguments to the *HVC_REGISTER_KVM* hypercall. An alternative implementation that we considered was creating a new *HVC_PASS_MEM* hypercall to pass these 1M regions to the corevisor, and getting the hostvisor to call *HVC_PASS_MEM* before calling *HVC_REGISTER_KVM*. However, we ultimately choose to discard this alternative design because it leads to a more complicated implementation. In particular, the corevisor would have to verify a one-to-one correspondence between calls to *HVC_PASS_MEM* and *HVC_REGISTER_KVM*. It would also have to handle malicious hostvisors that try to call *HVC_REGISTER_KVM* without calling *HVC_PASS_MEM*. Morevoer, adding an extra hypercall increases latency because it leads to more frequent world-switches between the hostvisor running in EL1 and the corevisor running in EL2.

The *HVC_REGISTER_KVM* hypercall eventually calls *register_kvm* [9]. Note that at this point we are in the corevisor, running at EL2. Within this function, we preprocess the eight 1M regions to ensure that the calls to *alloc_pages* in the hostvisor were successful, and to verify that the memory regions provided actually belong to the hostvisor. We then set the ownership of these memory regions to the corevisor, so that the host can no longer access or manipulate the guest VM's stage-2 page tables. In particular, we do the following:

```
for (i = 0; i < 8; ++i) {
    addr = page_starts[i];
    end = page_starts[i] + SZ_1M;

    /* ... */

    if (addr == 0) {
        // Hostvisor's alloc_pages() failed.
        __hyp_panic();
    }

    // Change the ownership of these pages
    // to the corevisor.
    do {
        index = get_s2_page_index(addr);
        owner = get_s2_page_vmid(index);

        if (owner != HOSTVISOR) {
            // The hostvisor shouldn't be
            // giving memory that belongs
            // to the corevisor or to a VM.
            __hyp_panic();
        }

        set_s2_page_vmid(index, COREVISOR);
        addr += PAGE_SIZE;
        ++page_cnt;
    } while (addr < end);
}
```

We then build the VM's stage-2 page table using these eight 1M regions, by populating the relevant fields in the *struct el2_data* object used by the corevisor for VM management.

2.3 Reclaiming memory from the corevisor

When a VM is destroyed, FlexAlloc reclaims the eight 1M memory regions that were previously allocated to the corevisor for the VM's stage-2 page table. Returning this memory to the hostvisor ensures that it does not remain unused in corevisor memory. Unlike the case of VM creation, there is no hypercall made directly to the corevisor when a VM is destroyed. Hence, we create a new *HVC_DESTROY_KVM* hypercall that mirrors the *HVC_REGISTER_KVM* hypercall used during VM creation. In particular, on ARM architectures, if the *CONFIG_VERIFIED_KVM* macro is defined, we call *hypsec_arch_destroy_vm*. This function in turn calls *hypsec_destroy_kvm*, which ultimately invokes the *HVC_REGISTER_KVM* hypercall with the corresponding VMID. We do not pass the base addresses of the eight 1M memory regions as arguments to this hypercall, since the corevisor can access them directly by reading the relevant fields of the *struct el2_data* object. Once we are in the corevisor running in EL2, we return the eight memory regions to the hostvisor. We then return to the hostvisor running in EL1, which can then call *__free_pages* on each of the eight 1M regions.

3 Evaluation

We evaluate FlexAlloc by running it on a QEMU-emulated Raspberry Pi 4. In particular, we run VMWare Fusion Professional Version 13.0.0 on a 2018 x86-64 Macbook Pro running MacOS Ventura 13.6.1. Using VMWare, we create a VM with 4 processor cores and 12 GB memory running Ubuntu 22.04.3 LTS. On this VM, we run QEMU 6.2.0 to emulate a Raspberry Pi 4. This emulated Raspberry Pi 4 serves as our host machine, on which we run an SeKVM with the FlexAlloc extension. Our guest VMs run vanilla Linux v5.4.

