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

D4.1 Design & Implementation of Tools for Unikernel Deployment

Due date of deliverable: 30 September 2019
Actual submission date: 30 September 2019

Start date of project	1 January 2019
Duration	36 months
Lead contractor for this deliverable	University of Liège (ULiège)
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. The objective of this deliverable is to describe the current architecture of the toolchain as well as the different tools that have been developed to build and orchestrate unikernels. The current toolchain contains two fully-functional tools (dependency analysis tool and automatic build tool) that have been tested with different schemes and configurations. Others tools are still in research and will be released in a next milestone.

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	D4.1 Compilation toolchain
Editor	University of Liège (ULiège)
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2019 Participants in project UNICORE

Executive Summary

This is the first version of the UNICORE D4.1 document, ”Design & Implementation of Tools for Unikernel Deployment”. This document describes the current architecture of the UNICORE toolchain as well as the different tools that have been developed to build and orchestrate unikernels. The approach taken is to describe the general structure of the UNICORE toolchain and then explain each tool in a separate way. The main objective of the UNICORE toolchain is to provide a set of tools to automatically build unikernels by extracting OS primitives and selecting micro-libs and third-party libraries. Extracting components and building lightweight operating system are not the only tasks of the toolchain. Indeed, the toolchain must also provide tools to verify the correctness and security of unikernels. Furthermore, the tool must be able to profile and update configuration parameters to improve unikernels performance. In total, five different tools must be integrated to the toolchain: decomposition tool, dependency analysis tool, automatic build tool, verification tool and performance optimization tool. In addition to the UNICORE toolchain, a library pool will also be maintained. This component handles all the micro-libs (*e.g.*, schedulers, memory allocators, ...) that will be used by applications. The pool consists of a set of micro-libs and can be divided into three sub-pools: main libraries pool, platform libraries pool and architecture libraries pool. The first contains all libraries related to drivers, network stack, ... The second one contains all the libraries for a particular target platforms such as Xen, KVM and bare metal. Finally, the last one provides libraries dedicated to a particular computer architecture (*e.g.*, x86, ARM, ...). In this way, the toolchain will select the relevant micro-libs from these pools, link them against the application, eliminate used code and finally produce a UNICORE unikernel image. The current toolchain is designed to run on a UNIX/LINUX platform and is written in Golang. For now, two tools are fully-functional and have been tested with different schemes and configurations. The first one, the dependency analysis tool aims to break down a binary application by analysing its symbols, its system calls and its dependencies. The tool is a console-based application and relied on static and dynamic analysis. It was decided that the best way to gather information was to examine application binaries (source code not always accessible, independent of the programming language, binary rewriting techniques, ...). Using a static and dynamic analysis allows to retrieve as much information as possible. Indeed, results of the static analysis can be limited if the binary is stripped or obfuscated. It is why a second analysis is performed on the application. This one is dynamic since it requires running the program. To gather all symbols and dependencies of a process, it is necessary to explore all possible execution paths of the application. Nevertheless, exploring all execution paths is in general infeasible. Fortunately, it was possible to approach this scenario by using a heuristic approach which requires high application code coverage. Tests with expected input and fuzz-testing techniques have been used in order to find the most possible symbols/dependencies. Although this approach allows to gather process information, it is also a limitation since it explores only a subset of the execution paths of a program. That is why it is combined with a static analysis of the binary file. The current version of the dependency analysis tool allowed to collect results of more than 2500 applications. The second

functional tool is the automatic build tool. The tool is also divided into two components: a controller written in Go and the Unikraft build system which relies upon the existing automake component to build binaries. The build controller is directly integrated to the toolchain and interacts with the Unikraft build system. These main operations consist of matching and selecting micro-libs from the pool, generating configuration files and moving source files to a Unikraft repository. Concerning the Unikraft build system, it consists of a set of Makefiles to run the build, and a set of kConfig files to drive the menu. These two tools acts in a coordinated way in order to build unikernels. Having two different components allows to preserve the Unikraft build system without having to modify it. For now the automatic tool is fully functional with lightweight and ”basic” applications. Using complex applications results into compilation errors due to the small number of μ-libraries. With the development of new micro-libraries, the tool shall support more applications. The other tools (*e.g.*, decomposition, verification and performance optimization tool), are still in research phase therefore only analysis and hypothesis have been made about their behaviour. They will be developed in next milestones. All in all, this first document related to WP4 delivers a first usable version of the UNICORE toolchain which allows to build unikernels in a easy way. An initial release of the open source tool set has been deployed in a git repository in order that it can be used by developers. A wiki as well as a README are available in order to help developers to use or even contribute on the UNICORE toolchain.

List of Authors

Authors	Gaulthier Gain and Cyril Soldani (ULiège), Felipe Huici (NEC), Razvan Deaconescu (UPB)
Participants	ULiège, NEC, UPB
Work-package	WP4 - Toolstack implementation
Security	PUBLIC
Nature	R
Version	1.0
Total number of pages	50

Contents

List of Figures

List of Tables

1 Introduction

Deploying applications is the core business of the IT industry. Indeed, network operators, CDNs, and even Internet providers need to be able to quickly deploy their applications to deliver high-performance services to the market. These services are typically deployed on shared hardware hosted in public and/or private clouds.

Since the advent of cloud computing services, IT departments have embraced virtual machines (VMs) as a way to lower costs and increase efficiencies. Although VMs can limit the number of physical machines, they have some drawbacks. Indeed, they can take up a lot of system resources. These are very heavy since they require a complete image of the operating system to run. This quickly adds up to a lot of RAM and CPU cycles. Memory and disk space are wasted, and starting or stopping virtual machines takes several seconds at best.

For these reasons, the software industry has adopted containers to replace virtual machines, with the aim of further improving the performance of shared hardware and thus reducing costs. With containers, instead of virtualizing the underlying computer like a VM, just the OS is virtualized. Containers are as economical as traditional operating system processes, which means that they can start, stop, or migrate in less than a second. Nevertheless, as they share the host OS kernel and, usually, the binaries and libraries, the attack surface is quite large and containers are thus subject to a lot of vulnerabilities.

Offering a great trade-off between size and isolation, a new model has been defined to replace virtual machines and containers: unikernels. Also known as lightweight VMs, they are specialized VMs that include only the minimum feature(s) to run a specific application. Unikernels are thus the smallest lightweight virtual machines that can be created. In a unikernel, the application is compiled only with the necessary components of the operating system. The attack surface and the size are thus considerably reduced, resulting in better performance and protection.

The fundamental disadvantage of this approach is that it is necessary to manually port application(s) to the underlying operating system. For example, the nginx web server can be ported as a unikernel by selecting and extracting the right operating system components and primitives. Once the migration is done, verification and optimization processes are needed. Indeed, developers must verify and optimize the code and the resulting binary in order that the application is operational and optimized on the underlying platform and architecture, which is a very tedious and complex job for developers.

To circumvent these drawbacks, the aim of UNICORE is to develop a toolkit comprising a set of tools to automatically build images of operating systems targeting a single (or multiple) applications that are optimized to run on bare metal or as virtual machines. In this way, the resulting image(s) will have lower image size, boot time, and amount of memory used. This will ensure both strong isolation and performance comparable to solutions such as containers.

1.1 Objectives

The main objective of this deliverable is to analyse and explain the UNICORE toolchain. Each tool developed will be detailed and its general operation will be explained. As the toolchain is being developed, some tools have not yet been developed and are therefore not operational. In this case, an analysis as well as hypothesis concerning the behaviour of the tool are mentioned.

1.2 Organization

This document is organized as follows. Chapter 2 gives a general overview of the UNICORE toolchain, and discusses its general principles. Chapter 3 describes the analysis concerning the decomposition tool. Chapter 4 gives an overview of the dependency analysis tool. Chapter 5 describes the automatic build tool. Chapter 6 and 7 respectively provide an analysis of the verification tool and the optimization tool. Finally, Chapter 8 summarises what has been achieved and any shortcomings that have been identified.

2 General Overview

This chapter will first define the general principles of the UNICORE toolchain and will briefly present the tools to develop and to incorporate in the toolkit. Furthermore, general requirements will be discussed in this chapter.

2.1 UNICORE Toolchain

The UNICORE toolchain will provide a set of tools to automatically build images of operating systems targeting applications. In a general way, the toolchain will build unikernels by extracting OS primitives and selecting micro-libs and third party libraries. Figure 2.1 illustrates the high-level concept of UNICORE.

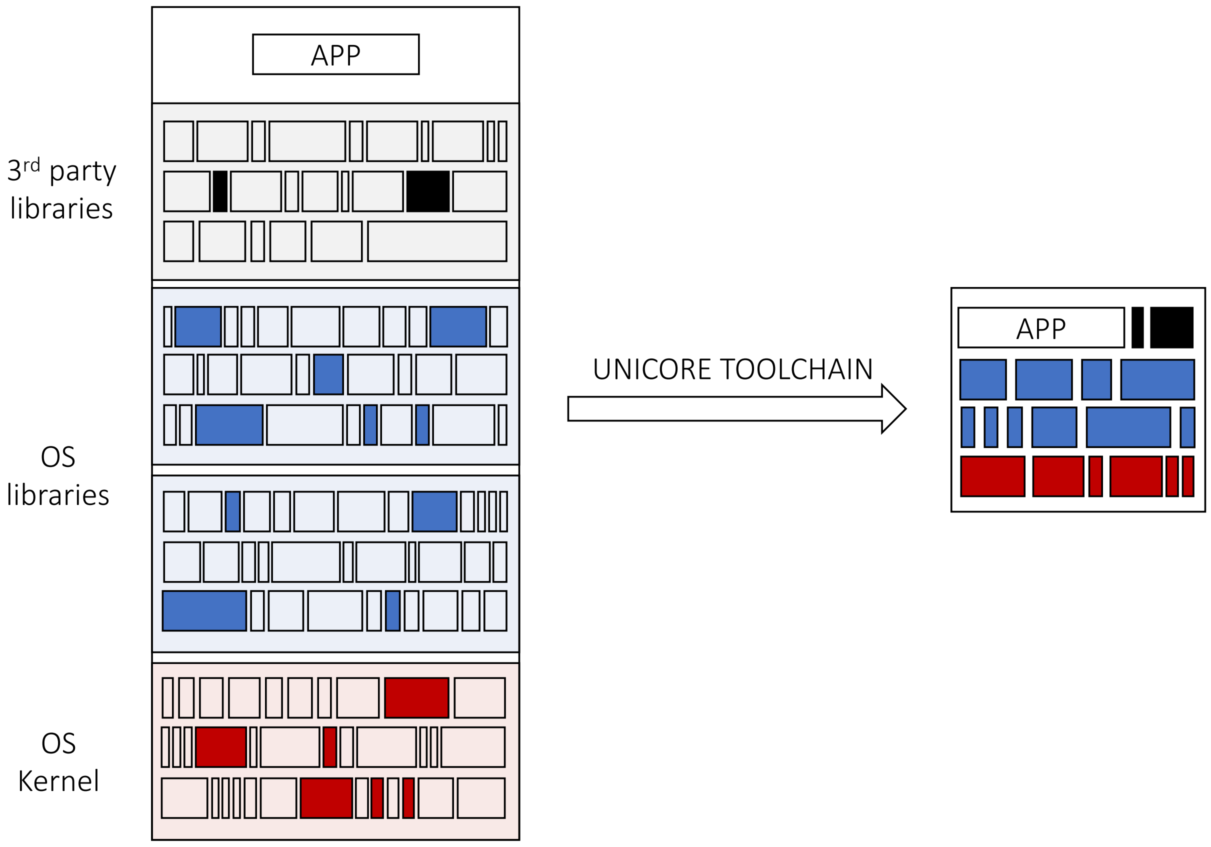


Figure 2.1: High-level overview concept of UNICORE

In order to be able to build secure and reliable unikernels, several tools must be integrated into the toolchain. This one will include the following tools:

(i) **Decomposition tool** to assist developers in breaking existing monolithic software into smaller components.

