Dark Shadows: User-level Guest/Host Linux Process Shadowing

Peter Dinda Akhil Guliani
Department of Electrical Engineering and Computer Science
Northwestern University

***Abstract*—The concept of a *shadow process* simplifies the design and implementation of virtualization services such as system call forwarding and device file-level device virtualization. A shadow process on the host mirrors a process in the guest at the level of the virtual and physical address space, terminating in the host physical addresses. Previous shadow process mechanisms have required changes to the guest and host kernels. We describe a shadow process technique that is implemented at user-level in both the guest and the host. In our technique, we refer to the host shadow process as a *dark shadow* as it arranges its own elements to avoid conflicting with the guest process's elements. We demonstrate the utility of dark shadows by using our implementation to create system call forwarding and device file-level device virtualization prototypes that are compact and simple.**

***Keywords*-operating system, virtualization, shadow process**

I. Introduction

The term *shadow process* has been coined in several domains, most notably in intrusion detection (e.g., [11], [13]), and in distributed computing environments (e.g., [21]). As one might guess, in all these domains, the concept is that of a process that replicates some aspects of an original process. In this paper, we define a shadow process as one that replicates the virtual and physical address space of the original process while also allowing the inclusion of additional code and data, and having an independent control flow. And in particular, we are interested in cases where the original process is a *guest process* running within a guest OS in a virtual machine, and the shadow process runs within the host operating system. More specifically, we consider a guest process running in a Linux-like guest OS running under a VMM that is embedded in a Linux-like host OS.[1]

In this kind of virtualization context, shadow processes have considerable utility because they make it possible to selectively grant the guest process access to features available to a host process without requiring a massive development effort. This makes it possible to implement quite powerful services for a guest, again without requiring a massive development effort. Perhaps the best example is Sani et al's work on device file virtualization [17], [15], [16], which we elaborate on in Section VI. For the high performance computing community, their demonstration of how to provide access to a black-box GPU to a guest process without any special hardware support, porting of drivers, etc, is the most intriguing. It is quite common in applying virtualization in the HPC context to have either (a) no way to make a device visible to the guest in a controlled way, and/or (b) no drivers for the device in the guest. A core idea in device file virtualization is that interactions with the device, at the level of system calls, are forwarded to a shadow process on the host, which executes them. The drivers are available in the host, access to the device is controlled by normally available mechanisms in the host, and yet the guest can use the device seamlessly.

Previous shadow process work of this kind has required modifications to the host and guest kernels. Since the shadow process concept requires the inspection of guest page tables and the creation of page tables for the shadow process that mirror those of the guest process, it might appear that this is a hard requirement. In this paper, we show how to implement the shadow process concept using entirely user-level means, and no modifications to the guest and host kernels.

Why is this useful? First, in principle it would allow the creation of services that are independent of any particular VMM or other virtualization model (e.g., containers [4] or a partitioned host [14]). Second, it would ease the development of services for virtual machines as these could be implemented at user-level within a shadow process. Finally, it would ease the practical deployment of shadow process-based services, particularly in Linux-like HPC environments such as CNL [9].

Our user-level shadow process technique produces *dark shadows*. A dark shadow shares the virtual and physical address space of its guest process. The techniques for achieving this simply require the ordinary system call interface, access to introspection mechanisms (specifically `/proc`), and the ability to `mmap()` physical memory, all of which can be controlled and secured using standard Unix mechanisms. The shadow is dark in that it appears to contain nothing other than the mirrored address space. A key contribution is achieving this behavior while allowing a service to run within the shadow. This is accomplished by compile-, link-, and run-time techniques that encapsulate the service and make it mobile within the address space, and thus make it possible for the service to be located at unused addresses,

This project is made possible by support from the United States National Science Foundation through grant CCF-1533560 and from Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the United States Department of Energy's Office of Science.

[1] Our proof of concept uses the Palacios VMM [12] embedded in a stock Red Hat 6.5 environment (Linux 2.6.32 kernel) as a kernel module and running a Linux guest. This arrangement is quite similar to KVM and our results would be immediately applicable there as well.

and thus avoiding causing a conflict.

Our contributions are as follows:

- We describe the underlying mechanisms of the dark shadow technique and how they fit together to create the dark shadow technique. (Section II)
- The technique depends on the tractability of recreating page tables using the `mmap()` mechanism. We present an empirical study of over 1.2 million processes on production machines that argues for its tractability. (Section III)
- We analyze the security and trust aspects of the dark shadow technique and argue that access control and security can be achieved using standard Unix mechanisms. (Section IV)
- We describe the design and implementation of a system call forwarding service that allows guest processes to make host system calls, even those involving direct and indirect pointer arguments. (Section V)
- We describe the design and implementation of a user-level device file virtualization service using an NVIDIA GPU as our example. (Section VI)

It is important to be clear about the scope of this work and its claims. First, the concept of a shadow process is an old one, and its use for services such as system call forwarding and device file virtualization also predates this work (Paradice [15] is the most important point of comparison). Our work is an engineering design study to show how a shadow process mechanism can be implemented at user-level. The second clarification is what is meant by user-level: we mean that no source code changes to the guest and host kernels are involved in the dark shadow technique; both host and guest kernels are out-of-the-box binaries, as is the VMM. Additional kernel modules may be involved in the host kernel in implementing a service, but these also do not require any kernel source changes or recompilation.

II. Mechanisms

The overview of the dark shadow model and technique is given in Figure 1. The goal is to map the user portion of a guest process's virtual address space into a host process's address space with the invariant that both the physical *and virtual* addresses match in both processes. This is achieved in part by making the host process's own components (e.g., code, data, stack, heap), *mobile*. This set of components, which we call the *dark shadow capsule*, then moves out of the way of any given guest process mapping. A service is then implemented in the capsule with the assumption that it has the virtual address space of the process at its disposal.

