This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3122896

A Syscall-Level Binary-Compatible Unikernel

Pierre Olivier[1], Hugo Lefeuvre[1], Daniel Chiba[2], Stefan Lankes[3], Changwoo Min[4], and Binoy Ravindran[4]

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 system call-level binary compatibility with Linux applications. It is composed of a hypervisor and a lightweight kernel layer emulating the load- and runtime 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. HermiTux's design principles are architecture-independent and we present a prototype on both the x86-64 and ARM aarch64 ISAs, targeting various cloud as well as edge/embedded deployments. We demonstrate HermiTux's compatibility over a range of native C/C++/Fortran/Python Linux applications. We also show that it offers a similar degree of lightweightness compared to other unikernels, and that it performs similarly to Linux in many cases: its performance overhead averages 3% in memory- and compute-bound scenarios, and its I/O performance is acceptable.

Index Terms—Unikernels, Virtualization, Operating Systems

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, etc. Their potential application domains are plentiful: cloud- and edge-deployed micro-services/SaaS/FaaS-based software [1], server applications [2], NFV [3], IoT [4], HPC [5], etc. Although they are presented as a secure and attractive alternative to containers, unikernels still struggle to gain significant traction in industry and their adoption rate is quite slow [1]. One of the major reasons is the difficulty, and sometimes impossibility, of porting legacy/existing applications to current unikernel models [1], [2], [6]–[8].

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 relinking with a unikernel layer is not suitable. Even when sources are available, considering unikernel models supporting legacy programming languages (C/C++) [9], [10], porting a medium/large-sized or complex codebase can still be difficult [1], [2], [6], [8]. 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 [7]. Because it is currently the burden of the application programmer, 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 [5] 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, etc.), compilers (GCC and LLVM), full optimizations (`-O3`), and stripped/obfuscated binaries. It supports multithreading and Symmetric Multi-Processors (SMP), checkpoint/restart and migration. 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 majority of existing unikernels do not provide any kind of binary compatibility. Still, a couple of systems [10], [11] offer such binary compatibility by interfacing at the level of the C library, acting similarly to a dynamic loader. This prevents them for supporting a wide range of applications requiring OS services through system calls made without going through the C library. Contrary to these works, and in order to maximize compatibility, HermiTux is compatible at the *system call* level which is a standardized interface used by all applications and libraries compiled for

- *Contact: pierre.olivier@manchester.ac.uk*
- [1]*The University of Manchester,* [2]*Qualcomm,* [3]*RWTH Aachen University,* [4]*Virginia Tech*

Linux.

The first challenge HermiTux tackles is *how to provide system call-level 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.

Because of the wide range of unikernel application cases, HermiTux aims to be compatible in both server and embedded virtualization scenarios. Thus, our system is developed for the Intel x86-64 and ARM aarch64 (ARM64) Instruction Set Architectures (ISAs). The fundamental principles of HermiTux's design are architecture independent. However, its implementation as well as the design of the binary rewriting/analysis techniques we use to bring back unikernel benefits are ISA specific. These differences are described in this paper.

The contributions presented in this paper are:

- A new unikernel model designed to execute native Linux executables while maintaining the classical unikernel benefits;
- Two prototype implementations of that model on the x86-64 and aarch64 architectures;
- An evaluation of these prototypes comparing their performance to Linux, containers, and other unikernel models: OSv [10], Rump [9] and Lupine Linux [11].

This paper is organized as follows: 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

2.1 Unikernels

A unikernel [2] 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 [12]. Some unikernels are also written in languages providing memory-safety guarantees [2]. Concerning performance, unikernel system calls are fast because they are common function calls: there is no costly world switch between privilege levels [5]. 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 [1].

All these benefits make that the application domains for unikernels are plentiful. They are a perfect fit for the datacenter [1], [2] that runs the majority of cloud applications requiring high degree of isolation, or compute-intensive jobs necessitating high performance and low OS overheads. Furthermore, the reduced resource usage of unikernels make them uniquely suited for embedded virtualization [4], [12], a domain of growing importance with the emergence of paradigms such as edge computing and IoT. Because the application domains of unikernels include both server and embedded machines, the system presented in this paper targets two ISAs: Intel x86-64, which is unarguably the dominant architecture in the datacenter, and aarch64, widely used in embedded devices.

2.2 Porting Existing Applications to Unikernels

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 [2], which in many scenarios is unacceptable. Third, considering the unikernels supporting legacy languages, the task still represents a significant challenge [1], [6], [8] 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 [6]. In many cases the lack of a feature/library means that the application cannot be ported at all. 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.

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.

2.3 The Lightweight Virtualization Design Space

The lightweight virtualization design space includes unikernels, security-oriented LibOSes such as Graphene [13], [14], and containers with software [15] and hardware [16] hardening techniques. HermiTux requires no application porting effort, and further differs from other binary-compatible systems because of 2 reasons. First, as a unikernel HermiTux runs hardware-enforced (Extended Page Tables) VMs, an isolation mechanism that is fundamentally stronger than software-enforced isolation (containers/software LibOS). It is shown by the current trend of running containers within VMs for security (clear containers [17]) and the efforts to strengthen containers' isolation (such as gVisor [15]). This is generally used as a security argument in favor of unikernels versus containers [1], [18]. Second, SGX-based isolation such as used in Graphene-SGX [14] has a non-negligible performance impact 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.

2.4 System Call-Level Binary Compatibility

Two existing unikernels already claim binary compatibility with applications, OSv [10] and Lupine Linux [11]. It is important to note that both offer binary compatibility at the *standard C Library* (libc) level: the unikernel includes a dynamic loader that catches at runtime the calls to the libc functions such as `printf`, `fopen` and redirects them to the kernel.

