A Binary-Compatible Unikernel

Pierre Olivier
Virginia Tech, USA
polivier@vt.edu

Daniel Chiba*
Qualcomm Technologies Inc., USA
danchiba@vt.edu

Stefan Lankes
RWTH Aachen University, Germany
slankes@eonerc.rwth-aachen.de

Changwoo Min
Virginia Tech, USA
changwoo@vt.edu

Binoy Ravindran
Virginia Tech, USA
binoy@vt.edu

Abstract

Unikernels are minimal single-purpose virtual machines. They are highly popular in the research domain due to the benefits they provide. A barrier to their widespread adoption is the difficulty/impossibility to port existing applications to current unikernels. *HermiTux* is the first unikernel providing binary-compatibility with Linux applications. It is composed of a hypervisor and lightweight kernel layer emulating OS interfaces at load- and runtime in accordance with the Linux ABI. HermiTux relieves application developers from the burden of porting software, while providing unikernel benefits such as security through hardware-assisted virtualized isolation, swift boot time, and low disk/memory footprint. Fast system calls and kernel modularity are enabled through binary rewriting and analysis techniques, as well as shared library substitution. Compared to other unikernels, HermiTux boots faster and has a lower memory/disk footprint. We demonstrate that over a range of native C/C++/Fortran/Python Linux applications, HermiTux performs similarly to Linux in most cases: its performance overhead averages 3% in memory- and compute-bound scenarios.

CCS Concepts • **Software and its engineering → Virtual machines**; **Operating systems**.

Keywords Unikernels, Linux Kernel, Binary Compatibility. Virtualization, Operating Systems

ACM Reference Format:
Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3313808.3313817

*This work was done while Daniel Chiba was at Virginia Tech.

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313817

1 Introduction

Unikernels have become popular in academic research, in the form of a virtualized LibOS model bringing numerous benefits: increased security, performance improvements, isolation, cost reduction, ease of deployment, etc. Their potential application domains are plentiful: cloud- and edge-deployed micro-services/SaaS/FaaS-based software [8, 29, 30, 41, 63], server applications [30, 39, 40, 63, 80], NFV [14, 40–42], IoT [14, 16], HPC [31], efficient VM introspection/malware analysis [79], and regular desktop applications [56, 68]. While they are presented as a secure and attractive alternative to containers [41, 55, 78], unikernels still struggle to gain significant traction in industry and their adoption rate is quite slow [41, 71]. One of the major reasons is the difficulty, and sometimes impossibility, of porting legacy/existing applications to current unikernel models [7, 21, 40, 41, 51, 59, 61, 71].

In situations such as the use of compiled proprietary code, the unavailability of an application's sources makes it impossible for a user to port and run it using any of the existing unikernel models. Such binaries are generally stripped and obfuscated, thus disassembling and re-linking with a unikernel layer is not suitable. Even when sources are available, considering unikernel models supporting legacy programming languages (C/C++) [8, 24, 28, 31, 42], porting a medium/large-sized or complex codebase can still be difficult [7, 21, 40, 41, 51, 59, 61, 71]. This is due to factors such as incompatible/missing libraries/features, complex build infrastructures, lack of developer tools (debuggers/profilers), and unsupported languages. Porting complexity is further increased as that process requires expertise in both the application and the considered unikernel model [71]. Because it is currently the burden of the application programmer [59], we believe that this significant porting effort is one of the biggest roadblocks preventing wide-spread adoption of unikernels.

The solution we propose is a unikernel that offers binary compatibility for regular (i.e. Linux) applications, while keeping classical unikernel benefits. It allows the development effort to be focused on the unikernel layer. In this context, we present a prototype named HermiTux, an extension of the

HermitCore [31] unikernel, which is able to run *native* (no recompilation/relinking) Linux executables as unikernels. By providing this infrastructure, HermiTux *transforms the porting effort from the application programmer into a supporting effort from the unikernel layer developer*. In this model, not only can unikernel benefits be obtained transparently for native Linux applications, but furthermore it is now possible to run previously un-portable applications such as proprietary software. With HermiTux, the effort to port and run a legacy application as a unikernel is non-existent, even when its sources are unavailable. HermiTux supports statically and dynamically linked executables, is compatible with multiple languages (C/C++/Fortran/Python), compilers (GCC and LLVM), full optimizations (`-O3`), and stripped/obfuscated binaries. It supports multithreading and Symmetric Multi-Processors (SMP), checkpoint/restart and migration. Finally, HermiTux offers intuitive debugging and profiling tools. We demonstrate HermiTux on a set of native Linux applications on the x86-64 architecture. Their performance running in HermiTux is mostly similar to a Linux execution.

The first challenge HermiTux tackles is *how to provide binary compatibility?* To that end, HermiTux sets up the execution environment and emulates OS interfaces at runtime in accordance with Linux's Application Binary Interface (ABI). A custom hypervisor-based ELF loader is used to run a Linux binary alongside a minimal kernel in a single address space Virtual Machine (VM). System calls made by the program are redirected to the implementations the unikernel provides. A second challenge HermiTux faces is *how to maintain unikernel benefits while providing such binary compatibility?* Some come naturally (small disk/memory footprints, virtualization-enforced isolation), while others (fast system calls and kernel modularity) pose technical challenges when assuming no access to sources. To enable such benefits, HermiTux uses binary rewriting and analysis techniques for static executables, and substitutes at runtime a unikernel-aware C library for dynamically linked executables. Finally, HermiTux is optimized for low disk/memory footprint and attack surface, which are as low as or lower than existing unikernel models.

The contributions presented in this paper are: (1) a new unikernel model designed to execute native Linux executables while maintaining the classical unikernel benefits; (2) a prototype implementation of that model; (3) an evaluation of this prototype comparing its performance to Linux, containers, and other unikernel models (OSv [28] and Rump [24]).

In this paper, we give some background and motivation in Section 2. In Section 3, we present the design of HermiTux, then give implementation details in Section 4. A performance evaluation is presented in Section 5. We present related works in Section 6, before concluding in Section 7.

2 Background and Motivation

Unikernels & Applications Port. A unikernel [40] is an application statically compiled with the necessary libraries and a thin OS layer into a binary able to be executed as a virtualized guest on top of a hypervisor. Unikernels are qualified as: (A) *single purpose*: a unikernel contains only one application; and (B) *single address space*: because of (A), there is no need for memory protection within the unikernel, consequently the application and the kernel share a single address space and all the code executes with the highest privilege level.

Such a model provides significant benefits. In terms of security, the strong isolation between unikernels provided by the hypervisor makes them good candidates for cloud deployments. Moreover, a unikernel contains only the necessary software needed to run a given application. Combined with the very small size of the kernel, this leads to a significant reduction in the application attack surface compared to regular VMs [39]. Some unikernels are also written in languages providing memory-safety guarantees [9, 40, 74]. Concerning performance, unikernel system calls are fast because they are common function calls: there is no costly world switch between privilege levels [31]. Context switches are also swift as there is no page table switch or TLB flush. In addition to the codebase reduction due to small kernels, unikernel OS layers are generally modular: it is possible to configure them to include only the necessary features for a given application. Small size and modularity lead to a reduction in resource usage (RAM, disk), which translates into cost reduction for the cloud user, and high per-host VM density for the cloud provider [37, 41].

Porting existing software to run as a unikernel in order to reap these benefits can be difficult or even impossible. First, in some situations, the unavailability of an application's sources (proprietary software) makes porting it to any existing unikernel impossible, as all require recompilation/relinking. Second, porting legacy software to a unikernel that supports only modern programming languages requires a full application rewrite in that target language [40], which in many scenarios is unacceptable. Third, considering the unikernels supporting legacy languages, the task still represents a significant challenge [7, 21, 41, 51, 59, 61, 71] for multiple reasons. A given unikernel supports a limited set of kernel features and software libraries. If a feature, library, or a particular version of a library required for an application is not supported, the application would need to be adapted [7, 51]. In many cases the lack of a feature/library means that the application cannot be ported at all [50]. Moreover, unikernels use complex build infrastructures and it can be burdensome to port the large build infrastructures of some legacy applications (large/generated Makefiles, autotools/cmake environments) to unikernel toolchains. The same goes for changing the compiler or build options. In

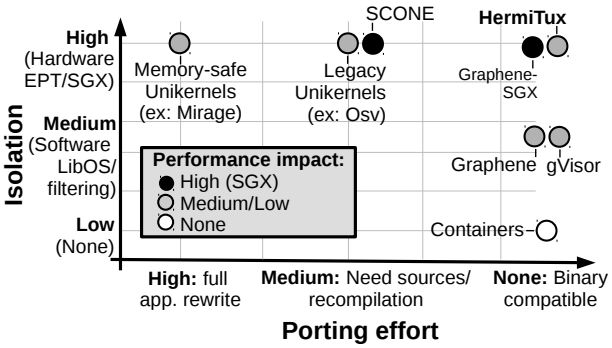

Figure 1. Lightweight Virtualization Design Space.

addition, unikernels lack practical developer tools such as debuggers and profilers [61], further complicating the porting process. The porting effort is made even more complex by the fact that it necessitates expertise in both the application and the considered unikernel model [71].

