H2020-ICT-2018-2-825377

UNICORE

UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments

Horizon 2020 - Research and Innovation Framework Programme

D3.3 API, Library and Security Primitives Implementation - Initial

Due date of deliverable: June 30, 2020

Actual submission date: June 30, 2020

Start date of project	1 January 2019
Duration	36 months
Lead contractor for this deliverable	University POLITEHNICA of Bucharest (UPB)
Version	1.0
Confidentiality status	”Public”

Abstract

The goal of the EU-funded UNICORE project is to develop a common code-base and toolchain that will enable software developers to rapidly create secure, portable, scalable, high-performance solutions starting from existing applications. The key to this is to compile an application into very light-weight virtual machines – known as unikernels – where there is no traditional operating system, only the specific bits of operating system functionality that the application needs. The resulting unikernels can then be deployed and run on standard high-volume servers or cloud computing infrastructure.

In the deliverable we provide the initial implementation of the UNICORE APIs, along with an initial set of libraries. This set supports multiple applications (nginx, Redis, SQLite) available for the project use cases. We recall the overall design of UNICORE APIs together with a list of external libraries and applications. External libraries and applications are linked to required UNICORE APIs to create specialized (small and fast) unikernel images. We also describe the initial implementation of security and safety primitives.

Target Audience

The target audience for this document is **public**.

Impressum

Full project title	UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments
Title of the workpackage	3.3 API, Library and Security Primitives Implementation - Initial
Editor	University POLITEHNICA of Bucharest (UPB)
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2020 Participants in project UNICORE

Executive Summary

UNICORE uses modularity to enable specialization, splitting OS functionality into components that only communicate across well-defined API boundaries. The goal is to obtain performance via careful API design and static linking, rather than short-circuiting API boundaries for performance.

In order to create specialized unikernel images, UNICORE consists of micro-librariers, exposing an API. UNICORE APIs or core APIs provide OS-like functionality (file system, memory management, scheduling, platform specifics). External libraries are rich libraries providing required support for actual applications to run.

Micro-libraries (UNCORE APIs and external libraries) are tied together with application code to create specialized unikernel images to run on different platform.

Current APIs provided functionality enabling well known applications such as Nginx, Redis and SQLite to work. Moreover, popular interpreters are provided for programming languages such as Python, Lua, Ruby. Support for these applications and languages rely on common libraries (such as OpenSSL or Musl libc) or specific runtime libraries that are currently supported in UNICORE. Each addition of a new library extends the list applications that can be built and run as an UNICORE image.

Supported platforms (linuxu, KVM, Xen) rely on micro-libraries themselves, called platform APIs. The UNICORE build system provides the necessary options to target a unikernel image for a specific platform and architecture.

UNICORE components modularity provide specialization of unikernel builds which in turn provides performance with respect to image size, boot time and execution time. Measurements show either comparable or superior results for UNICORE application builds against other unikernels or general-purpose OSes.

With their reduced size, unikernel images are potentially more secure. Nevertheless, security and safety features are added in UNICORE, such as secure allocators and memory randomization, further increasing the appeal of UNICORE a security-focused solution.

List of Authors

Authors	Răzvan Deaconescu and Costin Raiciu (UPB), Felipe Huici and Simon Kuenzer (NEC), Gaulthier Gain and Cyril Soldani (ULiège), Cristiano Giuffrida and Herbert Bos (VUA)
Participants	UPB, NEC, ULiège, VUA
Work-package	WP3 - Core Implementation
Security	PUBLIC
Nature	R
Version	1.0
Total number of pages	35

Contents

List of Figures

List of Tables

1 Introduction

UNICORE uses modularity to enable specialization, splitting OS functionality into components that only communicate across well-defined API boundaries. The goal is to obtain performance via careful API design and static linking, rather than short-circuiting API boundaries for performance.
In order to create specialized unikernel images, two important components are provided:

- **Micro-libraries and pools**: Micro-libraries (micro-libs, for short) are software components which implement one of the core UNICORE APIs, where all libraries in a *pool* implement the same API and are thus interchangeable. In addition, micro-libraries can provide functionality from external library projects (*e.g.* OpenSSL, musl, Protobuf [1], *etc.*), applications (*e.g.* SQLite, Redis, BIND, Memcached, *etc.*), or even platforms (*e.g.* SoloFive, AmazonFirecracker, Raspberry Pi 3). Micro-libraries can be arbitrarily small (*e.g.* a library with basic boot code), or large (*e.g.* musl/libc support).

- **Build toolchain**: This provides a KConfig-based menu for users to select which micro-libraries to use in an application build, for them to select which platform(s) and CPU architectures to target, and how to configure each of the individual micro-libs (if desired). The build system then takes care of compiling all of the micro-libs, linking them, and producing one binary per selected platform.

Micro-libraries (UNICORE APIs and external libraries) are tied together with application code to create specialized unikernel images to run on different platform. The overall view of software components used is shown in Figure 1.1.
Micro-libraries are organized into two large groups: core or *internal* libraries, and *external* ones. Core libraries (UNICORE API) generally provide functionality typically found in operating systems. Such libraries are grouped into categories such as memory allocators, schedulers, filesystems, network stacks and drivers, among others. Core libraries are part of the main repository which also contains the build tool and configuration menu.
In contrast, external libraries consist of existing code not specifically meant for the core OS features; this includes standard libraries such as a libc or OpenSSL, but also run-times like Python. In support of modularity, external libraries are also separate repositories, allowing for the development of components without having to modify the main unikernel repository.
Chapter 2 presents UNICORE APIs (i.e. core libraries). Chapter 3 presents external libraries and applications supported in UNICORE. Chapter 4 shows security and safety primitives. Chapter 5 concludes.

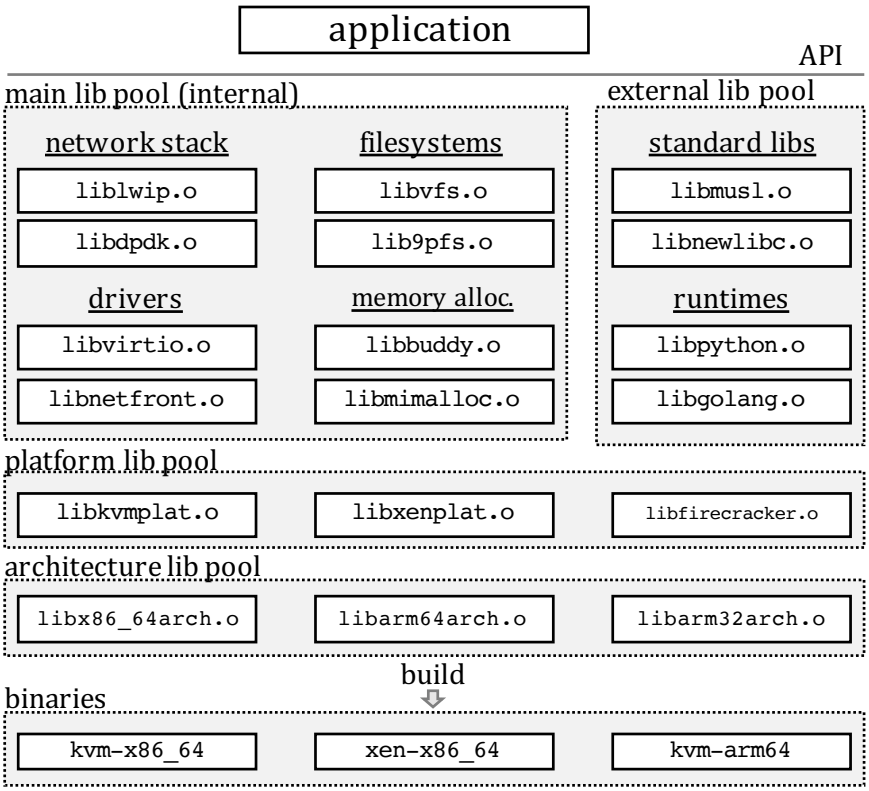

Figure 1.1: UNICORE APIs architecture: all components are micro-libraries

2 UNICORE APIs

UNICORE APIs provided the essential functionality to create specialized unikernel images. The APIs provide functionality typically found in operating system kernels that can be selectively tied together with external libraries and applications to create a final binary file. The resulting file is run on a UNICORE supported platform, typically a hypervisor.