Such an method of interfacing implies the assumption that *all syscalls are made through the libc*, which does not hold true when considering the wide variety of modern application binaries. We analyzed the entirety of Debian 10 x86-64 repositories (`main`, `contrib` and `non-free`) and counted 553 ELF executables including at least one invocation of the `syscall` instruction: these represent programs that perform system calls without going through the libc, and that as such would not be supported by libc-level binary compatible unikernels. This limited libc-level compatibility prevents these systems from running a relatively large range of applications that would highly benefit from execution as unikernels. Just to give a few examples, a plethora of cloud services are written in Go, a language that performs most system calls without going through a standard C library. Furthermore, due to the lack of compatibility at the system call level, OSv does not support the most popular HPC shared memory programming framework, OpenMP[1]. Finally, libc-interfacing precludes support for static binaries.

HermiTux represents an attempt to push the degree of compatibility of unikernels further by interfacing at a much more standard and unanimously used interface: the system call level.

3 System Design

The design of HermiTux is based on the following assumptions: 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 [19], [20], 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 [21]. These rules are partially ISA-specific, and 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 file, and the particular register state (ISA-specific) 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: these are obviously ISA-specific. Finally, Linux applications also expect to communicate with the OS by reading and writing various virtual filesystems (`/proc`, `/sys`, etc.) [22] as well as through a memory area shared with the kernel: the vDSO/vsyscall.

3.1 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, namely fast system calls

1. https://github.com/cloudius-systems/osv/issues/590

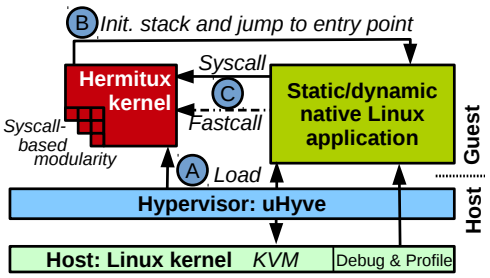

Fig. 1. Overview of the HermiTux system, composed of Uhyve hypervisor running on a Linux host, modified to load both the HermiTux kernel and the Linux binary into a single-address space VM.

and modularity, without assuming access to the application sources using binary rewriting and analysis techniques for static executables. For dynamically compiled programs, we use a unikernel-aware shared standard C library that is loaded at runtime in place of the original standard library the program was linked against. HermiTux's kernel is developed as an extension to the HermitCore unikernel [5].

Figure 1 presents a high-level view of the system. We rely on a custom lightweight hypervisor named Uhyve that runs in a Linux host and leverages the KVM interface to create a virtual machine. Uhyve was originally developed for HermitCore, and was extended in the context of HermiTux. At launch time, Uhyve allocates memory for the guest to use as its physical memory. Next, the hypervisor loads the HermiTux kernel (Ⓐ on Figure 1) at a specific location in that area, and then proceeds to map 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 at the highest level of privileges: ring 0 in x86-64 and exception level 1 for aarch64.

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, i.e. using the `syscall` x86-64 instruction or the `SVC` (supervisor call) instruction for aarch64. The kernel catches such invocations by implementing a system call handler that identifies the system call invoked, determines parameters from CPU registers, and invokes the HermiTux implementation of the considered system call Ⓒ.

3.2 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 applications. 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 at 0x200000, *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: 34 bits, which is higher than vanilla Linux (28 bits) as well as PaX/grsecurity hardened kernels (33 bits).

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 that it comes shipped with the application binary and its shared library dependencies. Contrary to KylinX [23], 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 [24] such as Flush+Reload or Prime+Probe. Moreover, if security is less of a concern and memory usage is constrained, Kernel Same page Merging (KSM) [25] is an efficient and standard way to solve that issue. While dynamic binaries are favored by Linux distribution maintainers [22], static executables still provide benefits in terms of performance [26] and compatibility [27]. Thus we aim to support both linking types.

3.3 Runtime Binary Compatibility

In a unikernel, system calls are common function calls. In HermiTux, the application performs system calls using the (ISA-specific) Linux convention, unsupported by existing unikernels. HermiTux implements a system call handler invoked when the specific instruction triggering a system call is executed by the application. The handler redirects the execution to the internal unikernel implementation of the invoked system call (Ⓒ on Figure 1). Interfacing with the application at the system call level is at the core of the runtime 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 only a very 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 [28]. It has also be shown that one can support 90% of a standard distribution's binaries by implementing as few as 200 system calls [22]. To support the applications presented in the evaluation section (Section 5), our prototype implements 107 system calls. Source-compatible unikernels such as OSv [10] or Rumprun [9]

also showed that supporting a relatively large portion of the Linux system call API does not lead to a significantly large codebase size or attack surface.

3.4 Unikernel Benefits & Isolation

System call latency in unikernels is low as they are common 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 instruction used to perform a system call used in unmodified Linux binaries. In both x86-64 and aarch64 ISAs, that instruction relies on an exception. The latency of such an operation is significantly higher than that of a common call instruction.

Without assuming access to the application sources, we rely on two techniques to offer fast system calls in HermiTux (*Fastcall* in Figure 1). For static binaries, we use binary instrumentation to rewrite the system call instructions found in the application with regular function calls to the corresponding unikernel implementations. This process is ISA-specific and is detailed for both x86-64 and aarch64 in Section 4. For dynamically linked programs, we observe that in a vast majority of application binaries, most of the system calls are made by the standard C library. With that in mind, in HermiTux dynamic binaries are linked at load time against a unikernel-aware C standard library that we designed, in which all the system calls are replaced by function calls to the kernel. We call this technique library substitution. It bears a resemblance to the way the OSv [10] kernel and LibC interface with an application, however in our case we do not write a Libc from scratch but adapt automatically an existing one using a code transformation tool, Coccinelle [29]. Compared to writing a Libc from scratch, we believe this solution not only provides a larger support for Libc functionalities, but is also more robust and future proof.