We believe that this large porting cost, combined with the fact that it is the responsibility of the application programmer, represents an important factor explaining the slow adoption of unikernels in the industry. One solution is to have a unikernel provide binary compatibility for regular executables while still keeping the classical unikernel benefits such as small codebase/footprint, fast boot times, modularity, etc. This new model allows unikernel developers to work on generalizing the unikernel layer to support a maximum number of applications, and relieves application developers from any porting effort. Such an approach should also support developer tools such as debuggers. In that context, HermiTux allows running Linux binaries as unikernels, while maintaining the aforementioned benefits.

HermiTux in the Lightweight Virtualization Design Space. Using Figure 1, we demonstrate that HermiTux occupies a unique location in the lightweight virtualization design space. In that space we include unikernels, security-oriented LibOS such as Graphene [68, 70], and containers [45] with software [19] and hardware [2] hardening techniques. In addition to the points mentioned in the previous paragraph concerning porting effort, HermiTux differs from other binary-compatible systems because of the combination of 2 reasons. First, hardware-enforced isolation such as the use of SGX or EPT is fundamentally stronger than software-enforced isolation (containers/software LibOS), as shown by the current trend of running containers within VMs for security (clear containers [13]) and the efforts to strengthen containers isolation (such as gVisor [19]). This is generally used as a security argument in favor of unikernels versus containers [54]. Concrete examples include ret2usr [27], ret2dir [26], and Meltdown [36] attacks, which containers are subject to but not unikernels due to the guest running in a completely separate address space from the host. The minimal nature of unikernels allows them to have a reduced attack surface and to avoid vulnerabilities present in management systems software such as shells (Shellshock), absent from unikernels but included in the TCB of most containers. Second, SGX-based isolation such as used in Graphene-SGX [70] has a non-negligible performance impact [81] that is fundamentally higher than the very low performance overhead of current direct-execution, hardware-enforced virtualization techniques leveraged in HermiTux.

HermiTux enables a wide range of applications to *transparently* reap unikernel benefits without any porting effort: there is no need for code modification and the potential complexities of maintaining a separate branch. Given the security and footprint reduction features provided by unikernels, this is highly valuable in today's computer systems landscape where software and hardware vulnerabilities regularly make the news, and where datacenter architects are seeking ways to increase consolidation and reduce resource/energy consumption. Being binary compatible allows HermiTux to be the only way to run proprietary software (whose sources are not available) as unikernels. Finally, HermiTux allows commodity software to reap traditional benefits of VMs such as checkpoint/restart or migration without the associated overhead of a large disk/memory footprint.

The HermitCore Unikernel. In this work we significantly extend the unikernel implementation of HermitCore [31, 32]. Written in C, its sources are statically linked with application code to create a unikernel binary. In HermitCore, while most system calls are processed by the kernel, file-system related ones (`open`, `read`, etc.) are forwarded to the host and use its file-system for simplicity reasons. HermitCore includes a C standard library (Newlib [48]), libraries for multi-threading (Pthreads Embedded [57]) and TCP/IP (LWiP [38]) support. Device drivers for typical network interfaces are also included. Uhyve (Unikernel Hypervisor) is a minimal hypervisor developed by HermitCore's authors, designed to run that unikernel on a Linux host, using the KVM interface [67]. It is an adaptation of the *Ukvm* hypervisor [64, 76], tailored for unikernels. Motivations for its development are the reduction of the attack surface and boot times compared to complex hypervisors such as QEMU/KVM. Uhyve's fast initialization, combined with HermitCore's swift boot time, lead to initialization times of tens of milliseconds.

Because of its advantages in terms of low complexity/attack surface/boot time, we chose the HermitCore/Uhyve couple as a basis for HermiTux. Note that all of HermiTux design principles, and most of its implementation techniques, are easily applicable to other unikernels.

3 System Design

We assume that the sources of the binaries we consider are unavailable. We make no assumption about the compiler used, the level of optimizations, or whether the binary is

stripped or obfuscated. Thus, disassembling and reassembling, generally considered quite unreliable [72, 73], is not a suitable solution. We rather decide to offer binary compatibility with a commodity OS, Linux.

The Linux ABI. To offer binary compatibility, HermiTux's kernel needs to comply with the set of rules constituting the Linux ABI [43]. These rules can be broadly classified into load-time and runtime rules. Load-time rules include the binary format supported (ELF), which area of the 64 bit address space is accessible to the application, the method of setting up the address space by loading segments from the binary, and the specific register state and stack layout (command line arguments, environment variables, ELF auxiliary vector) expected at the application entry point. Runtime rules include the instruction used to trigger a system call, and the registers containing its arguments and return value. Finally, Linux applications also expect to communicate with the OS by reading and writing various virtual filesystems (`/proc`, `/sys`, etc.) [69] as well as through a memory area shared with the kernel: the vDSO/vsyscall.

System Overview. HermiTux's design objective is to *emulate the Linux ABI at load- and runtime while providing unikernel principles*. Load time conventions are ensured through the use of a custom ELF loader. Runtime convention are followed by implementing a Linux-like system call handler in HermiTux's kernel. vDSO/vsyscall and virtual filesystems access are emulated with methods described further. Finally, HermiTux maintains some unikernel benefits (fast system calls and modularity) without assuming access to the application sources using binary rewriting and analysis techniques for static executables, and substitution to a unikernel-aware shared C library at runtime for dynamically compiled programs.

Figure 2 presents a high-level view of the system. At launch time, the hypervisor allocates memory for the guest to use as its physical memory. Next, the hypervisor loads the HermiTux kernel (Ⓐ on Figure 2) at a specific location in that area, and then proceeds to load the loadable ELF segments from the Linux binary at the locations indicated in the ELF metadata. After that loading process, control is passed to the guest and the kernel initializes. Page tables are set up in order to build a single address space containing both the kernel and the application. Following the unikernel principle, the kernel and the application both execute in protection ring 0.

After initialization, the kernel allocates and initializes a stack for the application following the Linux loading convention, then jumps to the executable entry point Ⓑ whose address was read at load time from the ELF metadata. During application execution, system calls will be executed according to the Linux convention, using the `syscall` x86-64 instruction. The kernel catches such invocations by implementing a system call handler that identifies the system call

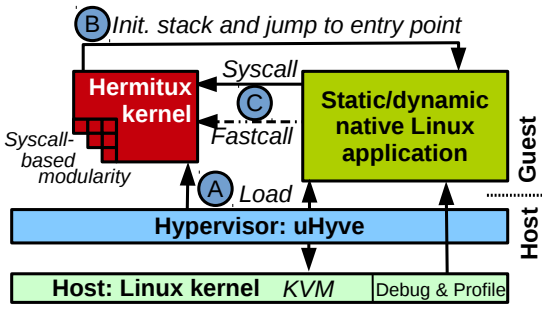

Figure 2. Overview of the HermiTux system.

invoked, determines parameters from CPU registers, and invokes the HermiTux implementation of the considered system call Ⓒ.

Load-Time Binary Compatibility. At load time, the hypervisor copies the kernel and Linux application loadable ELF segments in guest memory at the virtual addresses indicated in ELF metadata. Both application and kernel will run in a single address space so we need to ensure that statically and dynamically allocated code/data for both entities do not overlap. This is done by locating the kernel outside of the memory region dedicated for application. Contrary to Linux, which dedicates the upper half of the 48 bit virtual address space to the kernel, and because of the very small virtual/physical memory requirements of HermiTux's kernel, we can locate it *below* the area reserved for the application. This gives the application access to the major part of the virtual address space and has the interesting side-effect of enabling very high entropy for randomized mappings.