Using FlexAlloc, we boot up a guest VM and verify that it is able to start up without any kernel panics. We use *print_string* to verify that we are passing eight distinct 1M regions to the corevisor when the VM is created. We test a case where the 1M regions are not contiguous, and verify that our modified page table iterators are able to successfully handle a VM stage-2 page table that is non-contiguous. We also shut down the guest VM and verify that all eight 1M regions are returned to the hostvisor. We compare the base address of each region allocated during VM creation against the base address of each region reclaimed during VM shutdown, and verify that they match each other.

In order to verify that the hostvisor will generally be able to find eight distinct contiguous 1M regions when booting a VM, we evaluate how many times we can successfully call *alloc_pages(1M)* before a null pointer is returned. To reproduce our tests, simply compile *host-sekvm-project* with the *ALLOC_STRESS_TEST* macro defined. Across our three trials, after booting host SeKVM with the FlexAlloc extension, we were able to achieve between 7049 and 7054 successful 1M allocations from calls to *alloc_pages(1M)*. Since each VM only needs eight successful allocations for its stage-2 page table, we believe that the upper bound on the number of VMs we can create before running into *alloc_pages(1M)* failures is sufficiently high.

We also verify the changes in the corevisor's memory footprint by checking the number of pages owned by the corevisor, using the *s2_pages* array in *el2_data* at key points when a VM is created and destroyed. On a host running SeKVM without the FlexAlloc extension, we observe that the corevisor owns 65549 pages before booting a VM, while running a VM, and after destroying a VM. On a host running SeKVM with the FlexAlloc extension, we observe that the Corevisor owns 32768 pages before VM boot, 34817 pages during the life of the VM, and 32768 pages after destroying the VM. Note that FlexAlloc also drastically reduces the size of the statically-allocated *STAGE2_PAGES_SIZE* region in the linker script, since it now only needs to contain the stage-2 page tables of the hostvisor and the corevisor. These findings show that the corevisor in FlexAlloc is able to successfully receive, use, and return pages given to it by the hostvisor for constructing a VM's stage-2 page table.

4 Shortcomings and Future Work

4.1 Shortcomings in design

One key shortcoming of FlexAlloc is that it is not formally verified. In this sense, FlexAlloc is less like SeKVM and more like HypSec. It provides isolation guarantees in principle, but cannot be logically proven to be secure. In particular, there are a few major flaws in FlexAlloc when it comes to isolating guest VMs from the host machine. For instance, the current implementation of FlexAlloc does not use *memset* to zero out the memory given to it by the hostvisor for the VM's stage-2 page table. This presents a danger if the MMU tries to read from a page table entry that should not actually exist, but that has previously been written into the memory region by the hostvisor. We also occasionally observed stalls when shutting down VMs and booting new ones, which we believe is caused by our failure to *memset* the pages to zero. For example, if we shut down a VM and boot up a new one, the second VM's stage-2 page table could contain pages previously used by the first VM's stage-2 page table. These pages could contain non-zero data, leading to undefined behavior when interpreted as page table entries.

Another flaw is that FlexAlloc does not verify that the eight 1M regions are actually distinct and do not overlap with one another. Hence, FlexAlloc still fundamentally relies on a non-malicious hostvisor, and would require further patches to be fully secure. To fix this, FlexAlloc could simply have an additional check verifying that all of the physical addresses received from the host are unique, 1M-aligned, and non-overlapping.

Finally, our design does not fully implement dynamic memory partitioning between the corevisor and hostvisor, outside of VM stage-2 page tables. As mentioned in the introduction, another area in which corevisor memory scales with the number of VMs is in maintaining *struct el2_vm_info* objects that record the metadata associated with each guest VM. Currently, we still statically allocate a fixed-size array of *struct el2_vm_info* objects in corevisor memory [10]. This essentially preserves the hard cap on the number of guest VMs that can be run. An easy fix would be to keep this array as a statically-allocated array, while increasing the number of elements. Since each *struct el2_vm_info* object is only 768 bytes, this would not waste a significant amount of memory even if fewer than sixteen VMs are created. Moreover, each VM also requires a *struct kvm*. Hence, to lift the sixteen-VM cap present in SeKVM, we would need to either dynamically allocate the memory for this struct, or increase the size of the statically-allocated region for VM metadata. The latter approach is probably reasonable. Since the changes to the VM stage-2 page tables already decrease the size of the statically-allocated region by 128 MB, re-inflating this region to accomodate 32+ VMs would still result in substantial memory savings. The former approach is possible as well. In the subsection on Future Work, we propose DPM as a potential solution to the issue of bringing dynamic allocation to EL2 metadata structures that are not page-aligned.