UNICORE APIs are implemented as micro-libraries, small pieces of software providing one functionality. As shown in Figure Fig. Figure 1.1 they fill the main library pool (i.e. core libraries).

An API consists of a header file defining the actual API as well as an implementation of some generic/compatibility functions, if any, that are common to all micro-libraries under a specific pool. This functionality is *also* a UNICORE micro-library, and is placed under `lib/uk[poolname]` (*e.g.* `lib/ukalloc` for the memory allocator API, `lib/uknetdev` for the network driver API). We then use the naming convention `uk[category][name]` (*e.g.* ukschedcoop) to name an actual library implementing an API.

The fact that these APIs are modular, pluggable micro-libraries drives the specialization benefit of unikernel images. Another consequence of UNICORE API's modularity is that applications can plug into UNICORE at different levels in its stack: for instance, for backwards compatibility and ease-of-porting reasons, we could plug in an application such as Memcached on top of a network stack and socket API or directly over the low-level network micro-library (i.e., entirely foregoing a network stack) for a UDP-based, performance-oriented Memcached specialized implementation. This is akin to some of the concepts outlined in [2], and we will present results from these mechanisms later on in this paper.

2.1 Core APIs

The core APIS fill roles in operating systems. A short description is shown in Table 2.1. We then present a description of each.

uktime is the time information API, mostly used to get the current time. Sleep-related functionality is provided in functions such as `sleep` and `nanosleep`. **uktime** relies on platform support to get the wall clock.

uktimeconv is a simple API to convert and get time duration (such as days in a month, whether there is leap year).

ukswrand uses software support to provide basic randomization features. It is essentially a pseudo-random number generator.

uksglist uses a scatter/gather list is a list of vectors with each describing the location and length of one contiguous physical memory region. Scatter gather lists are helpful for setting up I/O requests with devices, such as network devices.

ukdebug allows debugging of unikernel code, including other APIs.

ukboot is the bootloader API. It provides a unitary interface to boot a unikernel images, while its implemen-

API	Role
ukalloc	Abstraction for memory allocators
ukargparse	Simple argument parser
ukblkdev	Block driver interface
ukboot	Unikernel bootstrapping
ukbus	Abstraction for device buses
ukdebug	Debugging and tracing
uklibparam	Library arguments
ukmmap	mmap system call
uknetdev	Network driver interface
uksched	Abstraction for schedulers
uksglist	Scatter Gather List
ukswrand	Software random number generator
uktime	Time functions
uktimeconv	Time conversion functions
vfscore	VFS Core Interface

Table 2.1: UNICORE APIs

tation uses the underlying platform code. It provides features such as printing / logging, memory dumping and stack trace investigation.

ukargparse is a simple API to parse arguments passed as an array of strings, typically command line arguments for applications ported on top of the UNICORE API.

ukblkdev provides block device (persistent functionality) interface. It uses a queue-based implementation to manage I/O requests for block devices. As other APIs, it is a wrapper implementation with most of the implementation being part of the actual block device implementation. The block device implementation uses the API exposed through the structures `uk_blkdev`, `uk_blkdev_queue`, `uk_blkdev_ops` and others.

ukmmap is the memory mapping API, allowing for page table management and configuration of the runtime unikernel memory layout.

ukmmap exposes the typical OS memory mapping API in the functions `mmap`, `unmmap`, `remap`.

vfscore offers the interface for managing filesystems, typically found in the VFS (Virtual Filesystem Switch) framework in general OSes (Linux, FreeBSD). The actual filesystem implementation (such as 9fs or ramfs) is located in a different library. VFS configuration allows the selection of the actual filesystem implementation for the final image.

vfscore exposes functions that map to common file I/O system calls (`read`, `readdir`, `close`, `closedir`).

uksched is UNICORE's scheduling API. It allows for multiple schedulers to be present and active *within* one running instance. On creation, each thread is assigned a scheduler, allowing customization of threads

potentially using different schedulers (*e.g.* a real-time scheduler for the real-time part of an application, a co-operative scheduler for a performance critical part). On the other extreme, UNICORE builds may run with **no** scheduler, essentially running a task to completion from an interrupt handler in order to increase performance by eliminating scheduling overheads (and ill-effects of having stacked schedulers when running in a virtualized environment); we will show the effects of such an approach in the evaluation section.

uksched exposes the `uk_sched` and `uk_thread` data types as abstractions for a scheduler implementation (including scheduling queues and scheduling parameters) and thread implmentation. Common scheduler functionality is exposed as function pointers in the `uk_sched` structure (for example there is no exposed `uk_sched_yield` function; it's an internal function calling corresponding `uk_sched` function) or functions for managing threads (`uk_sched_thread_create`, `uk_sched_thread_switch` *etc.*).

uknetdev API: the networking sub-system aims to decouple (1) the device driver side (e.g., virtio-net, tapif, netfront) from (2) the network stack or low-level networking application (which we will call simply "application" in the following for ease of presentation).

Regarding the former, easily swapping network stacks is something that is not common in commodity OSes; instead, drivers are usually implemented for a particular network stack. The aim of this API is to decouple these two components in order to allow drivers to be reused across platforms.

For the latter, a networking application or network stack should be able to run unmodified on a different platform with different drivers. Because we are addressing a wide range of use cases, the API should not restrict any of them nor become a potential performance bottleneck for high performance workloads. We derived part of the design from Intel DPDK's `rte_netdev` API. However, because its focus is on high performance rather than efficient resource usage, we provide modified interfaces and allow applications to operate drivers in polling, interrupt-driven, or mixed mode.

In addition, `uknetdev` leaves memory management to the application while supporting high performance features like multiple queues, zero-copy I/O, and packet batching. We adopt DPDK's approach of letting the application fully operate and initialize the driver; drivers do not do run any initialization routine on their own. Instead, we provide API interfaces for applications to provide necessary information (e.g., supported number of queues and offloading features) so that the application code can specialize by picking the best set of driver properties and features. Drivers register their callbacks (e.g, send and receive) to a `uk_netdev` structure which the application then uses to call the driver routines.

ukalloc API is composed of three layers: (1) a POSIX compliant external API, (2) an internal allocation interface called `ukalloc`, and (3) one or more backend allocator implementations. The external interface is motivated by backward compatibility to facilitate the porting of existing applications to UNICORE. In the case of the C language, the external API is exposed by a modified standard library which can be nolibc (a minimal, UNICORE libc implementation), newlib or musl. The external allocation interface acts as a compatibility wrapper for the UNICORE-specific internal allocation interface, which in turn redirects allocation

requests to the appropriate allocator backend. The internal allocation interface therefore serves as a multiplexing facility that enables the presence of multiple memory allocation backends within the same unikernel. UNICORE supports four allocation backends: a buddy system, the Two-Level Segregated Fits (TLSF) real-time memory allocator, tinyalloc, and Mimalloc; further, there is an ongoing effort to port Oscar [3], a protection scheme based on page permissions.

UNICORE internal allocation interface exposes `uk_` prefixed versions of the standard POSIX allocation interface: `uk_malloc()`, `uk_calloc()`, etc. In contrast to POSIX, these functions require the caller to specify which allocation backend should be used to satisfy the request.

A view of the UNICORE APIs and their respective interfaces (functions and data structures) is shown in Table 2.2.