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): 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-reuse 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

HermiTux is build on top of HermitCore [5] with 15K additional LoC on top of HermitCore's 20K LoC. It supports both x86-64 and aarch64. Although our system's design principles are architecture independent, a (small) subset of its implementation is architecture-specific.

Loading and Initialization. The hypervisor sets up the VM and loads both the kernel and the application in memory, according to the ELF metadata in the binaries. If the application supports PIC/PIE it is loaded at a random location. Next the kernel initializes and creates a page table defining a single address space. After initialization, the kernel creates a task for the application. The application will share its stack with the kernel, so the stack is filled with elements (command line parameters, etc.) according to the ABI convention, with a series of push operations on x86-64. Doing the same thing is not practical on aarch64 as this ISA only supports 16 bytes-aligned push operations, and many elements we wish to push are 8 bytes in size. Thus, we fill a temporary buffer that ends up being copied to the stack with potentially one byte of padding.

System Call Handling. The kernel installs and implements a system call handler adhering to the Linux ABI: system call number in `%rax`/`%x8`; arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r9`, `%r8`/`%x0`-`%x5`; and return value in `%rax`/`%r0` for x86-64/aarch64, respectively. The handler saves the register's contents, calls the implementation of the invoked system call, and restores the register before returning. It is optimized: many 'world switch' operations are unnecessary in a unikernel (for example stack switches). When returning, we can also avoid costly instructions such as `sysret` on x86-64 and replace it with a simple jump.

HermiTux supports currently 107 system calls. Many are only partially supported, for example, `ioctl` only supports the necessary commands for LibC initialization. 4K LoC are dedicated to the system call layer, showing that HermiTux can keep a small unikernel codebase while supporting a wide range of applications as presented in the evaluation section. With the supported system calls, HermiTux is able to emulate Linux's support of networking, filesystem, multithreading and synchronization, memory mappings, process management, break management, signals, time management, and scheduling.

Fast System Calls. In its basic form, HermiTux uses a traditional system call handler and thus looses the unikernel benefit of low-latency system calls. To recover that feature for dynamically compiled binaries, we link at runtime against a unikernel-aware standard C Library. The unikernel-aware C library loaded at runtime with dynamic binaries is adapted from Musl Libc. We use the Coccinelle [29] tool to describe high level code transformation rules updating system call invocations into function calls to HermiTux's kernel. With a small set of rules (80 lines), we are able to update 97.5% of the 500+ system call invocation within the entire library. We

confirmed the success of this method over different versions of Musl released multiple years apart.

Regarding static binaries, we resort to binary rewriting, realized statically to avoid any runtime overhead. Our goal is to replace the occurrences of the system call instruction (`syscall` for x86-64, `SVC` for aarch64) by function calls to the kernel. For x86-64, an ISA whose instructions have variable sizes, the main challenge lies in the small size of `syscall`: 2 bytes. It is too small to allow replacement by any kind of `call` or jump-like instruction without overwriting the next instruction(s) in the code segment. To address that issue, we overwrite each occurrence of the `syscall` instruction as well as the next instruction(s) with a `jmp` to a snippet of code that we developed. This code is in charge of first adapting the Linux syscall ABI convention to the function call system-V convention (i.e. moving `%r10` to `%rcx`). Second, the system call implementation in the kernel is invoked with a common function call instruction, `callq`. Finally, the instructions following the `syscall` that were originally overwritten are replayed, before jumping back to instruction following the last overwritten instruction. Although this process includes a few operations in addition to the function call, it is still much faster than a traditional system call invocation.

On the other hand, aarch64 is a fixed-size instruction set and thus does not suffer from the same issue as x86-64. Intuitively the system call instruction `SVC` can be simply overwritten with a function call, i.e. a `BL` (Branch and Link) without side-effects on the following instructions. The actual challenge for aarch64 lies in an important ABI point: contrary to x86-64 that stores the return address on the stack, aarch64 holds it in the special register `%x30`. Thus, when we replace the system call instruction `SVC` with the function call one `BL`, `BL` overwrites `%x30` holding the return address of the current function with the address that the newly inserted function (i.e. the system call implementation) is supposed to return to.

In that context, one may think that we would loose the possibility to return from functions invoking system calls which of course would break the program. However, we realized that in many cases this issue could be tackled without resorting to a complex solution such as the one we used for x86-64. First, in the common case, a function invoking a system call is also calling other functions, which mandates that the value of `%r30` is saved on the stack by the compiler-generated code and restored at the time of returning from the function in question, thus even if overwriting `SVC` with `BL` looses the value of `%r30`, it will be properly restored before returning. Second, in the relatively common case where `SVC` is directly followed with a return instruction `RET`, then `SVC` can be overwritten with a simple branch instruction `B`: as this function preserve `%x30`, when the system call implementation return, it will simply return to the function initially invoking the system call. Third, a small number of system calls such as `exit` never return thus loosing `%x30` is acceptable. Combined, these 3 cases cover more than 90% of the system call invocation in a standard libc (Musl). We used the angr [30] binary analysis tool to identify them and perform safe replacements of system calls by function calls. The 10% syscalls left go through the standard trap-based handling mechanism.

System Call-based Modularity. With the growing support for the Linux ABI, the subset of HermiTux's codebase that concerns system call implementation is relatively large: it currently represents about 25% of the entire unikernel codebase. To bring back the "modularity" feature of unikernels into HermiTux, we propose the compilation of tailored kernels containing only the implementation of the system calls required for an application. This is achieved by having as much as possible of each system call's processing code implemented within its own compilation unit (C source file), and using preprocessor directives to enable/disable calls to the system call implementations (`sys_*`) from the generic system call handler as necessary at build time.