4.2 Shortcomings in evaluation

We also do not fully evaluate our implementation. For one, we do not stress-test our implementation by running a large number of VMs. In addition, we do not verify whether fragmenting the stage-2 page table results in any degradation in VM performance. Given the limitations of our test environment, we only considered metrics related to the corevisor's memory footprint, and did not investigate performance metrics related to execution time. Ideally, we would run a memory-intensive benchmark on our guest VMs to verify that our iterators work properly under a high memory load across multiple VMs.

4.3 Future work

A possible area of future work is exploring other allocators designed for allocating large chunks of contiguous memory. We fragment the VM stage-2 page tables into eight distinct 1M pages, because *alloc_pages* does not allow us to allocate contiguous regions larger than 4M. We also wanted to be reasonably certain that the size of our regions would not impeed the host's ability to launch VMs. It might be possile to avoid this by looking at other Linux memory allocators explicitly designed for the purpose of large contiguous allocations and adding such an allocator to the host.

4.4 Dynamic Page Manager

4.4.1 Overview

To enable dynamic allocation of EL2 memory in general beyond just the scope of a VM's stage-2 page tables, we also designed but did not fully implement a dynamic page manager (DPM). DPM is a system that exposes an interface for other "services" within EL2 to utilize dynamic allocation for their memory regions, while hiding the implementation of actually receiving memory from and returning memory to the host. This would be another interesting area of future work. Please see the *dpm* branch of the linked repository for a partial DPM implementation.

By implementing an interface for EL2 to dynamically "request" and "return" arbitrary regions of contiguous memory from EL1, the corevisor would no longer be limited to using statically-allocated, fixed-size regions. Instead, other services within EL2 could use DPM to dynamically request regions of contiguous memory. These services would be responsible for managing the contents of these pages once DPM allocates these pages to that specific service. The service would be required to signal DPM when that contiguous region is no longer in use, using an interface that is similar to the kmalloc and kfree model.

DPM operates at the contiguous region granularity rather than the page granularity because a lot of the existing SeKVM code assumes contiguous regions. If DPM were to instead piece together pages to form fragmented regions, all of this other code would break. Thus DPM can actually be thought of as a memory-region manager rather than a page manager. DPM is structured in such a way that a second page-granularity version of DPM could operate alongside the region-granularity version of DPM, thus allowing some services to remain dependent on contiguous regions while others adapt to use separated non-contiguous regions, in order to achieve better memory utilization by EL2.

4.4.2 Interface

Our design of DPM exposes the following additional hypercalls to the hostvisor running at EL1:

- Allocating pages to EL2: *hvc_give_pages(base_phys_addr_of_region, size_of_region)*. Note that the size must be page-aligned.
- Reclaiming pages from EL2: *hvc_reclaim_pages(base_phys_addr_of_output_struct)*. The output struct is provided by the hostvisor, and should contain an array of physical addresses. Services at EL2 should record the physical addresses of regions being handed back to the hostvisor in the output struct. The hostvisor can then read from the output struct and free the relevant pages. Every physical address that EL2 records in this output struct is considered freed and no longer managed by DPM.

Our design of DPM also exposes the following interface to services running at EL2 that want dynamically-allocated memory:

- Requesting a contiguous region of a given size from the DPM pool: *dpm_alloc_region(size)*. Returns an address to the region if one was available in the pool, and 0 to indicate an error which the requesting service must propagate back to EL1.
- *dpm_free_region(starting_region_addr, optional: size)*: Signals that a previously-allocated region can be returned to the DPM pool.