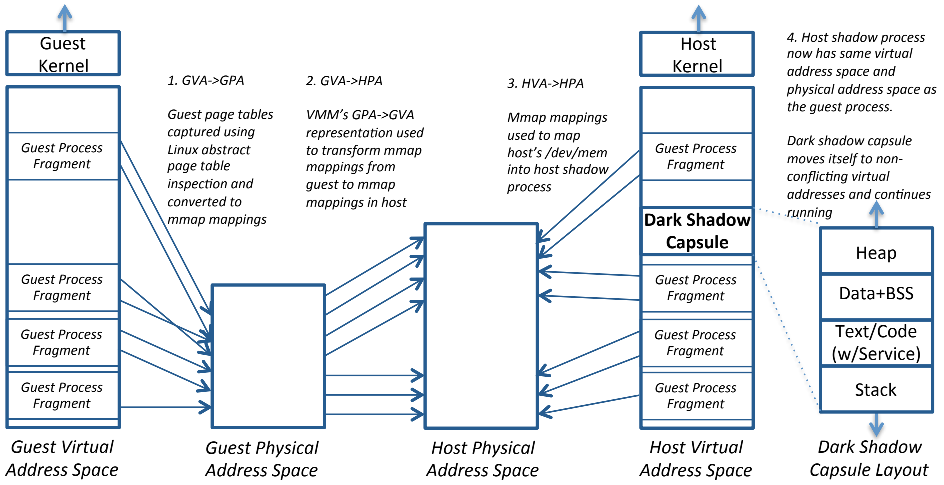
The implementation of this model from guest page table discovery, to translation, to construction of the host page tables, to the mobile capsule, has been designed with the requirement that no host or guest kernel changes are to be made, including no kernel modules. The sole exception is the VMM itself, which we assume can perform guest physical address to host physical address translation for us. This is enabled by the Linux's page table introspection mechanism and the ability to `mmap()` physical memory. Using the enumerated points of Figure 1, we describe how (1) the guest page table is discovered, (2) translated to be usable in the host, (3) mapped into the host process, and (4) how the dark shadow capsule's migratory capability works. We also describe the requirements for services.

Guest process page table discovery: We extract the salient information in the guest's process's page table, namely the guest virtual to guest physical address mapping (GVA→GPA), using the guest Linux's memory map and abstract page table mechanism. This mapping is then compactly represented by run-length-encoding simultaneous runs of virtually and physically contiguous pages. The mapping itself is designed to be readily transportable and to itself be easily mapped into an address space for use.

For a process with process id `pid`, Linux provides an easily parsed representation of its memory map in `/proc/pid/maps`. We iterate over the regions of the memory map. For each page within a region, we consult `/proc/pid/pagemap`, which is the abstract virtual to physical mapping of the process. We join this information with that in `/proc/kpageflags` and `/proc/kpagecount`, which describe the properties of the physical pages, as well as additional attributes the guest kernel has given them. The outcome is the (GVA→GPA) page mapping we need, at the granularity of the smallest page size (4 KB for x64).

We next compactly represent the page mapping and make it suitable for reconstruction in the host. This is achieved through run-length encoding (RLE). We scan the mapping, finding runs in which both virtual page numbers (VPN) and physical page numbers (PPN) are consecutive. This readily detects both "natural" contiguity in the mappings and "artificial" contiguity that results from the guest kernel promoting a mapping to large or huge pages. Each run is encoded with its starting VPN, starting PPN, length, and physical page attributes. The entire GVA→GPA mapping is reduced to an array of these runs. We refer to such an array as the *GVA→GPA map* for the process. The array can easily be written to a file or otherwise transported since it is a pointer-free blob.

Our tool that implements this process can be used either as a user-level library, for example linked into an LD_PRELOAD library that implements part of a service or as a user-level command-line tool in the guest. In either case, the only requirement in the guest is that the LD_PRELOAD library or the command-line tool is executed with sufficient privileges to read the elements of the `/proc` filesystem noted above. By default, our tool considers all mapped regions in `/proc/pid/maps` that are within the "user half" of the virtual address space (i.e., the canonical lower



Guest Kernel
Guest Process Fragment
Guest Process Fragment
Guest Process Fragment
Guest Process Fragment
Guest Virtual Address Space
1. GVA->GPA
Guest page tables captured using Linux abstract page table inspection and converted to mmap mappings
Guest Physical Address Space
2. GVA->HPA
VMM's GPA->GVA representation used to transform mmap mappings from guest to mmap mappings in host
Host Physical Address Space
3. HVA->HPA
Mmap mappings used to map host's /dev/mem into host shadow process
Host Kernel
Guest Process Fragment
Dark Shadow Capsule
Guest Process Fragment
Guest Process Fragment
Guest Process Fragment
Host Virtual Address Space
4. Host shadow process now has same virtual address space and physical address space as the guest process.
Dark shadow capsule moves itself to non-conflicting virtual addresses and continues running
Heap
Data+BSS
Text/Code (w/Service)
Stack
Dark Shadow Capsule Layout

Figure 1: Overview of model. The host virtual address (HVA) space of the shadow process mirrors the guest virtual address (GVA) of the guest process, and the HVA→HPA mapping mirrors the GVA→HPA mapping. The shadow processes's service resides in a mobile "dark shadow capsule" that moves to non-conflicting addresses in the host virtual address space. This mobility makes it possible to precisely replicate the user portion of the guest process's virtual address in the shadow process on the host, and vice-versa.

half of the x64 address space), except for the vDSO.[2] However, this can easily be restricted to specific mapped regions, for example if the service we are building needs access only to specific regions.

Translation and transport: The GVA→GPA map is not sufficient to construct the shadow process on the host, and must be translated into a *GVA→HPA map*, where HPA refers to host physical addresses. The VMM maintains the GPA→HPA mapping, so it can perform this translation for us. In our Palacios VMM, while the GPA→HPA mapping can be arbitrary, it is most commonly the case that the guest physical address space is mapped using a small number of contiguous chunks of host physical addresses, typically conforming to the NUMA boundaries of the host. As a consequence, the GVA→GPA map resulting from the translation typically is the same size as the GVA→GPA map produced in the guest.

Existing mechanisms within the VMM allow us to transport the GVA→HPA map to the host. In our case, our in-guest tool or library can either hypercall Palacios to do so, or it can place the map on a filesystem shared with the host.

It is important to note that at any time, it is straightforward to validate a GVA→HPA map with the VMM. The core invariant to check is whether any HPA in the map extends outside the HPAs allocated to the guest by the VMM, which is readily done.