(ii) **Dependency analysis tool** to analyse existing, unmodified applications to determine which set of libraries and OS primitives are absolutely necessary for correct execution.

(iii) **Automatic build tool** to match the requirements derived by the dependency analysis tools to the available libraries constructed by the OS decomposition tools.

(iv) **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.

(v) **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 combination of these tools is shown in Figure 2.2 and represents a unikernel toolkit able to automatically build efficient, customized, verifiable and secure specialized operating systems and virtual machine images. With a such toolchain, applications become independent of the system against which they are built, not only providing vertical but also horizontal scaling across diverse infrastructures (*e.g.*, Linux, FreeBSD, etc).

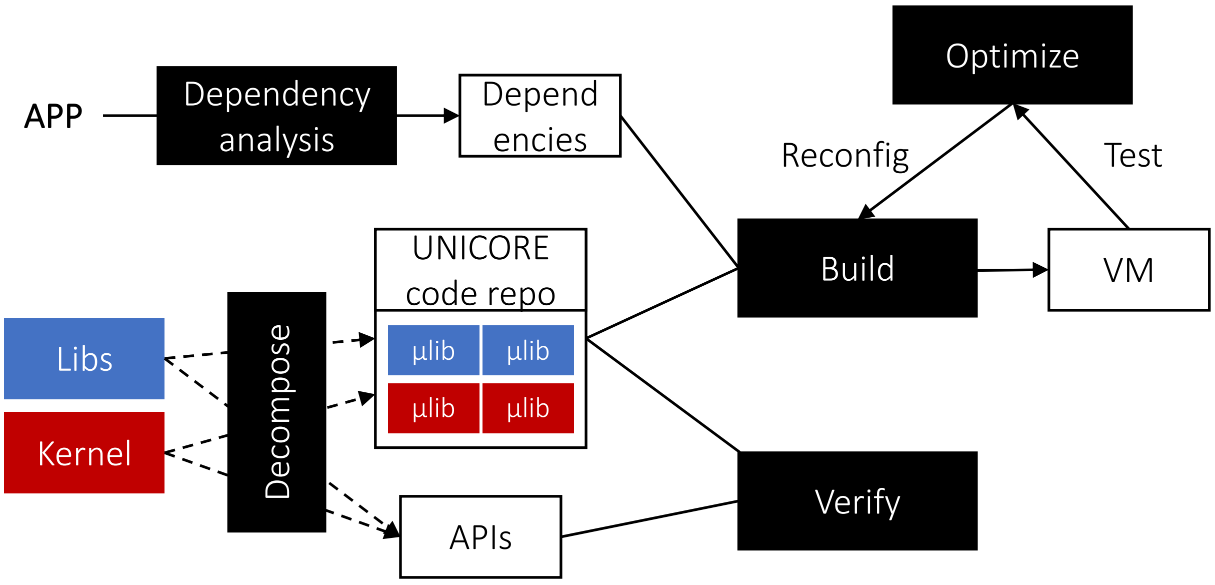


Figure 2.2: The UNICORE toolchain

The upper part of the figure shows the workflow for users of UNICORE: a dependency analysis tool will examine and extract dependencies from existing, unmodified applications. The user will then invoke the UNICORE build tool, specifying one or multiple target platforms (*e.g.*, a hypervisor such as Xen or KVM). The tool will use the dependencies to select the relevant micro-libs from a library pools (see Section 2.2), link them against the application, eliminate used code and produce a UNICORE unikernel image which can then be deployed using standard provisioning tools such as Openstack, Docker or Kubernetes. Finally, the resulting unikernel will be automatically profiled and the configuration parameters for the micro-libs changed to improve performance, in an iterative process driven by the optimization tool.

2.2 UNICORE library pools

In addition to UNICORE toolchain, a library pools will also be maintained. This component handles all the micro-libs that will be used by applications. The pool consists of a set of micro-libs and can be divided into three sub-pools:

(i) **Main lib pool** will contain all the libraries related to drivers, network stack, memory allocator, ...

(ii) **Platform lib pool** will contain all the libraries for a particular target platforms such as Xen, KVM and bare metal.

(iii) **Architecture lib pool** will provide libraries dedicated to a particular computer architecture.

This type of procedure allows great flexibility and better efficiency since micro-libraries are smaller. It is therefore easier to verify their behaviour. Figure 2.3 illustrates the UNICORE library pools.

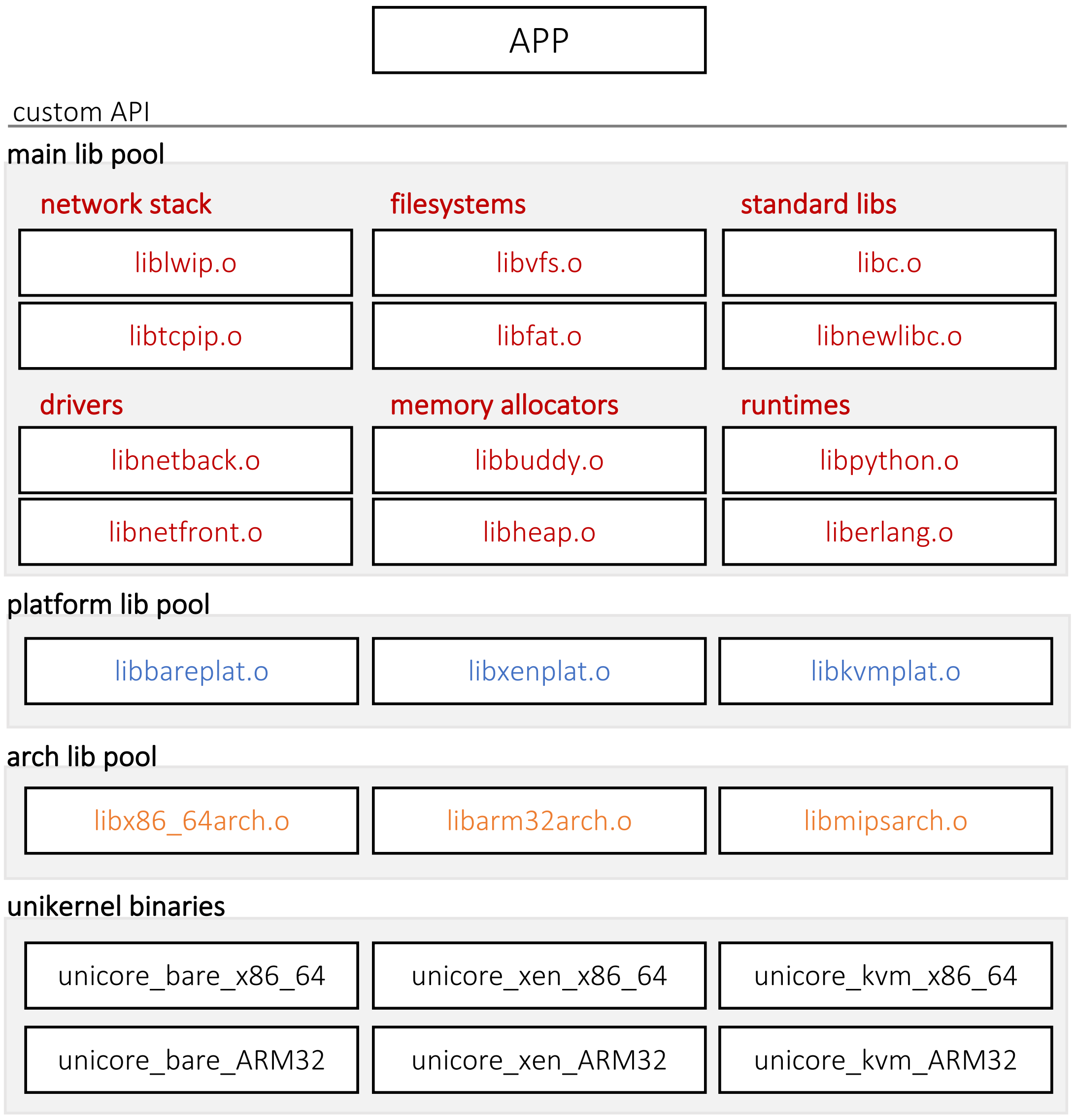


Figure 2.3: The UNICORE library pools

2.3 General requirements

In general, the toolchain has been designed to run on a UNIX/LINUX platform. In future versions, another platform such as Windows can also be considered to support the toolchain. The toolkit has been designed with the Golang language[1] (also known as Go). The choice of the programming language of the toolchain has been analysed at length. The trade-off was to find a powerful but easy-to-use programming language. Higher level languages such as Java[2] and Python[3] are poorly efficient compared to C and C++. Nevertheless, large C/C++ projects can therefore be relatively difficult to maintain due to languages limitations[4]. Go brings best of both the worlds. Like lower level languages, Go is compiled language. It offers thus good performance combined with a gradual learning curve. The requirements of the toolchain as well as the tools are further explained in deliverable D2.1[5].

2.4 Deployment

An initial release of the open source tool set has been deployed in a git repository in order that it can be used by developers. This one is available at the following address: https://github.com/unikraft/tools. A wiki as well as a README have been written in order to help developers to use or even contribute on the UNICORE toolchain. All files are licensed under the BSD license.

