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

# D2.5 Platform Integration

Due date of deliverable: 30 June 2021

Actual submission date: 30 June 2021

| | |
|---|---|
| Start date of project | 1 January 2019 |
| Duration | 36 months |
| Lead contractor for this deliverable | Nextworks s.r.l. (NXW) |
| Version | 1.0 |
| Confidentiality status | Public |

**Abstract**

The UNICORE project is developing tools to enable lightweight VM development to be as easy as compiling an app for an existing OS, thus unleashing the use of the next generation of cloud computing services and technologies. With UNICORE toolchains for unikernels, software developers will be able to easily build and quickly deploy lightweight virtual machines starting from existing applications.
This deliverable reports on the integration activities executed in the UNICORE project.

**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 | WP2 - Platform Design and Evaluation |
| Editor | Nextworks s.r.l. (NXW) |
| Project Co-ordinator | Emil-Ioan Slusanschi, UPB |
| Technical Manager | Felipe Huici, NEC |
| **Copyright notice** | © 2021 Participants in project UNICORE |

# List of Authors

| | |
|---|---|
| Editors | Gino Carrozzo (NXW), Gabriele Scivoletto (NXW) |
| Participants | NEC, UPB, ULIEGE, IBM, VUA |
| Work-package | WP2 - Platform Design and Evaluation |
| Security | PUBLIC |
| Nature | R |
| Version | 1.0 |
| Total number of pages | 32 |

# Contents

**List of Authors** **3**

**List of Figures** **5**

**List of Tables** **6**

**1 Executive Summary** **7**

**2 Introduction** **8**

**3 UNIKRAFT Integration and Deployment** **10**

3.1 GitHub code repository . . . . . 10

3.1.1 The UNIKRAFT kernel repository . . . . . 11

3.1.2 The kraft toolstack repository . . . . . 12

3.2 The UNIKRAFT external libraries . . . . . 14

3.3 UNIKRAFT sample applications . . . . . 14

3.4 Patch tracking via Patchwork and GitHub Pull Request . . . . . 16

3.5 Continuous Integration via Concurse-CI . . . . . 17

3.6 Release Management . . . . . 18

**4 UNICORE toolstack integration** **20**

4.1 Security and isolation primitives . . . . . 20

4.2 Deterministic execution support . . . . . 21

4.3 Compilation Toolchain . . . . . 23

4.4 Symbolic Verification Support . . . . . 24

4.5 Orchestration Tools Integration . . . . . 26

**5 Guidelines for deployment** **27**

5.1 Krafting functions with UNIKRAFT . . . . . 27

5.2 Bug filing and support . . . . . 29

**6 Conclusions** **30**

**7 Abbreviations and Definitions** **31**

7.1 Abbreviations . . . . . 31

7.2 Definitions . . . . . 31

**References** **32**

# List of Figures

3.1 The UNIKRAFT GitHub repository. . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2 UNIKRAFT builds specialized OSses just selecting the application specific modules and libraries. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.3 High level view of the micro-libraries needed for building and running unikernels. . . . . . . 13
3.4 The implementation of the Kraft CLI. . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.5 The Patchwork Web Interface. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.6 Example of Pull Request. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.7 Result of the integration tests via unikraft-bot. . . . . . . . . . . . . . . . . . . . . . . . 18
3.8 Software Development life cycle in UNIKRAFT. . . . . . . . . . . . . . . . . . . . . . . 19

4.1 Symbolic Verification Support architecture. . . . . . . . . . . . . . . . . . . . . . . . 25

# List of Tables

3.1 Krafted libraries in UNIKRAFT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.2 UNIKRAFT releases. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

# 1 Executive Summary

One of the main goals of the UNICORE Consortium is to work on the integration of all the different libraries, components and tools developed within the Project in order to build and release consolidated prototypes of the UNIKRAFT/UNICORE toolstack.

The main collectors of public artifacts from this process are:

- The UNIKRAFT GitHub page: `https://github.com/unikraft`
- The UNIKRAFT project webportal: `https://unikraft.org/`
- The UNICORE project webpage: `https://unicore-project.eu/`

This deliverable reports on the integration activities executed in the UNICORE project.

The document specifically provides insights on the toolkits adopted to manage the software configuration management aspects of the UNIKRAFT/UNICORE project, and also provides guidelines for building and deploying krafted functions. Mechanisms for submitting developed artifacts on UNIKRAFT as well as the available channels and policies for interactions with the UNIKRAFT core developer teams are described.

# 2 Introduction

Continuous Integration and Continuous Delivery (CI/CD) are common practices for building, testing, and releasing modern software and services.
CI/CD refers to workflows which generally consists of various steps, executed sequentially and then closed in a loop to implement the incremental improvement:

- Continuous Integration workflow
  (i) Plan
  (ii) Code
  (iii) Build
  (iv) Test
  (v) *back to Plan*
- Continuous Delivery workflow
  (i) Release
  (ii) Deploy
  (iii) Operate
  (iv) Monitor
  (v) *back to Plan*

It is common sense that CI/CD practices have increased efficiency of releases, have reduced bugs and errors in software ahead-of-time, and allow to produce more reliable software products with less overhead for code reviews, testing and validation.
This deliverable reports on the integration activities organised within the UNICORE project in relation to the main software stream around UNIKRAFT [1]. The different areas of development of the project are covered and the related integration activities described. The deliverable also provides a complete guideline for building and deploying general software applications in mixed UNICORE environments, taking advantage of the automation and optimization features offered by the system.
The document is organised as follows:

- Section 3 describes the UNIKRAFT integration and deployment environment, documenting the software configuration management platform in use (GitHub), the patch tracking system, the continuous integration environment and the release management approach.
- Section 4 describes the UNICORE toolstack integration aspects, and covers specifically the parts related to security and isolation primitives, the deterministic execution support modules, the symbolic verification support, the orchestration tools.

- Section 5 documents the guidelines for building and deploying krafted functions with UNIKRAFT, and provides information on how to raise bugs and request support to the UNIKRAFT development team.
- Section 6 contains the document conclusions.

# 3 UNIKRAFT Integration and Deployment

## 3.1 GitHub code repository

UNIKRAFT [1][2][3] is a project hosted by the Linux Foundation. Thanks to the open source nature of the project, the source code is distributed in the form of public repositories in GithHub, the most important and widely used web-based version-control and collaboration platform for software developers.

The url of the project is `https://github.com/unikraft`.

The UNIKRAFT repository in the public GitHub project contains all the source code files needed to compile a fully functional unikernel based on the UNIKRAFT framework. The building process has to be configured in order to fit the requirements of the different platforms and architectures.

The contribution policy to UNIKRAFT in GitHub allows anyone who is interested in working on the development of UNIKRAFT modules to do so and submit their work in the form of patches. The project is mailing-list driven, meaning that the contributors should submit the patches to the proper mailing list (`minos-devel@list.xen.org`) and CC the corresponding maintainer. The list of the maintainers for every sub-project is written in the file MAINTAINERS.md under the */unikraft/unikraft* repository in GitHub.

The patch submission procedure follows the Xen Projects one: the main branches are master and staging.

For a correctly formatted patch submission, the git tools (in particular git format-patch and git send-email) is strongly recommended and makes the patch review processes easier.