API	Data Structures	Functions
devfs	struct devinfo, struct devops, struct driver, struct device, struct partition_table_entry	device_create, device_open, device_close, device_read, device_ioctl, device_info
ukalloc	struct uk_alloc, struct metadata_ifpages	uk_alloc_register, uk_alloc_get_default, uk_alloc_set_default, uk_malloc_ifpages, uk_free_ifpages, uk_realloc_ifpages, uk_posix_memalign_ifpages, uk_calloc_compat, uk_memalign_compat, uk_realloc_compat, uk_palloc_compat, uk_pfree_compat
ukargparse		uk_argnparse
ukblkdev	struct uk_blkreq, struct uk_blkdev_conf, struct uk_blkdev_info, struct uk_blkdev_queue_info, struct uk_blkdev_queue_conf, struct uk_blkdev_ops, struct uk_blkdev_cap, struct uk_blkdev_event_handler, struct uk_blkdev_data, struct uk_blkdev, struct uk_blkdev_sync_io_request	uk_blkdev_drv_register, uk_blkdev_count, uk_blkdev_get, uk_blkdev_id_get, uk_blkdev_drv_name_get, uk_blkdev_state_get, uk_blkdev_get_info, uk_blkdev_configure, uk_blkdev_queue_get_info, uk_blkdev_queue_configure, uk_blkdev_start, uk_blkdev_queue_submit_one, uk_blkdev_queue_finish_reqs, uk_blkdev_sync_io, uk_blkdev_stop, uk_blkdev_queue_release, uk_blkdev_drv_unregister, uk_blkdev_unconfigure
ukboot	struct thread_main_arg	ukplat_entry_argp, ukplat_entry, main, uk_version
ukbus	struct uk_bus	uk_bus_count, _uk_bus_register, _uk_bus_unregister, uk_bus_list
ukdebug	struct uk_tracepoint_header, struct _vprint_console, struct out_dev	_uk_vprintd, _uk_printd, _uk_vprintk, _uk_printk, uk_hexdumpsn, uk_hexdumpf, uk_hexdumpd, _uk_hexdumpd, _uk_hexdumpk, _uk_asmdumpd, _uk_asmdumpk, uk_trace_buffer_free, uk_trace_buffer_writepracepoint_header, struct _vprint_console, struct out_dev
uklibparam	struct param_args, struct uk_param, struct uk_lib_section	uk_libparam_parse, _uk_libparam_lib_add
ukmmap	struct mmap_addr	mmap, munmap, mremap

uknetdev	struct uk_netbuf, struct uk_hwaddr, struct uk_netdev_info, struct uk_netdev_queue_info, struct uk_netdev_conf, struct uk_netdev_rxqueue_conf, struct uk_netdev_txqueue_conf, struct uk_netdev_ops, struct uk_netdev_event_handler, struct uk_netdev_data, struct uk_netdev	uk_netbuf_init_indir, uk_netbuf_alloc_indir, uk_netbuf_alloc_buf, uk_netbuf_prepare_buf, uk_netbuf_free_single, uk_netbuf_free, uk_netbuf_disconnect, uk_netbuf_connect, uk_netbuf_append, uk_netdev_drv_register, uk_netdev_count, uk_netdev_get, uk_netdev_id_get, uk_netdev_drv_name_get, uk_netdev_state_get, uk_netdev_info_get, uk_netdev_einfo_get, uk_netdev_rxq_info_get, uk_netdev_txq_info_get, uk_netdev_configure, uk_netdev_rxq_configure, uk_netdev_txq_configure, uk_netdev_start, uk_netdev_hwaddr_set, uk_netdev_hwaddr_get, uk_netdev_promiscuous_get, uk_netdev_promiscuous_set, uk_netdev_mtu_get, uk_netdev_mtu_set, uk_netdev_rxq_intr_enable, uk_netdev_rxq_intr_disable
uksched	struct uk_sched, struct uk_thread_attr, struct uk_thread, struct uk_waitq_entry	uk_sched_default_init, uk_sched_register, uk_sched_get_default, uk_sched_set_default, uk_sched_create, uk_sched_start, uk_sched_idle_init, uk_sched_thread_create, uk_sched_thread_destroy, uk_sched_thread_kill, uk_sched_thread_sleep, uk_sched_thread_exit, uk_thread_init, uk_thread_fini, uk_thread_exit, uk_thread_wait, uk_thread_detach, uk_thread_set_prio, uk_thread_get_prio, uk_thread_set_timeslice, uk_thread_get_timeslice, uk_thread_block_timeout, uk_thread_block, uk_thread_wake, uk_thread_attr_init, uk_thread_attr_fini, uk_thread_attr_set_detachstate, uk_thread_attr_get_detachstate, uk_thread_attr_set_prio, uk_thread_attr_get_prio, uk_thread_attr_set_timeslice, uk_thread_attr_get_timeslice
uksglist	struct uk_sglist_seg, struct uk_sglist, struct sgsave	uk_sglist_count, uk_sglist_alloc, uk_sglist_free, uk_sglist_append, uk_sglist_append_sglist, uk_sglist_build, uk_sglist_clone, uk_sglist_length, uk_sglist_split, uk_sglist_join, uk_sglist_slice, uk_sglist_append_netbuf
ukswrand	struct uk_swrand, struct uk_swrand	getrandom, uk_swrand_def, uk_swrand_init_r, uk_swrand_randr_r, uk_swrandr_gen_seed32
uktimeconv	struct uktimeconv_bmkclock	uktimeconv_days_in_month, uktimeconv_is_leap_year, uktimeconv_bmkclock_to_nsec
uktime	struct timeval, struct timespec, struct itimerval, struct timezone, struct tm, struct itimerspec, struct utimbuf	clock_getres, clock_gettime, clock_settime, gettimeofday, nanosleep, setitimer, sleep, timegm, times, usleep, utime, timer_create, timer_delete, timer_settime, timer_gettime, timer_getoverrun

vfscore	struct vfscore_file, struct dentry, struct mount, struct vfscore_fs_type, struct vfsops, struct vnode, struct vattr, struct vnops, struct uio, struct task, struct fdtable, struct pipe_buf, struct pipe_file	vfscore_alloc_fd, vfscore_put_fd, vfscore_install_fd, vfscore_get_file, vfscore_put_file, mount, vfscore_nullop, vfscore_release_mp_dentries, vfscore_vget, vfscore_uiomove, vfscore_vop_nullop, vfscore_vop_einval, vfscore_vop_eperm, vfscore_vop_erofs, open, creat, write, uk_syscall_e_write, uk_syscall_r_write, close, read, uk_syscall_e_read, uk_syscall_r_read, mkdir, fsync, fstat, uk_syscall_e_fstat, uk_syscall_r_fstat, flock, fhold, fdrop, fget, ftruncate, stat, chmod, fchmod, fchown, dup, dup2, dup3, sync, vfscore_mount_dump, umount, umount2, link, unlink, getcwd, chown, chroot, chdir, fstatat, statfs, lstat, lchown, openat, opendir, readdir, readdir_r, readdir64, closedir, pread, pwrite, pwritev, readv, uk_syscall_e_readv, uk_syscall_r_readv, writev, truncate, mknod, preadv, ioctl, fdatasync, fdopendir, dirfd, rewinddir, telldir, seekdir, rmdir, fchdir, symlink, statvfs, fstatvfs, access, faccessat, readlink, uk_syscall_e_readlink, uk_syscall_r_readlink, fallocate, lseek, uk_syscall_e_writev, uk_syscall_r_writev, umask, dentry_alloc, dentry_init, dentry_lookup, dentry_move, dentry_remove, drele, vrele, vput, vref, vflush, dref, fcntl, readdir_r, readdir64_r, fstatfs, eaccess, euidaccess, rename, __xmknod, __xstat, __lxstat, vn_access, vn_add_name, vn_del_name, vn_lock, vn_lookup, vn_setmode, vn_settimes, vn_stat, vn_unlock, vfs_busy, pipe, pipe2, mkfifo, futimes, futimesat, utimensat, futimens, utimes, lutimes, posix_fadvise, scandir

Table 2.2: Interfaces of UNICORE APIs

2.2 Platform APIs

Following the principle of everything-as-a-library, support for different platforms is also implemented as individual libraries; this means that code for QEMU/KVM, SoloFive/KVM, AmazonFirecracker, Xen containers and Linux user-space are all micro-libraries in their own right. This is also the case for CPU-specific code (`x86_64`, `arm32` and `arm64`) and drivers (*e.g.* virtio, netfront, *etc.*).