3 Decomposition Tool

The decomposition tool will be used to break down monolithic libraries such as libc and operating system primitives (*e.g.*, memory allocators, network stack, ...) into a set of small modules that can be selected from a libraries pool to build unikernel(s). The tool will help developers in decomposing the software, and is targeted at the UNICORE consortium and not the software community at large.
Another objective was to identify dependencies between functions and libraries, as well as dependencies between different libraries that are known by package managers such as dpkg, yum or apt. Then, these relations are displayed in graphical form to help the human expert understand the interactions between different components. For this part, it was chosen to merge this identification procedure within the dependency analysis tool (see Chapter 4).

3.1 General overview

The decomposition tool is still in research phase therefore only some assumptions are established. Two areas of research were considered and needs to be further analysed.

3.1.1 Decomposing Software into Subsystems

Architecture of large and complex systems is structured into a series of packages. One goal of an architectural decomposition is to provide a way to better understand the source. Considering the Linux kernel, this one implements a number of important architectural attributes. At a high level, and at lower levels, the kernel is layered into a number of distinct subsystems. This model proposes a regulatory system that classifies services based on their common characteristics. Major components of the Linux kernel are illustrated in Figure 3.1.

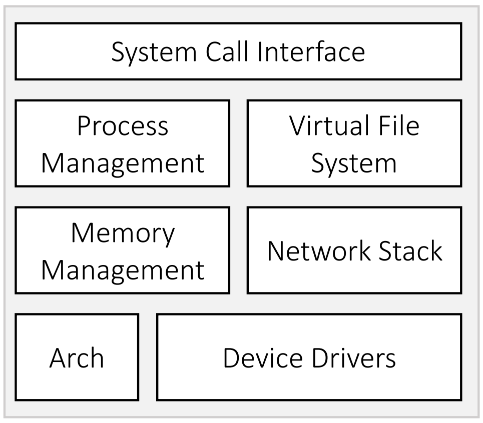


Figure 3.1: Architectural perspective of the Linux kernel

The decomposition process consists of three main steps:

(i) Treat all source files from specified directories built into the Linux kernel as subsystems and perform incremental decomposition isolating one kernel subcomponent at a time;

(ii) When kernel subsystems have been isolated, patterns recognition techniques can be used to extract relevant files and blocks of code. The idea is to provide as input, all the symbols that are used by a particular application and that are unknown by the UNICORE build system (see Chapter 5);

(iii) These components are then integrated with each other where unknown functions and symbols are replaced by stubs.

The first objective of the tool is thus to help experts to understand the interactions between different components and to obtain a first skeleton of a micro-library. After this semi-automatic extraction, developers will have to work on their own by implementing and verifying all stub functions to have fully functional modules.

3.1.2 Using existing tools

Another approach is to use open-source tools like Clang Static Analyser[6] and to integrate them to the toolchain. Such tool(s) would allow to perform an analysis of the application flow code. Once the application flow code has been identified, a graph can be generated in order to help developer(s) to extract the identified sub-systems.

4 Dependency Analysis Tool

The objective of the dependency analysis tool is to gather, for the target applications, which software in the operating system they actually use. Such software will include shared libraries, other applications, core kernel components, kernel modules, and so forth. The tool needs to find a sufficient, but minimal superset of other software that must be installed for the application to work correctly.

4.1 General overview

A first functional dependency analysis tool has been developed. This one aims to break down a binary application and analyses its symbols, its system calls and its dependencies. The tool is a console-based application and relied on static and dynamic analysis.

The tool has been designed to examine binary files. Indeed, source code is not always accessible. Moreover, this way of doing is independent of the programming language and is thus easier to use to gather binary information. Finally, gathering data of binary is also useful for binary rewriting techniques[7]. Nevertheless, code parsing tools can also be considered in future release(s) even if they require more time and resources to develop. Figure 4.1 represents a high-level overview of the tool.

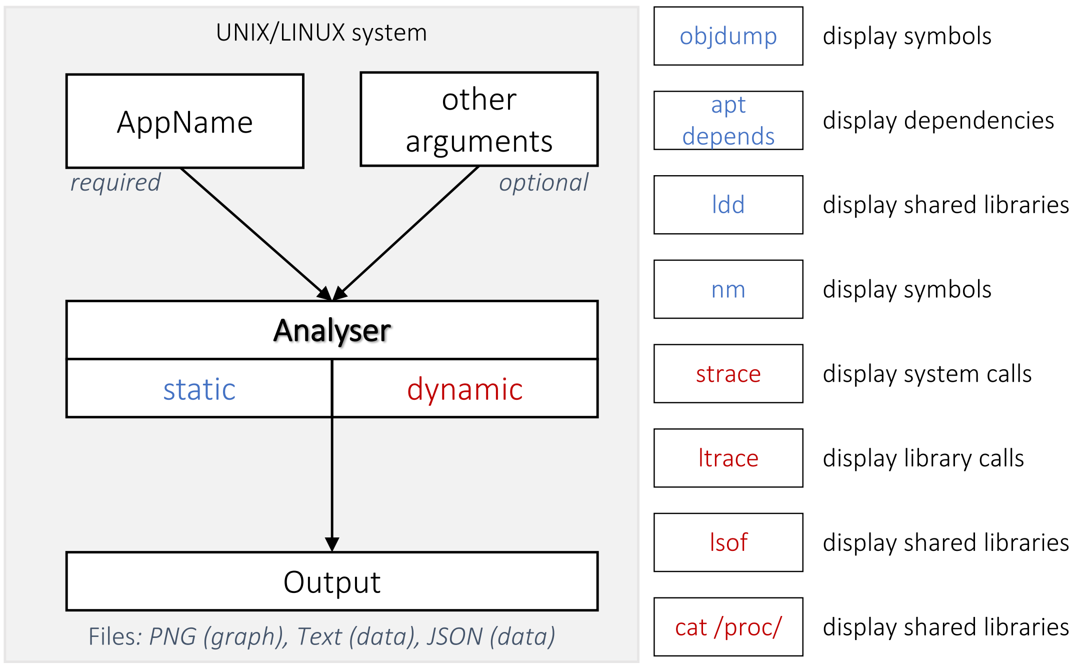


Figure 4.1: High-level overview of the UNICORE dependency analysis tool

Firstly, the dependency analysis tool requires the application name as an argument to gather binary information. This one can either be absolute or simply passed as an executable file name (*e.g.*, sqlite). In that case, the tool will search in the path environment variable to locate the executable file and gets its absolute path (*e.g.*, /usr/bin/sqlite). Others arguments will be discussed later.

A static analysis is performed on the binary file of the application. It allows to recover all the symbols which compose it. In order to perform such a task, the tool uses several internal commands such as `nm`, `objdump`, `apt-cache depends` and `ldd`. The output of each command is then parsed and various information is stored into adequate data structures. However, the result of this analysis can be limited since binaries can be stripped or obfuscated.

A second analysis is thus performed on the application. This one is dynamic since it requires running the program. It allows to collect various information such as system calls, library calls and shared libraries. To gather all the symbols and dependencies of a process, it is necessary to explore all possible execution paths of the application. Nevertheless, exploring all execution paths is in general infeasible. Fortunately we can approach this scenario, by using a heuristic approach which requires high application code coverage. Tests with expected input and fuzz-testing techniques have been used in order to find the most possible symbols and dependencies (see Section 4.2). As for the static analysis, several internal commands are executed: `strace`, `ltrace`, `lsof`, `cat` and `/proc/pid`. The difference here is that the program is executed. Binary information is also saved into several data structures. When both analysis are completed, a JSON file is automatically generated. For example, considering the SQLite[8] program, a summary of the resulting JSON file is shown below:

```
"static_data": {
    "dependencies": {
      "liblzma5": [
        "libc6"
      ], ...
    }
    "shared_libs": {
      "libpthread.so.0": [
        "libc.so.6"
      ], ...
    },
    "system_calls": {
      "access": "",
      "exit": ""
      , ...
    },
    "symbols": {
      "access": "GLIBC_2.2.5"
      , ...
    }
}
```

Listing 4.1: data gathered from static analysis

```
"dynamic_data": {
    "shared_libs": {
      "libpcre.so.3": [
        "libpthread.so.0",
        "libc.so.6"
      ], ...
    },
    "system_calls": {
      "access": "",
      "arch_prctl": ""
      , ...
    },
    "symbols": {
      "access": "",
      "fflush": ""
      , ...
    }
}
```

Listing 4.2: data gathered from dynamic analysis

A complete detailed output of the SQLite program is shown in Appendix 8. In addition to the JSON file, the tool can automatically generate dependencies and shared libraries graphs. Such graphs allow to illustrate the relation between the dependencies. For example, considering again SQLite, Figure 4.2 represents all shared libraries (as well as their dependencies) acquired during the dynamic dependency analysis.

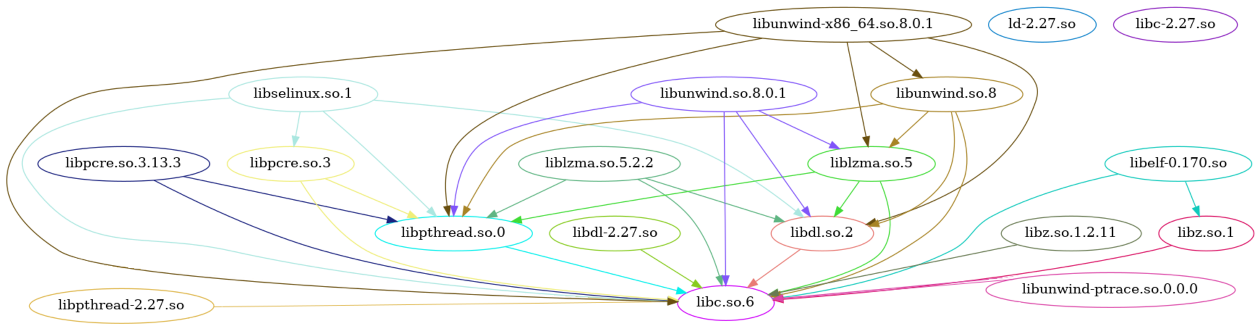


Figure 4.2: Shared libraries required by SQLite (dynamic analysis)

Required dependencies gathered during the static analysis procedure are shown in Figure 4.3.