To leverage this functionality and build a kernel tailored for a given application it is necessary to know the entire set of system calls that can possibly be invoked by the said application. To that aim we decide to rely on static analysis. As we do not assume access to the application source code, we resort to decompiling the binary. Armed with the knowledge of the system call invocation ABI convention, we explore system call sites in the decompiled machine code and determine at these points what is the value present in the register holding the system call identifier: `%rax` for x86-64 and `%r8` for aarch64. This technique works for both statically and dynamically compiled binaries, as for the latter it can be applied to the application binary as well as to libraries. We use Dyninst [31] for x86-64 and Angr [30] for aarch64 to decompile the binaries and obtain the CFG. We iterate on the instruction flow backwards until we find the value loaded in `%rax`/`%r8` identifying a system call. In the vast majority of cases this identifier comes from a constant in the original code and this search is straightforward. For Glibc, we found one call site where this value 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.

In addition to the syscall-based modularity, we also enabled modularized coarse-grained components that were originally (in HermitCore) included in all builds, such as the LWIP TCP/IP stack.

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 lightweightness benefits of unikernels, i.e. low disk/memory footprints and fast boot times? How does it compare to other lightweight virtualization solutions regarding these metrics?* (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 other lightweight virtualization solutions?* (Section 5.3);

We evaluate HermiTux over multiple macro- and micro-benchmarks. The proposed system is compared to several lightweight virtualization solutions, including a Linux VM

running an Alpine distribution on top of the Firecracker hypervisor, Docker, and three unikernels models focusing on compatibility with existing applications: Lupine Linux [11], *OSv* [10] and *Rumprun* [9]. For each of them we use the latest version available on their respective git repositories. Note that on the contrary to HermiTux, none of these unikernels are binary compatible with Linux at the system call level (Lupine's "pure" unikernel form is enabled by Kernel Mode Linux[2] – KML that forces the interfacing to take place at the level of the libc). Lupine and OSv run on top of Firecracker, and Rump on top of Solo5 for compatibility reasons. Lupine does not support aarch64. In network-bound setups, we also run all VMs on top of Qemu for performance reasons. The macro-benchmarks we used include C/Fortran/C++/Python NPB [32], PARSEC [33] and the Python Performance Benchmark Suite[3]. We also build an edge computing benchmark based on PARSEC's StreamCluster compute kernel. Micro-benchmarks include redis-benchmark and LMbench[4] to measure system call latency.

We wish to assess HermiTux's efficiency in both datacenter/cloud and edge contexts, and we run experiments on both x86-64 and aarch64 architectures. The x86-64 machine is an Intel Xeon E5-2637 (3.0 GHz, 64 GB RAM), running Ubuntu Server 16.04 with Linux v4.4.0 as the host. It is a typical server found in the datacenter. The aarch64 machine is a LibreComputer LePotato single board computer, with an aarch64 CPU clocked at 1.5 GHz and 2 GB of RAM. It runs Ubuntu 18.04 with Linux v4.19.0 as the host. It is representative of a certain class of low-power embedded systems found at the edge of the cloud. Unless otherwise stated, the compilers used are GCC/G++ v6.3.0 (x86-64) and v8.3.0 (aarch64), and the `-O3` level of optimizations is used.

In addition to the experiments presented here, we also validated our syscall-level binary compatibility by confirming HermiTux's basic support for additional languages such as Rust, Lua, and Nim.

5.1 Lightweightness: Footprint Reduction & Boot Time

Boot Time. This metric is critical for unikernels [1], [12], [34], in situations where reactivity and/or elasticity is required. Boot and destruction latencies have been measured in various ways in related work. Although hypervisor initialization time can sometimes be non negligible, guests can run on top of various virtual machine monitors and we chose to exclude hypervisor initialization time from our study and only consider guest boot time. We thus define the boot time as the latency between the moment the hypervisor starts to execute guest code when the unikernel is launched and the moment when the first instruction of user code is run after guest kernel initialization. To that aim, we instrumented both the hypervisors (Uhyve, Firecracker and Solo5) and the guest kernels. The hypervisors are modified to take a timestamp right before the start of guest execution. The guest kernels are instrumented by inserting right after the kernel boot process a trap to the hypervisor which in turn takes a timestamp. For Docker, we used `docker events` to compute the difference between the *container start* and *container die* events.

The results are presented on Figure 2. HermiTux inherits the fast and optimized boot time of its basis, HermitCore: 33 ms on x86-64, and 5ms on aarch64. On x86-64 it is moderately slower than OSv (13 ms) and Rump (17 ms), but much faster on aarch64 (34 ms for OSv, 50 ms for Rump). HermiTux also boots quite faster than Docker: 3x for x86-64, 26x for aarch64. Regarding Lupine, as mentioned in the related paper [11] its boot time is impacted by the KML patch: with KML Lupine's application can enjoy fast system calls however the boot time is 3x that of HermiTux: 94 ms. Without KML, it drops to 41 ms. Unsurprisingly, the traditional kernel (Alpine) numbers are much higher, being 20x (x86-64) and 237x (aarch64) higher than HermiTux'.

Memory Usage. A low memory footprint is one of the promises of the unikernel model. Similar to boot time, various ways have been used by related works to measure RAM usage. Once again we chose to exclude hypervisors' internal memory footprint and as such we define RAM usage as the minimal amount of memory one can give to a VM for the execution of a dummy "hello world" program. We used this method for the unikernels and the Alpine VM, and use `docker stat` for Docker.

The results are presented on Figure 2. The minimalist design of HermiTux, inherited from HermitCore's, allows to offer a low memory footprint: 11 MB on x86-64 and aarch64. Rump has a slightly smaller memory usage on x86 (8 MB) but it is more than twice higher on aarch64 (24 MB). OSv's RAM footprint is also higher than HermiTux on both ISAs: more than 2x on x86-64 and 1.3x on aarch64. On x86-64 Lupine's footprint is 1.8x higher than HermiTux'. Unsurprisingly, the Alpine VM has the higher memory usage on both ISAs, 34 MB, and the docker container has the lowest, 6 MB.