Platform APIs provide the required support for running UNICORE applications on existing infrastructures. Built-in support is currently provided for KVM and Xen and external support is provided for AmazonFirecraker and SoloFive. As a test case or use cases withing a running Linux platform instance, there is the linuxu API platform. Platform API for the final image is configured via the build system.

KVM support allows unikernel images to run on top of x86 and ARM CPUs on the KVM hypervisor. Similarly, Xen hypervisors on x86 and ARM are able to run UNICORE images. Specific memory layouts, device address and bootloaders are implemented for each platform in the final unikernel image.

The linuxu UNICORE platform creates a unikernel image as a classical ELF static binary. The ELF binary is

loaded using the operating system (Linux loader) as a normal user process. It interacts via system calls with the Linux kernel; the Linux kernel takes the role of the hypervisor used by Xen and KVM.

3 Libraries and Applications

External libraries are similar to UNICORE APIs, with a modular design. They live outside the main tree and are adapted (i.e. ported) from existing source code to facilitate the running of existing applications. Below we present libraries and applications currently supported by UNICORE APIs.

3.1 Libraries

Table 3.1 lists the libraries ported on top of UNICORE APIs. Some of them are standalone general-purpose libraries, while others are helper libraries for applications. Most applications (shown in Section 3.3) use a specific library; *e.g.* **nginx** uses the **lib-nginx** library, and **redis** uses the **lib-redis** library. Note that some libraries aren't fully ported and may require extensive testing to ensure proper functionality of provided interface.

3.2 Porting Libraries

Porting external libraries and applications relies the target library's build system, instead of having to write the project MakefileUK and related files. With this approach, the target build system is used to generate object or archive files, and the outputs are linked into the final linking step.

Pitfalls with this approach may be that the (external) build system may build against glibc instead of UNICORE's own standard C library (newlib), or that the target application may rely on dynamic libraries.

To support musl, which depends on the availability of Linux syscalls, we extended the internal UNICORE `syscall_shim` library to also generate a system call interface at standard C library level. In this way, we can link to system call implementations directly when compiling application source files natively.

In Table table 3.2, we show results of trying this approach on a number of different applications and libraries when building against musl and newlib. As can be seen, this approach is not effective with newlib ("std" column), but it *is* with musl: most libraries build fully automatically. For those that do not, the reason has to do with the use of glibc-specific symbols. To address this, we build a glibc compatibility layer based on a series of musl patches [4] and 20 other stubs we add by hand. With this in place, as shown in the table ("compat layer" column), this layer allows for almost all libraries and applications to compile and link. For musl that is good news: as long as the syscalls needed for the applications to work are implemented, then the image will run successfully (for newlib the stubs would have to be implemented). Note that related work [5, 6] reports that in the region of 100 syscalls are enough to run a rich set of mainstream application.

In all, this approach is quite effective in porting the library or application itself, as long as their dependencies can be met by UNICORE APIs.

3.3 Applications

Applications currently working with UNICORE are either simple test applications such as **app-hello** or **app-httpreply**. They are shown in Table 3.3 together with the list of libraries they depend upon.

Library	Role
lib-pcre	port of the Perl Compatible Regular Expressions library
lib-compiler-rt	port of compiler-rt, a runtime library
lib-newlib	port of newlib, a C standard library
lib-openssl	port of the OpenSSL libraries
lib-intel-intrinsics	port of Intel intrinsics
lib-libuv	port of the libuv library, for asynchronous I/O
lib-fft2d	port of the fft2d library, 2 fast Fourier transform library
lib-eigen	port of eigen, C++ template library for linear algebra
lib-pthreadpool	port of pthreadpool, pthread-based thread pool for C/C++
lib-nnpack	port of nnpack, acceleration package for neural networks
lib-zydis	port of the Zydis disassembler library
lib-lwip	port of the lwip network stack
lib-libelf	port of libelf from the ELF toolchain
lib-gemmlowp	port of Google's gemmlowp library
lib-farmhash	port of Google's FarmHash library
lib-flatbuffers	port of Google's FlatBuffers library
lib-libcxx	port of the C++ standard library
lib-nginx	port of NGINX
lib-pybind11	port of the pybind11 library
lib-redis	port of Redis in-memory data structure store
lib-arm-intrinsics	port of ARM intrinsics
lib-micropython	port of Micropython, Python for embedded devices
lib-ruby	port of Ruby
lib-sqlite	port of SQLite
lib-mbedtls	port of the Mbed TLS library
lib-python3	port of Python 3
lib-click	port of the Click modular router
lib-libgo	port of the Go language
lib-libunwind	port of libunwind, an unwinder library
lib-libcxxabi	port of the C++ ABI (read-only mirror)
lib-lzma	port of lzma compression
lib-duktape	port of duktape/JavaScript
lib-lua	port of the Lua language
lib-dnnl	port of Intel Math Kernel Library for deep neural networks
lib-boost	port of the boost library
lib-gcc	port of the GNU Compiler Collection libraries
lib-wamr	port of WAMR, Intel's WebAssembly Micro Runtime
lib-googletest	port of the Google testing and mocking framework
lib-psimd	port of psimd, portable SIMD intrinsics
lib-libfxdiv	port of fxdiv, a library for division via fixed-point multiplication by inverse

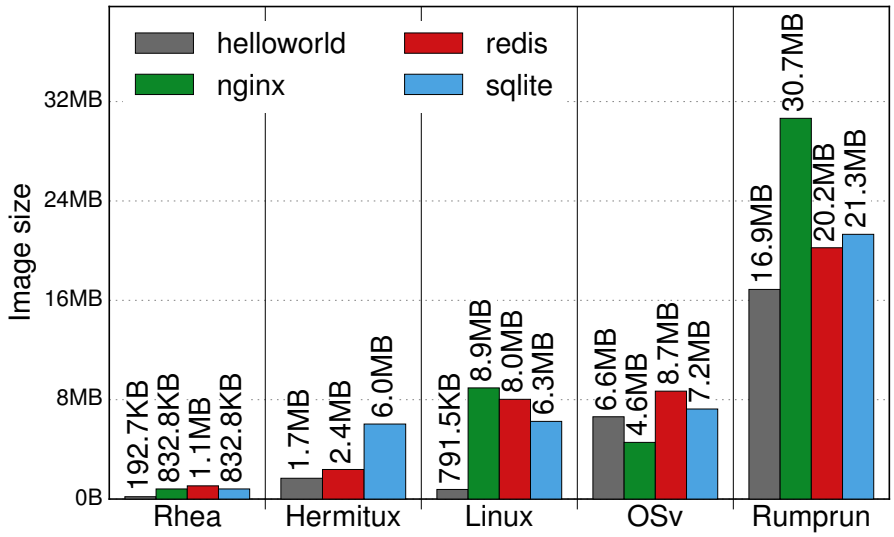

Figure 3.1: Image sizes for representative applications with UNICORE APIs and other OSes.

app-hello implements a simple "Hello, world!" printing to test basic functionality of UNICORE APIs. Likewise, **app-httpreply** tests UNICORE APIs including the lwip (Lightweight IP) library. **app-lua**, **app-micropython**, **app-python3**, **app-ruby** implement support for the Lua, Micropython, Python3 and Ruby interpreters. **app-nginx** implements the Nginx web server. **app-redis** implements the Redis key-value database. **app-sqlite** implements the SQLite in-file database engine. **app-wamr** implements the WebAssembly Micro Runtime.

3.4 Measurements

To highlight the advantage of UNICORE for reducing the binary image size when on disk, and boot-time and memory footprint at runtime, We built binaries for some of above apps using leading unikernels, Linux and UNICORE APIs. We show image sizes in Figure 3.1. UNICORE-based builds are named Rhea from the current version name. Image sizes range ranging from 100KB (HelloWorld) to 1.1MB (Redis), one or two order of magnitude smaller than other unikernels, and one order of magnitude smaller than the associated Linux binary with statically linked standard C library.