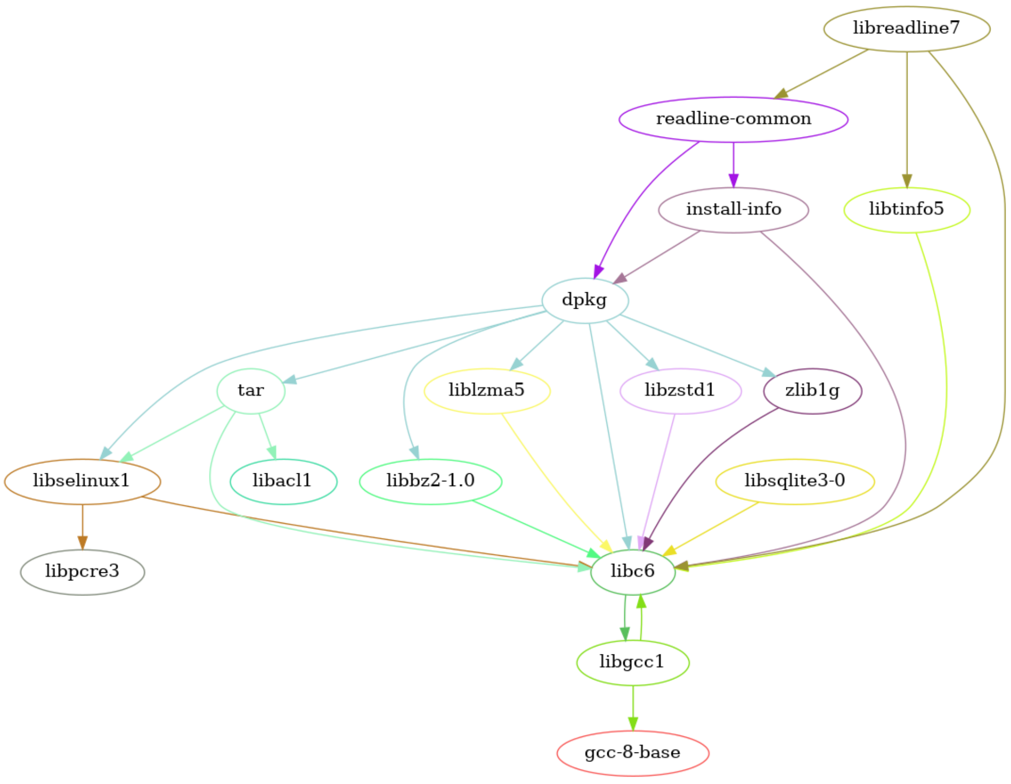


Figure 4.3: Dependencies required by SQLite (static analysis)

4.2 Tests and Fuzz-Tests

For the dynamic analysis, different tests were used to explore execution paths of a particular application. In order to do it, the dependency tool provides two different ways: either passing a test file as argument which contains internal commands to test the current application (*e.g.*, SQL queries for a database, ...) or either to perform manually tests by specifying a duration to test.

For each application, it is thus necessary to manually write the tests or test the program. In addition, tests can use the default configuration of an application. As a result, it is possible that not all execution paths are tested.

4.3 Results

The tool made it possible to gather some results. These were obtained by running the tool on a multitude of applications. These are divided into several categories:

(i) HTTP servers;

(ii) Database servers;

(iii) DNS servers;

(iv) Mail servers;

(v) Monitoring tools;

(vi) Miscellaneous.

Static and dynamic analysis have been performed on each category except the last one. Indeed, only static analysis has been executed on the miscellaneous category. It contains various applications from the popcorn debian[9] website. A big majority of applications in this folder are basic utilities and tools such as xxd, hexdump, ...

Considering database servers, it was possible to collect the following results. These results show how easy it is to gather useful data using the dependency analysis tool. In order to have a clean display, some symbols, system calls or dependencies are skipped in some tables. Furthermore, only results about database servers are shown. It should be considered that these results are also available for other categories. Table 4.1 represents the most used shared libraries by databases.

%	**Shared libraries**
84.62	libpthread-2.27.so, ld-2.27.so, libc-2.27.so
76.92	libdl-2.27.so
69.23	libpthread.so, libm-2.27.so
53.85	libnss_files-2.27.so, libm.so, libdl.so
46.15	libz.so
38.46	libnsl.so, libgcc_s.so, libnsl-2.27.so, libnss_compat-2.27.so, libnss_nis-2.27.so, libnss_files.so, libstdc++.so
30.77	librt-2.27.so, libresolv-2.27.so
23.08	libcrypt-2.27.so, liblz4.so, libresolv.so, libsasl2.so, libcrypto.so, libssl.so
15.38	librt.so, libffi.so, liblber-2.4.so, libldap_r-2.4.so, libwind.so, libgmp.so, libnettle.so, libroken.so, libhx509.so, libcrypt.so, libp11-kit.so, libcom_err.so, libhcrypto.so, libheimntlm.so, libkrb5.so, libasn1.so, libgssapi.so, libtasn1.so, libheimbase.so, libunistring.so, libidn2.so, libgnutls.so, libhogweed.so, libsqlite3.so, libpcre.so, libnss_dns-2.27.so, libnss_mdns4_minimal.so, liblzma.so, libicuuc.so, libicudata.so
7.69	**43** others

Table 4.1: Usage of shared libraries for database servers

It also possible to visualize which system calls are the most used by database servers. Table 4.2 represents these results.

%	**System calls**
84.62	arch_prctl, set_robust_list, mprotect, close, rt_sigaction, rt_sigprocmask, mmap, write, fstat, openat, futex, brk, socket, access, set_tid_address, read, prlimit64, munmap, execve
76.92	stat, clone, lseek
69.23	setsockopt, bind, fcntl
61.54	uname, connect, fsync, getpid, listen, getsockname
53.85	getcwd, getdents, ioctl
46.15	unlink, lstat, geteuid, accept, sendto, getuid, exit_group, epoll_ctl, epoll_wait
38.46	poll, chdir, umask, madvise, sysinfo, pipe
30.77	nanosleep, sched_yield, gettid, recvfrom, pread64, pwrite64, getppid, fdatasync, dup, dup2, readlink, sched_getaffinity, ftruncate, rt_sigreturn, select, getrusage, rename, getpeername, wait4, recvmsg, epoll_create1
23.08	exit, mkdir, fallocate, setsid, clock_getres, getegid, getgid, getsockopt, statfs
15.38	setuid, shutdown, rt_sigtimedwait, setgid, kill, getrandom, epoll_create, pipe2, faccessat, WIFEXITED, sendmsg, chmod, fadvise64, flock, accept4
7.69	getpriority, rmdir, setgroups, setpriority, times, io_setup, io_submit, io_getevents, htons, getresgid, getresuid, pselect6, creat, fchown, bash, chown, getpgrp, fchmod, msync, prctl, writev, eventfd2, mlock, timerfd_settime, mincore, timerfd_create, sigaltstack, renameat, getdents64, unlinkat, mkdirat, setitimer, shmget, shmdt, shmat, fchdir, socketpair, sendmmsg, setpgid

Table 4.2: Usage of system calls for database servers

After system calls, the most used library calls (symbols) can also be retrieved. Table 4.3 shows their usage.

%	**Library calls**
69.23	pthread_mutex_lock, pthread_mutex_unlock
61.54	pthread_mutex_init, strlen, memset, close, fclose, __errno_location, malloc, getenv, __ctype_b_loc
53.85	strtol, memcmp, read, getpid, time, memcpy, strncmp, calloc, memmove, strchr, free, fcntl
46.15	sigaction, sigemptyset, pthread_create, strcpy, gethostname, pthread_cond_init, strcmp, gettimeofday, strdup, write, getaddrinfo, __snprintf_chk
38.46	chdir, fflush, geteuid, fileno, __memcpy_chk, strncpy, strrchr, __cxa_atexit, pthread_key_create, pthread_attr_init, strcasecmp, strstr, fopen, open, signal, sysconf, socket, __vsnprintf_chk, realloc, setsockopt, bind
30.77	sigfillset, pthread_setspecific, unlink, closedir, fopen64, opendir, open64, pthread_mutexattr_settype, pthread_mutexattr_init, pthread_mutex_destroy, pthread_attr_getstacksize, umask, __fprintf_chk, fgets, clock_gettime, strtoul, fputs, pthread_sigmask, sigaddset, dup, snprintf, __xstat, memchr, getcwd, freeaddrinfo, pipe, exit, setlocale, getuid, getrlimit
23.08	pthread_getspecific, gmtime_r, pthread_attr_setstacksize, localtime_r, dup2, pthread_cond_signal, __xstat64, __cxa_guard_acquire, __cxa_guard_release, getrlimit64, pthread_cond_broadcast, setrlimit64, sscanf, lseek64, pthread_cond_wait, strftime, _setjmp, log, setsid, __sprintf_chk, uname, strtod, fwrite, listen, epoll_ctl, __fxstat64, fsync, rename, fork, __fxstat, ioctl, readdir, qsort, fdatasync, lseek, __strcpy_chk, mmap, feof
15.38	**77** others
7.69	**337** others

Table 4.3: Usage of library calls for database servers

It is also possible to perform further analysis on the gathering results. Indeed, it is possible to discriminate which applications are threaded. Table 4.4 shows which program uses the pthreads library, the `fork` function or both.

Application	use pthreads	use fork
mysql(Server)	✓	✓
mariadb		
virtuoso	✓	✓
memcached	✓	✓
groonga	✓	✓
sqlite	✓	✓
firebird	✓	✓
sun-javadb-core		
mongodb	✓	✓
postgres		✓
redis	✓	✓
influxdb	✓	
mysql(Client)	✓	

Table 4.4: Usage of `fork` and pthread library by application

4.4 Limitations and improvements

As mentioned above, fuzz testing is in itself a limitation. Indeed, it explores only a subset of the execution paths of a program. That is why it is combined it with a static analysis of the binary file. Nevertheless, this one has also some limitations (binary stripped or obfuscated). A considerable improvement will be to develop a testing framework to test a large number of applications in order to gather as much information as possible. Another limitation concerns the type of application. If the application is not an executable file (ELF), some gathering procedures are omitted. Its due to some limitations of the ptrace module. More specially with the library calls procedure. This limitation reduces the number of applications that use particular symbol(s)/library(s). Nevertheless, only a small part of applications is impacted. Therefore, it allows to get significant results in the dependency analysis part.

5 Automatic Build Tool

The objective of the automatic build tool is to automatically build an OS image (kernel and filesystem) that can run, out of the box, the target application(s).

5.1 General overview

As for the dependency analysis, a first version of the automatic build tool has been developed. The tool is also divided into two components: a controller written in Golang and the Unikraft build system which relies upon the existing autoconfigure/automake[10] combination to build binaries and the existing package management tools such as apt on Ubuntu or yum on Redhat. Figure 5.1 represents these two components.