Shadow process page table instantiation: Once the GVA→HPA map is available in the host, the shadow process instantiates it by using the `mmap()` system call to map the specified regions of `/dev/mem` into its host virtual address (HVA). An `mmap()` call is made for each region in the GVA→HPA map. The host kernel will instantiate page table entries to reflect these mappings on an as-needed basis. The effect is the same as if we had built page tables directly in the host kernel ourselves.

The result is that the GVA→HPA map is merged with the shadow process's original HVA→HPA map. As a consequence any valid pointer in the guest (a GVA) is now a valid pointer in the host. Furthermore, since it maps identically to the ultimate hardware physical addresses, it can also be passed directly or indirectly to host device drivers that use DMA. No pointer swizzling or data copying is needed.

Making the shadow process dark: The heart of the dark shadow technique lies in the merger of the GVA→HPA map and the shadow processs's own HVA→HPA map. Simply put, there must be no conflict for virtual addresses in the shadow process between the two maps—the two maps must mesh without overlap.

The dark shadow technique achieves this by making it possible for the shadow process to dynamically reconfigure itself at runtime to avoid overlap. When given a new GVA→HPA map, the dark shadow relocates its code, data, stack, etc, to non-conflicting addresses, and continues running in the new location. Thus, a service can run within the dark shadow alongside the guest process's memory and address space while able to operate directly on it.

Dark shadowing is accomplished by a combination of compile-, link-, and run-time mechanisms. In our implementation, the dark shadow code is a template that at its core invokes the service. Much of the complexity of dark shadowing is hidden from the service developer provided a small set of requirements, given later, is maintained.

The entry point for the dark shadow executable (e.g., `_start()`) goes directly into our bootstrap code to assure it runs prior even to the C runtime. This allows it to know the precise execution conditions (the starting page of the

[2] The virtual Dynamic Shared Object (vDSO) is a kernel interface library mapped by the kernel into the address space.

kernel supplied stack, where the _start() code itself is located within the current page, etc). Additional invariants are guaranteed by the use of a custom linker script that results in the initial code, data, and bss locations being clear. All code is compiled with position independence, resulting in all control flow and data access being PC-relative. This allows for relocation of the code and data as we execute it. During bootstrap, the process uses a hand-coded assembly interface to invoke system calls, and has only a small set of C library functions that are built in.

Using this minimal and clear execution environment, the dark shadow executable then maps in or otherwise acquires the GVA→HPA map. It then builds up its own map, using /proc/self/maps to determine where the host kernel has placed its initial heap, stack, vDSO, and other runtime elements. Note that with address space randomization these elements can change from run to run and so cannot be determined statically. Combining the two maps, the process computes a new mapping of its own elements that will not conflict. This mapping constitues a contiguous capsule in which the text, data, and bss are sandwiched between the heap (which grows to higher addresses above them) and the stack (which grows to lower addresses below them).

To instantiate this mapping, the dark shadow code first allocates space for the destination using mmap(). Next, it copies its text, data, bss, and stack to the new addresses. It then performs a stack switch using custom assembly that resembles that of a thread context switch in a threading package. At this point, by construction, the code is aware that it is on the very first page of the stack, which simplifies this initial stack switch, as well as subsequent ones. In the next step, we compute and execute an indirect jump to get to the next instruction within the new copy of the text. This completes the dynamic relocation of the capsule.

Now that the dark shadow is executing in the relocated code using the relocated data, bss, stack, and heap, it discards the originals. It does this by using munmap() to remove all user-space (lower-half) memory mappings that were originally enacted by the kernel. This amounts to removing the original text, data/bss, stack, heap, and vDSO mappings. At this point, the only mappings in the user-space are the dark shadow capsule and the GVA→HPA map. The latter is then also relocated if it conflicts. We do not make it part of the dark shadow capsule since subsequent updates may change its size. Conceptually, an update may require us to find new homes for the capsule and the map. Both can live at any address, so the main thing we need to know on an update is how large the capsule currently is and the size of the new map. The new map can be temporarily placed into memory at any non-conflicting location.

The next step is for the dark shadow to instantiate the GVA→HPA map. It does this simply by opening /dev/mem and then mmap()ing each entry in the map using MAP_FIXED. That is, the GVA in the entry provides the target virtual address, the HPA provides the offset into /dev/mem, the run length provides the length, and the MAP_FIXED option forces to kernel to use our target virtual address. By virtue of the dark shadow's relocation, nothing else is mapped at our target virtual address, and so the mmap() request will succeed. After all the entries are completed, the address space of the dark shadow consists of the user address space of the guest process, the capsule, and the map.

Control is now passed to the service, which is a part of the capsule. The service can now use any virtual address that is valid in the guest process (and that has an instantiated page table entry in the guest), and it will refer to the same ultimate memory location.

The service will check to see if maintenance is needed. If so, it is obligated to call back to the relocation code to allow it to examine the new map and dynamically relocate the capsule if needed.

Service requirements: The service embedded in the dark shadow template must currently have the following properties:

- It and any libraries it depends on must be statically linked with the dark shadow template.
- It and any libraries it depends on must be compiled with position independence (e.g., -fPIC in gcc).
- It must never store pointers into the stack. Handles are acceptable. (A handle is a pointer-to-a-pointer with an associate/disassociate protocol.)
- If it must use the heap, it must do so with handles.

The requirements for using handles are due to potential future relocations. If the service developer knows future relocations cannot happen (i.e, if the map never changes), they can use pointers without concern. Otherwise, before calling back to the relocation code, the service should disassociate all handles. After the relocation completes, the service can then reassociate them. It is important to understand that this handle requirement applies only to pointers within the capsule that point to code or data within the capsule. Pointers within the capsule to guest process code or data outside the capsule can never change due to relocation.

Map maintenance: It is important to point out that a GVA→HPA map reflects the state of the guest page tables at the time the map was acquired. A memory mapping may exist in the guest, but not yet have a page table entry. This page table entry may be created later. Similarly, a memory mapping may added, removed, or updated, resulting in changes to the page table entries.

It is the service's responsibility to detect changes in the guest and forward them to the shadow process for instantiation. This can be done by construction. For example, in an HPC environment, the guest process may already have invoked mlockall() to pin its memory before the initial map is extracted. Alternatively, a LD_PRELOAD library might be a component of the service, and it might use mlock() to assure any arguments about to be made visible to the shadow process are pinned, then forward an updated map. Of note, the mechanism of Section V could be used to intercept all mmap() system calls and edit them to add the MAP_POPULATE flag. This forces the eager creation of the page table entries implied by the mmap(). The VMM itself could also determine updates to the map by monitoring guest page tables themselves, as is already done for shadow paging, for example.