Build times are also small (Figure 3.2): UNICORE-based builds have competitive build times of a few minutes on first build (which includes dependencies) and under a minute for application-only builds. The build times are comparable with Rump and Linux, and significantly faster than OSv and HermiTux (where dependencies' build time dominates).

The advantage of UNICORE components stems from their modular nature, which differs from the monolithic approach taken by other unikernels, and from their ability to employ link-time size optimizations that are not possible when a syscall indirection layer is used. UNICORE image sizes are halved by gcc's dead-code elimination and link-time optimization (see Figure 3.3).

Small image sizes are not only useful for minimizing disk storage, but also to enable quick boot times for VMs based on those images. LightVM [7] has shown that it is possible to boot a no-op VM in around 2ms,

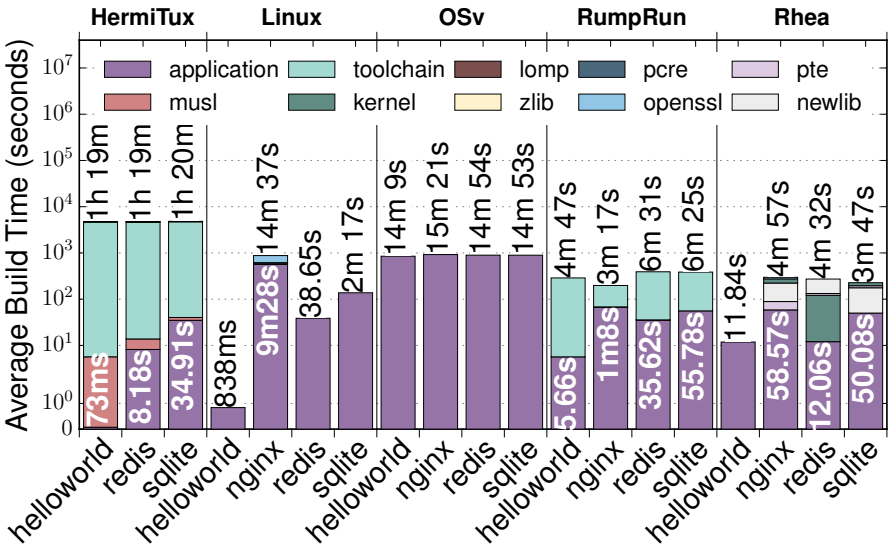

Figure 3.2: Time taken to build a unikernel application.

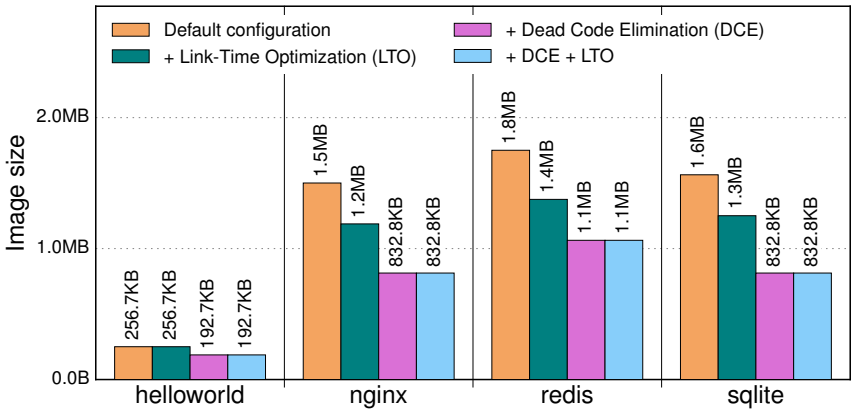

Figure 3.3: Image sizes of UNICORE-based applications

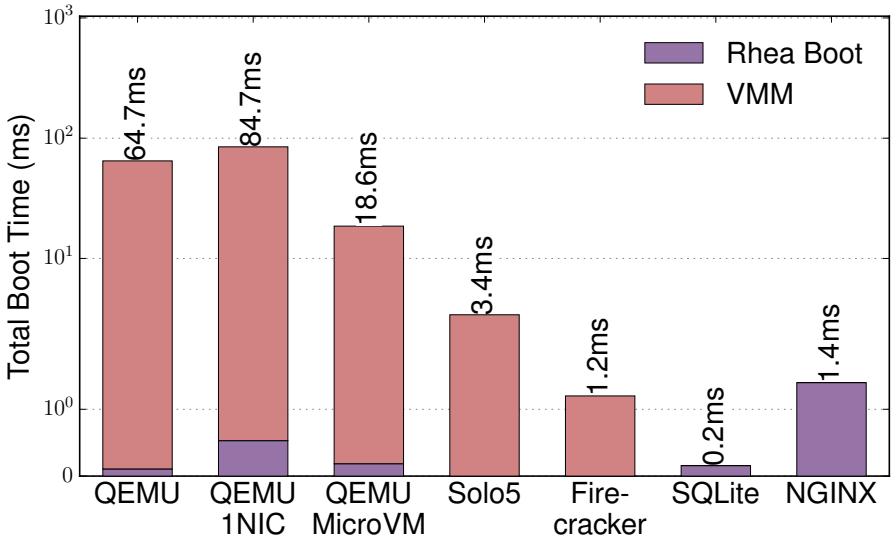

Figure 3.4: Boot time for UNICORE-based images with different virtual machine monitors

with a heavily optimized Xen tool stack. We use standard virtualization tool stacks instead, and wish to understand how quickly UNICORE-based VMs can boot. When running experiments, we measure both the time taken by the VMM (e.g. Firecracker or QEMU) and the boot time of the VM, measured from when the first guest instruction is run until main is invoked.

The results are shown in Figure 3.4 showing how long a *Hello world* needs to boot with different VMMs. Note that UNICORE images boot time ranges from tens to hundreds of microseconds when the VM has no devices, up to 1ms when the VM has one networking interface; other unikernels boot similarly fast, so we omit their results. This compares to around 200ms for Alpine Linux running on Firecracker, showing the clear benefit of having a small VM image.

Overall, the total VM boot time is dominated by the VMM, with Firecracker and Solo5 being the fastest (1-3ms), QEMU Microvm around 25ms and QEMU the slowest at around 65ms. We further plot UNICORE-based VM boot times for SQLite and Nginx, which add 1-2 ms to the total boot. These results show that UNICORE can be readily used in scenarios where just-in-time instantiation of VMs is needed.

	musl			newlib		
	Size (MB)	std	compat. layer	Size (MB)	std	compat layer
lib-axtls	0.336	✗	✓	0.432	✗	✓
lib-bzip2	0.296	✓	✓	0.364	✗	✓
lib-c-ares	0.304	✓	✓	0.432	✗	✓
lib-duktape	0.700	✓	✓	0.772	✗	✓
lib-farmhash	0.232	✓	✓	0.276	✓	✓
lib-fft2d	0.356	✓	✓	0.396	✗	✓
lib-helloworld	0.232	✓	✓	0.256	✓	✓
lib-libucontext	0.232	✓	✓	0.276	✓	✓
lib-libunwind	0.232	✓	✓	0.276	✗	✓
lib-lighttpd	0.796	✗	✓	0.916	✗	✓
lib-lighttpreply	0.256	✓	✓	0.296	✓	✓
lib-memcached	0.524	✓	✓	-	✗	✗
lib-micropython	0.527	✓	✓	0.628	✗	✓
lib-nginx	1.13	✗	✓	1.20	✗	✓
lib-open62541	0.248	✗	✓	0.804	✗	✓
lib-openssl	2.98	✗	✓	3.01	✗	✓
lib-pcre	0.344	✓	✓	0.380	✗	✓
lib-python	4.75	✗	✓	4.81	✗	✓
lib-redis-client	0.640	✗	✓	0.801	✗	✓
lib-redis-server	1.26	✗	✓	1.42	✗	✓
lib-ruby	6.84	✗	✓	6.93	✗	✓
lib-sqlite	1.22	✓	✓	1.31	✗	✓
lib-zlib	0.348	✓	✓	0.404	✗	✓
lib-zydis	0.276	✓	✓	0.328	✗	✓

Table 3.2: Porting on externally-built archives using musl and newlib.