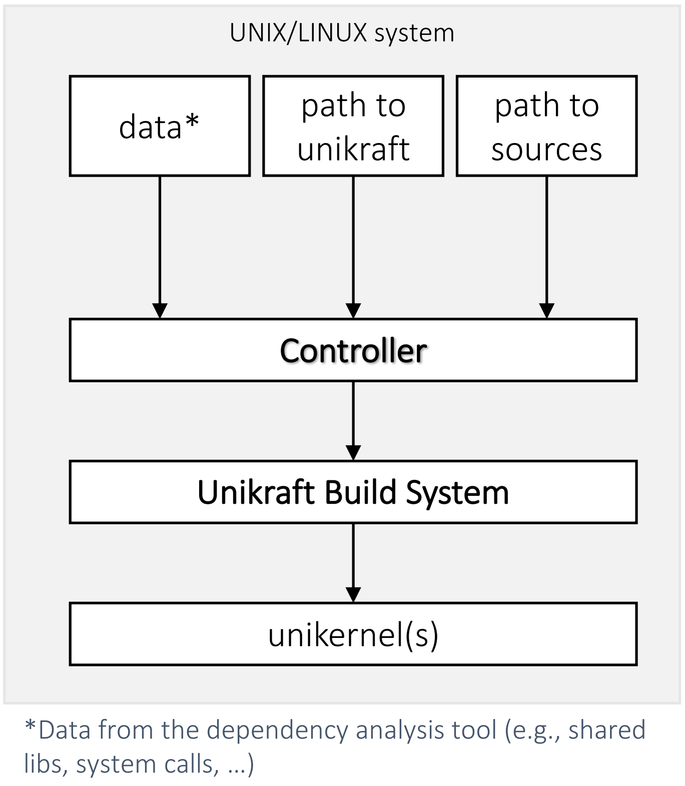


Figure 5.1: High-level overview of the UNICORE automatic build tool

Having two different components allows to preserve the Unikraft build system without having to modify it. It allows to get a greater flexibility since each component has a very specific behaviour.

5.1.1 Controller component

The build controller is directly integrated to the toolchain and is also written in Go. This component requires three arguments as input: all the pieces of data (*e.g.*, shared libraries, system calls, library calls, ...) gathered by the dependency analysis tool, the path to the application source files and the path to the Unikraft workspace (see Section 5.1.2).

In a first time, the tool detects the programming language used by the application and moves all source and header files into a newly created folder within the Unikraft workspace. All headers files are added into a specific folder. Afterwards, the matching process between shared libraries (gathered from the dependency analysis process) and the micro-libs (from the library pool) is performed. The matching is quite simple and is based into two parts:

(i) First, a simple match is performed based on the name of the shared library and the name of the μ-library. For example, if the *libpthread.so* shared library is used by an application, the resulting μ-library that will be selected from the library pool is *pthread-embedded*.

(ii) There are also an additional matching that selects the right μ-library according to symbols (function names) of the application. For example, an application which uses network related functions such as `socket`, `gethostbyname`, ... requires a networking library. In that case, the tool will automatically select the *lwip* library. Internally, a function matching score based on symbols usage is computed to select the right μ-libraries. Indeed, the tool will read from the library pool, the *exportsyms.uk* file of each library (see Section 5.1.2), save their symbols and then perform the matching. Note that for external libraries, symbols file is directly read from git repositories avoiding to download, on the host system, all the content of the μ-library.

Selecting the right μ-libraries is performed by fetching and cloning those from an external git-repository platform: xenbits.xen.org. If those are already available on the host system, the cloning process is skipped and local repositories are directly considered by the tool.
Once the μ-libraries are selected, configuration files and Makefiles are automatically generated and the tool calls the make utility to build the application unikernel. In that case, several scenarios are to consider:

(i) The simplest scenario is that the compilation and linking process went smoothly. In this case, one or several resulting image(s) is/are created in a build directory.

(ii) If the linking process fails due to undefined references, the tool will handle this case by automatically created a new source file where all undefined references will be replaced by stub functions. Developer(s) should implement those to have a fully working application.

(iii) Finally, the worst case concerns a failed compilation. Indeed, if there are series of compiler errors, these are displayed on the console and developer(s) must manually fix them.

All these operations are illustrated in Figure 5.2.

5.1.2 Unikraft Build system component

As explained earlier, we chose to rely on the existing Unikraft build system. The current version of Unikraft supports multiple platforms (*e.g.*, Xen, KVM, Solo5, Firecracker) and CPU architectures (*e.g.*, ARM, x86). It consists of three basic components:

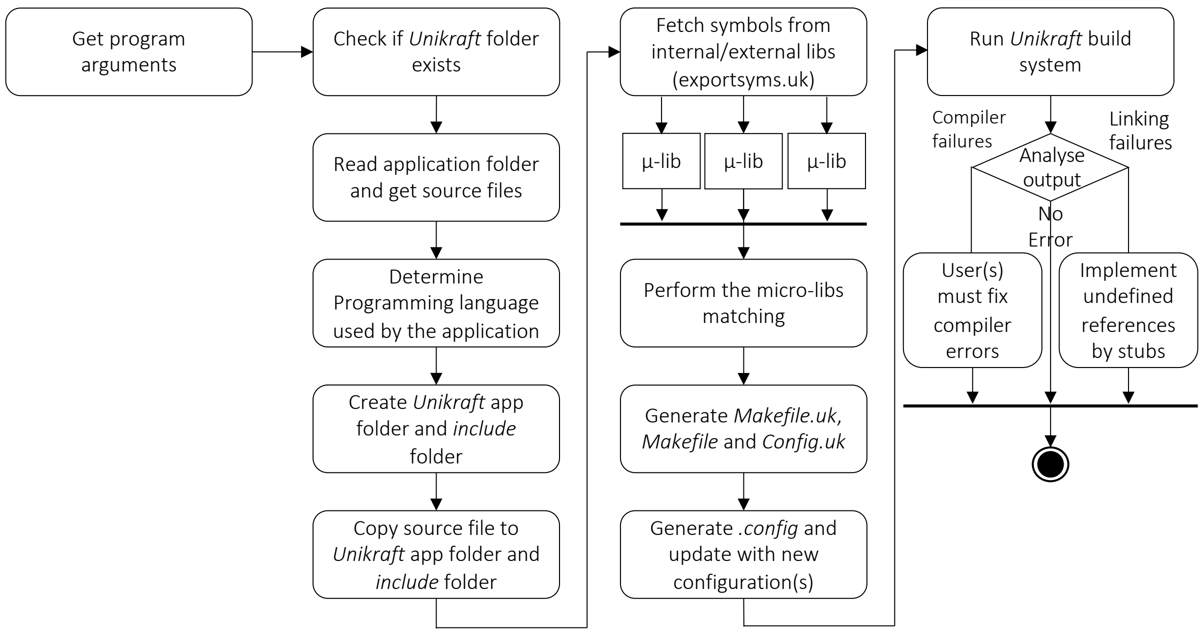

Get program arguments
Check if Unikraft folder exists
Read application folder and get source files
Determine Programming language used by the application
Create Unikraft app folder and include folder
Copy source file to Unikraft app folder and include folder
Fetch symbols from internal/external libs (exportsyms.uk)
µ-lib
µ-lib
µ-lib
Perform the micro-libs matching
Generate Makefile.uk, Makefile and Config.uk
Generate .config and update with new configuration(s)
Run Unikraft build system
Compiler failures
Linking failures
Analyse output
No Error
User(s) must fix compiler errors
Implement undefined references by stubs

Figure 5.2: Flow chart of the UNICORE automatic build tool (controller part)

(i) **Library Pools** are Unikraft modules, each of which provides a basic piece of functionality.

(ii) **Configuration Menu**: Inspired by Linux's Kconfig system, this menu allows users to pick and choose which libraries to include in the build process, as well as to configure options for each of them, where available. Like Kconfig, the menu keeps track of dependencies and automatically selects them where applicable.

(iii) **Build Tool**: Based on make, it takes care of compiling and linking all the relevant libraries, and of producing images for the different platforms selected via the configuration menu.

Unikraft contains a well structured hierarchy of repositories and packages. The general architecture is represented in Table 5.1.

Packages	Description
arch	contains all the build configuration files related to a particular architecture (x86/arm)
doc	contains the official documentation
include	contains the header files for architecture (x86, ARM) and platform (KVM, Xen, Linux userspace) interface
lib	contains all the internal libraries (nolibc, ukdebug, ukargparse, ...)
plat	contains all the code related to a particular platform (KVM, Xen, Linux userspace)
support	contains the files related to the Kconfig menu and additional scripts

Table 5.1: Unikraft architecture

An overview of the build system architecture is shown in Figure 5.3. As previously mentioned, Unikraft is a modular system based on micro-libraries, some of which are *internal*, meaning that they provide low-level functionality such as operating system primitives or CPU architecture specific code and are part of the main

Unikraft repository; and *external* libraries, which tend to provide higher-level functionality (e.g., openSSL, protobuf, etc.) and have their own repository fully independent of the main Unikraft one.