HermiTux supports dynamically compiled binaries as follows: when the loader detects such a binary, it loads and passes control to a dynamic loader that in turns loads the application as well as the library dependencies, and takes care of the symbols' relocations. Because of its binary compatibility, in HermiTux the dynamic loader is an unmodified version of a regular Linux dynamic loader. We assume it comes shipped with the application binary and its shared library dependencies. Contrary to KylinX [80], in HermiTux we take the decision not to share in the virtual address space the dynamic libraries between multiple unikernels. This is mostly for security reasons as shared memory between VMs makes them potentially vulnerable to side-channels [20] (for example Flush+Reload or Prime+Probe). Moreover, if security is less of a concern and memory usage is constrained, Kernel Same page Merging (KSM) [1] is an efficient and standard way to solve that issue. While dynamic binaries are favored by Linux distribution maintainers [69], static executables still provide benefits in terms of performance [15] and compatibility [10]. Thus we aim to support both linking types.

Runtime Binary Compatibility. In a unikernel, system calls are common function calls. In HermiTux, the application performs system calls using the Linux convention, unsupported by existing unikernels kernel. HermiTux implements a system call handler invoked when the `syscall` instruction is executed. It redirects the execution to the internal unikernel implementation of the invoked system call (Ⓒ on Figure 2). Interfacing with the application at the system call level is at the core of the binary compatibility provided by HermiTux. It means that our prototype, currently tested on software written in C/C++/Fortran/Python, can easily be extended to other languages and runtimes.

Vanilla HermitCore supports a small number of system calls, and we had to significantly extend this interface in HermiTux. In a unikernel context, developing system call support for unmodified Linux applications might raise concerns in terms of codebase size and complexity increase. It could also be, intuitively, a very large engineering effort to end up re-implementing Linux. Yet for our work this does not represent a full re-implementation of the Linux system call interface: while it is relatively large (more than 350 system calls), applications generally use only a small subset of that interface [58]. It has also be shown that one can support 90% of a standard distribution binaries by implementing as few as 200 system calls [69]. To support the applications presented in the evaluation section (Section 5), our prototype implements 83 system calls.

Unikernel Benefits & Isolation. System call latency in unikernels is low as these are function calls. Despite a system call handler optimized for the unikernel context, we observed that in HermiTux this latency still does not approach that of function calls: it is due to the `syscall` instruction that relies on an exception.

Without assuming access to the application sources, we rely on two techniques to offer fast system calls in HermiTux (*Fastcall* in Figure 2). For static binaries, we use binary instrumentation to rewrite the `syscall` instructions found in the application with regular function calls to the corresponding unikernel implementations. For dynamically linked programs, we observe that (1) most of the system calls are made by the C library and (2) the C library interface is very well defined so it is possible to link at runtime a program with a shared C library that is not the one the program was compiled against. We then use for dynamically linked programs a unikernel-aware C library we designed, which is making direct function calls to the kernel in the place of system calls. We call this technique library substitution. In the spirit of HermiTux, the development effort outside of the unikernel kernel/hypervisor should be nonexistent or at worst minimal. A popular C library (Musl LibC) is made unikernel-aware in a fully automated way with the help of code transformation tools.

Modularization is another important benefit of unikernels. Because of our binary compatibility goal, the system call codebase is relatively large in HermiTux. We design the kernel so that the implementation of each system call can be compiled in or out at kernel build time. In addition to memory footprint reduction, this has security benefits that are stronger than traditional system call filtering (such as seccomp [11]): it is not only impossible to call the concerned system calls, but the fact that their implementation is entirely absent from the kernel means that they cannot be used in code reused attacks. To compile a tailored kernel for a given application whose sources are not necessarily available, we designed a binary analysis tool able to scan an executable and detect the various system calls that can be made by that program.

HermitCore forwarding filesystem calls to the host raises obvious concerns about security/isolation. We implemented a basic RAM filesystem, *MiniFS*, within HermiTux's kernel, disabling any dependence on the host in that regard. Building a full-fledged filesystem is out of the scope of this work, however MiniFS's simple implementation is sufficient to run with honorable performance the benchmarks used in our performance evaluation (see Section 5), including Postmark. MiniFS also emulates pseudo files with configurable per-file read and write functions: we emulate `/dev/zero` with a custom read function filling the user buffer with zeros, `/dev/cpuinfo` with a read function populating the buffer with Linux-like textual information about the CPU, etc.

4 Implementation

Additions and modifications to the HermitCore [31] infrastructure in order to build HermiTux represent a total of 10675 lines of code. Binary system call rewriting and identification tools represent 1268 additional LoC.

HermiTux currently only supports the Intel x86-64 architecture. However its design principles also apply to other ISAs. Although a lot of implementation work is architecture specific (installation of a system call handler, system call rewriting and identification, etc.), there is no fundamental roadblock to adapt HermiTux to other ISAs.

4.1 Linux Binary Loader

Virtual Address Space Setup. The Linux ABI dictates that the region of the address space allowed for access by user space code ranges from `0x400000` to `0x7FFFFFFFFFFF` [17, 35]. While `0x400000` is arbitrarily set as the lower user level accessible area by default linker scripts, the virtual memory below that address is actually free to use. We locate the kernel below at `0x200000`, as the small HermiTux kernel fits in these 2 MB. The original HermitCore kernel was previously located at `0x800000`, and we relocated it by changing the kernel linker script and updating hardcoded references in the kernel code. The address space of a running HermiTux

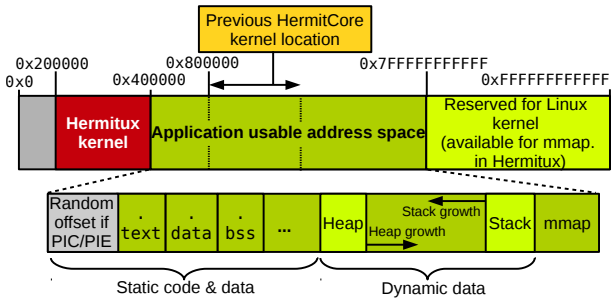

Figure 3. Virtual address space layout of HermiTux.

unikernel is illustrated in Figure 3. As one can observe, the application now has access to close to the entire 48 bit address space. This gives us the unique opportunity of obtaining randomized mappings of very high entropy (higher than Linux) for shared libraries and PIE programs.

Loading Process and Initializing Kernel. At load time, Uhyve initializes and allocates memory for the guest. It then loads the kernel by reading the kernel executable file and copying each loadable ELF section at the right offset in guest memory. The native Linux executable is also loaded following the same principles (see Ⓐ on Figure 2). For security reasons the application is loaded at a random offset if it supports PIC/PIE. Its entry point address and loaded section sizes are written in a specific location in guest memory to be retrieved by the kernel when it boots.

Next, control is passed to the guest and the kernel initializes. During this process, the kernel memory subsystem prepares a stack and a heap for the application in the corresponding region of the address space. The kernel uses information it got from the hypervisor about the location and sizes of the application's loadable ELF sections (`.text`, `.data`, etc.) in order to set the location of the stack and heap not to overlap with these (see Figure 3). The initialization kernel stack is located in the kernel memory region. After the application starts running, it shares its stack with the kernel as in unikernels system calls are common function calls. The kernel's dynamically allocated memory, including heap and ISR stacks, is located in the kernel memory region.

After kernel initialization, a task is spawned to execute application code. Before jumping to the application entry point, the application stack must be filled with specific data for the application to be able to initialize (Ⓑ in Figure 2). Indeed, the entry point of an application generally corresponds to low-level library initialization. This bootstrap code accesses the stack to recover important information concerning the application command line arguments (`argc` and `argv`), environment variables, and ELF auxiliary vectors. The kernel pushes these elements on the stack in reverse order: auxiliary vectors, environment variables, `argv`, then `argc`. The default behavior is to have the values of these elements forwarded from the host, but it is also possible to edit/filter them, for example to avoid leaking sensitive information through environment variables. The Linux vDSO shared object purpose is pointless in a unikernel as the application and kernel share a single unprotected address space. The absence of the vDSO is indicated to the C library on the stack through the ELF auxiliary vectors.

If the binary is dynamically compiled, a dynamic loader is first loaded and run by HermiTux's kernel. Because of our binary compatibility the dynamic loader comes unmodified from a standard Linux distribution (for example `ld-linux.so` for GLibC). The kernel adds the application binary name within `argv` and the dynamic loader proceeds to load this binary and its shared library dependencies, and performs the relocations at runtime. On the kernel side this requires comprehensive support for the `mmap` system call.