Application	Required Libraries
app-hello	
app-httpreply	lwip
app-lua	newlib, lua
app-micropython	newlib, lwip, micropython
app-nginx	pthread-embedded, newlib, lwip, nginx
app-python3	pthread-embedded, lwip, zlib, libuuid, newlib, python3
app-redis	pthread-embedded, newlib, lwip, redis
app-ruby	pthread-embedded, libunwind, compiler-rt, libcxx, libcxxabi, newlib, lwip, ruby
app-sqlite	pthread-embedded, newlib, sqlite
app-wamr	wamr, pthread-embedded, lwip, newlib

Table 3.3: Applications Ported on UNICORE

4 Security and Safety Primitives

T3.2 and T3.3 focus on adding support for security / safety and deterministic execution in UNICORE. In this section we highlight features present or in development with respect to security and deterministic execution.

4.1 Security and Isolation Primitives

Memory errors rank highly among the most dangerous vulnerabilities in C/C++ programs. Temporal errors, such as use-after-free, are particularly insidious, as detecting them efficiently during the execution is challenging. To efficiently detect all such use-after-free issues, we built Wilde, a secure memory allocator that detects any attempt to use a pointer to an object that has already been freed. In particular, we ensure upon allocation time that objects are mapped to their own virtual address page(s), even if they are sharing the same physical page. When the program frees the object, we invalidate the virtual page so that any attempt to dereference a pointer to it triggers a segmentation fault. Unfortunately, doing so in a process on a monolithic operating system such as Linux is expensive, since the page manipulations require the program to trap into the kernel. Instead, we are using virtualization technology to allow page table manipulation directly, without requiring expensive mode switches.

4.1.1 Use-after-free (UAF) Vulnerabilities

A use-after-free vulnerability exists whenever a program allows a pointer to a previously deallocated block of memory (that is, a dangling pointer) to be dereferenced. After deallocation, the allocator eventually reuses memory for new objects. As a result, the dangling pointer and the new live pointer refer to the same block of memory. This means that writes to one object will corrupt the other and reads may either leak sensitive data from the other object or cause data from the other object to be misinterpreted.

Use-after-free errors are both common and very difficult to prevent. They frequently happen easily in more complex environments. For instance, consider the example in Listing 4.1, where the program tries to log an error, but in doing so, the erorr handler accesses `line` which had already been freed.

```
1  oid read_str() {
2      ...
3      getline(&line, &length, stdin);
4      ...
5      if (error) {
6          free(line);
7          aborted = 1;
8      }
9      ...
10     if (aborted) {
11         log_printf("Line contained error: %s\n", line);
12     }
13 }
```

Listing 4.1: Use-after-free error

Use-after-free errors are particularly common in C++ programs in what is known as VTable hijacking. In C++, virtual method tables (VTables) help select an implementation of a virtual method at runtime. If a program creates a dangling object pointer, an attacker can craft an input that causes the program to allocate blocks of memory and store fake VTable pointers into them, a process known as heap or stack spraying (or massaging in a more targeted form). Afterwards, calls to virtual methods use the fake VTable pointer supplied by the attacker, hijacking control flow. This example demonstrates how a simple and hard to find bug can lead to an attacker taking over control.

4.1.2 The Default Allocator

At the time of writing, the UNICORE memory model provides an identity mapping, i.e., the physical memory space is mapped one to one to virtual memory. For instance, if we consider the x86-64 KVM implementation, the basic layout consists of the top two levels of page tables containing a single page table entry each and the third level containing 512 page table entries for 2 MB pages (known as huge pages). In other words the full address space is 1 GB in size. The default allocator is essentially a basic buddy allocator for which security was emphatically not a design goal and provides memory allocations that are not just predictable, but also universally readable, writable, and executable.

4.1.3 The Wilde Allocator

Our design implements a fully external system level allocator running in an environment provided by UNICORE. All memory allocation and de-allocation requests made by the application are handled by the Wilde library which provides safe aliases for the memory which it in turn fetches from the default allocator.

A first simple improvement over the original allocator is to mark memory allocated with `malloc` non executable. A second improvement, targets use-after-free vulnerabilities and guarantees that virtual memory will not be reused (even if physical memory is) and that all objects will be mapped on separate virtual pages. Every time the program issues an allocation request, we relay the request to the underlying allocator, and then create an alias (a virtual page pointing to the same memory) which we return to the program. Upon a free, we destroy the alias and guarantee that this particular alias is never re-used. This guarantee only works because of the huge 48-bit address-space of the AMD64 architecture, which provides in 256Tb of virtual RAM space. We implemented Wilde as a library, which hooks into the allocator API and uses the `cr3` register to gather information about the current memory layout (ensuring that we needed no modifications to the UNICORE unikernel itself). Note that this is possible in systems such as UNICORE, since the library has ring-0 access to the page table data structures. To keep the attack surface to a minimum, we opted for storing metadata separate from the program-requested memory.

We protect memory by registering the appropriate access rights, which given the virtual memory mappings, amounts to setting some page table bits. In addition we added guard pages around every alias to make sure every overflow is caught. This means stack overflows cause a segfault rather than corrupt the machine state in unpredictable ways. Moreover, objects are allocated as close to the guard page above it as possible to increase

the probability of catching buffer overflow errors.

In a preliminary evaluation, we tested the Wilde allocator on a server with the AMD Ryzen 7 2700X processor, with 32 gigabyte of RAM, using a set of the performance benchmarks and found that the overhead compared to baseline UNICORE is always less than 20% in 33 out of 35 benchmarks, and 40 and 60% for the remaining two, respectively. We emphasize that the results are preliminary and we are still working on our allocator and its evaluation.

4.1.4 Crash Recovery

The Wilde allocator can detect arbitrary use-after-frees at run time and halt the execution to prevent exploitation. However, this strategy alone would translate every use-after-free exploitation attempt into a crash, ultimately leading to denial of service. To address this problem, we complemented Wilde with a crash recovery solution, which can also be used to automatically recover from other unrelated crashes.

Our solution is based on efficient checkpoint-restart techniques enabled by modern virtualization extensions. At a high level, we periodically checkpoint the state of the unikernel (memory and registers) according to a predetermined policy (e.g., every 1 second, at every `vmexit`, etc.). Then, when a crash is detected, we restart execution from the last checkpoint to bring the system back to a consistent state. Optionally, we can execute dedicated error-handling code to rule out the re-occurrence of the crash upon re-execution (e.g., rejecting problematic future inputs).

Checkpointing registers is inexpensive, but checkpointing memory pages can incur non-trivial overhead. However, on modern Intel processors with virtualization extensions, we can leverage hardware features such as Page Modification Logging (PML) to efficiently implement *memory checkpointing*. PML, originally designed for efficient virtual machine (VM) migration, can transparently log all the addresses of the memory pages dirtied over a predetermined execution interval. The log can then be accessed (and reset) by the hypervisor as necessary.

Our solution repurposes PML for incremental memory checkpointing, saving all the unikernel dirtied pages included in the log and resetting the log at every checkpoint. Starting from an initial full memory checkpoint of the unikernel, this checkpointing strategy is efficient and guarantees we can always restore the last checkpoint by walking the log and fetching the last available copy of each of the contained pages. We implemented a prototype of our design using the PML interface available in QEMU/KVM, which already handles the case of log overflows (i.e., triggering a `vmexit` and storing partial logs in a bitmap data structure). Our preliminary experimental results suggest our PML-based checkpointing implementation introduces negligible performance overhead when using moderate checkpointing frequencies (e.g., 1 second).