III. Evidence of map file workability

Our mechanism relies on the tractability of reconstructing the virtual address to physical address mapping of the page table of the guest process using the mmap() system call. Of course, this is not at all the purpose of mmap() nor the map region data structure that underlies it in the kernel. We now describe the issue in more detail and report on a study that provides empirical evidence that suggests our technique is tractable almost all of the time.

Issue: The main purpose of mmap() is to associate runs of virtual addresses with either runs of offsets within a file or with anonymous memory. The map region data structure can be thought of as a list of such associations. The kernel then incrementally builds page table entries from this list. It is critical to understand that a single memory region in a process may translate to a large number of page table entries that map to non-contiguous physical pages. In the dark shadow technique, we attempt to represent a large number of page table entries using memory regions, the *opposite* of the normal usage. We could conceivably need to have as many memory region (mmap() requests) as there are page table entries.

The key to tractability is to be able to exploit *runs* of page table entries that represent virtual to physical mappings that are both virtually *and physically* contiguous. For example, consider the sequence of mappings $(1 \rightarrow 5, 2 \rightarrow 6, 3 \rightarrow 7)$. This sequence of three page table entries can be run-length-encoded as $(1 \rightarrow 5 \times 3)$. That encoding can then be implemented as a single mmap() request (a single map region). If the page table entries of the guest process (the GVA→GPA mapping) can be practically compressed in this way, the result would be arguably tractable numbers of map regions in the shadow process (which produce the HVA→HPA mapping. [3]

Study: To study this issue, we evaluated the page tables produced by Linux in production environments. We developed a tool that periodically dumps the page tables of all of the processes on a machine, and then attempts to compress them using the run-length-encoding technique described above. That is, for each process, we can compare the raw page table representation of the virtual to physical address mapping with the best mmap()-based reconstruction of it.

We ran our tool every 15 minutes for a period of 19 days on two heavily used servers in our department.

Murphy is a Dell R410 server equipped with 128 GB of memory. It runs Red Hat 6.7 (stock Red Hat-provided 2.6.32 kernel) and Oracle 11g Enterprise 11.2, as well as Apache and other tools needed to build Oracle-based web applications. During the time of the study it was being used to teach a databases course in which 50 students were simultaneously developing applications based on running analysis queries on FEC political contribution data. No throttling was involved. Murphy gives an example in which there are simultaneously many users and processes that have vast virtual address spaces. Each Oracle process on the machine has over 50 GB of mapped memory. At peak utilization, there are over 150 of these processes (which have many shared mappings).

Hanlon is a Dell T620 server equipped with 128 GB of memory, and NVIDIA K20 and Intel Phi co-processors. It runs Red Hat 6.7 (stock Red Hat-provided 2.6.32 kernel) and the toolchains needed to support the coprocessors. During the study, it was extensively used in an introductory computer systems course by about 150 students.

Results: We collected the statistics of >770,000 processes on Murphy, and >480,000 processes on Hanlon.

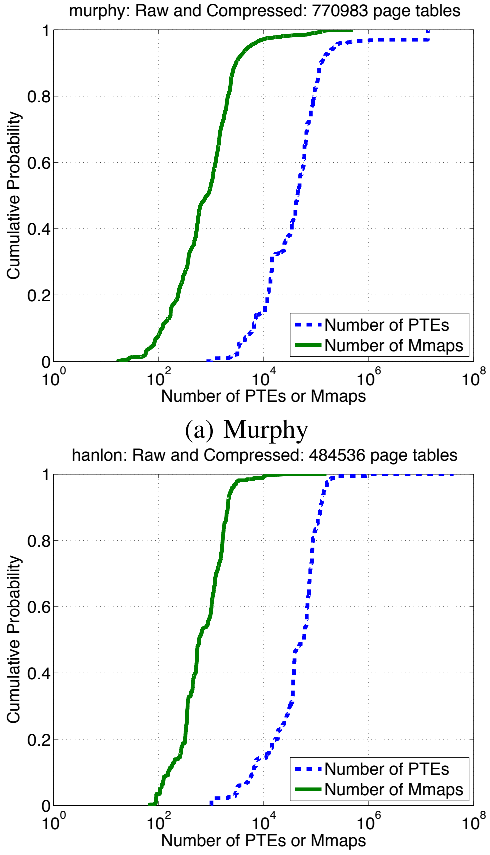
Figure 2 compares the distributions of the sizes of the raw page table representations[4] and the compressed mmap representations. The most important things to observe is that the compressed mmap representation is typically two orders of magnitude smaller than the raw page table entry representation, and that the mmap representation is almost always compact in absolute terms.

On Murphy, the 95th percentile mmap representation is 5430 mmap entires. Even the 99th percentile is ∼88000 entries. Hanlon's processes can be represented even more compactly—the 95th and 99th percentiles are 2389 and 9751 entries, respectively. Linux maintains the mmap entries in a red-black tree that can easily support these numbers of entries efficiently.

Figure 3 shows the distribution of the compression ratios (defined as the ratio of the number of mmap entries to the number of raw page table entries). On Murphy, the 95th and 99th percentiles are 0.0844 and 0.2315, respectively. On Hanlon, compression is typically even better—these percentiles are 0.0777 and 0.1274.

[3] An astute reader will note that the GPA→HPA mapping of the VMM is also critical, since we are really trying to represent the whole GVA→HPA mapping in the shadow process. Our VMM, Palacios, is typically configured to do GPA→HPA mapping using large contiguous chunks, and so the additional layer of translation does not appreciably change the compression problem. Other VMMs can be configured similarly. It is also important to note that VMMs like Palacios can use large (2MB), huge (1 GB), and will use future gigantic (512 GB) nested page table entries in order to reduce TLB pressure. The use of these larger pages also reduces the amount of physical non-contiguity that can be introduced by the GPA→HPA mapping.

[4] That is, the number of 4KB page table entries marked as present. Regardless of which page sizes are used, Linux's abstract page table mechanism shows us behavior at 4KB granularity.