Image Size. Finally, we compared the disk image sizes of a simple "hello world" application for each solution. For the compiled unikernels it is simply the size of the unikernel binary. For HermiTux it is the sum of the kernel and application binaries. We also report the disk image size of the application binary itself (that would be the footprint of an empty container), and of an Alpine container and VM.

Results are presented in Figure 2. Varying the ISAs for the systems that support it leads only to minor disk footprint variations. As one can observe HermiTux offers a very low image size: 1.2 MB on x86-64 and 530 KB on aarch64. Once again this benefits comes from HermitCore's minimalist design. It is similar or better than the other unikernel image sizes that are all below 5 MB apart from OSv on ARM which is 7 MB. The Alpine VM has the highest footprint: 35 MB, and for the Alpine container it is relatively low: 5 MB. Note that the application binary size is negligible (about 10KB), indicating that most of it is taken by systems software such as the kernels.

We can conclude by stating that the lightweightness benefits of unikernels are preserved in HermiTux, as it is on par and sometimes better than state-of-the art unikernels.

System Call Level Modularization. We analyzed a set of applications compiled against the Musl libc with our system call identification tool and compiled for both x86-64 and aarch64 a set of HermiTux kernels, each tailored for an

2. http://www.yl.is.s.u-tokyo.ac.jp/~tosh/kml/
3. https://pyperformance.readthedocs.io/
4. http://lmbench.sourceforge.net/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3122896

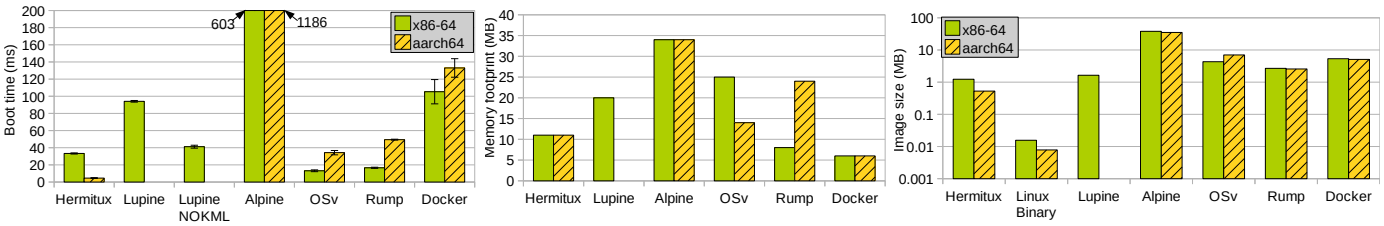

Fig. 2. Boot time, memory usage and image size comparison for several virtualization solutions running on x86-64 and aarch64. HermiTux runs on Uhyve. Lupine, Alpine, and OSv run on Firecracker, and Rump runs on Solo5. NOKML means KML patch disabled.

TABLE 1
System call-based modularity efficiency.

Program	Number of system calls	x86-64 kernel `.text` size reduction	aarch64 kernel `.text` size reduction
Minimal	4	21.22 %	29.26 %
Hello world	9	19.91 %	27.42 %
PARSEC Blackscholes	15	17.68 %	24.50 %
Postmark	27	16.02 %	22.55%
Sqlite	33	11.34 %	16.44%
Full syscalls support	107	-	-

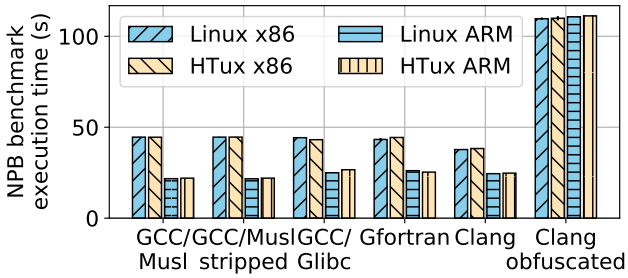

Fig. 3. Execution times for various scenarios, x86 runs BT, ARM CG.

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` function returns directly. Results show that compiling a tailored kernel can lead to a significant reduction in the kernel code size. For example, for Blackscholes, this code reduction is 24% for aarch64, and 17% for x86-64. More system call intensive applications see a smaller size reduction: 16% for aarch64 and 11% for x86-64 with 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 a program from the NPB [32] suite under different configurations. We varied the compiler (GCC v6.3.0 for x86-64 and v8.3.0 for aarch64, as well as LLVM [35] v4.0.1 for both ISAs), the C library (Musl and Glibc), and the language the benchmark is written in: NPB has C and Fortran 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 [36], 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. The experiments were run on both the x86-64 server and the aarch64 embedded board. We chose NPB BT class A for x86-64 and CG class A for aarch64, because these run for a sufficiently long time (tens of seconds) on each machine.

Execution times for Linux and HermiTux are very similar, as presented in Figure 3: for all experiments and in both architectures, the difference between Linux and HermiTux stays below 2%.

For x86-64 one can also observe that compiling with LLVM brings about 15% performance improvement. It is worth noting that the version of LLVM we use (v4.0.1) is slightly more recent than the version of GCC used for x86-64 (v6.3.0). For aarch64, GCC is about 12% faster than LLVM. In that case the version of GCC we used for that ISA (v8.3.0) is much more recent than LLVM's version (v4.0.1).