4.2 Memory Randomization Support

For additional protection against memory disclosure attacks, memory randomization support is being integrated in UNICORE, patches undergoing. The memory randomization support is to be connected to the `mmap` UNICORE API for full control of the UNICORE runtime memory layout. Memory randomization

helps prevent “known address” attacks such as return to libc attacks [8] or shellcode injected on stack attacks [9], in general, all the buffer overflow attacks. Memory randomization consists of adding support for ASLR (Address Space Layout Randomization) and PIE (Position Independent Executable).

Address space layout randomization (ASLR) [10] is a security technique that randomizes the memory layout of a program. Every time a program is run, segments of the address space like the stack, heap, or libraries are placed at a random address in memory. If the program that we run is position-independent, then the text, data, and bss segments can also be randomized. Randomization of the code in memory is an essential security feature, and it is what makes ASLR so powerful. The randomization of memory addresses means that an attacker no longer knows where the code needed by him it is placed.

A position-independent executable (PIE) is an executable that can be placed anywhere in memory and still run correctly. A position-independent executable contains position-independent code (PIC). Position-independent code uses relative addressing, data references are usually made through the global offset table [11], and function calls are done using the procedure linkage table [11]. In general, to obtain a position-independent executable, it is only necessary to add the `-fPIC` compilation flag and the `-pie` linking flag, but in some cases, some parts of the code have to be modified.

4.2.1 PIE Support in UNICORE

In the default state, UNICORE code is loaded at a constant address, which is set by a linking script, every time the unikernel is run. The aim is to randomize the placement of the code from the unikernel image when loaded in memory. To achieve this, there is a requirement for:

- the unikernel image to be built with PIC support
- a randomization engine that will generate a different address for our code every time the unikernel is run
- a loader that will load the unikernel at the above random generated address

To obtain a position-independent executable (PIE), it usually requires the passing of the `-fPIC` and `-pie` compilation and linking flags. For UNICORE changes are required in certain assembly source code files, such as Listing 4.2 and updates the build system.

```
1     andl $(~(X86_CR0_EM | X86_CR0_TS)), %esi
2     orl $(X86_CR0_MP | X86_CR0_NE | X86_CR0_WP), %esi
3     movq %rsi, %cr0
4     fninit
5 #if __SSE__
6     orl $(X86_CR4_OSFXSR | X86_CR4_OSXMMEXCPT), %edi
7     movq %rdi, %cr4
8     ldmxcsr (mxcsr_ptr)
9 #endif /* __SSE__ */
```

Listing 4.2: entry64.S code that is not position-independent

The loader is a small piece of code that needs to be called by the underlying platform and then load the unikernel image itself. In the default build, UNICORE is directly loaded by the platform. The solution was to split UNICORE images into a minor image that is non-PIE with basic code to boot and incorporating the loader; we also call this *bootloader*. The larger part of the UNICORE image, including most of the UNICORE APIs, external libraries and application code is built as a PIE executable.

For random address generation to load the unikernel proper to, the loader uses the UNICORE `ukswrand` API.

The static randomization process will have three parts, as we can see in Figure 4.1. The first part will be to load the bootloader into memory. The bootloader will contain the non-PIC boot code, the randomization engine, and the loader. The second part will be unikernel proper.

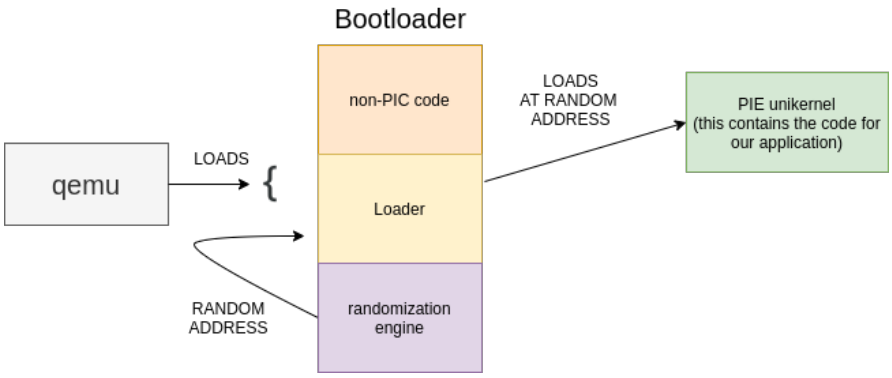

Figure 4.1: Loading scheme of the code

PIE is currently support in linuxu (simply by using `-fPIC`, `-pie` flags) and KVM (using the bootloader / unikernel split above). In KVM the using of the two images split is enabled by the ability to path the bootloader image as the kernel image and the unikernel proper as the initial randmisk (initrd) image as below:

```
qemu-system-x86_64 -kernel bootloader -initrd unikernel-proper
```

Listing 4.3: Passing an initrd file to qemu

4.2.2 Performance

Position-independent code can be slightly slower than non-position-independent code due to the relative addressing that has to be done, use of the global offset table, or the procedure linkage table. To test this on the PIE unikernel, we used Redis and SQLite.

For Redis we measured the throughput in `operations/second`. The tests were done using the `SET` and `GET` operations. Results are shown in Figure 4.2. The difference between the default unikernel and the PIE unikernel builds is negligible.

For SQLite we measured how much time does a certain number of `insert` operations take. As how in Figure 4.3, results are similar to Redis, that is the default unikernel and the PIE unikernel builts have similar performance.

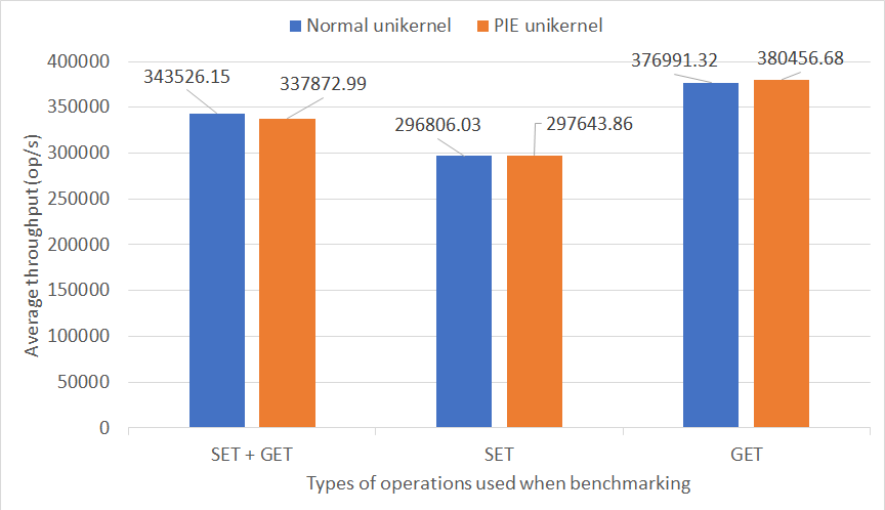

Figure 4.2: Redis throughput

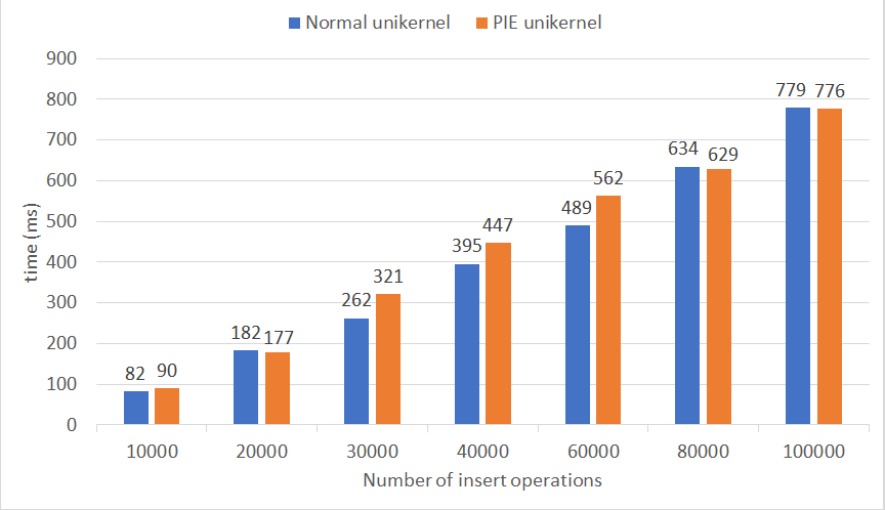

Figure 4.3: SQLite time for insert operations

4.3 Deterministic Execution Support

In order to validate the solution for deterministic execution of smart contracts, including running in a UNICORE-specific constrained environment, we created a set of programs that behave in a non-deterministic manner.