4.2 Handling Linux System Calls

System Call Handler. We implemented system call support according to the Linux convention [43] on x86-64 [22]: they are made using the `syscall` instruction. Before its invocation, a unique number identifying this system call is loaded in the `%rax` register, and the system call parameters values are set in order in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.

The HermiTux system call handler is simple and optimized: according to the unikernel principle, the application lives in protection ring 0 in an address space shared with the kernel. Thus, many classical 'world switch' OS operations are not necessary in our case (for example segment selector or stack switches). The handler is implemented using a combination of assembly and C. The assembly code saves all the registers on the stack and calls a C function which retrieves these values, and based on the value in `%rax` redirects control to the corresponding system call implementation within the kernel. On return, the assembly function restores all the registers and sets the return value in `%rax`. A regular OS would then call the `sysret` instruction. On the contrary, we perform a simple jump to the address in `%rcx`, which was set by the CPU to the next application instruction upon `syscall` invocation. This results in an optimization as a jump is significantly faster than `sysret`. Moreover, `sysret` automatically switches the privilege level to ring 3 [22], and as a unikernel we need to stay in ring 0.

System Calls Development. Vanilla HermitCore has support for a very limited number of system calls (16), and thus applications. In HermiTux we had to implement support for many additional system calls. Currently we support a total of 83 system calls, 67 of these being developed on top of the original HermitCore kernel. Some of them are partially supported: for example, `ioctl` only supports the necessary commands for LibC initialization. The total number of LoC dedicated to the system call layer is 3265. it shows that HermiTux can keep a small unikernel codebase while supporting

a wide range of applications as presented in the performance evaluation section.

Basic signal support is provided by HermiTux through the `signal`, `rt_sigaction` and `kill` system calls. The `errno` global variable is transparently supported by HermiTux: the C library loads for each thread the address of the Thread Local Storage (TLS) memory region in the `%fs` register using the `arch_prctl` system call. In TLS resides a thread descriptor containing, amongst other things, `errno`. Multithreading (pthreads) is provided by implementing `clone` support with the `CLONE_VM` flag, and basic support for `futex` used as the central locking/synchronization primitive in numerous modern pthread implementations. As mentioned earlier the vDSO is disabled in HermiTux. Although it is considered insecure, we noticed some programs called *vsyscall* [12] without checking its presence or absence in the ELF auxiliary vectors. Our solution in HermiTux consists in setting up page tables so that accesses to the vsyscall page will always page fault. In the page fault handler, we redirect the control flow on return from the exception to the corresponding system call which can be identified according to the faulting address (which is by convention hardcoded). `set/getpriority` are supported by mapping the Linux *nice* values (-20 to 20) to HermiTux priorities (0 to 31) with an inverse proportional relationship.

The non-fixed mappings offered by our `mmap` implementation are fully randomized. It means that for PIC/PIE programs we are able to offer ASLR features for program and library code/data segments. An interesting benefit of unikernels here is that because the application and kernel share and can use the entire 48 bit virtual address space, we can offer a very high entropy for these randomized mappings: it is currently of 34 bits for HermiTux which is higher than vanilla Linux (28 bits) as well as PaX/grsecurity hardened kernels (33 bits) [46].

The number of system calls supported by HermiTux can be seen as relatively low compared to the full interface – more than 400 system calls for a modern Linux kernel. However it is sufficient to support the wide range of applications presented in the evaluation section. Moreover, implementing more syscalls in HermiTux is principally a matter of engineering. Existing legacy unikernels [24, 28] have shown that it is possible for a unikernel to support a large number of applications while maintaining a small codebase/footprint.

Like the majority of other unikernels, HermiTux does not support creating a new process through `fork()`. It is true that support in unikernels of multi-process applications is an important roadblock to their widespread adoption. However, a solution have already been proposed in KylinX [80], where new unikernels are spawned upon `fork` calls and inter-unikernel IPCs are implemented. Such a solution can also be adapted within HermiTux, which we scope out as future work.

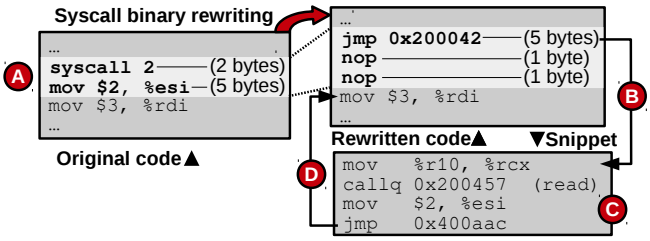

Figure 4. System call rewriting example.

Fast System Calls. Even if the HermiTux system call handler is optimized, its execution is still slower than a regular unikernel's system call latency (function call) because of the underlying mechanism behind the `syscall` instruction: an exception, introducing a significant delay.

HermiTux offers two techniques to decrease such latency, one for each type of binaries (static/dynamic) considered. Both techniques assume no access to the application sources.

For statically linked executables we use static binary rewriting: the idea is to replace in the code each `syscall` instruction with a direct call to the corresponding system call implementation in the kernel.

However we cannot directly replace the `syscall` instruction with a `call` because of their sizes: `syscall` (2 bytes long) is smaller than `call` (5 bytes). The rewriting process is illustrated on Figure 4 and works as follows: we overwrite each `syscall` along with one or more instructions after it (Ⓐ on Figure 4) with a `jmp` instruction Ⓑ, jumping to a snippet of assembly code Ⓒ that performs the following operations: (1) copy `%r10` into `%rcx`: this is necessary due to the difference in calling conventions between system calls and function calls [43]; (2) Call the implementation of the given system call within HermiTux. This is possible thanks to our ability to identify which system call is being made at a given address (see Section 4.3); (3) On return from the system call function, the instructions that were overwritten with `jmp` are executed (these are copied within the snippet during the rewriting process); (4) Return control to the address in the application code Ⓓ immediately following these instructions. Note that this technique completely bypasses the system call handler, further reducing latency.

A small number of system calls invocations cannot be overwritten. For example, if the `syscall` instruction is near the end of a function $f1$ and there is no space to insert the `jmp` instruction before the start of the next function $f2$: doing so would break any function pointer set to $f2$ at runtime. We make use of Dyninst [75] to identify such scenarios, and as a result, this optimization only works for non-stripped binaries.

For dynamically linked binaries, low-latency system calls are obtained by forcing the dynamic loader to link at runtime a custom C library we developed, in which systems calls are common function calls to the kernel. As the spirit of

HermiTux is to avoid any effort outside of unikernel kernel and hypervisor development, this C library is created by automatically replacing all system calls made by a mainstream C library, Musl, by function calls to HermiTux's kernel. This is done in a fully automated way: in the C library most system calls are made through standardized macros. We use the Coccinelle [34, 53] code transformation tool to replace each macro invocation with a call to a wrapper generated within the C library. This wrapper takes as parameters the system call number and parameters, and simply uses an array of function pointers, indexed by system call numbers, to invoke the system call implementation within the kernel. The code initializing that array is generated after kernel compilation when the address of each syscall implementation is known. The use of Coccinelle gives a very robust, future-proof and comprehensive method to create a unikernel-aware C library in a fully automated way: with a simple semantic patch of less than 80 lines describing the code transformations needed, we are able to update 97.5% of the 500+ system call invocations within the entire library. We also confirmed the success of this method over different versions of Musl released multiple years apart. The same technique can be adapted to other C Libraries such as GLibC.

4.3 Other Unikernel Benefits & MiniFS

Kernel Size Reduction, System Call-based Modularity. In addition to fast system calls, unikernel benefits include small codebase and memory/disk footprint, as well as the ability to customize a kernel to contain only what is needed by an application (modularity). HermiTux targets support for native Linux binaries while still providing such benefits. We significantly reduced the size and footprint of the kernel through the combination two straightforward techniques. First, we added support in the Uhyve hypervisor to load a kernel binary compressed using `gzip`. The kernel size can further be reduced through the use of the `strip` utility.

Vanilla HermitCore is not modular. We modularized HermiTux to include or exclude at kernel compile time coarse grain functionalities such as network support. Moreover, as we implemented support for new system calls for binary compatibility, we realized that the system calls codebase was growing to reach a relatively large portion of the total kernel code: it currently represents about 20% of the number of LoC in the kernel (excluding LWIP), and is expected to grow larger as new system calls are implemented to support more applications. Thus, we added the possibility to compile a kernel containing only the implementation of selected system calls, i.e. only the ones needed by a given application. This is achieved by placing each system call implementation into its own compile unit (C source file) and using preprocessor macros to enable or disable system calls invocation from the system call handler. At link time, the base kernel object files are linked with the selected system calls' object files.