The combination of obfuscation options we chose leads to a 146% slowdown on x86-64 and a 449% slowdown on aarch64. Such performance degradation is similar for Linux and HermiTux in both ISAs. They are 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 on x86-64. We ran on our x86-64 server 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 [37] lightweight interpreter. Results are presented on Figure 4 where execution times are normalized to the execution time of Linux: 1 on the y-axis represents Linux's

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3122896

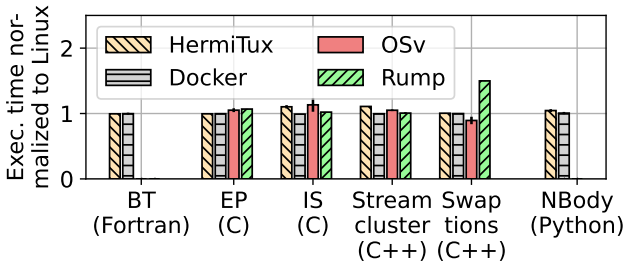

Fig. 4. NPB/PARSEC/Python performance suite execution time normalized to Linux' on x86-64.

execution time. OSv and Rump do not support Fortran or Micropython. These benchmarks have not been ported to run on Lupine, but we expect a runtime similar to Linux since the system call latency is not a bottleneck.

HermiTux performs similarly to Linux: the average difference between HermiTux and Linux runtime over all benchmarks is 2.7% (including NPB/PARSEC/Python benchmarks not shown here for space reasons). The overhead observed for HermiTux is slightly higher for a few benchmarks (e.g., IS). The reason is the very short runtime of these 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 with 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.

Network Performance on x86-64. 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. On our x86-64 server, we ran Redis within the previously mentioned virtualization solutions. As Firecracker's network support is not yet on par with Qemu's, for the related VMs we report results for both hypervisors. We ran the redis-benchmark from the host with 20 parallel clients and a key size of 2 bytes to stress the system in an extremely latency-sensitive scenario. We also varied artificially the network latency between the client and the server using `tc`, with 0, 0.2, 1, and 5 additional ms of latency, corresponding to various LAN setups representative of a certain class of Redis deployments.

The results are presented on Figure 5. Without any added network latency this experiment represents an extremely latency-sensitive scenario and HermiTux is slower than the competitors. It is on average 2.47x (GET) and 2.76x (SET) slower than the competitors running on Qemu, as well as 1.34x (GET) and 1.13x (SET) slower when they run on Firecracker. This is due to multiple factors, particularly the un-optimized network driver and TCP/IP stack (LWIP) of HermiTux, combined with the lack of support for *virtio*. It is likely that given mature support for networking, HermiTux

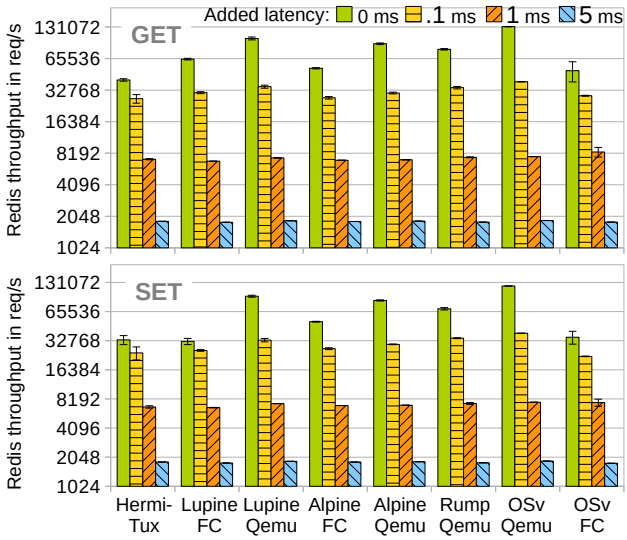

Fig. 5. Redis performance. Note the logarithmic scale on the Y-axis, and that we voluntarily start this axis at 1024 to better show differences. Results annotated "FC" are run on the Firecracker hypervisor.

would perform similarly to the competitors. Furthermore, by adding the network latency that is unavoidable in many deployment scenarios for Redis, the performance differences between the various systems decreases. For example, by adding a mere 0.2 ms, for GET HermiTux is only 1.2x slower than the competitors on Qemu. Starting from 1 ms of latency, that slowdown becomes negligible (¡ 3%). This shows that HermiTux is still viable in many networking scenarios. A further argument is developed in the next paragraph, showing our systems' efficiency in throughput-bound scenarios.

Edge Computing Benchmark on aarch64. Edge computing promises to bring the computational power that is currently centralized in the datacenter closer to the data sources (end user, IoT devices, etc.), with the aim to reduce service latencies. Edge nodes are expected to be much more heterogeneous than traditional servers, and can include in particular embedded devices [38] of various ISAs such as aarch64. Unikernels are good candidates for these multi-tenant and resource constrained environments, thus we evaluate the aarch64 port of HermiTux with an edge computing benchmark.

Real-time data analytics is one of the most important types of workloads at the edge [39]. We created an edge benchmark by adapting PARSEC's StreamCluster, an application performing online k-means clustering, to receive its input data from the network rather than from a file. With this benchmark we aim to reproduce a scenario typical of edge computing, in which some end user/IoT device produces data and sends it for processing to an edge node.

In this experiment a host server (the Potato aarch64 board) is representing the edge node and runs StreamCluster in a VM, HermiTux or a standard KVM VM. The client is represented by a separate machine that sends the data set to the VM, which in turns runs the clustering algorithm on that data. Both machines are connected through a local network. We define the execution time of one iteration as the time between the start of the data set transfer and the

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3122896

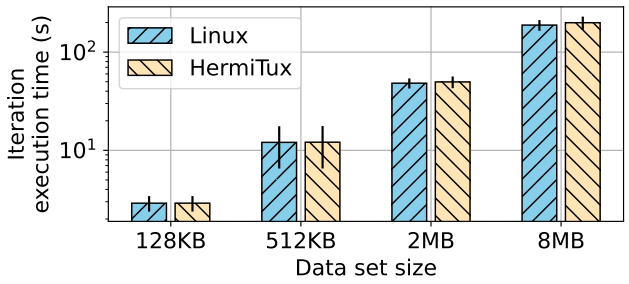

Fig. 6. Execution time of one iteration of the edge benchmark on aarch64 with varying input data set sizes.