The Commit Message follows a predefined format:

**[selector]/[component name]: [Short message]**

Where [selector] can be one of the following:

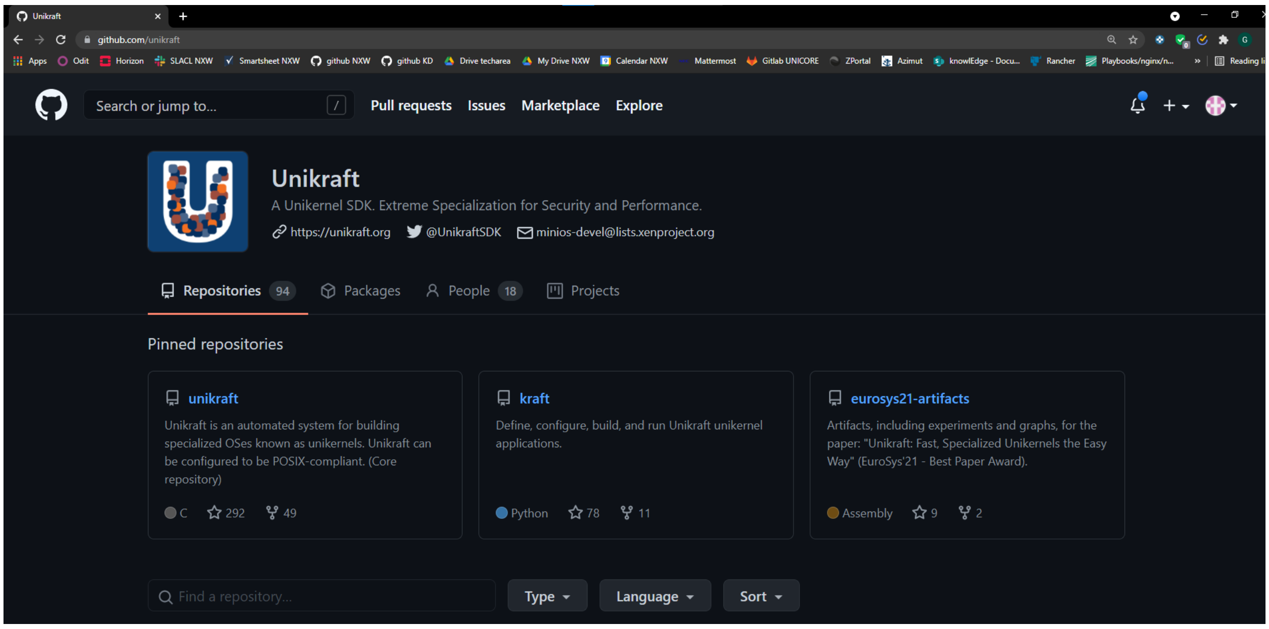
- *arch*: Patch for the architecture code in arch/, [component] is the architecture (e.g, x86) applies also for corresponding headers in include/uk/arch/
- *plat*: Patch for one of the platform libraries in plat/, [component] is the platform (e.g, linuxu). This applies also for corresponding headers in include/uk/plat/
- *include*: Changes to general UNIKRAFT headers in include/, include/uk
- *lib*: Patch for one of the UNIKRAFT base libraries (not external) in lib/, [component] is the library name without lib prefix (e.g, fdt)
- *doc*: Changes to the documentation in doc/, [component] is the corresponding guide (e.g., developers)
- *build*: Changes to build system or generic configurations, [component] is optional

UNIKRAFT is organized into libraries where each might be individually licensed. In general, each source file should declare who is the copyright owner and under which terms and conditions the code is licensed.

The main license of the project is the following BSD 3-clause license. It applies in particular to source code files that do not declare a license and where there is no license information file (e.g., LICENSE, COPYING) placed in the same or corresponding root folder.

The files in *plat/xen/include/xen* of the repository *unikraft/unikraft* were copied from the Xen sources, so they are licensed under MIT (for more information, its possible to read the file *plat/xen/include/xen/COPYING*).

UNIKRAFT's build and configuration system is based on Buildroot, which is licenced under GPLv2. In addition, the UNIKRAFT Kernel repository supplies a copy of the Linux Kernel's checkpatch.pl script to help committers adhere to the coding style. This means that a number of files in this repository are GPLv2-licenced.


Unikraft

A Unikernel SDK. Extreme Specialization for Security and Performance.

https://unikraft.org @UnikraftSDK minios-devel@lists.xenproject.org

Pinned repositories

unikraft
Unikraft is an automated system for building specialized OSes known as unikernels. Unikraft can be configured to be POSIX-compliant. (Core repository)
C 292 49

kraft
Define, configure, build, and run Unikraft unikernel applications.
Python 78 11

eurosys21-artifacts
Artifacts, including experiments and graphs, for the paper: "Unikraft: Fast, Specialized Unikernels the Easy Way" (EuroSys'21 - Best Paper Award).
Assembly 9 2

Figure 3.1: The UNIKRAFT GitHub repository.

The most relevant repositories of the project are:

- **UNIKRAFT kernel repository** (`https://github.com/unikraft/unikraft`), which contains the source code related to the core part of the unikernel itself, such as platform specific related code (i.e arm vs x86) or architecture specific related code (i.e Linux KVM vs Solo5);
- **kraft repository** (`https://github.com/unikraft/kraft`), which contains the source code related to the UNIKRAFT Toolstack implementation

### 3.1.1 The UNIKRAFT kernel repository

The main goal of the UNIKRAFT framework is to automatically build OSes tailored to the needs of specific applications, based around the concept of small, modular libraries, each providing a part of the functionality commonly found in an operating system, as shown in Figure 3.2.

Following the concept in Figure 3.2, the UNIKRAFT kernel repository is organized as follows:

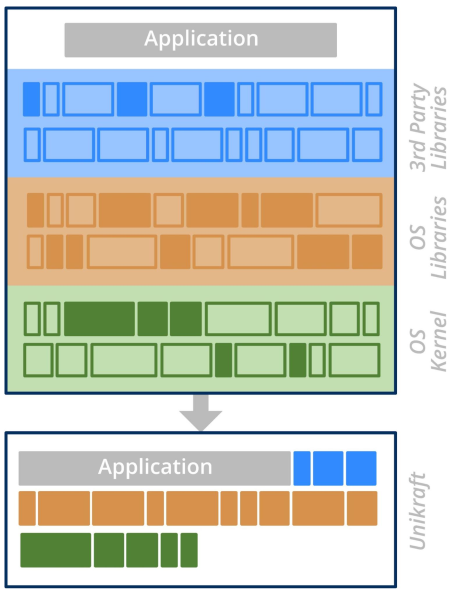


Figure 3.2: UNIKRAFT builds specialized OSses just selecting the application specific modules and libraries.

- **Architecture specific directory (arch/)**: contains all the tools and source code files related to the architecture-specific kernel building processes. Therefore, the directory is composed, so far, by an arm subdirectory (*arm/*) and the x86 one (*x86/*)
- **Platform specific directory (plat/)**: contains all the tools and source code files related to the platform-specific kernel building processes, including the virtualization drivers if required by the platform chosen. Actually, the platform supported (the repository has the proper subdirectory for each of them) are
  - **Linux KVM**: Virtualized using the QEMU tool
  - **Linux User Space**: Running on a bare-metal linux platform
  - **XEN Hypervisor**
- **OS Libraries directory (lib/)**: contains all the libraries who make the platform modular, since there were all the micro-libraries and tools needed for building the platform-independent layer of the system, as well as all the system calls that have been krafted so far. Figure 3.3 shows a high level view of the hierarchical level of the micro-libraries.

### 3.1.2 The kraft toolstack repository

The kraft tools is responsible for implementing all the tools needed for building and managing the unikernel using the UNIKRAFT Framework.

Kraft is a command-line utility, therefore, once it has been installed, it provides a full set of shell commands for defining, configuring, building, running and debugging UNIKRAFT applications.