(a) Murphy

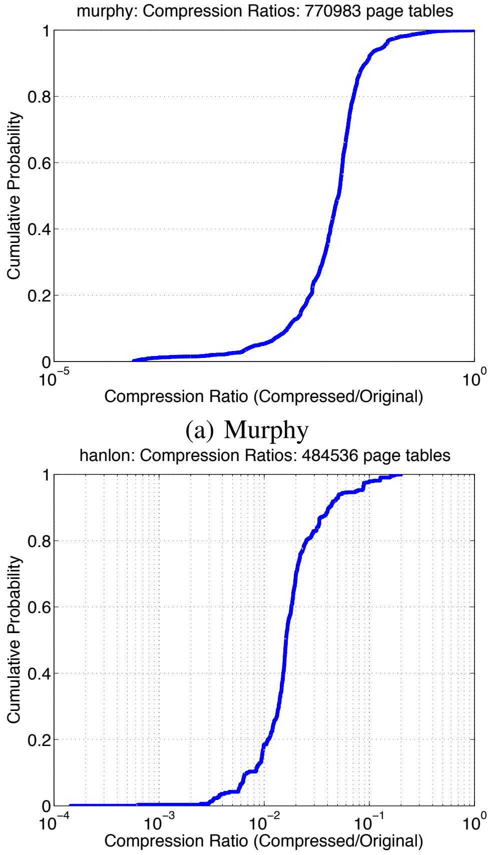
(b) Hanlon

Figure 2: Comparison of page table entry representation of address space and the compressed mmap representation needed to reconstruct it. In practice, the vast majority of processes' page tables can be replicated with a small number of `mmap()` requests that map `/dev/mem`. The ability to efficiently replicate specific page tables indirectly at user-level via `mmap()` is a central requirement for the dark shadow technique.

This data is quite promising for our purposes, but it could be that "large" processes, those with many raw page table entries, achieve less compression. This is fortunately not the case. Figure 4 plots compression ratios versus size (number of raw page table entries). There is little relationship between the two. In fact, the coefficient of correlation in both cases is actually slightly negative (-0.097), suggesting that if anything, compression gets slightly better with size.

The extreme of this can be seen by considering the "notch" at the extreme right of the CDF for Murphy (Figure 2(a)). This is due to the Oracle processes. These are actually compressed extensively as Oracle and Linux are using large pages, which guarantee at least 2 MB regions of virtually and physically contiguous addresses. In Figure 4(a), these are the small set of points at the extreme right, which show very high levels of compression.

Our conclusion is that in practice the virtual address to physical address mapping of a Linux process in a guest can be reconstructed with a tractable number of `mmap()` requests mapping `/dev/mem` in the shadow process running in the host. Even in the worst case we examined, 95% of processes could be represented with fewer than 6,000 `mmap()`s. There are rare exceptions to this, but they do not stem from size. Indeed, larger processes typically use larger page sizes, creating more opportunities for compression in the `mmap()` representation.



(a) Murphy

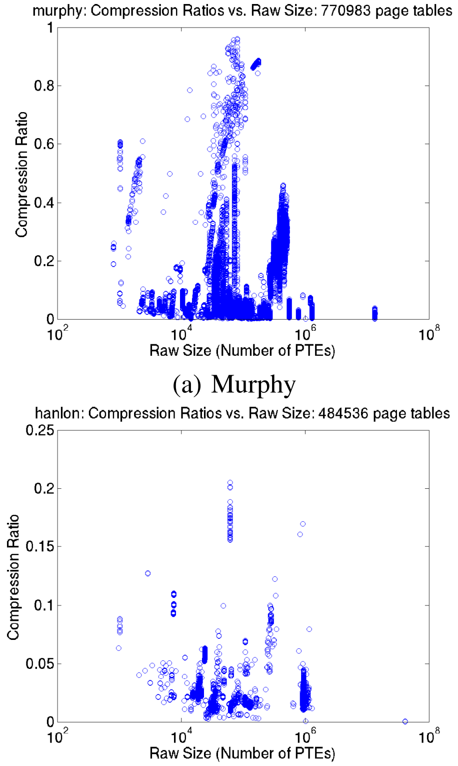
(b) Hanlon

Figure 3: Distribution of achieved compression ratios. The virtual to physical mappings of most processes' page tables can be compactly represented with `mmap()` requests that map `/dev/mem`. This is central to the practicality of the dark shadow technique.

IV. Security and Trust

The dark shadow mechanism can be secured using existing Linux mechanisms.

Guest process page table discovery is already limited to those users and groups that have permissions on its `/proc/pid/` entries, which is based on the effective user id of the guest process within the guest. That is, someone in the guest can access the guest process page tables only if they can access the guest process itself. The guest-wide information in `/proc/kpagemap` and `/proc/kpageflags` requires more privileged access, but the page table extraction code can itself be encapsulated in a `setuid` executable through which the guest administrator provides the needed privileges, while maintaining



(a) Murphy

(b) Hanlon

Figure 4: Compression ratios versus original size. There is little relationship between the effectiveness of representing a process's page table with `mmap()` requests that map `/dev/mem`, and the size of the portion of the address space that is occupied. In fact the largest processes are generally the most compressible. This is due to increased use of larger pages, either directly by the process or indirectly by the kernel, for larger processes. The user-level dark shadow technique is effective even for very large processes, such as the Oracle processes at the extreme right of (a).

the permissions check on the process-specific data, and restricting GVA→GPA translations to only those GVAs deemed legitimate.

Translation and transport is governed by the VMM. The VMM can simply refuse to do any GPA→HPA translation for a GPA that is invalid for the guest. In this way, no map can be conveyed to the dark shadow process that addresses host physical addresses outside of the guest.

The VMM (or host kernel) as a trusted party can provide a binding mechanism between guest processes and host dark shadow implementations. In essence, the guest process and dark shadow process are cooperative and thus can share a secret which the VMM or host kernel can then validate. For example, the VMM could associate hashes of the dark shadow implementations with hashes of the guest executables. Alternatively, the VMM can just provide a communication channel between the two so that the secret can be validated by internal means.

The heart of security and trust in the system is in shadow process page table instantiation. As described, the dark shadow must be able to `mmap()` segments of `/dev/mem`, the physical address space of the host. Palacios (and other tools) configure the kernel to allow this, and the translation and transport mechanism can guarantee it provides a safe list of regions to map (none with HPAs outside of the guest), but we trust the dark shadow not to make other mappings.