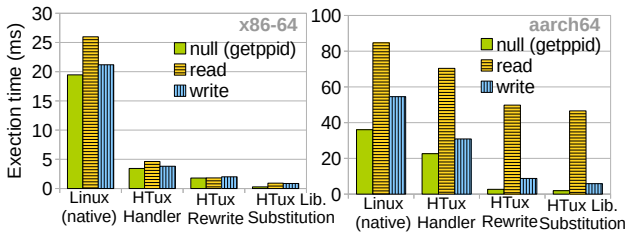

Fig. 7. LMbench system call latency on x86-64 (left) and aarch64 (right).

end of the clustering process on that data. Varying the data set size, we run 10 iterations for each size on each system (HermiTux/Linux KVM) and report the average execution time of an iteration. The results are presented on Figure 6.

As one can see, the performance of HermiTux is similar to Linux': the average difference over all data set sizes is 2.3%. Although we saw in the previous experiment (Redis) that HermiTux's network performance is somewhat inferior to Linux, we noticed that for this edge benchmark the data set network transfer only represents a very small part of the iteration execution time: indeed, because of the low processing power of the embedded board we use, the computation phase represents the major part of the work. Increasing the data set size does not change this fact as the network and processing latencies grow proportionally.

System Call Latency. We used LMbench3 [40] 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 7 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.

Concerning x86-64, the system call latency is on average 5.6x lower in HermiTux (handler) compared to Linux. It is due to multiple 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.

Concerning aarch64, HermiTux system call latency optimizations also bring significant speedups: binary rewriting system call invocations gives a latency reduction of 8.4x (for `null`) over using a regular system call handler. Substituting a unikernel aware C library brings a reduction of 11.7x. which is only 1.3x faster than binary rewriting. As previously mentioned, with aarch64, most system call invocations can be binary-rewritten with a simple function call or branch operation without the need for additional operations such as in x86-64. As a result, the speedups for libc substitution and binary rewriting are relatively similar in aarch64. Overall, the absolute latencies are higher on aarch64 due to the low processing power of the embedded board compared to the x86-64 server. The relatively high latency for the `read` system call can be explained by the fact that LMbench uses `/dev/zero` as target for the file operations. Reading from this file corresponds to zeroing the user buffer passed to `read`, a much more expensive operation (`memset`) than writing to this file which corresponds to a no-op. The difference between `read` and `write` is higher on aarch64 due to a difference in the (machine-dependent) throughput of `/dev/zero`.

We ran additional experiments not shown here for space reasons. We also evaluated HermiTux's multi-threading support running the OpenMP version of NPB CG/LU/MG and observed that performance was similar to Linux. We also ran a filesystem (Postmark) and a database (SQLite) benchmark and observed that HermiTux's numbers were on par with Linux and Docker. Finally, we validated our system's support for checkpoint/restart/migration by checkpointing and restarting the NPB benchmarks.

Overall, these results 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.

6 Related Works

Rumprun [9] and OSv [10] 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 [10] 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 [1], 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). Still, in absolute, it is unlikely that a kernel such as Linux can achieve the same degrees of lightweightness and

attack surface reduction compared to unikernels built from scratch such as HermiTux.

Lupine [11] is a unikernel version of Linux that reduces kernel size through configuration and eliminating the user/kernel boundary with the Kernel Mode Linux patch. Although it claims binary compatibility, it is important to note that contrary to HermiTux that is binary compatible at the system call level, Lupine compatibility is achieved at the standard C library level through a dynamic loader and a modified version of Musl Libc. As a result, contrary to HermiTux, with Lupine some unikernels benefits (such as fast system calls) cannot be achieved for programs that do not dynamically link against Musl such as static binaries. UKL [41] is another unikernel-version of Linux, however it is still under development. As our experimental comparison with Lupine shows, it is unlikely that even a heavily shrinked down version of large monolithic OS such as Linux can achieve the same degree of lightweightness as a unikernel built from scratch such as HermiTux.

Graphene [13] is a LibOS running on top of Linux, capable of executing unmodified, multi-process applications. Graphene's security can be enhanced with Intel SGX [14], but this involves significant overhead (up to 2x). While binary compatibility comes for free in containers and in some software LibOSes such as Graphene, 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 [1], [18], which comes at a very low performance overhead. Google proposes gVisor [15], a Go framework addressing containers' security concerns by providing some degree of software isolation through system call filtering/interposition. This frameworks comes at a non-negligible performance overhead [42], and is not able to reach the same level of isolation provided by the VMs used in the context of unikernels [43].

Dune [44] 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 [22] on x86-64 classify system calls by popularity. Such knowledge 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.

Finally, contrary to HermiTux, some of the systems referenced here (OSv, LightVM, X-Containers, Graphene-SGX, Lupine) only support a single ISA, x86-64.

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 grants N00014-16-1-2104, N00014-16-1-2711, and N00014-16-1-2818; BMBF grant 01IH16010C; and EPSRC grant EP/V012134/1. Hugo Lefeuvre is partly supported by NEC labs Europe.

References

[1] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 218–233. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132763

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

[3] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 459–473. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616491

[4] B. Duncan, A. Happe, and A. Bratterud, "Enterprise iot security and scalability: how unikernels can improve the status quo," in *IEEE/ACM 9th International Conference on Utility and Cloud Computing*, ser. UUC 2016. IEEE, 2016, pp. 292–297.