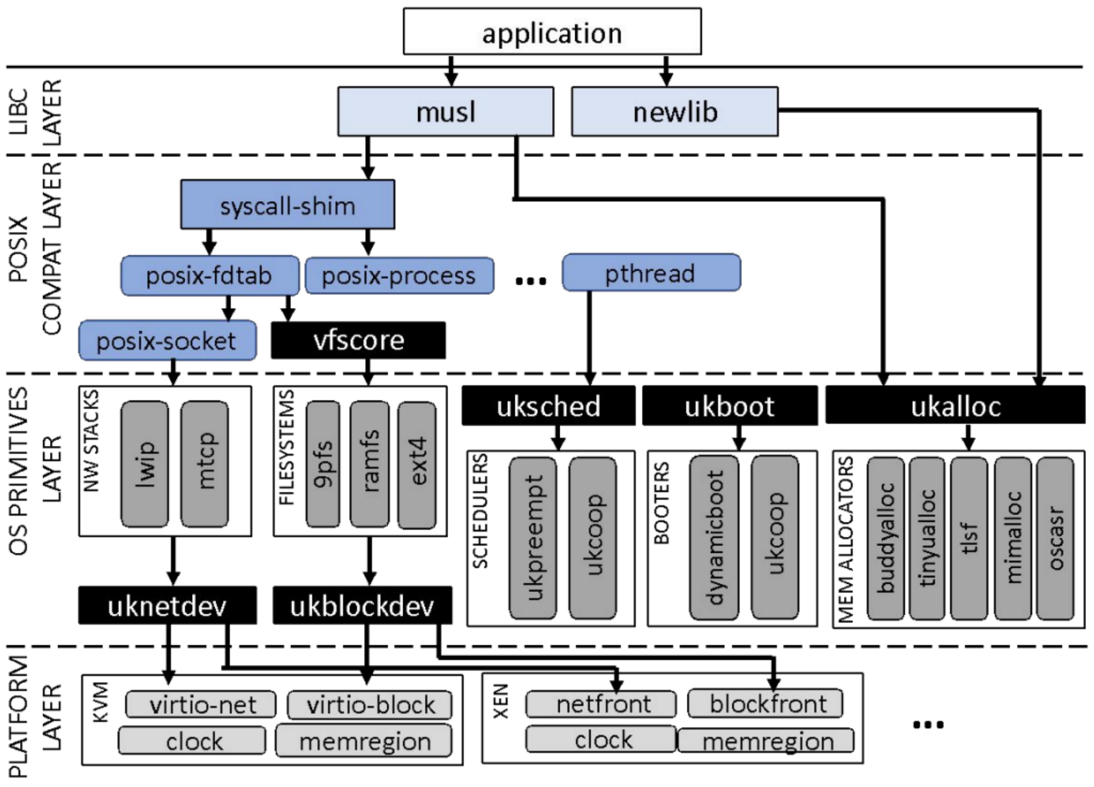
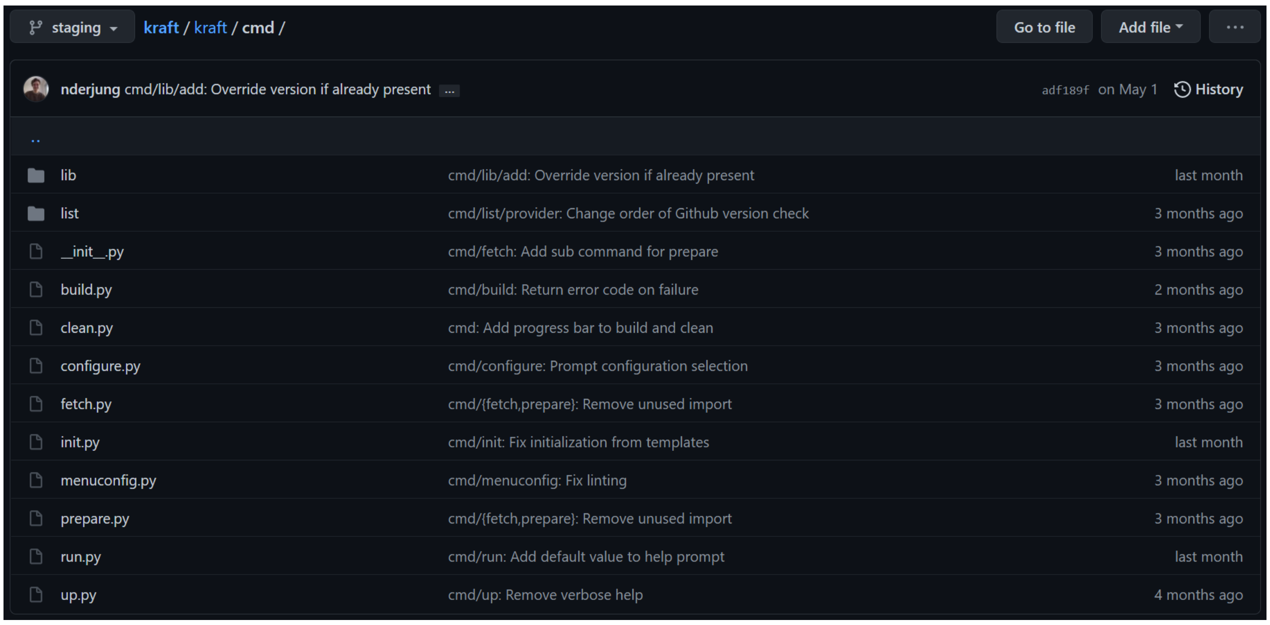


Figure 3.3: High level view of the micro-libraries needed for building and running unikernels.

The tool is written in Python and it is released as a Python package: in the root directory, there are all the python configuration files needed for installing the package in a Linux based environment, such as the mandatory *setup.py* for the installation script and the *requirements.txt* for preparing the environment installing the dependencies.

The implementation of the CLI that kraft offers, is located in the subdirectory *kraft/cmd/*.



nderjung cmd/lib/add: Override version if already present ... adf189f on May 1

| | | |
|---|---|---|
| .. | | |
| lib | cmd/lib/add: Override version if already present | last month |
| list | cmd/list/provider: Change order of Github version check | 3 months ago |
| __init__.py | cmd/fetch: Add sub command for prepare | 3 months ago |
| build.py | cmd/build: Return error code on failure | 2 months ago |
| clean.py | cmd: Add progress bar to build and clean | 3 months ago |
| configure.py | cmd/configure: Prompt configuration selection | 3 months ago |
| fetch.py | cmd/{fetch,prepare}: Remove unused import | 3 months ago |
| init.py | cmd/init: Fix initialization from templates | last month |
| menuconfig.py | cmd/menuconfig: Fix linting | 3 months ago |
| prepare.py | cmd/{fetch,prepare}: Remove unused import | 3 months ago |
| run.py | cmd/run: Add default value to help prompt | last month |
| up.py | cmd/up: Remove verbose help | 4 months ago |

Figure 3.4: The implementation of the Kraft CLI.

## 3.2 The UNIKRAFT external libraries

Along with the os-libraries that have been krafted during the project, a set of external libraries have been correctly rebuilt and made available for being used in a unikernel application.
As shown in Figure 3.2, the main goal of the UNIKRAFT framework is to combine, following a modular path, os kernel modules with both os and 3rd party libraries in order to generate a fully functional unikernel with a minimal footprint. For achieving this goal, the UNIKRAFT GitHub Project has been populated with the most commonly used libraries ready to be used with the UNIKRAFT Toolstack. All of them are stored in a separate repository who has the name who start with *lib-**.
Currently, the libraries reported in table 3.1 have been correctly krafted.

## 3.3 UNIKRAFT sample applications

Along with the porting of the 3rd party libraries, several applications have been compiled and have been made available and ready to be executed. These samples can also act as a starting point for new UNIKRAFT applications: for example, the application app-python3 have been used for the UNICORE Smart Home use case as a starting point for the implementation of an MQTT driver written in Python, using both UNIKRAFT Libraries (such as lib-pthread-embedded) and also external krafted libraries. Moreover, the Application app-nginx has been used as a benchmark for the validation of the UNIKRAFT Framework and the results are shown in the EuroSys'21 - Best Paper Award *UNIKRAFT: Fast, Specialized Unikernels the Easy Way*.
All of the UNIKRAFT applications are stored in a separate repository who has the name who start with *app-**.
Currently, the available application samples include:

- **app-httpreply**: A simple HTTP echo server example for UNIKRAFT
- **app-helloworld**: A simple "Hello World" application written in C
- **app-duktape**: kraft-ready repo for building JavaScript/duktape applications with UNIKRAFT
- **app-lua**: kraft-ready repo for building Lua applications with UNIKRAFT
- **app-helloworld-go**: kraft-ready repo for building Go applications with UNIKRAFT
- **app-python3**: UNIKRAFT Python3 app repo
- **app-helloworld-cpp**: kraft-ready repo for building c++ applications with UNIKRAFT
- **app-wamr**: Web Assembly (WAMR) on UNIKRAFT
- **app-click**: Click Modular Router on UNIKRAFT
- **app-redis**: Redis on UNIKRAFT
- **app-sqlite**: SQLite on UNIKRAFT