To tailor a HermiTux kernel for a particular application, we need to be able to identify which system calls can be made by the application, for which we assume only the binary is available (no sources). One option would include running the application under the `strace` utility. However, this solution is highly dependent on the code coverage of the sample test runs. For HermiTux, we need to be able to identify all of the system calls being made by an application, since excluding the wrong one could lead to a crash at runtime.

For static executables we choose to statically analyze the binary and find, for each `syscall` invocation, what is at that point the value in the `%rax` register (containing the system call identifier). Generally it is loaded with an immediate value right before the `syscall` instruction. However the value may come from another register or follows a more complicated path prior to the `syscall` instruction. We use Dyninst [75] to analyze the control flow of the program starting from the `syscall` instruction and going backward until we find the value we are looking for. This technique works well for applications compiled against the Musl C library: we were able to determine all system calls being made in a large set of applications. For GLibC, we found one call site where the value that was loaded in `%rax` came from memory, making it impossible to identify statically. Looking at the corresponding C code allows to easily determine that it was in fact a `read` system call. To tackle such scenarios, we created a lookup table that returns the system calls being made by library functions that contain such statically unidentifiable system calls. These cases are extremely rare (just one for all applications we tested) and requires minimal human effort (a quick source code inspection). Moreover, the source code within the GLibC behind them does not change very often. This system call identification is combined with our tailored kernel building technique in a fully automated process. Concerning dynamically compiled programs, it is possible to identify the system calls needed by first analyzing which functions from library dependencies are called and then running our system call identification tool on these functions within the libraries.

These lightening techniques are efficient: a stripped and compressed HermiTux kernel without network support, with full system call support (minus the network-related ones) has an on-disk size of 57 KB (versus 2.9 MB for the full version).

MiniFS. We implemented MiniFS within HermiTux's kernel. It adheres to the standard UNIX filesystem interface, supporting the classical functions `open`, `read`, `write`, etc. As a RAM filesystem, files data and metadata are allocated in memory mapped areas. A trusted system administrator can specify at load time some files to be imported from the host (for example data files to be processed by the application or shared library dependencies to be loaded at runtime). A small number of pseudo files are emulated by MiniFS: `/dev/{null|zero|random|urandom}` and simple versions

of `/proc/{cpuinfo|meminfo}`. This is sufficient to run the programs presented in the evaluation, and MiniFS pseudo file support can easily be extended by writing additional custom per-pseudo-file read/write functions.

Developer Tools. HermiTux supports GDB debugging. We leveraged code from Ukvm [64] to implement a GDB remote *stub* [65] in Uhyve. A HermiTux application and kernel can be debugged transparently from a GDB client in the host. The code was adapted for HermiTux in multiple points. In particular, the stub needs to access guest memory from the hypervisor, we developed a specific guest to host address translation method: contrary to solo5/ukvm, HermiTux/Uhyve guest virtual address space is not directly mapped. We also implemented guest and host code to trap to GDB in the case of an unrecoverable exception/page fault within the guest. Regular debugging features are supported such as breakpoints, single-step execution, memory/register inspection, etc. We also implemented a simple profiler in HermiTux: with a configurable frequency, Uhyve injects an interrupt within the guest, and the corresponding handler samples the guest instruction pointer `%rip`. Samples are gathered by Uhyve at the end of the execution and written to a file on the host. We developed an analysis tool taking this file as input, reading the DWARF debugging metadata in both the kernel and the application, and outputting the most time-consuming functions/lines of code.

5 Evaluation

The objective of the performance evaluation is to answer the following questions: First, *can HermiTux run native Linux binaries while still keeping the benefits of unikernels such as low disk/memory footprints and fast boot time?* (Section 5.1). Second, as we focus on native/legacy executables, *can HermiTux execute binaries that are written in different languages, stripped, obfuscated, compiled with full optimizations and different compilers/libraries?* (Section 5.2). Finally, *how does HermiTux's performance compare with Linux, containers and other unikernel models?* (Section 5.3);

We evaluate the performance of HermiTux over multiple macro-/micro-benchmarks, comparing it to a Linux VM, Docker [45], and two unikernels models focusing on compatibility with existing applications: *OSv* [28] and *Rumprun* [24]. Please note that on the contrary to HermiTux, both are not binary compatible with Linux. Macro-benchmarks include C/Fortran/C++/Python NPB [3, 62], PARSEC [6], Python Performance Benchmark Suite [66] and Postmark [25] benchmarks. Micro-benchmarks include redis-benchmark, a SQLite unikernel, and LMbench [44] to measure system call latency.

Experiments were run on a server with an Intel Xeon E5-2637 (3.0 Ghz, 64 GB RAM), running Ubuntu Server 16.04 with Linux v4.4.0 as the host. OSv and Rumprun unikernels versions are the latest available on their respective git repositories and run on top of Qemu [5] 2.5.0 with KVM acceleration. The Linux VM is also an Ubuntu 16.04 distribution. Unless otherwise stated, the compilers used are GCC/G++ v6.3.0 and the `-O3` level of optimizations is used.

5.1 Footprint Reduction & Boot Time

Image Size. We compared the sizes of the unikernel on-disk images for minimal Linux VMs, HermiTux, OSv, Rumprun and Docker for a simple "hello world" program. We also include numbers for MirageOS [40] for information. For Linux we choose a stripped-down Ubuntu 16.04 distribution from Vagrant repository, and of a distribution designed to be minimal: Tiny Core Linux. For Docker we measured the downloaded image size for Ubuntu 16.04. We measured the binary image size for classical unikernels, and the sum of application and kernel sizes for HermiTux. Results are in Figure 5. As one can observe HermiTux offers the lowest image size, being significantly smaller than Ubuntu (650x) and Docker (96x) images. It is also 23x and 3x smaller than OSv and rump images, respectively. Please note that in some situations the actual disk size of container images can be reduced with the use of layered filesystems.

Boot and Destruction Time. These metrics are critical for unikernels [39, 41, 49], in situations where reactivity and/or elasticity is required. We measured the total execution time of a dummy application whose `main` function returns directly: such time encompasses both boot and destruction latencies. We measured time from the moment the unikernel/container management tool is invoked (Uhyve for HermiTux, `docker` for Docker, `Capstan` for OSv and `Rumprun` for Rump) until the control is passed back to the terminal after the application's execution. We also measured boot and destruction time of a minimal Linux VM. Results are on Figure 5. Once again HermiTux gives the best results as it benefits from the minimalist design of both Uhyve and the OS layer: the sum of boot and destruction times is one to two orders of magnitude faster than Docker, OSv and Rump. The slow startup/halt latency for these is mainly due to the long setup and teardown of the heavyweight Qemu/KVM for OSv and Rump, and the Docker management engine.

Memory Usage. A low memory footprint is one of the promises of the unikernel model. We determined the lowest amount of RAM required for a hello world application to execute within a minimal Linux VM, in HermiTux, OSv and Rumprun. For Docker, we measured the memory usage of a container running the program using `docker stats`. Results are presented on Figure 5. The minimalist design of HermiTux allows it to offer a low memory footprint: 9MB. Rumprun also exhibits a similarly small memory. On the contrary, OSv's memory footprint is 8 times higher than with HermiTux. This is due to the additional libraries and systems software present in OSv's unikernel image.

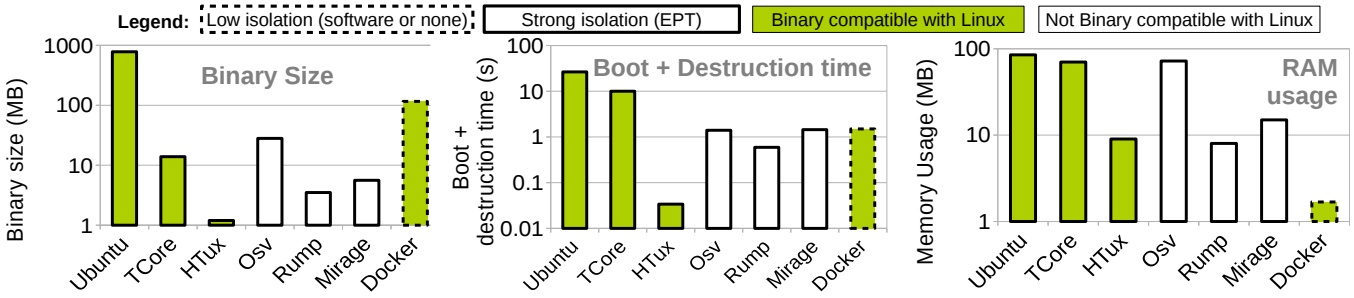

Figure 5. Binary size, boot time and memory usage comparison for a stripped-down Ubuntu distribution (Ubuntu), Tiny Core Linux (TCore), HermiTux (HTux), OSv, Rump, MirageOS and Docker (note the logscale on the y-axis).