The mechanism of conferring trust on a dark shadow executable is based on the fact that root privileges are needed to `mmap() /dev/mem`. The system administrator can thus select, by making them `setuid` root, which specific executables are given this privilege. As we describe later, dark shadow-based services can often be extremely compact given the address space invariants, so in practice the amount of code to trust will be small. Alternatively, a trust framework, with the dark shadow executable signed with a trust chain, could be used to determine whether the executable could be trusted on the specific system based on its validated provenance.

If a non-trust-based approach is needed, a tiny host kernel module, similar to that described in Section V, could be introduced whose sole purpose would be to validate all `mmap()` calls from a dark shadow process, comparing notes with the VMM to discard spurious `mmap()`s.

Another approach to assuring that the dark shadow process never `mmap()`s to disallowed host physical addresses is to use a mechanism identical to that described in Section V to intercept `mmap()` system calls made by it. This would constitute an `LD_PRELOAD` library that the host administrator would require. The library would transform illegal `mmap()` system calls into no-ops (and alert the administrator).

V. Service: System Call Forwarding

In this service, the guest process has selective access to system calls within the host via the shadow process running on top of the host. That is, some of the guest process's system calls are resolved using the guest while others are forwarded to the shadow process and executed by it. Note that no modifications to the guest kernel, host kernel, or the VMM are made to implement this service.

System call migration is not a new concept. Our point here is to describe how guest process to host shadow process system call forwarding can be implemented straightforwardly at user-level using the dark shadow technique. Perhaps the most relevant related work is that in hybrid kernels such as mOS [20] and IHK/McKernel [18] which combine both lightweight kernel (LWK) functionality and Linux functionality within the same kernel. Linux system calls made in the LWK context are then forwarded to a Linux context for execution. The FlexSC [19] system call model inherently makes the migration of system calls within a single kernel easier. Blue Gene/L nodes run an LWK that shares an address space with applications and forwards system calls to a specialized I/O node [1]. The Pisces Co-

Kernel architecture [14] and XEMEM [10] allow multiple distinct kernels, including VMMs, to run simultaneously with their end-user processes sharing memory regions.

Dark shadow's simplifications: Because the dark shadow mechanism allows the shadow process to precisely mirror the user-level virtual address address space of the guest process and its mapping to physical addresses, the construction of a system call forwarding service is greatly simplified. Consider, for example, forwarding a `write(int fd, void *buf, size_t n)` system call. Executing this forwarded call requests that the host kernel read GVAs `buf` through `buf+n`. Without dark shadow, the `buf` argument will have to be translated to its corresponding HVA in the shadow. Furthermore, if the range of addresses spans a page boundary, and the GVA→HPA mapping is not contiguous across the page boundary, the write will need to be sharded. With the dark shadow mechanism, on the other hand, *the* `write()` *call can simply be executed as is, with no translation.* Another way to think about this is that system call forwarding without the dark shadow mechanism is much like an RPC mechanism—we need what amounts to an RPC stub to do translation and/or copy out, sharding, reassembly, and copy in. With the dark shadow mechanism, in comparison, it is much like any function call—we just need to get control flow to the dark shadow and back.

Where the dark shadow mechanism comes particularly into its own is in dealing with system calls that have opaque arguments, for example the `ioctl()` system call. This call includes on opaque argument which can be a pointer which can in turn point to a pointer-based data structure. The semantics of an `ioctl()`, indeed even of its arguments, depend on the system and the type of object (e.g., file, device, etc) on which `ioctl()` is being invoked. For this reason, building RPC-like stubs for `ioctl()` is extremely challenging and programmer-intensive. In comparison, with the dark shadow mechanism, we can *simply use the opaque arguments verbatim.* If they happen to be GVAs, directly or indirectly, these will just be valid in the shadow, since the shadow's HVAs and HVA→HPA mappings will be identical to the guest's GVA→HPA mappings.

Given how the dark shadow mechanism lets the service developer simply assume that shadow process HVAs and guest process GVAs are identical, the only remaining elements of the system call forwarding service are control flow, as we describe below.

System call interception in the guest process: To intercept system calls from the guest process, we have developed a `LD_PRELOAD` library that comprises ∼500 lines of C. This uses the GCC/ld.so constructor mechanism to force the execution of an initialization function at library load time, which occurs well before even other shared libraries are loaded, and well before the user's `main()` begins. The initialization function creates a child thread, directly using Linux's `clone()` system call for general compatibility. The child thread, called the *monitor*, then uses the kernel's `ptrace` interface to attach itself to the parent. The `ptrace` interface is intended to support the construction of debuggers. We use it to intercept system call events. One event, the *syscall entry*, is sent to the monitor just before a system call starts in the guest kernel, and another, the *syscall exit*, is sent to it just before it returns from the guest kernel. At each of these events, the monitor can modify the system call that the guest kernel sees. Note that a system call comprises the values in seven well-known registers, one indicating the system call number, and the others being up to six arguments to the system call.

The monitor sees all system calls, including invocations of `clone()` that the guest process may make, for example as the underlying mechanism for library functions like `pthread_create()` It detects these and also attaches itself to the newly created threads. Invocations of `clone()` or `fork()` that create separate processes are not followed. In this way, we observe all system calls made by all threads within the guest process.

On a syscall entry for a system call that we want to intercept, the monitor modifies the system call number register to the `getpid()` number, an idempotent no-op. It then forwards the original system call (the contents of the seven registers) to the dark shadow using a transfer mechanism we describe later. It then waits until the transfer mechanism indicates completion of the forwarded system call. At this point, it issues the no-op system call in the guest, and waits for a syscall exit. The syscall exit handler then patches the return value of the no-op guest system call with the completion value of the real forwarded system call, and allows the guest thread to continue. From the guest thread's perspective, the original system call has now returned.

System call execution in the shadow process: The other end of the transfer mechanism resides in the dark shadow capsule. The code here waits for a system call, in the form of a message containing the seven register values that define it, to arrive. When a message arrives, an assembly stub is called that unpacks the register values into their corresponding actual registers, and then issues a `syscall` instruction, which causes the system call to launch on the host kernel. The next instruction, executed after the kernel executes its corresponding `sysret` simply stores the return value (i.e., RAX). The capsule then hands this completion message back to the transfer mechanism.