Table 3.1: Krafted libraries in UNIKRAFT

| Unikraft Library | Orig. Library | | Unikraft Library | Orig. Library |
|---|---|---|---|---|
| lib-lwip | lwip | | lib-libelf | ELF toolchain |
| lib-libsodium | libsodium | | lib-pybind11 | lib-pybind Redis |
| lib-newlib | newlib | | lib-redis | Redis |
| lib-compiler-rt | compiler-rt | | lib-dnnl | Intel Math Kernel Library for DNNs |
| lib-gcc | GNU Compiler Collection libraries | | lib-musl | musl libC |
| lib-libcxx | C++ standard library | | lib-duktape | duktape/JavaScript |
| lib-tlsf | TLSF (general-purpose memory allocator) | | lib-bzip2 | bzip2 |
| lib-shfs | shfs | | lib-mbedtls | Mbed TLS library |
| lib-pthread-embedded | pthread-embedded | | lib-sqlite | SQLite |
| lib-dafny | Dafny language | | lib-c-ares | c-ares |
| lib-nettle | Nettle cryptographic library | | lib-lzma | lzma compression library |
| lib-libhogweed | libhogweed | | lib-boost | boost library |
| lib-libuv | libuv | | lib-lvgl | lvgl |
| lib-arm-intrinsics | ARM intrinsics | | lib-nnpack | nnpack |
| lib-libtasn1 | libtasn1 | | lib-open62541 | OPC UA implementation |
| lib-tinyalloc | thi.ng/tinyalloc | | lib-protobuf | Googles protobuf |
| lib-mimalloc | mimalloc | | lib-ruby | Ruby |
| lib-libicu | ICU | | lib-micropython | Micropython |
| lib-tflite | TensorFlow Lite | | lib-intx | intx |
| lib-fft2d | fft2d | | lib-libgo | Go language |
| lib-gemmlowp | Google's gemmlowp | | lib-wamr | WAMR |
| lib-farmhash | lib-flatbuffers | | lib-googletest | Google testing and mocking framework |
| lib-zydis | Zydis disassembler | | lib-libunwind | libunwind |
| lib-axtls | axTLS | | lib-http-parser | http-parser |
| lib-pcre | Perl Compatible Regular Expressions library | | lib-openssl | OpenSSL |
| lib-libuuid | libuuid | | lib-nginx | NGINX |
| lib-zlib | zlib | | lib-googlebenchmark | Google Benchmark |
| lib-lua | Lua language | | lib-libfp16 | half-precision floating point formats conversion |
| lib-libucontext | ucontext.h API | | lib-intel-intrinsics | Intel intrinsics |
| lib-libcxxabi | C++ ABI | | lib-libfxdiv | fxdiv |
| lib-psimd | psimd | | lib-eigen | libeigen |
| lib-pthreadpool | pthreadpool | | lib-click | Click modular router |
| lib-python3 | Python 3 | | | |

- **app-ruby**: Ruby on UNIKRAFT
- **app-micropython**: Micropython on UNIKRAFT
- **app-nginx**: Nginx on UNIKRAFT
- **app-elfloader**: Load and execute Linux ELF binaries
- **app-nettle-test**: Nettle cryptographic library in UNIKRAFT

## 3.4 Patch tracking via Patchwork and GitHub Pull Request

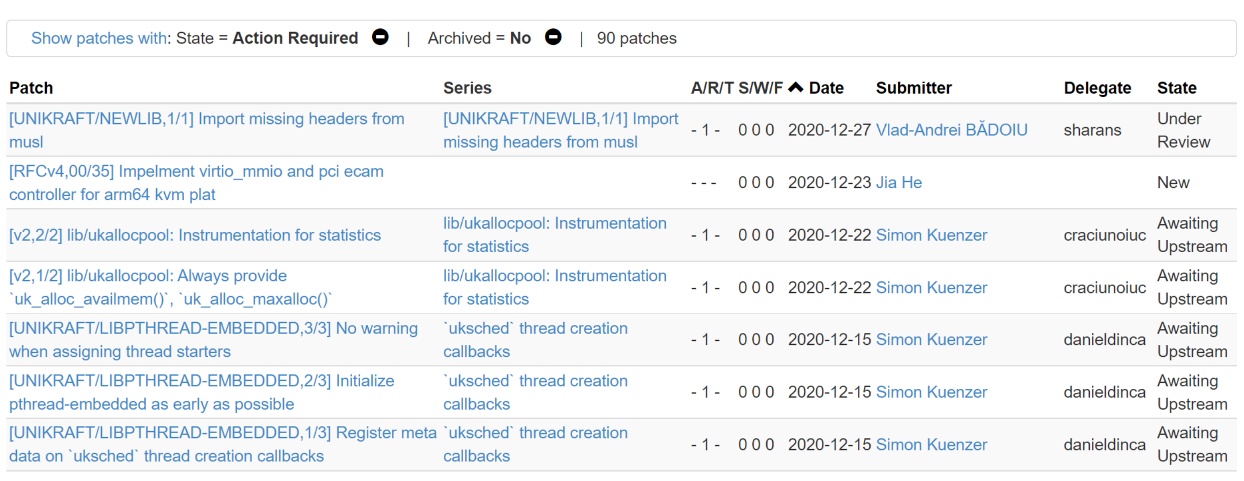
The UNIKRAFT software development cycle provides a mechanism for releasing the patchwork as soon as a bug is found or a new development is required.

For the first two years of UNICORE project, the Patch tracking tool that has been used was Patchwork [4]. Patchwork is free software, and is available from the Patchwork website (http://jk.ozlabs.org/projects/patchwork/).

The main goal of that Patch Tracking software is to facilitate the contribution and management of an open source project. The patches that have been sent to a mailing list are caught by the system and made available through a web-based interface. Any comments posted that reference the patch are appended to the patch page too.

The Patchwork system is reachable from the URL https://patchwork.unikraft.org/, and the version used is v2.1.0.postrc1-2-gb2106f3. The tools provide also a CLI named pwcllient: currently, it provides access to some read-only features such as downloading and applying patches.

Figure 3.5 shows how the Patchwork Web Interfaces looks like.



Show patches with: State = **Action Required** | Archived = **No** | 90 patches

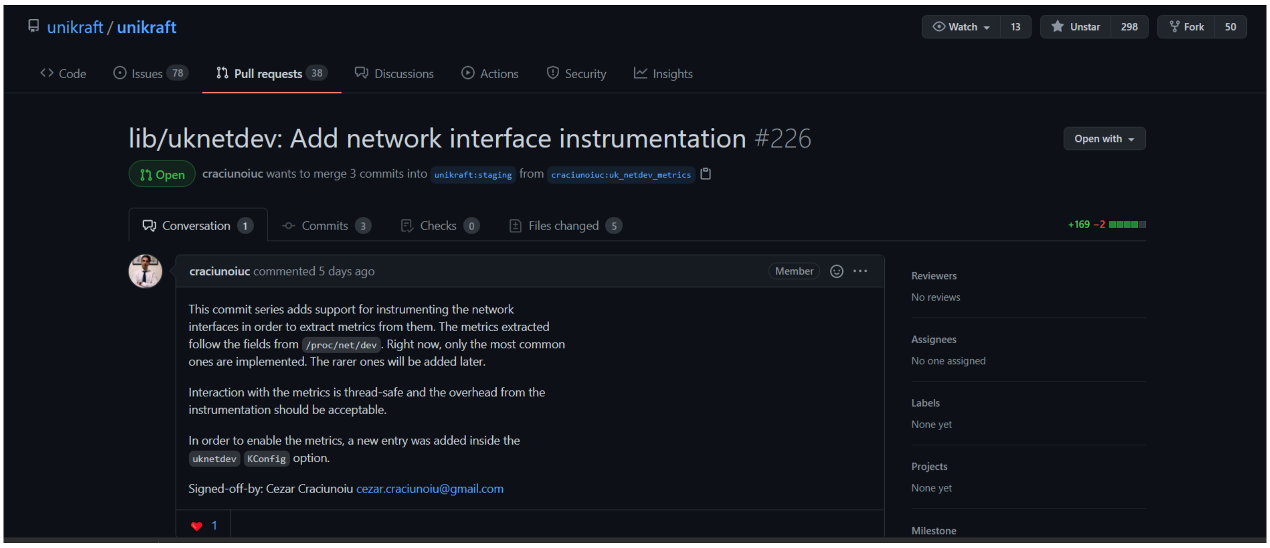
| Patch | Series | A/R/T | S/W/F | Date | Submitter | Delegate | State |
|---|---|---|---|---|---|---|---|
| [UNIKRAFT/NEWLIB,1/1] Import missing headers from musl | [UNIKRAFT/NEWLIB,1/1] Import missing headers from musl | - 1 - | 0 0 0 | 2020-12-27 | Vlad-Andrei BĂDOIU | sharans | Under Review |
| [RFCv4,00/35] Impelment virtio_mmio and pci ecam controller for arm64 kvm plat | | - - - | 0 0 0 | 2020-12-23 | Jia He | | New |
| [v2,2/2] lib/ukallocpool: Instrumentation for statistics | lib/ukallocpool: Instrumentation for statistics | - 1 - | 0 0 0 | 2020-12-22 | Simon Kuenzer | craciunoiuc | Awaiting Upstream |
| [v2,1/2] lib/ukallocpool: Always provide \`uk_alloc_availmem()\`, \`uk_alloc_maxalloc()\` | lib/ukallocpool: Instrumentation for statistics | - 1 - | 0 0 0 | 2020-12-22 | Simon Kuenzer | craciunoiuc | Awaiting Upstream |
| [UNIKRAFT/LIBPTHREAD-EMBEDDED,3/3] No warning when assigning thread starters | \`uksched\` thread creation callbacks | - 1 - | 0 0 0 | 2020-12-15 | Simon Kuenzer | danieldinca | Awaiting Upstream |
| [UNIKRAFT/LIBPTHREAD-EMBEDDED,2/3] Initialize pthread-embedded as early as possible | \`uksched\` thread creation callbacks | - 1 - | 0 0 0 | 2020-12-15 | Simon Kuenzer | danieldinca | Awaiting Upstream |
| [UNIKRAFT/LIBPTHREAD-EMBEDDED,1/3] Register meta data on \`uksched\` thread creation callbacks | \`uksched\` thread creation callbacks | - 1 - | 0 0 0 | 2020-12-15 | Simon Kuenzer | danieldinca | Awaiting Upstream |