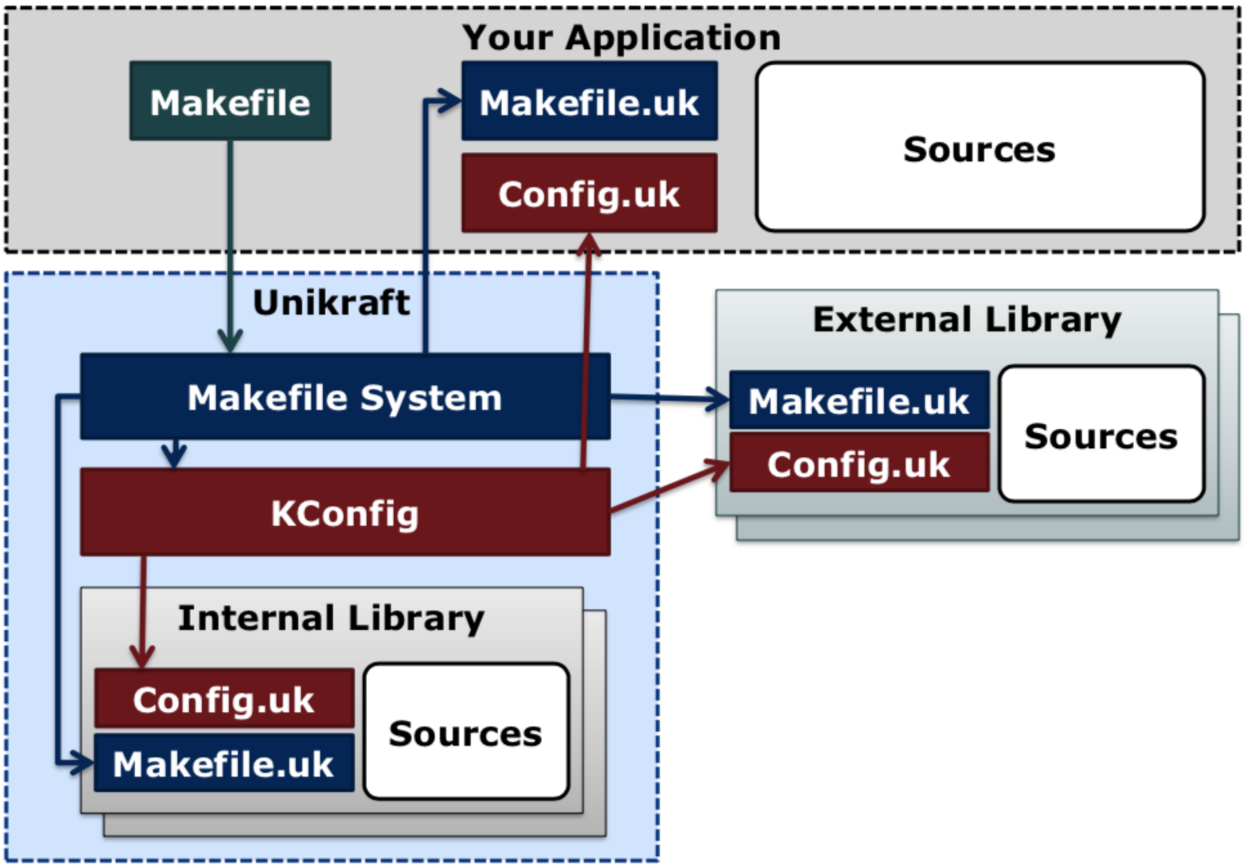
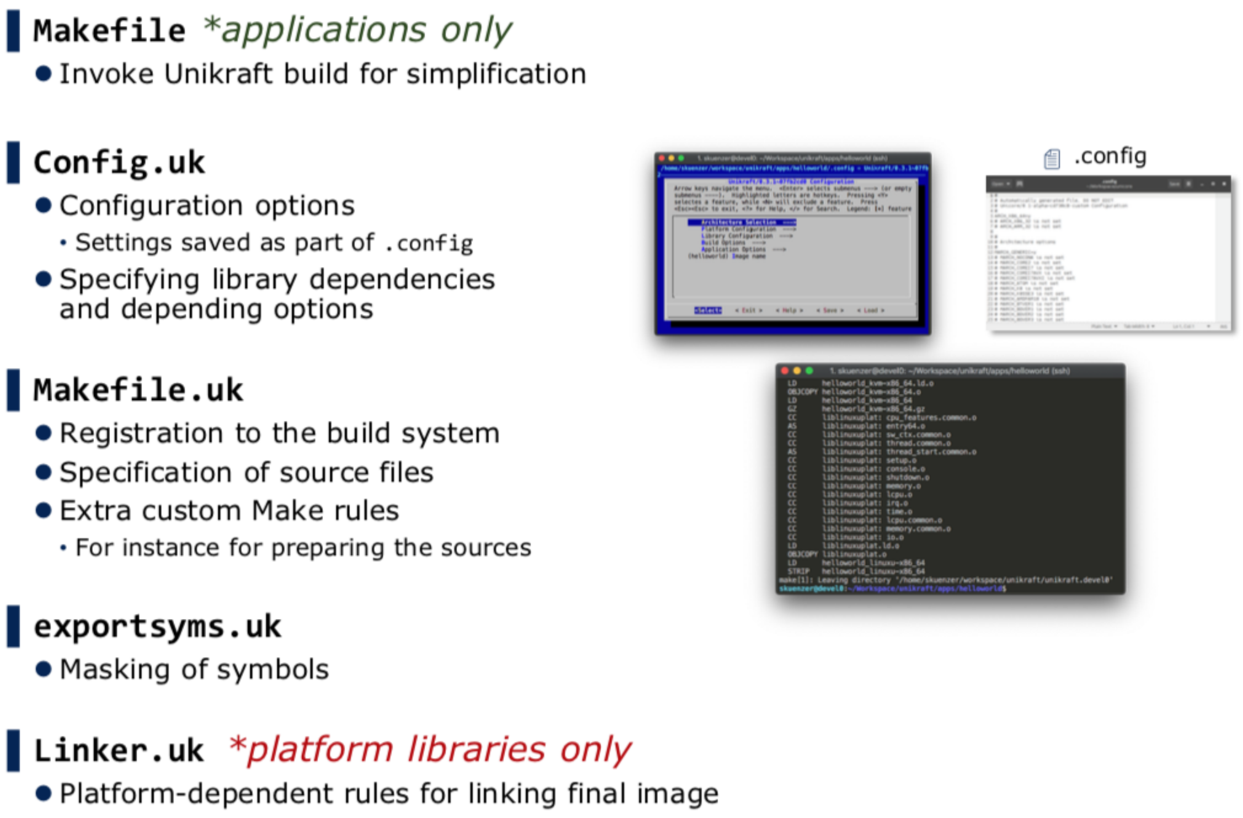


Figure 5.3: Unikraft's build system architecture

The Unikraft build system, contained within the `support` sub-directory of the main Unikraft repository, consists of a set of Makefiles to run the build, and a set of kConfig files to drive the menu and generation of `.config` configuration files.

As shown in Figure 5.4, a Unikraft library, whether internal or external, consists of a set of required files:

- **Config.uk**: Specificies nodes/items for the library that should be populated in the kConfig menu.
- **Makefile.uk**: The library's main Makefile. This is a standard Makefile, except it must follow certain Unikraft-specific formatting and variable rules. It must also specify the library's source code files.
- **exportsyms.uk**: Only symbols listed in this file actually make it to the final image. This is to ensure that functions that are private to the library (in other words, not belonging to the library's public API), are not exposed to the other libraries in the build, thus preventing potential name clashes. Clashes can easily occur since most libraries are essentially individual, independent software projects that often don't expect to be built with other projects, and so are not properly namespaced.
- **Linker.uk**: This file applies only to platform libraries (platform libraries provide support for generating Unikraft images than can run on technologies such as KVM, Xen, or OCI containers). Where needed,



Makefile **applications only*
- Invoke Unikraft build for simplification

Config.uk
- Configuration options
  - Settings saved as part of .config
- Specifying library dependencies and depending options

Makefile.uk
- Registration to the build system
- Specification of source files
- Extra custom Make rules
  - For instance for preparing the sources

exportsyms.uk
- Masking of symbols

Linker.uk **platform libraries only*
- Platform-dependent rules for linking final image

```
1. skuenzer@devel0: ~/Workspace/unikraft/apps/helloworld (ssh)
/home/skuenzer/workspace/unikraft/apps/helloworld/.config - Unikraft/0.3.1-07fb
              Unikraft/0.3.1-07fb2cd0 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

         Architecture Selection  --->
         Platform Configuration  --->
         Library Configuration  --->
         Build Options  --->
         Application Options  --->
     (helloworld) Image name

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