The system call forwarding service code in the dark shadow capsule comprises ∼160 lines of C and assembly, of which 30 is simply the system call assembly stub. This extreme economy is possible because the dark shadow mechanism creates and maintains a virtual address space and mapping to physical addressees within the dark shadow that is identical to those in the guest process.

Transfer mechanism: To transfer system calls and responses, we leverage an existing Palacios mechanism originally developed for GEARS called the host hypercall interface [8]. This interface allows a hypercall (a call from the guest to the VMM) to be implemented as a separate host kernel module or as a host user space process instead of directly in the VMM. We do the latter. When the dark shadow process starts in host user space, the capsule uses the host hypercall interface to register itself as the implementer of a specific hypercall number on the specific guest. The capsule then iterates a `select()`/`read()` `write()` cycle to wait for a hypercall, fetch its content (the seven registers of the system call), and then write the result (the one register indicating the return value).

The monitor transfers a system call simply by copying the seven registers of the system call to the argument registers defined for a Palacios hypercall, and then invoking it. Both blocking and nonblocking hypercalls are available.

There are ∼100 lines of C and assembly code involved in the transfer mechanism between the capsule and the `LD_PRELOAD` library.

Alternatives: The transfer mechanism could be changed to one using a mechanism like Xen's I/O rings [2] in which memory shared between the guest process and the dark shadow process would be designated for communication. In this way, no hypercalls would be needed, eliminating the latency of going through the VMM for an intercepted system call. The latency would be that of a memory-based synchronization. Since the entire user portion of the guest process's address space is already in the dark shadow process, only a mechanism to agree on an address is needed.

Our design is entirely user-level. If we allow ourselves a guest kernel module, we could avoid the `ptrace` mechanism using the fast selective system call interception module described elsewhere [8]. With this module, there is effectively zero overhead for system call interception. This module could be injected without guest cooperation and even protected from the guest [5]. These mechanisms are already available in the public Palacios codebase.

VI. Service: Device File Virtualization

Device file virtualization is a technique for allowing the guest access to devices for which drivers exist only for the host. The concept was proposed and developed by Sani, et al in the Paradice system [15] and extended by them for mobile computing access to remote devices in the Rio system [16]. Our implementation leverages our dark shadow technique and is influenced by an early design of Paradice, particularly its hybrid address space [17]. Our service differs in that it does not require any guest or host kernel changes. Paradice and its associated systems involve host and guest kernel source code changes and recompilations. We use the dark shadow technique to create a hybrid address space at user-level and implement our system at user-level in the host and guest without any source code changes to either kernel. Our intent here is to demonstrate the utility of the dark shadow technique, not to innovate in device virtualization.

Basic concept: The basic premise of device file virtualization is that for a Unix-like guest OS (e.g., Linux) running on top of a similar Unix-like host OS (e.g., Linux), a device—and its device driver—in the host can be made accessible to a process in the guest via the device file boundary. Consider a device such as an NVIDIA GPU. The device driver in the host creates the device special file `/dev/nvidia0` within the host. A user application, for example one produced using the CUDA toolchain, then interacts with device file to execute code on the GPU. This is done via a CUDA library that is linked with the user's code as part of the CUDA compilation process. The device file is the interface to device driver that resides in the host kernel.

In device file virtualization, this device file is projected via straightforward means[5] into the guest. Now a guest user application linked with the CUDA library can attempt to interact with the projected device file. Each of these interactions is a system call, and each is intercepted and forwarded to the VMM which in turn forwards it to a dark shadow process on the host. This dark shadow process then executes the system call, and the result is returned.

To be clear, the above process is *virtually identical* to the system call forwarding service we described in Section V, and thus can take identical advantage of the fact that the dark shadow mechanism keeps the shadow's virtual and physical user address spaces and mappings identical to that of the guest process. Like the system call forwarding service, the guest-side component is an `LD_PRELOAD` library that intercepts all system calls, filters them, and forwards some to the shadow. The only real difference in the implementation is in how to filter system calls. Here, we intercept `open()`, `stat()`, `close()`, and similar system calls based on the path name (e.g., `/dev/nvidia0`) involved. We also track the file descriptors returned by `open()` calls (and destroyed by corresponding `close()` calls) for the device file path, and then intercept all other system calls that involve these file descriptors. A file descriptor mapping table is also maintained so that we can merge file descriptors supplied by the guest kernel with those supplied by the host kernel without overlap.

mmap()ing the device: The above is sufficient for most devices and/or device drivers that do not support `mmap()`ing of the device file into the user address space. Note that this includes DMA to and from user addresses within the guest process. Recall that the user portion of the dark shadow address space is both virtually and physically identical to the guest process's address space. Therefore, any DMA made by a driver to support a system call made by the shadow process

[5] For example by using a preload library, or by simplifying creating an identical device special file in the guest using standard tools (e.g., *mknod*).

on the guest process's behalf will land in exactly the right "place" in the guest.

However, some devices and drivers *do* support and rely on `mmap()`ing of their device files. For example, the NVIDIA GPU kernel driver supports `mmap()` to map memory that is shared between the driver and the userspace CUDA library. Beyond memory, portions of the hardware device itself (e.g., the portions exposed by the PCI BARs) are mapped into the userspace by the CUDA library, via the device file, so that it can talk directly to the device.[6]

In our service as explained so far, such `mmap()` calls are correctly forwarded from the guest process to the shadow process, and are correctly executed there, but the result is that the correct mappings *exist only within the shadow process*. Hence, when the guest process attempts to access the mapped device, it fails.

To solve this problem, we handle such `mmap()` requests in a special way. When this system call is executed in the dark shadow, the handler there records both its return value (the virtual address in the shadow where the device state is being mapped) and also the physical address it is mapped to—it returns both the HVA and the HPA. The physical address can be determined by examining the shadow process's `/proc/pid/maps` after the `mmap()` has returned. The system call forwarding hypercall now completes, and the virtual and physical addresses of the `mmap()` are returned to the guest process, namely to the device file virtualization preload library. At this point, the mapping is valid and correct within the dark shadow, but not within the guest process.