4.4.3 Metadata

DPM manages regions of pages using a circular linked list (Figure 4). When an EL2 service requests a region of a specific size from DPM, DPM will search this list for an available match. If no matches are found, DPM returns an error indicating to the EL2 service that it must propogate the error back to the host. Upon receiving the error, the host will make an additional hypercall that passes a new region of some size to DPM. After restarting the original hypercall, DPM will successfully find a region to hand to the EL2 service. The code below illustrates the data structures used to store this metadata.

```
// DPM Overall Metadata
// Contained in struct el2_data
struct dpm {
    // Address to the first
    // dpm_region page
    u64 base_page_addr;

    // Number of regions currently in pool
    u64 region_count;

    // Number of pages currently in pool
    u64 page_count;
};

// DPM Region-Specific Metadata
struct dpm_region {
    // Addr of the beginning of the first
    // page in the region
    u64 start_addr;

    // Number of pages in region
    u64 page_count;

    // DXXX XXXA, where A is allocated flag,
    // D is DPM flag meaning that region
    // holds DPM metadata, X is available bit
    u8 flags;

    struct dpm_region* prev;
    struct dpm_region* next;
};
```

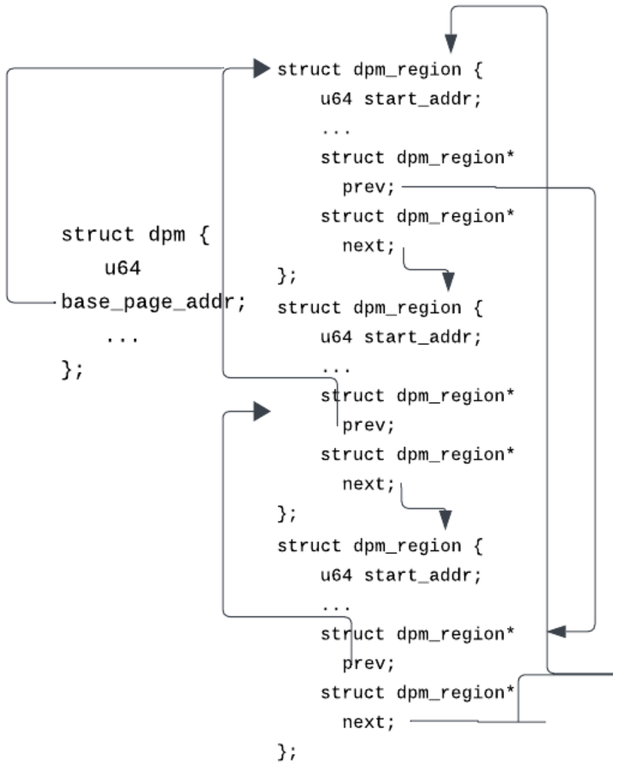

Figure 4: FlexAlloc: DPM Metadata Organization

If the host is under a lot of memory pressure, the host can request that DPM give control of available regions back to the host. The host does so by invoking a hypercall that passes a page *P*, which DPM will populate with an array of metadata pertaining to regions that it is handing back. DPM will first unmap this page *P* from the host. DPM will then search through the list mentioned above for all available regions and add them to *P*. Finally, DPM will remap *P* to the host and return. The host can then call *free_pages* on all of the regions listed in *P*.

It is important to ensure that DPM can receive the pages that it needs for its own metadata. If the host receives an error when trying to hand a region of some size to DPM, the host will know that it needs to first pass a region of size *PAGE_SIZE* to DPM in a seperate hypercall, and then retry the original region pass hypercall, before finally retrying the original hypercall pertaining to some other EL2 service. When DPM receives a region and notices that it is out of metadata space, it will search for a region of *PAGE_SIZE* to claim for itself before processing the larger region. If one is found, then DPM will claim it and process the region hand off as normal.

4.4.4 Considerations and Drawbacks

DPM allows services within EL2 to request arbitrary regions of contiguous memory at arbitrary points in time. This is valuable when we may or may not need additional pages. If we know some hypercall will definitely require EL2 to receive more pages, the error-check-retry pathway may be an unnecessary cost. For example, when we want to boot a new VM, we know EL2 will definitely need a memory region for the stage-2 page table of the VM. However, since regions returned to DPM can be re-allocated to other EL2 services as long as memory is not reclaimed by the host, a new VM could potentially reuse the region returned by a previously-shutdown VM. DPM is designed to be convenient to use, but its generalized strategy can lead to extra, avoidable work. We design DPM with programmability in mind over efficiency, in hopes that it would make full dynamic allocation attainable for SeKVM services by avoiding excessively-customized handoffs between the host and corevisor. Although we only partially implement DPM, it would be interesting to compare the real-world performance metrics between our current stage-2 dynamic allocation strategy, and a stage-2 dynamic allocation strategy built on top of DPM and a check-error-retry methodology.