The non-deterministic test suite is based on previous analysis of sources of non-determinism including uninitialized data, data types and data packing, architecture-specific instructions, I/O and random data, time-related measurements, scheduling.

The current set of samples is classified according to the source of non-determinism: program input, symbol address, uninitialized data, time, system specifics, scheduling, platform specifics. Samples are created in C, Python, Go and Java, conforming to our goal of providing deterministic execution across multiple platforms (languages, architecture, configuration).

Table 4.1 shows the non-determinism use cases together with their classes and where to mitigate them. The

mitigation will be handled in one component of the solution for deterministic execution.

One sample aims to getting non-determinism by using floating operations as software configuration and CPU specifics is likely to incur rounding and truncation. At this point, we haven't been able to create a sample with non-determinism; we use C and go programs and compiled them for x86_64, x86 and ARMv7 and results are identical. We plan to use a public floating point test bench to trigger non-deterministic behavior.

Consider a given (smart contract) program, we design a solution that integrates UNICORE in maximizing deterministic execution. The solution is shown in Figure 4.4. It includes four stages of compliance check:

(i) source code static analysis

(ii) local (developer) binary static analysis

(iii) remote (node) minimal binary compliance

(iv) runtime compliance enforcement (in a UNICORE constrained environment)

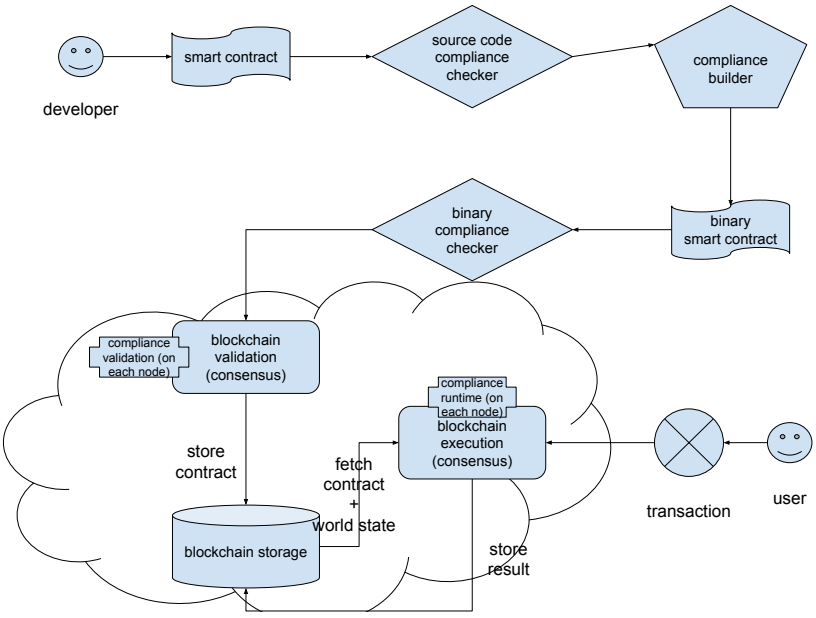

Figure 4.4: Deterministic Execution Support

The smart contract developer will have access to a build toolchain with deterministic support that incorporates the source code static analysis and the binary static analysis. The program will be checked locally for sources of non-determinism both at code level and executable / binary level, before being submitted to the blockchain. Before adding the smart contract to the blockchain, the internal distributed decision will run a minimal binary compliance phase on each blockchain node involved for quick vetting of the program. Once the vetting is passed the smart contract is stored on the blockchain.

Once a transaction triggers the use of the smart contract, a UNICORE-enabled runtime will execute the smart contract in a constrained compliant environment (such as disabling memory randomization, initializing

Test case	Test case class	Where to mitigate
random number	program input	runtime enforcement with seed
standard input	program input	runtime enforcement of closed stream or similar
environment variable	program input	runtime enforcement
program argument	program input	runtime enforcement (proper arguments provided by the validator)
address of environment variable	symbol address	source code checker + binary checker if we can
address of program argument	symbol address	source code checker + binary checker if we can
address of local variable	symbol address	source code checker + binary checker if we can
address of allocated variable	symbol address	source code checker + binary checker if we can
address of library function	symbol address	source code checker + binary checker if we can
address of global variable	symbol address	source code checker + binary checker if we can
address of program function	symbol address	source code checker + binary checker if we can
uninitialized local variable	uninitialized data	runtime enforcement
uninitialized global variable	uninitialized data	runtime enforcement
current time	time	runtime enforcement with some consensus on the current time (then provided like the seed)
duration	time	source code checker (blacklist)
process ID	system specifics	runtime enforcement
thread ID	system specifics	runtime enforcement
thread ordering	scheduling	runtime enforcement if possible, otherwise source code checker + binary checker
threaded arithmetic operations	scheduling	runtime enforcement if possible, otherwise source code checker + static checker
time of check to time of use (TOCC-TOU)	scheduling	runtime enforcement if possible, otherwise source code checker + static checker
threaded list operations	scheduling	runtime enforcement if possible, otherwise source code checker + static checker

Table 4.1: Test cases of non-determinism

all memory with zeroes, disabling scheduling, strict control of I/O, *etc.*). If results from a transaction are accepted by the distributed consensus reaching mechanism in the blockchain, the transaction is validated. While the solution is being developed, the current set of non-deterministic cases will be used to validate it.

5 Conclusion

This document summarizes the current state of the UNICORE APIs, libraries and applications. It also highlights current security and safety primitives and deterministic execution support in UNICORE.

Existing APIs, libraries and applications are enough for proof of concept builds and runs. We are able to measure unikernel images sizes, boot times and performance metrics using Redis, SQLite and nginx. In the current state, UNICORE components can be integrated in a variety of test cases.

Further work is including additional libraries and applications and stabilizing the UNICORE APIs with respect to performance and security.

References

[1] Google, “Protocol buffers - google’s data interchange format.” [Online]. Available: https://github.com/protocolbuffers/protobuf

[2] J. Litton, D. Garg, P. Druschel, and B. Bhattacharjee, “Composing abstractions using the null-kernel,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS'19. New York, NY, USA: ACM, 2019, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321450

[3] T. H. Dang, P. Maniatis, and D. Wagner, “Oscar: A practical page-permissions-based scheme for thwarting dangling pointers,” in *Proceedings of the 26th USENIX Security Symposium*, ser. USENIX Security'17. Vancouver, BC: USENIX Association, 2017, pp. 815–832. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang

[4] Openwall, “Implement glibc chk interfaces for ABI compatibility.” https://www.openwall.com/lists/musl/2015/06/17/1.

[5] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, “A binary-compatible unikernel,” in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019, pp. 59–73. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313817

[6] 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 ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 65–70. [Online]. Available: https://doi.org/10.1145/3141235.3141242

[7] 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

[8] S. El-Sherei, “Return-to-libc,” https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf, last accessed: 20 June 2020.

[9] D. Kapil, “Shellcode injection,” https://dhavalkapil.com/blogs/Shellcode-Injection/, last accessed: 20 June 2020.

[10] U. D. Sandra Henry-Stocker and N. World, “How aslr protects linux systems from buffer overflow attacks,” https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html, last accessed: 28 February 2018.

[11] G. Curell, “What is the symbol table and what is the global offset table?” https://www.codeproject.com/articles/1032231/what-is-the-symbol-table-and-what-is-the-global-of, last accessed: 9 June 2020.