```
1. skuenzer@devel0: ~/Workspace/unikraft/apps/helloworld (ssh)
  LD       helloworld_kvm-x86_64.ld.o
  OBJCOPY  helloworld_kvm-x86_64.o
  LD       helloworld_kvm-x86_64
  GZ       helloworld_kvm-x86_64.gz
  CC       liblinuxuplat: cpu_features.common.o
  AS       liblinuxuplat: entry64.o
  CC       liblinuxuplat: sw_ctx.common.o
  CC       liblinuxuplat: thread.common.o
  AS       liblinuxuplat: thread_start.common.o
  CC       liblinuxuplat: setup.o
  CC       liblinuxuplat: console.o
  CC       liblinuxuplat: shutdown.o
  CC       liblinuxuplat: memory.o
  CC       liblinuxuplat: lcpu.o
  CC       liblinuxuplat: irq.o
  CC       liblinuxuplat: time.o
  CC       liblinuxuplat: lcpu.common.o
  CC       liblinuxuplat: memory.common.o
  CC       liblinuxuplat: io.o
  LD       liblinuxuplat.ld.o
  OBJCOPY  liblinuxuplat.o
  LD       helloworld_linuxu-x86_64
  STRIP    helloworld_linuxu-x86_64
make[1]: Leaving directory '/home/skuenzer/workspace/unikraft/unikraft.devel0'
skuenzer@devel0:~/Workspace/unikraft/apps/helloworld$
```

Figure 5.4: Unikraft's library files, menu system, and resulting configuration file.

a platform library may specify a platform-specific linker script which will override Unikraft's default platform linker script.

- **Makefile**: Applicable to applications only, this file is used mostly to specify the location of Unikraft's main repository, as well as that of any external libraries and external platforms it may depend on.

Further shown in Figure 5.4 are screen captures of the system's menu, a sample resulting configuration file, and console output from a successful Unikraft build.

For further information, complete documentation of Unikraft can be found at the following address: Unikraft's Documentation.

5.2 Results

For now the automatic tool is fully functional with lightweight and "basic" applications such as SQLite and mini_httpd[11]. Using complex applications such as nginx or MySQL[12] result into compilation errors due to the small number of μ-libraries. With the development of new micro-libraries, the tool shall support more applications.

5.3 Limitations and improvements

As for the dependency analysis, the current version of the tool has some limitations. The main boundary of tool concerns the library pool. Indeed, for now only a small subset of μ-libraries are available. This limits thus the number applications that can be supported by the UNICORE automatic build tool. The more libraries are provided, the more applications can be ported as unikernels. This limitation is thus related to the library pool and not directly related to the automatic build tool. Another limit covers the controller part of the build system. Indeed, for now, the tool doesn't try to recover compilation error. Therefore, the user must fix all the issues himself. A significant improvement will be to handle this case by having several procedures of compilation. Another improvement will be to implement a new procedure to automatically convert application with complex build systems (*e.g.*, CMake[13], SCons[14], ...) to the Unikraft build system. Finally, an additional tool can be developed to unload unused modules and disables some configuration entries to weed out useless kernel components for the application at hand.

6 Verification Tool

The verification tool will ensure the correctness and security of unikernels. Indeed, the tool will ensure that the newly built application is equivalent to the initial one. In that case, heuristic methods will be used in order to check if the old and new application behave the same. The tool will also ensures security in ring 0 and the implementation of micro-libs.

6.1 Verify application behaviour

At this date, the tool has still not been developed, only the prerequisites have been analysed. Among these, the following assumptions can be made on verifying applications:

(i) The tool will be integrated to the current toolchain and thus will be written in Golang.

(ii) The idea is to run two different images of a same application: one as a unikernel and the other as a traditional application. It will then provide various inputs with different testing methods (*e.g.*, fuzz testing, stress testing, ...) to both and then analyse their behaviour and their output. Then, a matching score will be computed in order to ensure the correctness.

(iii) Even if the built application has the same behaviour than the original one, there can still remain bugs (e.g., buffer overflows). To protect against such attacks, UNICORE will use privileged processor instructions to implement highly efficient sand-boxing mechanisms to circumvent possible attacks.

6.2 Verify micro-libs implementation

In order to verify micro-libs implementation, we can consider formal verification methods.

Since the UNICORE project has been designed for different platforms and architectures, the host system should be able to execute and verify a unikernel on qemu-kvm or/and Xen. In the same way, it would be optimal to be able to test the resulting unikernel on several architectures (*e.g.*, ARM, x86, MIPS). Depending on the resources available, it may be appropriate for the toolchain to be hosted on remote servers in order to test all possible platforms and architectures.

In Figure 2.2, the verification tool uses, as input, unikernel libraries and APIs. Therefore, we are assuming that the application shown in this figure is the generic part, that which is not modified to fit the unikernel environment (*e.g.*, an unmodified Python program instead versus the modified Python interpreter and Python runtime).

There exist two ways that can be considered:

(i) The first approach is to consider the extended API provided by the unikernel (core libraries) and application-specific libraries (runtime and others). This API must be a subset of the API provided by the general purpose OS and libraries. Then, we fuzz the API on both platforms with the goal of validating equivalent functionality. This approach for unikernel core libraries is integrated with the overall security validation track in the project.

(ii) The second approach is grey-box fuzzing of the application running on top of the general-purpose OS and the application+specialized OS. We select a set of applications and fuzz them to increase coverage and validate equivalent functionality. These applications will call the underlying APIs in a realistic manner.

Equivalent functionality means that for a given input the program provides the same effects (including output). Effects are:

(i) Same output (standard output, files, networking, IPC), i.e. anything provided/ex-filtrated by the application outside of its address space.

(ii) Exit/error values: in case of successful run and/or errors and crashes, the application provides the same exit code.

(iii) Termination: application goes into an infinite loop (or equivalent) on both situations.

Furthermore, we can connect this to our work on evaluation of smart contract deterministic execution. There, we plan to use the same application running on different environments (hardware or configured environment) and then make sure it runs deterministically: i.e. it provides the same output for the same input.

7 Performance Optimization Tool

Optimizing unikernel code for a given application traditionally requires significant expert time and manual tweaking. The optimization process is done using a feedback loop that proceeds as follows:

(i) A unikernel is (manually) built, using the current code and set of configuration parameters.

(ii) The built unikernel is deployed in the infrastructure.

(iii) Performance of the application using the deployed unikernel is measured.

(iv) The code and configuration files must are (manually) modified based on the measured results.

(v) The whole process repeats.

While steps (i) to (iii) can be somewhat easily automated, step (iv) usually requires both expert knowledge, and a significant amount of trial and error.
To help with the optimisation of unikernels, the UNICORE project will provide a tool to help automatise this process as much as possible. The tool will comprise two parts: a tool that automates the build-deploy-measure part, and a tool that proposes new configuration parameters based on measured results.

7.1 The configuration space

To avoid having to modify the unikernel code between experiments, all changes should be driven by configuration variables, used at build or deploy time.
There can be several types of configuration variables that will influence the application performance:

Choice of μ-libs. As UNICORE decomposes application and OS functionality into a set of μ-libs, some of which are interchangeable, important parameters will be the chosen μ-libs. *E.g.*, one could replace a preemptive scheduler library by a cooperative one.

μ-lib/kernel parameters. *E.g.*, how big should page size be? Should we use polling or interrupts to communicate with a given device?

Deployment parameters. *E.g.*, should we use I/O passthrough? How many virtual CPUs?

Networking parameters. *E.g.*, Which TCP congestion control algorithm should be used? What size should the receive and send buffers be?

Application-specific parameters. Each application comes with its own set of parameters, that can influence performance (*e.g.* cache size, number of threads, *etc.*).

7.2 Performance measurement

Performance can be assessed though various KPIs (Key Performance Indicator), chosen by the user. Some example KPIs that could be measured are throughput, delay, instantiation time, response time, memory consumption, *etc*.

To be able to automate performance measurement, the user will have to provide a way to exert the application (*e.g.*, a script with traffic generation), and a way to measure performance. While exerting the application is very application-specific, many performance measurement such as instantiation time are shared by many applications. The framework will thus have built-in support for the most common ones.

7.3 The optimisation process

In some cases, one can make use of profiling information to guide parameter selection and code optimizations directly (*e.g.*, to improve branch prediction). In general, however, configuration optimisation if a non-trivial process.

For simple applications, the exploration of the parameter space could be done via an enumeration of all configurations.

However, as the number of possible configurations grows exponentially with the number of available parameters, this does not scale to larger, more complex applications.

To allow for the optimisation of larger configuration spaces, UNICORE will follow two approaches:

(i) As the optimisation tool will be split between a build-deploy-measure tool and a configuration optimiser, an end-user can dispense with the later and generate candidate configurations manually using expert knowledge. This approach only slightly improves the state-of-the-art, and still requires expert time, but is a useful fall-back to have in case the automated optimizer would perform poorly.

(ii) The automated configuration optimiser could use machine-learning and optimisation techniques to sift through the potential configurations, trying to reach high-performing configurations with as few experiments as possible. This approach is more of a research venue, and is not guaranteed to work, but preliminary results seem promising.

The fully automated approach would be based on some sort of reinforcement learning. Input features would be the configuration parameters, and output features the different KPIs. A predictive model could then be built, which could then be used as an oracle to guide an optimization process using a combination of the predicted KPIs as an objective function.

As the configuration search space is large, we will try to use *adaptive learning* to reduce the number of required experiments. Adaptive learning is a machine-learning technique which aims at improving the accuracy of a model, by choosing samples carefully. Given a model built using a given number of samples, the adaptive learning process will output a set of experiments that are likely to best improve the model. The process runs in a loop where experiments are chosen, measured, and used to refine the model. The new model is then used to provide new suggested experiments, *etc*. The process repeats until target model accuracy is reached, or experiment budget is exceeded.

In our case, we are not so much interested in model accuracy, than in finding high-performance configurations. There is no point in refining model accuracy in parts of the configuration space where the application performs poorly. By tweaking the adaptive learning process, we might be able to jointly learn and optimise, further reducing the number of required experiments.

Alternatively, we will also investigate the use of *Bayesian optimisation* to find good configurations with as few experiments as possible.

8 Conclusion

This document describes the first milestone concerning design & implementation of tools for unikernel deployment. It begins by introducing the general structure of the UNICORE toolchain as well as its main objectives. This one allows to understand the purpose of the different tools and how they interact with each other. Subsequently, a discussion on the chosen technologies were established. Each tool is then explained. In this release, two tools are fully-functional and have been tested with different schemes and configurations: the dependency analysis tool and the automatic build tool. They are already integrated to the toolchain and are available in open source in a git repository in order that they can be used by developers. The other tools (*e.g.*, decomposition, verification and performance optimization tool) are still in research phase therefore only analysis and hypothesis have been made. They will be released in next milestones.

Appendices

Complete output of the Dependency Analysis Tool

```
{
    "static_data":{
        "dependencies":{
            "dpkg":[
                "libbz2 -1.0",
                "libc6",
                "liblzma5",
                "libselinux1",
                "libzstd1",
                "zlib1g",
                "tar"
            ],
            "gcc-8-base":[


            ],
            "install-info":[
                "dpkg",
                "libc6"
            ],
            "libacl1":[


            ],
            "libbz2 -1.0":[
                "libc6"
            ],
            "libc6":[
                "libgcc1"
            ],
            "libgcc1":[
                "gcc-8-base",
                "libc6"
            ],
            "liblzma5":[
                "libc6"
            ],
            "libpcre3":[


            ],
            "libreadline7":[
            "readline-common",
            "libc6",
            "libtinfo5"
        ],
        "libselinux1":[
            "libc6",
            "libpcre3"
        ],
        "libsqlite3-0":[
            "libc6"
        ],
        "libtinfo5":[
            "libc6"
        ],
        "libzstd1":[
            "libc6"
        ],
        "readline-common":[
            "dpkg",
            "install-info"
        ],
        "tar":[
            "libacl1",
            "libc6",
            "libselinux1"
        ],
        "zlib1g":[
            "libc6"
        ]
    },
    "shared_libs":{
        "libc.so.6":[

        ],
        "libdl.so.2":[
            "libc.so.6"
        ],
        "libpthread.so.0":[
            "libc.so.6"
        ]
    },
    "system_calls":{
        "access":"",
        "chdir":"",
        "chmod":"",
        "exit":"",
        "fchmod":"",
        "fchown":"",
        "fcntl":"",
        "fsync":"",
        "ftruncate64":"",
        "getcwd":"",
        "geteuid":"",
        "getpid":"",
        "getrusage":"",
        "gettimeofday":"",
        "getuid":"",
        "mkdir":"",
        "mremap":"",
        "munmap":"",
        "read":"",
        "readlink":"",
        "rmdir":"",
        "signal":"",
        "symlink":"",
        "time":"",
        "unlink":"",
        "utimes":"",
        "write":""
    },
    "symbols":{
        "__assert_fail":"",
        "__ctype_b_loc":"",
        "__cxa_finalize":"GLIBC_2.2.5",
        "__errno_location":"",
        "__fxstat64":"",
        "__libc_start_main":"",
        "__lxstat64":"",
        "__stack_chk_fail":"",
        "__xstat64":"",
"access":"GLIBC_2.2.5",
"atoi":"",
"chdir":"GLIBC_2.2.5",
"chmod":"GLIBC_2.2.5",
"close":"",
"closedir":"",
"dlclose":"",
"dlerror":"",
"dlopen":"",
"dlsym":"",
"exit":"GLIBC_2.2.5",
"fchmod":"GLIBC_2.2.5",
"fchown":"GLIBC_2.2.5",
"fclose":"",
"fcntl":"GLIBC_2.2.5",
"fflush":"",
"fgetc":"",
"fgets":"",
"fopen64":"",
"fprintf":"",
"fputc":"",
"fputs":"",
"fread":"",
"free":"",
"fseek":"",
"fsync":"GLIBC_2.2.5",
"ftell":"",
"ftruncate64":"GLIBC_2.2.5",
"fwrite":"",
"getcwd":"GLIBC_2.2.5",
"getenv":"",
"geteuid":"GLIBC_2.2.5",
"getpid":"GLIBC_2.2.5",
"getpwuid":"",
"getrusage":"GLIBC_2.2.5",
"gettimeofday":"GLIBC_2.2.5",
"getuid":"GLIBC_2.2.5",
"isatty":"",
"localtime":"",
"lseek64":"",
"malloc":"",
"memcmp":"",
"memcpy":"",
"memmove":"",
"memset":"",
"mkdir":"GLIBC_2.2.5",
"mmap64":"",
"mremap":"GLIBC_2.2.5",
"munmap":"GLIBC_2.2.5",
"open64":"",
"opendir":"",
"pclose":"",
"popen":"",
"printf":"",
"pthread_create":"",
"pthread_join":"",
"pthread_mutex_destroy":"",
"pthread_mutex_init":"",
"pthread_mutex_lock":"",
"pthread_mutex_trylock":"",
"pthread_mutex_unlock":"",
"pthread_mutexattr_destroy":"",
"pthread_mutexattr_init":"",
"pthread_mutexattr_settype":"",
"putchar":"",
"puts":"",
"raise":"",
"read":"GLIBC_2.2.5",
"readdir64":"",
"readlink":"GLIBC_2.2.5",
"realloc":"",
"rewind":"",
"rmdir":"GLIBC_2.2.5",
"setvbuf":"",
"signal":"GLIBC_2.2.5",
"sleep":"",
"stderr":"",
"stdin":"",
"stdout":"",
"strchr":"",
            "strcmp":"",
            "strcspn":"",
            "strdup":"",
            "strlen":"",
            "strncmp":"",
            "strncpy":"",
            "strrchr":"",
            "strstr":"",
            "strtol":"",
            "symlink":"GLIBC_2.2.5",
            "sysconf":"",
            "system":"",
            "time":"GLIBC_2.2.5",
            "tolower":"",
            "unlink":"GLIBC_2.2.5",
            "utimes":"GLIBC_2.2.5",
            "write":"GLIBC_2.2.5"
        }
    },
    "dynamic_data":{
        "shared_libs":{
            "ld-2.27.so":[