Table 1. System call-based modularity efficiency.

Program	Number of system calls	Kernel `.text` size reduction
Minimal	5	21.87 %
Hello world	10	19.84 %
PARSEC Blackscholes	15	17.05 %
Postmark	26	14.36 %
Sqlite	31	11.34 %
Full syscalls support	64	00.00 %

System Call Level Modularization. We analyzed a set of applications with our system call identification tool and compiled a set of HermiTux kernels, each tailored for an application by supporting only the system calls made by that application. Table 1 presents the number of system calls made and the savings in terms of kernel code segment size reduction brought by the tailored kernel over a kernel with full system calls support. We chose the code segment size for metric as reducing its size enhances security: indeed, this segment is mapped with executable rights and is a potential target of code reuse attacks. In Table 1, *minimal* represents a kernel for an application with minimal system call usage: its `main` returns directly. Results show that compiling a tailored kernel can lead to a significant reduction in the kernel code size, for example it is more than 17% for Blackscholes. More system call intensive applications see a smaller size reduction: 11% for SQLite. We expect these numbers to grow as support for more system calls is added to HermiTux.

These experiments show that HermiTux offers low image sizes, RAM usage, boot time, and a modular kernel codebase, while being binary-compatible with Linux applications.

5.2 Application Support: Compilation Scenarios

To demonstrate the generality of HermiTux, we compiled NPB BT class A under different configurations. We varied the compiler (GCC and LLVM [33]), the C library (Musl and GLibC), and the language the benchmark is written in: NPB has C [62] and Fortran [3] implementations. Two additional configurations include (1) a stripped and (2) obfuscated binary. Obfuscation is typically used in scenarios where proprietary software is involved. It was achieved using Obfuscator-LLVM [23], an open-source tool applying obfuscation passes on the LLVM Intermediate Representation. We activated altogether these obfuscation techniques: instruction substitution, bogus control flow insertion, and control flow flattening. `-O3` optimization level was enabled for all configurations.

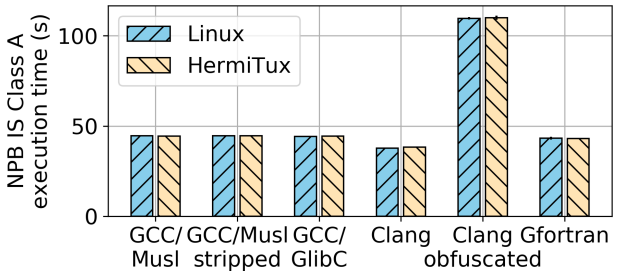

Figure 6. Linux and HermiTux NPB BT class A execution time for various compilation scenarios.

Execution times for Linux and HermiTux are very similar, as presented in Figure 6: the maximum difference is a 1.5% slowdown for Clang non obfuscated. One can also observe that compiling with LLVM brings about 15% performance improvement, and that the combination of obfuscation options we chose leads to a 146% slowdown. Such a slowdown is comparable for Linux and HermiTux, and it is to be expected due to the obfuscation overhead. Varying the C library and the language does not impact the performance of such a compute-/memory-intensive workload.

5.3 General Performance

Memory- and Compute-bound Benchmarks. We ran a set of benchmarks from NPB (BT/IS/EP), PARSEC (Swaptions and StreamCluster), and Python Performance Benchmark (Nbody). Note that HermiTux is able to run other programs from the benchmark suites, which results are not presented here for space reasons. To support Python, HermiTux runs the Micropython [47] lightweight interpreter. Results are

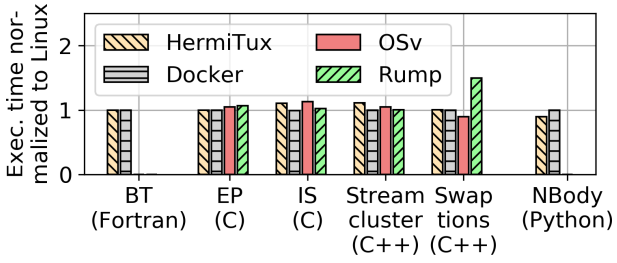

Figure 7. NPB/PARSEC exec. time normalized to Linux.

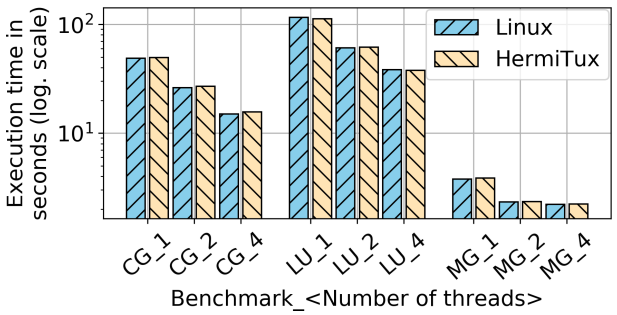

Figure 8. Multithreaded benchmarks results.

presented on Figure 7 where execution times are normalized to the execution time of Linux: 1 on the y-axis represents Linux's execution time. OSv and Rump do not support Fortran or Micropython.

HermiTux performs similarly to Linux: the average difference between HermiTux and Linux execution time over all benchmarks is 2.7% (including NPB/PARSEC/Python benchmarks not shown here because of space reasons). The overhead observed for HermiTux is slightly higher for a few benchmarks (for example IS). The reason is the very short runtime of this tests: a few seconds. In these cases, the benchmark is so short that I/O, in the form of printing to the standard output, becomes a significant source of latency (for our tests HermiTux such I/O is forwarded to the host).

Both Docker and OSv also present very similar results compared to Linux. It is also the case for Rump, however one can observe a significant slowdown (50%) for Swaptions. Rump lacking profiling tools, we were not able to pinpoint the exact reason for this degradation. One explanation could be that Rump toolchain is slightly older: it uses g++ v5.4.0 whereas all the other systems make use of the host g++ v6.3.0.

Multithreading and SMP. As unikernels do not support multiprocess applications, multithreading support is important to leverage multiple cores. We ran the mutlithreaded OpenMP version of NPB CG, LU and MG class B, using Intel Libiomp. We compared the execution time of HermiTux versus Linux for each program given 1, 2 and 4 threads (and as many VCPUs for HermiTux). Note that OSv does not support OpenMP [52] and the Rumprun unikernel does not support

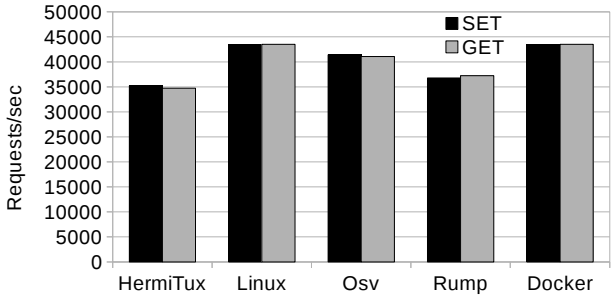

Figure 9. Redis performance.

SMP [60]. Results are presented in Figure 8. Once again, HermiTux exhibits identical performance compared to Linux: the average difference among these tests is 2.1%. Each benchmark scales similarly in HermiTux and Linux while given more threads and cores.

Network Performance. To assess HermiTux's network performance, we used Redis. Redis is a widely used key-value store server, and a perfect target for unikernel deployment in particular because of its security requirements as a server application. We ran Redis within HermiTux, a Linux KVM VM, a Docker container, and we compiled it for Rump and OSv. We bridged the virtual connections to the host physical network so that each VM is accessible from outside the host, and ran the `redis-benchmark` on an external machine on the local network. For that benchmark we used the following parameters: the number of clients was set to 10, the number of requests to 100000, and the data size to 10 bytes.