Figure 3.5: The Patchwork Web Interface.

Once the patch has been submitted, the tools stores not only the patch itself but also various metadata associated with the email that the patch was parsed from, for example the State, that track the current status of the element (it varies from project to project, but generally a minimum subset of new, rejected and accepted will exist. In the UNIKRAFT Project, the States are:

- New
- Under Review
- Awaiting Upstream
- Accepted
- Rejected

Currently, the number of submitted patches are 4235 (+31 archived).
Since the start of the third year of the UNICORE project, the Patch Tracking System has been moved to the GitHub Pull Request tool, which lets the developer tell others about changes that he has pushed to a branch in a repository on GitHub. Once a pull request is opened, he can discuss and review the potential changes with collaborators and add follow-up commits before the changes are merged into the base branch.
An example of a Pull Request is shown in Figure 3.6.



lib/uknetdev: Add network interface instrumentation #226

Open craciunoiuc wants to merge 3 commits into `unikraft:staging` from `craciunoiuc:uk_netdev_metrics`

craciunoiuc commented 5 days ago

This commit series adds support for instrumenting the network interfaces in order to extract metrics from them. The metrics extracted follow the fields from `/proc/net/dev`. Right now, only the most common ones are implemented. The rarer ones will be added later.

Interaction with the metrics is thread-safe and the overhead from the instrumentation should be acceptable.

In order to enable the metrics, a new entry was added inside the `uknetdev` `KConfig` option.

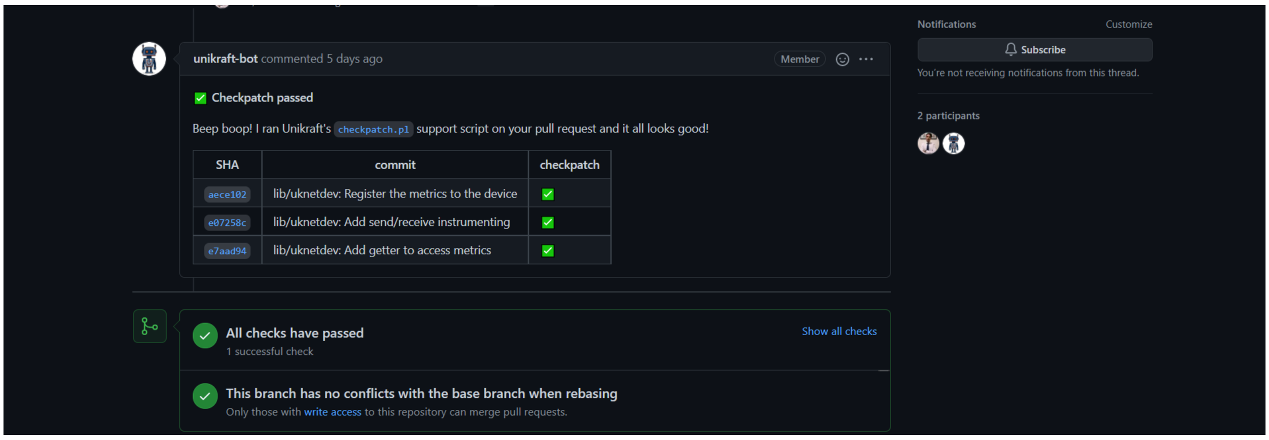
Signed-off-by: Cezar Craciunoiu cezar.craciunoiu@gmail.com

Figure 3.6: Example of Pull Request.

Once a pull request has been submitted, the CI/CD process (that will be explained in the next section) spawn a set of pipelines that make several integration test, the results is shown as an answer on the same Pull Request discussion, as shown in Figure 3.7.

## 3.5 Continuous Integration via Concurse-CI

The UNIKRAFT software development cycle also embeds mechanisms for continuously integrating and testing the code as soon as one of the repositories changes its status.
For that purpose, the tool chosen is Concurse-CI (`https://concourse-ci.org/`, [5]), a pipeline-based continuous thing-doer.



unikraft-bot commented 5 days ago

✓ Checkpatch passed

Beep boop! I ran Unikraft's `checkpatch.pl` support script on your pull request and it all looks good!

| SHA | commit | checkpatch |
|---|---|---|
| aece102 | lib/uknetdev: Register the metrics to the device | ✓ |
| e07258c | lib/uknetdev: Add send/receive instrumenting | ✓ |
| e7aad94 | lib/uknetdev: Add getter to access metrics | ✓ |

All checks have passed
1 successful check

This branch has no conflicts with the base branch when rebasing
Only those with write access to this repository can merge pull requests.

Figure 3.7: Result of the integration tests via unikraft-bot.

Actually, the word Pipeline in the range of CI/CD is a widely used term. Concurses Pipelines are built around Resources, which represent all external states and Jobs, which interact with them.
Concourses pipelines represent a dependency flow, kind of like distributed Makefiles Resources are used to express entities like source code, dependencies, deployments and any other external state.
The concourse web-based management interface of the UNIKRAFT Project is not publicly available, since its visibility is limited to the developer and maintainer of the project itself.
However, the idea behind the UNIKRAFT Framework is to build unikernel taking into account the following parameters:

(i) The architecture (armv7, arm64, x86_64)

(ii) The platform (KVM, XEN, Linux Bare Metal ..)

(iii) The set of libraries included in the building process

(iv) The set of configurations of each of the 3 entities at the points 1-3

For that purpose, the highly parameterizable nature of Concurrent-CI made the continuous integration and continuous testing of new modules suitable for all the scalability requirements. Whenever a new Pull Request (or any git commit) is submitted, a huge number of pipelines is generated based on all the possible configurations of architecture, platform, library and configurations and the result is shown in a web-based interface.

## 3.6 Release Management

The Release policy in UNIKRAFT is not included in the automatic DevOps operations explained above.
The the software development cycle can be summarized as depicted in Figure 3.8.
As soon as a Patch has been approved, the system is evaluated in order to check the current status of the system in terms of the number of new features.