5 Conclusions

We have designed and implemented FlexAlloc, a system for dynamic memory partitioning between the corevisor and hostvisor in SeKVM. In particular, we modify SeKVM so that the hostvisor allocates memory to the corevisor when a VM is created, and reclaims memory from the corevisor when a VM is destroyed. As a result, the memory for a VM's stage-2 page table does not need to be statically allocated to the corevisor upon bootup. This system reduces the wastage of memory when VM demand is low, and increases the number of VMs that can be run on a single machine when VM demand is high.

We also make a distinct contribution in terms of fragmenting a VM's stage-2 page table so that it can be composed of multiple noncontiguous regions of memory, instead of a single contiguous 8M region. Altogether, this system improves hardware utilization and reduces VM operating costs.

Acknowledgments

Shih-Wei Li, John S. Koh, and Jason Nieh implemented HypSec, which serves as the predecessor to SeKVM. Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui implemented and formally verified SeKVM. Xuheng Li wrote a helpful set of notes describing the boot and initialization process in SeKVM, as well as the steps needed to build and run SeKVM [7]. Jason Nieh and Xuheng Li provided helpful comments on the project and helped with the design of FlexAlloc. This paper was written as part of the class COMS E6118 Operating Systems II at Columbia University.

References

[1] Elixir Bootlin. mmzone.h in linux v5.4.55. https://elixir.bootlin.com/linux/v5.4.55/source/include/linux/mmzone.h#L27. Accessed: 2023-12-22.

[2] Elixir Bootlin. page_alloc.c in linux v5.4.55. https://elixir.bootlin.com/linux/v5.4.55/source/mm/page_alloc.c#L4718. Accessed: 2023-12-22.

[3] ARM Developer Documentation. Learn the architecture - aarch64 virtualization. https://developer.arm.com/documentation/102142/0100/Stage-2-translation. Accessed: 2023-12-21.

[4] kernel.org. Chapter 6: Physical page allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html. Accessed: 2023-12-22.

[5] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting cloud virtual machines from commodity hypervisor and host operating system exploit. In *Proceedings of the 28th USENIX Security Symposium*, pages 1357–1374, 2019. https://www.cs.columbia.edu/~nieh/pubs/security2019_hypsec.pdf.

[6] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Z. Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *Proceedings of the 30th USENIX Security Symposium.*, pages 3953–3970, 2021. https://www.usenix.org/system/files/sec21-li-shih-wei.pdf.

[7] Xuheng Li. Fall 23 Operating Systems 2 Homework. https://xuhengli.notion.site, 2023. Accessed: 2023-12-21.

[8] XuhengLi (GitHub username). el1.c in osdi23-paper114-sekvm. https://github.com/columbia/osdi23-paper114-sekvm/blob/ae/arch/arm64/hypsec_proved/el1.c. Accessed: 2023-12-21.

[9] XuhengLi (GitHub username). el2.c in osdi23-paper114-sekvm. https://github.com/columbia/osdi23-paper114-sekvm/blob/ae/arch/arm64/hypsec_proved/el2.c. Accessed: 2023-12-22.

[10] XuhengLi (GitHub username). hypsec_host.h in osdi23-paper114-sekvm. https://github.com/columbia/osdi23-paper114-sekvm/blob/ae/arch/arm64/include/asm/hypsec_host.h. Accessed: 2023-12-22.

[11] XuhengLi (GitHub username). kernel-pagetable.h in osdi23-paper114-sekvm. https://github.com/columbia/osdi23-paper114-sekvm/blob/ae/arch/arm64/include/asm/kernel-pgtable.h. Accessed: 2023-12-21.

[12] XuhengLi (GitHub username). vmlinux.lds.s in osdi23-paper114-sekvm. https://github.com/columbia/osdi23-paper114-sekvm/blob/ae/arch/arm64/kernel/vmlinux.lds.S. Accessed: 2023-12-21.