On Figure 9 we present the results, the rate in requests per second for Redis's GET and SET operations. As one can see, HermiTux's performance are slightly lower (18%) than Linux: this is due to multiple factors, including the un-optimized network driver and TCP/IP stack (LWIP) used with HermiTux that cannot compete with Linux's highly optimized network stack and virtual drivers. Rump's slowdown is relatively similar to the slowdown of HermiTux, while OSv is slightly better (5% slowdown). Docker's performance is marginally better than Linux, possibly because of a more direct access to the host network [18]. Note that Virtio is not yet supported in HermiTux and the enabling such paravirtualized drivers should bring performance gains.

System Call Latency. We used LMbench3 [44] to measure system call latency in HermiTux, for null (`getppid`), `read` and `write`. LMbench reports the execution time of a loop calling 100000 times the corresponding system call. Figure 10 shows the results for native Linux, HermiTux's system call handler, a static binary running in HermiTux with binary-rewritten system calls, and a dynamic binary running in HermiTux with our substituted unikernel-aware C library.

The system call latency is on average 5.6x lower in HermiTux (handler) compared to Linux. It is due to multiple

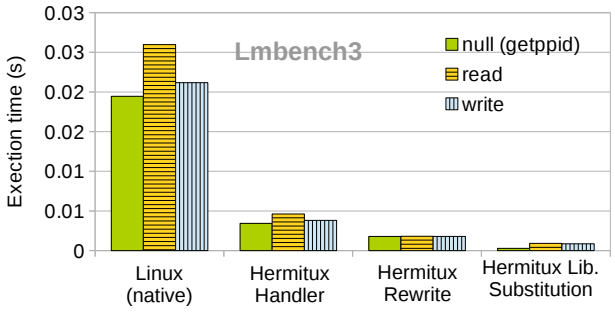

Figure 10. LMbench system call latency.

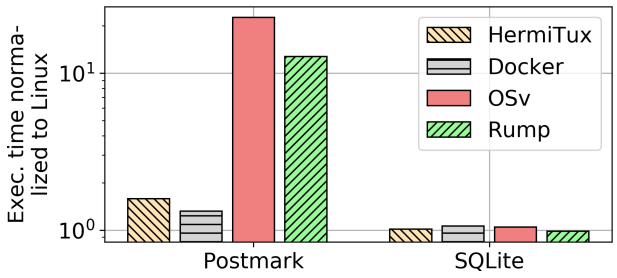

Figure 11. Execution times for the Postmark and SQLite benchmarks.

factors, including a simple and optimized handler implementation (for example there is no `sysret` in HermiTux) and a simpler implementation for the system calls themselves, speeding up their processing. Binary-rewriting the invocations of `syscall` in static binaries gives a 2.3x reduction over the regular handler in HermiTux: this is mainly due to the suppression of the interrupt overhead induced by the `syscall` instruction. Finally, substituting a unikernel-aware C library for dynamic programs brings a 5.7x latency reduction compared to HermiTux's handler: in that case system calls are common function calls. This is faster than our binary-rewriting technique (2.3x) because of the additional instructions this technique needs to execute.

Filesystem and Database Benchmarks. Postmark [25] represents the filesystem activity of a mail server, which is also a good application case for HermiTux. The benchmark is configured with the number of files set to 5 000, file size to 128 KB, read/write size to 4 KB, and the number of transactions to 20 000. Under this configuration, the total amount of data read and written are 1.3 and 1.9 GB, respectively. This means that with Linux, the entire data set falls into the page cache: this is voluntary as we want to stress the filesystem processing code rather than to have the disk be the bottleneck. Docker, OSv and Rump are also given enough memory to absorb the entire data set (we confirmed by experiment that OSv uses in-memory file access buffering, and Rump uses the rumpfs RAM filesystem). We also used a simple SQLite unikernel in which we measure the execution time of populating a local in-memory database with 2 millions records (with 1 integer and 1 128-bytes string fields).

Results are presented in Figure 11, where execution times are normalized to Linux'. Concerning Postmark, Docker's overhead is the lowest (1.3X), followed by HermiTux's (1.6X) demonstrating the relative efficiency of MiniFS. OSv and Rump have poor filesystem performance (more than 10X). Concerning SQLite, each system performs closely to Linux: all slowdowns are below 6%. Indeed no filesystem operation is involved as for our test we use an in-memory database.

These experiments show that HermiTux can bring the low footprint, fast boot time, and low system call latency of unikernels and be binary-compatible without a significant performance impact for a wide range of applications.

5.4 Miscellaneous Features

HermiTux's debugger and profiler were thoroughly validated. In unikernels the kernel and application live in the same, unprotected address space, so debugging and profiling both entities can be realized in concert [61]. Concerning GDB, both the application and kernel symbol files can be loaded using `add-symbol-file` and indicating the starting virtual address for each entity. We provide a GDB wrapper that makes debugging an application running under HermiTux no different from a regular application in Linux. We also validated our profiler by comparing its results for the NPB benchmarks with `perf` running in Linux. The orders of most time consuming functions reported are similar, and in terms of percentage of time spent in each of these the difference between HermiTux profiler and `perf` averages 2.06%. Our profiler was configured at a frequency of 10 samples per second (negligible performance degradation). We also validated HermiTux's checkpoint/restart feature by running it successfully over the NPB benchmarks.

6 Related Works

Rumprun [24] and OSv [28] are two unikernels focusing on compatibility with existing/legacy applications. Rump allows components of the NetBSD kernel to be used as libraries compiled with an application to create a unikernel. OSv [28] is designed from scratch, providing a specialized API for cloud applications, and supporting an unmodified Linux ABI. However, applications have to be recompiled as relocatable shared-objects. Consequently, both Rump and OSv require source code to be available and the build process of applications has to be extended to suit these unikernel models' requirements. With LightVM [41], authors show that the performance of unikernels are similar/better compared to containers and argue that porting to a unikernel requires significant effort. They propose Tinyx, a system which allows automated building of a stripped-down Linux kernel. HermiTux tackles the same problem by running unmodified

Linux executables on top of a unikernel layer whereby footprint and attack surface are significantly reduced compared to the Linux kernel (even a stripped down version). Red Hat recently announced working on a unikernel version of Linux, UKL [59], showing that there is a strong demand for more compatibility from unikernels, as well as the validity of a model in which the porting effort is on the kernel developer rather than the application programmer. UKL requires the application sources and thus it is not binary compatible. Moreover, even in a unikernel form, it is unclear if a heavyweight kernel such as Linux can achieve memory/disk footprints and attack surface that are as low as HermiTux'.

Graphene [68] is a LibOS running on top of Linux, capable of executing unmodified, multi-process applications. Graphene's security can be enhanced with Intel SGX [70], but this involves significant overhead (up to 2x). Google recently open-sourced Gvisor [19], a Go framework addressing container security concerns by providing more isolation. While binary compatibility comes for free in containers, we show that it is also doable in unikernels. Unikernels such as HermiTux are an interesting alternative to containers and software LibOSes as they benefit from the strong isolation enforced by hardware-assisted virtualization. In [77], the authors note that unikernels may run as processes as opposed to virtual machines for enhanced performance and compatibility with developer tools, without a fundamental loss of isolation. While these arguments are very compelling, we believe that HermiTux is an interesting alternative in scenarios where strong isolation (EPT) is needed, for example when considering vulnerabilities such as Meltdown [36], ret2usr [27], ret2dir [26], etc. Dune [4] uses hardware-assisted virtualization to provide a process-like abstraction, and implements in particular a sandboxing mechanism for native Linux binaries. It is important to note that its isolation model is quite different from HermiTux: Dune either redirects system calls to the host kernel or blocks them, which limits compatibility when blocking or decreases isolation when redirecting.

The authors of a Linux API study [69] analyze its usage and classify system calls by popularity. Such information can be used to prioritize system call development in HermiTux. A system call binary identification technique is also mentioned, but few implementation details are given, and authors report that identification fails for 4% of the call sites.

7 Conclusion

HermiTux runs native Linux executables as unikernels by providing binary compatibility, relieving application programmers from the effort of porting their software. In this model, not only can unikernel benefits be obtained for free in unmodified applications, but it is also possible to run previously un-portable software. HermiTux achieves this goal with, in most cases, negligible to acceptable overhead compared to Linux, and performs generally better than other unikernels (OSv, Rump) for unikernel-critical metrics. HermiTux is available online under an open-source license: https://ssrg-vt.github.io/hermitux/.

Acknowledgments

This work is supported in part by ONR under grants N00014-16-1-2104, N00014-16-1-2711, and N00014-16-1-2818.