[5] S. Lankes, S. Pickartz, and J. Breitbart, "Hermitcore: a unikernel for extreme scale computing," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS 2016. ACM, 2016.

[6] "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.

[7] S. Kuenzer, V.-A. Badoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu, C. Raducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, "Unikraft: Fast, specialized unikernels the easy way," ser. EuroSys'21. New York, NY, USA: ACM, 2021.

[8] "Unikernels are secure," 2017, https://news.ycombinator.com/item?id=14736909. Online, accessed 11/27/2017.

[9] A. Kantee and J. Cormack, "Rump kernels no os? no problem!" *USENIX; login: magazine*, 2014.

[10] A. Kivity, D. L. G. Costa, and P. Enberg, "Os v - optimizing the operating system for virtual machines," in *Proceedings of the 2014 USENIX Annual Technical Conference*, ser. ATC'14, 2014, p. 61.

[11] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[12] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels." in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 559–573.

[13] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys'14. ACM, 2014, p. 9.

[14] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC 2017, 2017, p. 8.

[15] Google, "Gvisor github webpage," 2018, https://github.com/google/gvisor, Online, accessed 05/03/2018.

[16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx." in *OSDI*, vol. 16, 2016, pp. 689–703.

[17] I. Corp., "Intel clear containers," 2018, https://clearlinux.org/documentation/clear-containers. Online, accessed 08/04/2018.

[18] R. Pavlicek, "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.

[19] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," 2017.

[20] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling." in *USENIX Security Symposium*, 2015, pp. 627–642.

[21] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface," *AMD64 Architecture Processor Supplement, Draft v0*, vol. 99, 2013.

[22] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "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, 2016, p. 16.

[23] Y. Zhang, J. Crowcroft, D. Li, C. Zhang, H. Li, Y. Wang, K. Yu, Y. Xiong, and G. Chen, "Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018.

[24] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017, pp. 217–233.

[25] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the linux symposium*. Citeseer, 2009, pp. 19–28.

[26] W. Dietz and V. Adve, "Software multiplexing: share your libraries and statically link them too," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 154, 2018.

[27] C. S. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "Slinky: Static linking reloaded." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 309–322.

[28] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "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*, ser. FEAST, 2017.

[29] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 247–260. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352618

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[31] C. C. Williams and J. K. Hollingsworth, "Interactive binary instrumentation," in *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, 2004.

[32] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[34] V. Nitu, P. Olivier, A. Tchana, D. Chiba, A. Barbalace, D. Hagimont, and B. Ravindran, "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*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050758

[35] C. Lattner and V. Adve, "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, 2004, p. 75.

[36] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, ser. SPRO'15, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[37] Micropython Contributors, "Micropython webpage," 2018, https://micropython.org/, Online, accessed 08/05/2018.

[38] R. Pettersen, H. D. Johansen, and D. Johansen, "Secure edge computing with arm trustzone." in *IoTBDS*, 2017, pp. 102–109.

[39] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[40] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[41] A. Raza, "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.

[42] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 121–135.

[43] H. Fingler, A. Akshintala, and C. J. Rossbach, "Usetl: Unikernels for serverless extract transform and load why should you settle for less?" in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2019, pp. 23–30.

[44] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged cpu features." in *Osdi*, vol. 12, 2012, pp. 335–348.

Pierre Olivier received the BS and MS degrees from the University of Western Brittany, Brest, France, in 2009 and 2011, and the PhD degree from the University of South Brittany, Lorient, France, in 2014. He was a Postdoc from 2015 to 2018 and a Research Assistant Professor from 2018 to 2019 at Virginia Tech, US, before joining the University of Manchester, UK, as a lecturer. His research interests include many areas of systems software.

Hugo Lefeuvre received the BS from Karlsruhe Institute of Technology, Germany, in 2020. He is working toward the PhD degree in computer systems at the University of Manchester, UK. His research interests include, among others, systems software, security, and networking.

Daniel Chiba received his MS degree in Computer Engineering from Virginia Tech in 2018, where his research centered on virtualization and unikernels. He currently works in the graphics software team at Qualcomm.

Stefan Lankes received a conferral of a doctorate from the RWTH Aachen University. Between 2007 and 2017, he was academic councilor at Chair for Operation Systems at the RWTH Aachen University. Since 2017, he is working as Academic Director at the Institute for Automation of Complex Power Systems, RWTH Aachen University. His research interests include operating systems, cloud computing and high performance computing.

Changwoo Min is an Assistant Professor of the Electrical and Computer Engineering Department at Virginia Tech, where his research focuses on many-core scalability and concurrency of in-memory and non-volatile memory systems. His prior research includes operating systems, storage systems, database systems, and system security. Before joining Virginia Tech in 2017, he was a research scientist in Computer Science at Georgia Institute of Technology. He received his Ph.D. degree from Sungkyunkwan University in 2014. Before starting his Ph.D., he developed various software products, including Linux-based mobile platform (Tizen), Java virtual machine (J9), and desktop operating system (OS/2) in Samsung Electronics and IBM Korea.

Binoy Ravindran is a professor of electrical and computer engineering at Virginia Tech, where he leads the Systems Software Research Group which conducts research on distributed systems, operating systems, virtualization, compilers, concurrency, and verification. His group has published more than 290 papers in these spaces, including eight best paper awards and nominations. Several of his group's results have been transitioned to the US DOD, in particular, the Navy. He has mentored six research faculty members, 14 postdoctoral scholars, and 18 PhD students, ten of whom currently hold tenured or tenure-track faculty positions. He is an ACM distinguished scientist, a former office of naval research faculty fellow, and serves or has served on the editorial boards of IEEE TC, IEEE TPDS, ACM TECS, IEEE D&T, and IEEE TSUSC.