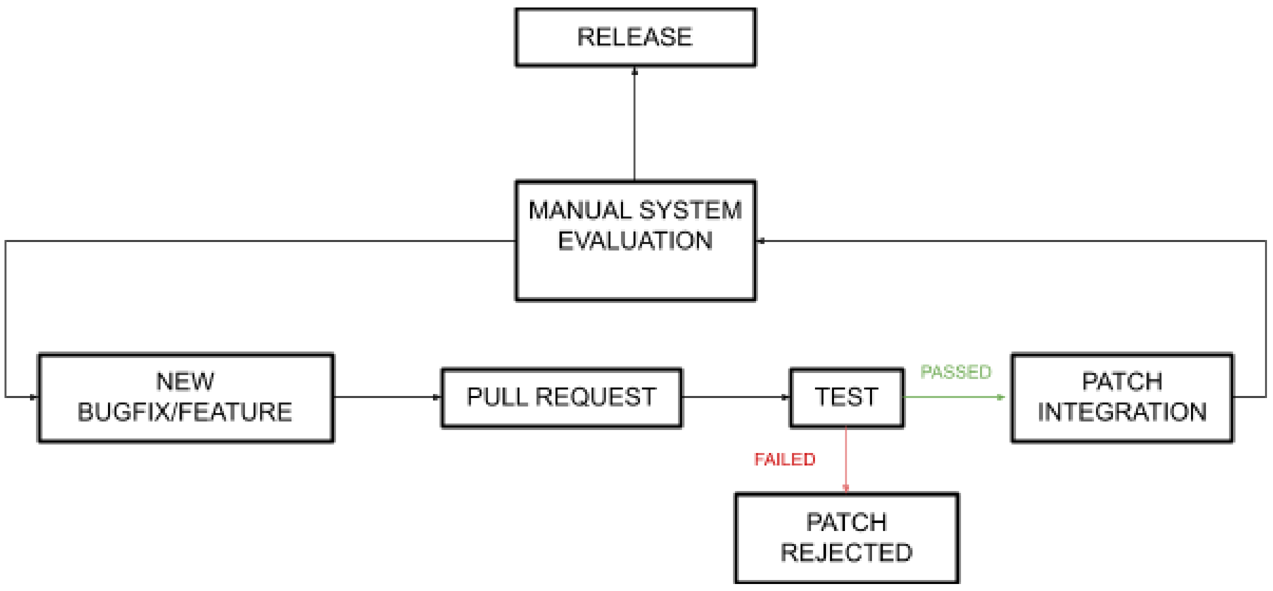


Figure 3.8: Software Development life cycle in UNIKRAFT.

A new release of the software was delivered only if the system is stable (after a huge number of tests) and with a good number of new features with respect to the last version released.

The releases delivered so far are reported in Table 3.2.

Table 3.2: UNIKRAFT releases.

| **VERSION** | **RELEASE NAME** | **RELEASE DATE** |
|---|---|---|
| v0.3.0 | Iapetus | February 20, 2019 |
| v0.3.1 | Iapetus | March 7, 2019 |
| v0.4.0 | Rhea | February 18, 2020 |
| v0.5.0 | Tethys | February 6, 2021 |

# 4 UNICORE toolstack integration

## 4.1 Security and isolation primitives

**FlexOS**. A UNIKRAFT variant called FlexOS has been designed which includes primitives for security and isolation.

FlexOS is a novel, modular OS design whose compartmentalization and protection profile can easily and cost-efficiently be tailored towards a specific application or use-case at build time, as opposed to design time as it is the case today. With FlexOS, the user can decide, at build time, which of the fine-grained OS components should be compartmentalized (e.g. the scheduler, TCP/IP stack, etc.), as well as how to instantiate isolation and protection primitives for each compartment and what data sharing strategies to use for communication between compartments. To that aim, we abstract the common operations required when compartmentalizing arbitrary software behind a generic API that is used to retrofit an existing LibOS into FlexOS. This API limits the porting effort of kernel and application components to a minimum. A solver automatically derives from that description as a set of conforming compartmentalization configurations that can be effortlessly instantiated through FlexOS flexible build.

**MPK Isolation Backend**. MPK is a mechanism present in recent Intel server CPUs and offers low-overhead intra-address space memory isolation at page granularity. Our MPK backend uses protection keys to isolate compartments and shared data. MPK permissions for the thread executing on a core are held in the PKRU register. Any compartment can modify its value, thus the MPK backend has to prevent such unauthorized writes; it can do so via runtime checks, static analysis, or page-table sealing. Conceptually, the implementation of MPK in FlexOS introduced isolation requirements for the scheduler and the Memory Manager (MM): the schedulers memory hold the value of the PKRU for threads that are not currently running and the MMs domain includes the page-table that holds the mapping between pages and protection domains. This implies that the scheduler and MM have to be trusted when using MPK, so in our implementation we used a verified scheduler implemented in Dafny, and we also introduced the option of using hardening mechanisms on schedulers/MMs implemented in C.

**EPT Isolation Backend**. Virtualization has been used in many works to support isolation within a kernel. Hardware-assisted virtualization is widely supported and provides strong security guarantees compared to MPK, at the cost of higher overheads. Our EPT backend generates one VM image per compartment. Images contain the minimum set of kernel functionality necessary to run the VM independently, along with a thin RPC implementation based on inter-VM notifications and a shared area of memory for shared heap/static data. This area is always at the same address and is mapped in all compart- ments (VMs) so that pointers to/in shared structures remain valid. Compartments do not share address spaces and run on different vCPUs. As a result, each compartment needs its own memory allocator and scheduler that have to be trusted. Our VM-based isolation backend currently uses the Xen hypervisor. VM-based gates place a function identifier as well as arguments in a predefined shared area of memory and issue an inter-VM notification. When the invoked compartment receives it, it executes the function, placing the return value in a predefined area of the shared memory, and notifying the caller that the RPC has succeeded.

**Toolchain**. The majority of the code transformations required for the build of a given FlexOS security configuration is realized through extensive use of Coccinelle. We leverage Coccinelles awareness of the C language semantics to de- scribe complex automated code transformations that would be hardly achievable through traditional search-and-replace tools or regular expressions: matching and replacement of various types of statically or dynamically allocated data including pointers and arrays, function calls and declaration with variable number of arguments, etc. The toolchains transformations achieve the following: 1) allocation of annotated shared static data n ELF sections such that they are mapped to the proper memory areas accessible from communicating compartments; 2) transformation of shared stack variable declarations into pointers referring either to the DSS or to a shared heap according to the data sharing strategy, as well as updating the statements reading from/writing to these variables into reads/writes to the corresponding location referred to by the pointer; 3) replacement of the calls to dynamic allocation functions (malloc, etc.) by calls to shared heaps for shared data, or to per-compartment heaps for private data; 4) identification of cross-components calls and their replacement by common function calls when they are in the same compartment, or by the proper cross-compartment gate according to the selected isolation mechanism; and 5) wrapping of functions annotated as callbacks in the proper cross-compartment call gates. As an example, the code transformation step modifies about 1 KLoC for a simple configuration (redis with 2 compartments, one for the TCP/IP stack, the other for the rest of the system, isolation with MPK and shared stacks). The other toolchain components are made up of a set of scripts and Makefiles driving the build process. Alongside code transformation the toolchain also generates a custom linker script declaring shared memory areas for static data. Finally, the toolchain includes a semi-automated performance evaluation platform that helps to explore the design space.

## 4.2 Deterministic execution support

Deterministic execution support is provided by native unikernel primitives such as programming language support, interaction with the outside world (networking, 9pfs), library configuration and build system. These are used to ensure minimalism, isolation and the optimal deterministic environment for running UNICORE applications as smart contract programs.

The build and deployment system in UNICORE relies on the kraft tool for configuration of components to be included in a UNICORE application image and parts of these components. kraft is a wrapper configuration tool on top of the basic KConfig+Makefile build system used in UNICORE. For fine tuning and control, the basic build system can be used. The build system and the UNICORE SDKs allow for both including or excluding certain APIs and for selecting a particular implementation for a given API (assumedly one that ensures improved determinism, isolation and minimalism).

The UNICORE development and execution environment enables developers to write programs in popular programming languages, such as Go, Python, C. We rely on that to provide a build environment for popular programming languages, allowing smart contract developers to use their preferred language. The UNICORE SDKs and build system have to provide:

- functionality: actual programming language support, allowing the execution of smart contract programs running in that particular programming language;
- configurability: ability to configure, replace and tune SDKs with the purpose of minimalism, isolation and determinism.

Configurability is provided as is by the UNICORE SDKs and build system. Functionality is available for C, C++ and partially for Go, Python and Lua.

A repository has been defined for storing smart contracts in multiple programming languages [6]. At this point support is provided for C and C++ smart contracts and partially in Go. Go smart contracts are running but without a networking interface for interacting with the outside world of the smart contract. A simple smart contract simply increments a value provided from the outside world and sends it back.