This research and development was supported by the German Federal Ministry of Education and Research under Grant 01IH16010C (Project ENVELOPE).

References

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, and others. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*, Vol. 16. 689–703.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, and others. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[4] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features.. In *Osdi*, Vol. 12. 335–348.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[7] Sören Bleikertz. 2011. How to run Redis natively on Xen. (2011). https://openfoo.org/blog/redis-native-xen.html. Online, accessed 11/27/2017.

[8] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 250–257.

[9] Cloudozer LLP. 2017. LING/Erlang on Xen website. (2017). http://erlangonxen.org/. Online, accessed 11/20/2017.

[10] Christian S Collberg, John H Hartman, Sridivya Babu, and Sharath K Udupa. 2005. SLINKY: Static Linking Reloaded.. In *USENIX Annual Technical Conference, General Track*. 309–322.

[11] Jonathan Corbet. 2009. Seccomp and sandboxing. *LWN. net, May* 25 (2009).

[12] Jonathan Corbet. 2011. On vsyscall and the vDSO. (2011). https://lwn.net/Articles/446528/, Online, accessed 08/05/2018.

[13] Intel Corp. 2018. Intel Clear Containers. (2018). https://clearlinux.org/documentation/clear-containers. Online, accessed 08/04/2018.

[14] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet'17)*. ACM, 36–41.

[15] Will Dietz and Vikram Adve. 2018. Software multiplexing: share your libraries and statically link them too. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 154.

[16] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT security and scalability: how unikernels can improve the status Quo. In *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UUC 2016)*. IEEE, 292–297.

[17] ELF 2015. Executable and Linking Format (ELF). (2015). http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-AMD64/LSB-Core-AMD64/elf-amd64.html. Online, accessed 11/24/2017.

[18] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 171–172.

[19] Google. 2018. Gvisor Github webpage. (2018). https://github.com/google/gvisor, Online, accessed 05/03/2018.

[20] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*. 217–233.

[21] Hacker News 2017. Unikernels are Secure. (2017). https://news.ycombinator.com/item?id=14736909. Online, accessed 11/27/2017.

[22] Intel Corporation. 2017. Intel 64 and IA-32 Architectures Software Developer Manual. (2017).

[23] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, Brecht Wyseur (Ed.). IEEE, 3–9. DOI: http://dx.doi.org/10.1109/SPRO.2015.10

[24] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[25] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark*. Technical Report. Technical Report TR3022, Network Appliance.

[26] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking Kernel Isolation.. In *USENIX Security Symposium*. 957–972.

[27] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks.. In *USENIX Security Symposium*, Vol. 16.

[28] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[29] Michał Król and Ioannis Psaras. 2017. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 134–144.

[30] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17)*. ACM, 15–29.

[31] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM.

[32] S. Lankes, S. Pickartz, and J. Breitbart. 2017. *A Low Noise Unikernel for Extrem-Scale Systems*. Springer International Publishing, Cham, 73–84. DOI: http://dx.doi.org/10.1007/978-3-319-54999-6_6

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[34] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 601–614.

[35] Linux Kernel Contributors 2017. Linux kernel documentation: x86_64 memory map. (2017). http://elixir.free-electrons.com/linux/v4.14.2/source/Documentation/x86/x86_64/mm.txt, Online, accessed 11/24/2017.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[37] Wei Liu. 2013. Improving Scalability of Xen: the 3000 Domains Experiment. Collaboration Summit. (2013). https://events.static.linuxfound.org/images/stories/slides/lfcs2013_liu.pdf

[38] LWIP 2017. LWIP Website. (2017). https://savannah.nongnu.org/projects/lwip/. Online, accessed 12/12/2017.

[39] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, and others. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 559–573.

[40] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[41] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233. DOI: http://dx.doi.org/10.1145/3132747.3132763

[42] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[43] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[44] Larry W McVoy, Carl Staelin, and others. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[45] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[46] Daniel Micay. 2018. Linux ASLR Comparison. (2018). https://gist.github.com/thestinger/b43b460cfccfade51b5a2220a0550c35, Online, accessed 12/12/2018.

[47] Micropython Contributors. 2018. Micropython webpage. (2018). https://micropython.org/, Online, accessed 08/05/2018.

[48] Newlib 2017. Newlib Website. (2017). https://sourceware.org/newlib/. Online, accessed 12/12/2017.

[49] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 1–14. DOI: http://dx.doi.org/10.1145/3050748.3050758

[50] OSv Contributors. 2013. OSv - execve(2) support. (2013). https://github.com/cloudius-systems/osv/issues/43, online, accessed 12/10/2018.

[51] OSv Contributors 2014. Porting native applications to OSv: problems you may run into. (2014). https://github.com/cloudius-systems/osv/wiki/Porting-native-applications-to-OSv. Online, accessed 05/02/2018.

[52] OSv contributors. 2016. OSv Issues: Thread-local storage doesn't work in PIE. (2016). https://github.com/cloudius-systems/osv/issues/352, Online, accessed 04/21/2018.

[53] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM, New York,

NY, USA, 247–260. DOI: http://dx.doi.org/10.1145/1352592.1352618

[54] Russell Pavlicek. 2018. Containers 2.0: Why unikernels will rock the cloud. (2018). https://techbeacon.com/containers-20-why-unikernels-will-rock-cloud. Online, accessed 08/05/2018.

[55] Max Plauth, Lena Feinbube, and Andreas Polze. 2017. A Performance Survey of Lightweight Virtualization Techniques. In *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing (ICN 2017)*. Springer, 34–48.

[56] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. DOI: http://dx.doi.org/10.1145/1950365.1950399

[57] Pthread Embedded 2017. Pthread Embedded Website. (2017). http://pthreads-emb.sourceforge.net/, Online, accessed 12/12/2017.

[58] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[59] Ali Raza. 2018. UKL: A Unikernel Based on Linux. https://next.redhat.com/2018/11/14/ukl-a-unikernel-based-on-linux/, Online, accessed 12/12/2018. (2018).

[60] Rump contributors. 2016. Rumpkernel FAQ. (2016). https://github.com/rumpkernel/wiki/wiki/Info:-FAQ, Online, accessed 04/21/2018.

[61] Florian Schmidt. 2017. uniprof: A Unikernel Stack Profiler. In *Proceedings of the ACM Special Interest Group on Data Communication Conference (Posters and Demos) (SIGCOMM'17)*. ACM, 31–33.

[62] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization (IISWC 2011)*. IEEE, 137–148.

[63] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2016)*. ACM, 44–49.

[64] Solo 5 2017. The Solo5 Unikernel. (2017). https://github.com/Solo5/solo5. Online, accessed 11/25/2017.

[65] Richard Stallman, Roland Pesch, Stan Shebs, and others. 1988. Debugging with GDB. *Free Software Foundation* 675 (1988).

[66] Victor Stinner. 2017. The Python Performance Benchmark Suite. (2017). http://pyperformance.readthedocs.io/, Online, accessed 08/04/2018.

[67] Josh Triplett. 2015. Using the KVM API. (2015). https://lwn.net/Articles/658511/. Online, accessed 11/25/2017.

[68] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. ACM, 9.

[69] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. 2016. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 16.

[70] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC 2017)*. 8.

[71] Unikraft 2017. Xen Website - Unikraft. (2017). https://www.xenproject.org/help/wiki/80-developers/207-unicore.html. Online, accessed 11/27/2017.

[72] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. (2017).

[73] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling.. In *USENIX Security Symposium*. 627–642.

[74] Adam Wick. 2012,. The HaLVM: A Simple Platform for Simple Platforms. Xen Summit. (2012,).

[75] Chadd C Williams and Jeffrey K Hollingsworth. 2004. Interactive binary instrumentation. In *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*.

[76] D. Williams and R. Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO, USA. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[77] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 199–211. DOI: http://dx.doi.org/10.1145/3267809.3267845

[78] Bruno Xavier, Tiago Ferreto, and Luis Jersak. 2016. Time provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2016)*. IEEE, 277–280.

[79] Xen Website. 2018. Google Summer of Code Project, TinyVMI: Porting LibVMI to Mini-OS. (2018). https://blog.xenproject.org/2018/09/05/tinyvmi-porting-libvmi-to-mini-os-on-xen-project-hypervisor/, Online, accessed 10/30/2018.

[80] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.

[81] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. 2016. On the performance of intel sgx. In *Web Information Systems and Applications Conference, 2016 13th*. IEEE, 184–187.