The preload library now needs to create an equivalent mapping within the guest. However, at this point, the HPA that the shadow supplied does not exist within the guest. The library issues a hypercall that requests access to the the HPA. Palacios validates the request and then attempts the mapping. This uses an existing mechanism in Palacios that most VMMs also have. It maps the HPA into an identical GPA. Recall that this is a device that is being mapped. This mapping will almost certainly succeed because this is a *device*, and it is not yet in the guest physical address space, and thus there is most probably nothing mapped at the destination address yet. After this step, the device is mapped into the guest physical address space at the same address at which it is mapped into the host physical address space.

The preload library now plays exactly the same trick that the dark shadow uses to set up its address space—it issues an `mmap()` (which will not be forwarded) that maps the relevant chunk (that at the HPA/GPA) of `/dev/mem` (the guest physical address space) into the guest virtual address space. Here it uses a `MAP_FIXED` request and supplies the HVA that was returned by the hypercall as the required target address. This `mmap()` must succeed since if the corresponding `mmap()` succeeded in the shadow (and it did), then it cannot be the case that there was some overlapping memory region in the guest. The end result is that the device is now mapped into the guest at precisely the same virtual address where it was mapped in the shadow. At this point reads or writes carried out by library or application code in the guest will correctly be made on the device.

It is important to point out that memory behavioral properties that an `mmap()`ed device might need, for example write combining, apply to HPAs. Whether the property is implemented via MTRRs or PAT on nested or shadow page table entries, this is an aspect of a VMMs mapping of GPAs to HPAs. The nested/shadow page tables take precedence over the guests page tables in this regard, and so the correct behavior results.

Our prototype implementation of this technique comprises ∼800 lines of C for the preload library, and ∼300 lines of C for the service implementation in the dark shadow framework.

Dark shadow's simplifications: It is important to understand that while the above succession of events may seem complicated, they occur at user-level in the guest and host, with the sole exception of the VMM editing the guest physical address space, a mechanism that already exists in all VMMs. This is made possible by the fact that the user portion of the address spaces of the guest and shadow processes can be kept identical, both physically and virtually, via the dark shadow technique. This in turn makes it possible to propagate mappings from one to the other without translation. The typical direction is from guest to host. For `mmap()`ing of devices, the direction is reversed.

VII. Conclusions and future work

We have described a technique for enabling shadow processes in a virtual machine monitor through user-level mechanisms that require no changes to the guest and host kernels. Shadow processes in turn simplify the creation of services such as system call forwarding and device file virtualization. The core aspect of our dark shadow technique is that the service is embedded in a mobile "capsule" which can place itself into the virtual address space of the shadow so that it does not conflict with any virtual address used by the guest process we are shadowing. The dark shadow proof-of-concept implementation can be found in the Palacios codebase, which can be accessed via v3vee.org. A related toolchain, HIO [3], within the Hobbes project can be found in the Hobbes repository at Sandia National Labs. Dark shadow-related techniques, particularly regarding address space mergers between a Linux process and another process, environment, or kernel, are used in our HVM [6] and Multiverse [7] tools that create and exploit VMs that can run multiple kernels simultaneously.

[6] What resides within the driver or the GPU at those addresses and the semantics of accessing it is opaque as far as we understand. Large components of both the CUDA library and the device driver are distributed as blobs.

REFERENCES

[1] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. Castaños, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An overview of the blue gene/l system software organization," in *Proceedings of the Euro-Par Conference on Parallel and Distributed Computing (EuroPar 2003)*, August 2003.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 164–177.

[3] N. Evans, B. Kocoloski, J. Lange, K. Pedretti, S. Mukherjee, R. Brightwell, M. Lang, and P. Bridges, "Hobbes node virtualization layer: System software infrastructure for application composition and performance isolation (poster)," in *Proceedings of the 28th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*, November 2016.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," IBM, Tech. Rep. IBM Research Technical Report RC25482 (AUS1407-001), July 2014.

[5] K. Hale and P. Dinda, "Guarded modules: Adaptively extending the vmm's privileges into the guest," in *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 2014)*, June 2014.

[6] ——, "Enabling hybrid parallel runtimes through kernel and virtualization support," in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, April 2016.

[7] K. Hale, C. Hetland, and P. Dinda, "Multiverse: Split-phase execution of virtualized hybrid runtimes," in *Proceedings of the 25th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2016)*, June 2016.

[8] K. Hale, L. Xia, and P. Dinda, "Shifting GEARS to enable guest-context virtual services," in *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)*, September 2012.

[9] L. Kaplan, "Cray CNL," in *FastOS PI Meeting and Workshop*, June 2007. [Online]. Available: http://www.cs.unm.edu/~fastos/07meeting/CNL_FASTOS.pdf

[10] B. Kocoloski and J. R. Lange, "Xemem: Efficient shared memory for composed applications on multi-os/r exascale systems," in *Proceedings of the 24^{th} International Symposium on High-Performance Parallel and Distributed Computing*, June 2015.

[11] K. Kourai and S. Chiba, "Hyperspector: Virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE 2005)*, June 2005.

[12] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing," in *Proceedings of the 24^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Apr. 2010.

[13] L. Liu and S. Chen, "Malyzer: Defeating anti-detection for application-level malware analysis," in *Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS 2009)*, June 2009.

[14] J. Oayang, B. Kocoloski, J. Lange, and K. Pedretti, "Achieving performance isolation with lightweight co-kernels," in *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)*, June 2015.

[15] A. A. Sani, K. Boos, S. Qin, and L. Zhong, "I/o paravirtualization at the device file boundary," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, March 2014.

[16] A. A. Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*, June 2014.

[17] A. A. Sani, S. Nair, L. Zhong, and Q. Jacobson, "Making i/o virtualization easy with device files," Rice University, Tech. Rep. 2013-04-13, 2013.

[18] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa, "Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures," in *Proceedings of the IEEE International Conference on High Performance Computing (HiPC 2014)*, December 2014.

[19] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[20] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An architecture for extreme-scale operating systems," in *Proceedings of the 4^{th} International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2014)*, June 2014.

[21] V. C. Zandy, B. P. Miller, and M. Livny, "Process hijacking," in *Proceedings of the 8th International Symposium on High Performance Distributed Computing (HPDC 1999)*, June 1999.