A more realistic scenario relies on implementing the Ed25519 signing algorithm thats used for e-voting systems and other common use cases for smart contracts. Its currently implemented in C in this repository [7] and relies on OpenSSL library support in UNIKRAFT. The implementation is going to be evaluated against EVM or other blockchain environments for running smart contracts.

Fine tuning of the UNICORE application is complementary to fine tuning of the VMM (Virtual Machine Monitor) for increased performance. Although the UNICORE isolation + determinism environment will incur overhead that may be inferior to other blockchain environments, the upside is the support for multiple programming languages via the UNICORE SDKs and build system.

Smart contract interface with the outside world for receiving transactions and providing results. There are two models that may be enabled by UNICORE for running smart contracts:

(i) continuous run: start smart contract, run continuously and receive transaction via a communication interface

(ii) start-top run: start smart contract with the transaction embedded in the smart contract (or passed as an argument), run it once and retrieve the results

The downside of the former approach is the interaction overhead between the smart contract and the outside world. The current networking interface employed provides unsatisfactory results. We are looking for a faster, shared memory-based approach.

The downside of the latter approach is the added time for starting a new unikernel instance each time. This however, comes with two advantages: the reduced resource consumption when the unikernel is not running and the added benefit of security as each instance is started anew and is not affected by previous runs. More-

over, this is closer to the unikernel-model of small, fast booting micro-VMs. We are looking into VMM configuration improvements to reduce the start-stop overhead.
Summarily, the UNICORE SDKs and build system provide the basic level of support for running smart contracts. Careful configuration of SDKs, VMM and running models, with the ongoing implementation of realistic smart contracts and addition of programming languages support will provide a complete good performance and feature rich environment.

## 4.3 Compilation Toolchain

The compilation toolchain contains a set of tools to automatically build images of operating systems targeting applications. The toolchain includes the following tools:

- Decomposition tool to assist developers in breaking existing monolithic software into smaller components.
- Dependency analysis tool to analyse existing, unmodified applications to determine which set of libraries and OS primitives are absolutely necessary for correct execution.
- Automatic build tool to match the requirements derived by the dependency analysis tools to the available libraries constructed by the OS decomposition tools. This one is composed of two components: a static analysis and a dynamic analysis.
- Verification tool to ensure that the functionality of the resulting, specialized OS+application matches that of the application running on a standard OS. The tool will also take care of ensuring software quality.
- Performance optimization tool to analyse the running specialized OS+application and to use this information as input to the automatic build tools so that they can generate even more optimized images.

The toolchain is written in golang/go. Once go is installed, it is necessary to run the Makefile (via make deps) to get all the required dependencies and in order to build the toolchain.
The toolchain contains the high-level architecture:

- **configFiles**: Contains sample configuration of specific applications.
- **testFiles**: Contains sample tests of a specific application for the dynamic analysis. The format used is a json which contains three different fields typeTest, timeMsCommand and listCommands. The first field represents the type of test which can be: exec (test via a classic execution like a script), stdin (test via stdin) or telnet (test via telnet). The timeMsCommand field defines the time of execution in ms. Finally, the listCommands field contains the different commands which will be executed during the dynamic analysis.

- **srcs**: Contains the go source of the toolchain.

As stated previously, the toolchain contains 5 different tools. You can either completely run the toolchain or specify which tool to use. The default behaviour executes all the tools (except the crawler one). To execute a specific tool, use one of the following arguments:

- *–dep*: runs only the dependency analysis tool.
- *–build*: runs only the automatic build tool.
- *–verif*: runs only the dependency analysis tool (prototype).
- *–perf*: runs only the dependency analysis tool (Not implemented).

Outside the toolchain:

- *–crawler*: run only the crawler tool. It creates a graph which represents dependencies of dependencies.
- *–memanalyser*: run only the memory analyser tool. It allows to analyse and disassemble binaries/object files (e.g., functions, instructions, etc) as well as identifying the mapping of microlibs.

Further information about the toolchain can be found in Deliverable 4.3: Design & Implementation of Tools for Unikernel Deployment.

## 4.4 Symbolic Verification Support

The main problem that were trying to solve is that the libraries that are part of UNIKRAFT, are neither safe nor correct since they are written in C and have not been formally verified, only manually tested. Moreover, UNIKRAFT introduces the idea of language mixing, and libraries written n different programming languages have different properties, e.g. Rust programming language libraries have built in memory safety.

The initial approach that we took in solving this problem was to use push-button verification based on symbolic execution. During our initial work, we discovered that it does not scale to proper components.

We devised a new innovative idea which consists of using runtime checks and hardware isolation such that we may integrate verified libraries into the pool of available UNIKRAFT libraries, but such that the properties of these libraries will hold at runtime and such that we will have both good performance and overall safety. Since UNIKRAFT has no correctness and safety proven for all its libraries, using a library that has been verified will cause its properties to be void when used directly alongside unverified C code. To solve this, we design a solution based on the idea of compartments. A compartment is a set of libraries that have compatible properties. For example, in the architecture of Figure 4.1 we have three compartments that gives us a configuration in which all the properties of the libraries hold:

In one compartment we have the network stack, in the second compartment the verified scheduler and in the third compartment we have the rest of the kernel and application.

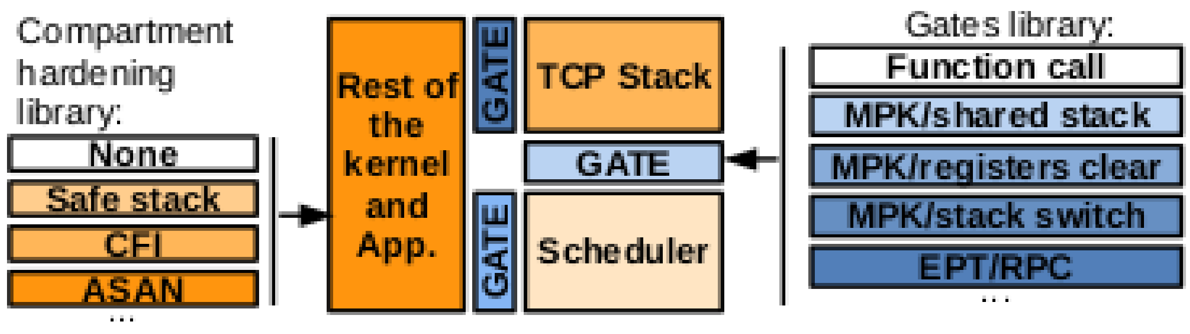


Figure 4.1: Symbolic Verification Support architecture.

The properties of a library written by the developer in a specialized metadata language. Such metadata are created manually for each library by its developer, a one-time and relatively low effort for the librarys author. The metadata purpose is to capture the effects upon the overall safety properties of running this library alongside other libraries in the same or in a different compartment. For instance, here is an example describing a formally verified scheduler that we have implemented in the Dafny programming language and integrated into UNIKRAFT:

```
[Memory Access] ::= R, W
[Call] ::= X
[API] ::=  (thread_add, SYMB) (thread_remove, SYMB)\
          (schedcoop_yield, SYMB)
[Requires] ::= X
```

The description concisely specifies that (1) the library accesses its own memory and a segment shared with other libraries (e.g. its callers), that (2) it only uses functions provided by the memory allocator, (3) which functions it exposes as its API, and that (4) it expects other libraries to be able to read its own memory (but not write to it) and be able to write in shared memory.

The metadata has been designed to be both simple and intuitively to use, e.g. R represents read, W write, X execute. Given two libraries and their metadata, we now have enough information to automatically decide whether they can run in the same compartment. If both libraries have no Requires clause, the answer is yes. If any of the libraries has such clauses, each clause can be automatically checked in the presence of the other library. In our example above, for its verified properties to hold, the scheduler expects others to only read, not write, to its own memory. A untrusted C component, on the other hand, could write to all memory it has access to (in its compartment) - thus breaking the expectation: as a result, these two libraries cannot be run in the same compartment. In order to maximize performance, we want to have the minimum number of compartments. Armed with information about pairwise incompatibility, selecting the smallest number of compartments can be reduced to the classical graph coloring problem: each library is a vertex, and an edge connects two incompatible libraries. Graph coloring assigns the smallest number of colors to the vertices of a graph such that no two adjacent vertices have the same color. For each color, we will instantiate a separate compartment that holds the libraries that have been painted with that color. In the worst case where all libraries have conflicts, each library will be instantiated in its own compartment.

We now introduce the idea of modifying the properties of libraries via software mechanisms which we call hardening mechanisms. We support several such mechanisms, among them Address Sanitization, Control Flow Integrity, Stack Protector and Undefined Behavior Sanitization. In certain cases, it is preferable from a performance or deployment point of view to use runtime instead of multiple compartments possibly only for a subset of the system/compartments. For control-flow integrity, the transformation is simple: libraries that previously declared X*(e.g. may be compromised and could jump at any address) are transformed into a list of predefined symbols where the list of functions is populated via a standard control-flow analysis of the library. The result of this step will be a list of libraries that have two versions: one with SH, and one without. We then iterate through all combinations of such library versions and run the graph coloring algorithm described above. This will result in as many colorings as there are possible combinations of libraries. Consider our example above: the unsafe C library will have two versions now, one with SH and one without SH. When put together with the scheduler in the same image, the SH version will be able to share a compartment with the scheduler, while the original version will require a separate compartment.

## 4.5 Orchestration Tools Integration

In order to make Unikernels easy to use and have a positive adoption by the community, one of UNICOREs goals is to integrate unikernels in some private cloud orchestrators and public cloud service providers.

Currently, UNIKRAFT supports three cloud providers: AWS, Google Compute and Digital Ocean. That means that it is possible to launch the resulting custom unikernels through UNIKRAFT in any of these three providers. To do so, first it is necessary to have an account and then configure UNIKRAFT with the keys so that it can communicate the endpoints of each of the platforms:

- **Amazon Web Services**: `https://github.com/unikraft/plat-aws` [8]
- **Google Compute Platform**: `https://github.com/unikraft/plat-gcp` [9]
- **Digital Ocean**: `https://github.com/unikraft/plat-digitalocean` [10]

As regards the private cloud orchestrators, UNICORE is working on the Integration of UNIKRAFT into OpenNebula and Kubernetes. UNIKRAFT is not fully integrated yet in these platforms, specially the Kubernetes due to the problems encountered in running unikernels on it, but there is a good progress so probably will be integrated in the next months.

# 5 Guidelines for deployment

## 5.1 Krafting functions with UNIKRAFT

Krafting a function with UNIKRAFT is a simple and automated procedure, thanks to the development of the kraft tool that includes all the commands needed to define, configure, build, run and debug UNIKRAFT applications.

With *kraft*, its possible to build environments for managing unikernels, as well as managing the dependencies of its build.

Installing the tool is straightforward, since its a python package hosted in a public repository in GitHub:

**$ pip3 install git+https://github.com/unikraft/kraft.git**

The Kraft tool works both with local and remote repositories: the (local or remote) link to the repositorys source code files, in combination with all the building information, are specified in the kraft.yml file. The following example shows how to specify local libraries and remote repositories for a custom UNIKRAFT application.

```
specification: '0.4'

UNIKRAFT: file:///home/developer/repos/unikraft/unikraft@3a8150d

libraries:
  mylib:
    version: devel/new-feature
    source: git://git.example.com/lib-mylib
```

Other files that should be present in the root directory of the application are:

- **Makefile.uk** A Kconfig target file you can use to create compile-time toggles for your application.
- **.config** The selection of options for architecture, platform, libraries and your application (specified in Makefile.uk) to use with UNIKRAFT.

The .config file can be obtained from an interactive Kconfig GUI with the command

**$ kraft configure –menuconfig**

When your unikernel has been configured to your needs, you can build the the unikernel to all relevant architectures and platforms using

**$ kraft build ./my-first-unikernel**

During the building process, all the artifacts are placed in the *build/* directory including intermediate object files and unikernel images.

**External Library Development**

Similarly to the case of krafting new applications, the UNIKRAFT framework offers the possibility to port an external library and makes it available to be used in an unikernel application.
Most of the work consists of creating a UNIKRAFT Custom Makefile called Makefile.uk that the framework uses to compile the source code.
The first thing to do is to call the UNIKRAFT addlib function to register the application with the system (note the letters lib: everything in UNIKRAFT is ultimately a library). This function call will also populate APPNAME_BASE (containing the directory path to the application sources) and APPNAME_BUILD (containing the directory path to the applications build directory):

**$(eval $(call addlib,appname))**

where the name would be replaced by the applications name. In case the main application code should be downloaded as archive from a remote server, the next step is to set up a variable to point to a URL with the application sources (or objects, or pre-linked libraries, see further below) - if required.
Together with the Makefile.uk file, a Config.uk should be created and properly filled in order to customize the entries of the Kconfig menu provided by the framework for easily configuring the application or library. An example of a Config.uk file is the following:

```
menuconfig LIBNAME
      bool "name: My lib"
      select LIBNOLIBC if !HAVE_LIBC
      select LIBUKALLOC
      default n

if LIBNAME

  ### Library settings go in here ###

endif
```

**Internal Platform Development**

In UNIKRAFT, along with applications and libraries, its possible to distinguish between different target platforms.
Platforms (like KVM or Xen) are also treated as libraries in UNIKRAFT, but:

- variables have to be namespaced (ex LIBKVMPLAT) Platforms have to be
- registered in the Makefile.uk like that

**$(eval $(call addplat_s,platname,$(CONFIG_PLAT_PLATNAME)))**

**$(eval $(call addplatlib,platname,libplatnameplat))**

- A Linker script (Linker.uk) has to be provided.

- The default linker script for a platform is provided using the UK_PLAT_PLATNAME_DEF_LDS variable in the Makefile.uk of the platform library.
- The platform files have to be placed under *plat/platname/* path in the UNIKRAFT Kernel repository.
- A platform have to implement interfaces defined in *include/uk/plat*.
- Platforms dont use any external source files.
- Platforms must not have dependencies on external libraries, i.e the UNIKRAFT repo must be able to be built on its own.
- All changes/additions to *include/uk/plat* and *include/uk/arch* have to be completely independent of any library (internal and external).

## 5.2 Bug filing and support

Previously, UNIKRAFT did not have any proper mechanism for filing bugs, other than writing an email to the Xen/mini-os mailing list.
Through UNICORE, the switch over GitHub has been implemented which allows - among other features - to submit and review patches, and report bugs or issues.
This not only provides a lot more visibility into current bugs, so that the community can help with common issues they have seen; but, it also works as a good tool for commenting on bugs and linking them to actual code.
In addition, a UNIKRAFT CI/CD system has been configured to optimise the integration, testing and release process. The CI/CD tool is under NEC operational control.
Finally, additional debugging facilities have been added to UNIKRAFT, not yet upstream in official release, which include support for running a gdb server within UNIKRAFT.

# 6 Conclusions

This deliverable reports on the integration of the UNICORE project in various areas development of the toolkit components.

The major integration workstreams of the UNIKRAFT/UNICORE development have been documented, which serve as a base for the deployment of functions krafted with UNIKRAFT for the UNICORE use cases.

The integration of  Performance Optimization Tools is not described in this document as it is planned for future releases of the UNICORE toolkit.

# 7 Abbreviations and Definitions

## 7.1 Abbreviations

| | |
|---|---|
| **DoA** | Description of Action |
| **EC** | European Commission |
| **FOSS** | Free Open Source Software |
| **IPR** | Intellectual Property Rights |
| **WG** | Working group |
| **WP** | Work Package |

## 7.2 Definitions

No definition is introduced by this document

# References

[1] The unikraft github page. [Online]. Available: https://github.com/unikraft

[2] The unikraft project webportal. [Online]. Available: https://unikraft.org/

[3] The unicore project webpage. [Online]. Available: https://unicore-project.eu/

[4] Patchwork tracking system. [Online]. Available: http://jk.ozlabs.org/projects/patchwork/

[5] Concurse-ci system. [Online]. Available: https://concourse-ci.org/

[6] Simple smart contract application on unikraft. [Online]. Available: https://github.com/cs-pub-ro/app-smart-contract-simple

[7] Unikraft openssl test application. [Online]. Available: https://github.com/cs-pub-ro/app-openssl

[8] Unikraft cloud script for amazon web services. [Online]. Available: https://github.com/unikraft/plat-aws

[9] Unikraft cloud script for google compute platform. [Online]. Available: https://github.com/unikraft/plat-gcp

[10] Unikraft cloud script for digital ocean cloud platform. [Online]. Available: https://github.com/unikraft/plat-digitalocean