            ],
            "libc-2.27.so":[

            ],
            "libc.so.6":[

            ],
            "libdl-2.27.so":[
                "libc.so.6"
            ],
            "libdl.so.2":[
                "libc.so.6"
            ],
            "libelf-0.170.so":[
                "libz.so.1",
                "libc.so.6"
            ],
        "liblzma.so.5":[
            "libdl.so.2",
            "libpthread.so.0",
            "libc.so.6"
        ],
        "liblzma.so.5.2.2":[
            "libdl.so.2",
            "libpthread.so.0",
            "libc.so.6"
        ],
        "libpcre.so.3":[
            "libpthread.so.0",
            "libc.so.6"
        ],
        "libpcre.so.3.13.3":[
            "libpthread.so.0",
            "libc.so.6"
        ],
        "libpthread-2.27.so":[
            "libc.so.6"
        ],
        "libpthread.so.0":[
            "libc.so.6"
        ],
        "libselinux.so.1":[
            "libpcre.so.3",
            "libdl.so.2",
            "libc.so.6",
            "libpthread.so.0"
        ],
        "libunwind-ptrace.so.0.0.0":[
            "libc.so.6"
        ],
        "libunwind-x86_64.so.8.0.1":[
            "liblzma.so.5",
            "libunwind.so.8",
            "libc.so.6",
            "libdl.so.2",
            "libpthread.so.0"
        ],
            "libunwind.so.8":[
                "libc.so.6",
                "liblzma.so.5",
                "libdl.so.2",
                "libpthread.so.0"
            ],
            "libunwind.so.8.0.1":[
                "libc.so.6",
                "liblzma.so.5",
                "libdl.so.2",
                "libpthread.so.0"
            ],
            "libz.so.1":[
                "libc.so.6"
            ],
            "libz.so.1.2.11":[
                "libc.so.6"
            ]
        },
        "system_calls":{
            "access":"",
            "arch_prctl":"",
            "brk":"",
            "close":"",
            "connect":"",
            "execve":"",
            "fstat":"",
            "getuid":"",
            "ioctl":"",
            "lseek":"",
            "mmap":"",
            "mprotect":"",
            "munmap":"",
            "openat":"",
            "prlimit64":"",
            "read":"",
            "rt_sigaction":"",
            "rt_sigprocmask":"",
            "set_robust_list":"",
            "set_tid_address":"",
            "socket":"",
            "write":""
        },
        "symbols":{
            "access":"",
            "fflush":"",
            "fopen64":"",
            "free":"",
            "getenv":"",
            "getpwuid":"",
            "getuid":"",
            "isatty":"",
            "malloc":"",
            "memcpy":"",
            "memset":"",
            "printf":"",
            "pthread_mutex_destroy":"",
            "pthread_mutex_init":"",
            "pthread_mutex_lock":"",
            "pthread_mutex_unlock":"",
            "pthread_mutexattr_destroy":"",
            "pthread_mutexattr_init":"",
            "pthread_mutexattr_settype":"",
            "puts":"",
            "realloc":"",
            "setvbuf":"",
            "signal":"",
            "strlen":"",
            "strncmp":""
        }
    }
}
```

Listing 1: data gathered from static analysis (complete)

Dependency Analysis Tool

```
-------------------------------------------------------------------
Analyze Program:  sqlite3
Full Path:  /home/unicore/sqlite_src/sqlite3
Background:  true
Options: sqlitetest.db
-------------------------------------------------------------------
ELF Class:  64 bits
Machine:  x86_64
Entry Point:  44112
-------------------------------------------------------------------
[(1) RUN STATIC ANALYSIS]
[(*) Gathering symbols from ELF file]
[(*) Gathering symbols & system calls rom ELF file]
[(*) Gathering shared libraries rom ELF file]
[(*) Gathering dependencies from apt-cache depends]
1) sqlite3-doc
2) sqlite3
3) sqlite3-pcre
Please enter your choice (0 to exit): 2
libc6
libgcc1
gcc-8-base
libreadline7
readline-common
dpkg
libbz2-1.0
liblzma5
libselinux1
libpcre3
libzstd1
zlib1g
tar
libacl1
libattr1
install-info
libtinfo5
libsqlite3-0
-------------------------------------------------------------------
[SUCCESS] [Data saved into sqlite3_output/static/sqlite3.txt]
[(2) RUN DYNAMIC ANALYSIS]
[INFO] [Kill 'sqlite3' if it is already launched]
[(*) Gathering system call from ELF file]
[INFO] [Run 'sqlite3' in background]
[INFO] [Waiting for external tests for 10 sec]
----------
[(*) Gathering shared libs]
[INFO] [PID 'sqlite3' : 3302]
[INFO] [Kill 'sqlite3']
[SUCCESS] ['sqlite3' Killed]
[(*) Gathering symbols from ELF file]
[INFO] [Run 'sqlite3' in background]
[INFO] [Waiting for external tests for 10 sec]
----------
[(*) Gathering shared libs]
[INFO] [PID 'sqlite3' : 3551]
[INFO] [Kill 'sqlite3']
[SUCCESS] ['sqlite3' Killed]
-------------------------------------------------------------------
[SUCCESS] [Data saved: sqlite3_output/dynamic/sqlite3.txt]
[SUCCESS] [JSON Data saved: sqlite3_output/sqlite3.json]
[SUCCESS] [Graph saved: sqlite3_output/static/
sqlite3_shared_libs.png]
[SUCCESS] [Graph saved: sqlite3_output/static/
sqlite3_dependencies.png]
[SUCCESS] [Graph saved: sqlite3_output/dynamic/
sqlite3_shared_libs.png]
-------------------------------------------------------------------
```

Figure 1: The current toolchain dependency analysis tool

Automatic Build Tool

```
[(3) AUTOMATIC BUILD TOOL]
[WARNING] [Unsupported extension for file: a.out]
[WARNING] [Unsupported extension for file: makefile]
[WARNING] [Unsupported extension for file: sqlite3]
[INFO] [Retrieving symbols of internal lib: devfs]
[INFO] [Retrieving symbols of internal lib: fdt]
[INFO] [Retrieving symbols of internal lib: nolibc]
[INFO] [Retrieving symbols of internal lib: posix-libdl]
[INFO] [Retrieving symbols of internal lib: ramfs]
[INFO] [Retrieving symbols of internal lib: ukalloc]
[INFO] [Retrieving symbols of internal lib: ukallocbbuddy]
[INFO] [Retrieving symbols of internal lib: ukargparse]
[INFO] [Retrieving symbols of internal lib: ukboot]
[INFO] [Retrieving symbols of internal lib: ukbus]
[INFO] [Retrieving symbols of internal lib: ukdebug]
[INFO] [Retrieving symbols of internal lib: uklock]
[INFO] [Retrieving symbols of internal lib: ukmpi]
[INFO] [Retrieving symbols of internal lib: uknetdev]
[INFO] [Retrieving symbols of internal lib: uksched]
[INFO] [Retrieving symbols of internal lib: ukschedcoop]
[INFO] [Retrieving symbols of internal lib: uksglist]
[INFO] [Retrieving symbols of internal lib: ukswrand]
[INFO] [Retrieving symbols of internal lib: uksysinfo]
[INFO] [Retrieving symbols of internal lib: uktimeconv]
[INFO] [Retrieving symbols of internal lib: ukunistd]
[INFO] [Retrieving symbols of internal lib: vfscore]
[INFO] [Retrieving symbols of external lib: eigen]
[INFO] [Retrieving symbols of external lib: libfp16]
[INFO] [Retrieving symbols of external lib: python]
[INFO] [Retrieving symbols of external lib: pthreadpool]
[INFO] [Retrieving symbols of external lib: libcxx]
[INFO] [Retrieving symbols of external lib: libuuid]
[INFO] [Retrieving symbols of external lib: newlib]
[INFO] [Retrieving symbols of external lib: libcxxabi]
[INFO] [Retrieving symbols of external lib: libunwind]
[INFO] [Retrieving symbols of external lib: googletest]
[INFO] [Retrieving symbols of external lib: lwip]
[INFO] [Retrieving symbols of external lib: intel-intrinsics]
[INFO] [Retrieving symbols of external lib: compiler-rt]
[INFO] [Retrieving symbols of external lib: libfxdiv]
[INFO] [Retrieving symbols of external lib: pthread-embedded]
[INFO] [Retrieving symbols of external lib: psimd]
[SUCCESS] [Match lib: newlib]
[SUCCESS] [Match lib: vfscore]
[SUCCESS] [Match lib: pthread-embedded]
[SUCCESS] [Match lib: uksysinfo]
[SUCCESS] [Match lib: ukunistd]
[SUCCESS] [Match lib: posix-libdl]
[INFO] [Library newlib already exists in folder/home/unicore/
unikraft/libs/]
[INFO] [Library pthread-embedded already exists in folder/home/
unicore/unikraft/libs/]
[WARNING] [build folder already exists. Delete it.]
[INFO] [Waiting command: make allnoconfig]

…

[INFO] [Warnings are written in file: /home/unicore/unikraft/apps/
sqlite3/sqlite3_warnings.txt]
[INFO] [Output is written in file: /home/unicore/unikraft/apps/
sqlite3/sqlite3_output.txt]
[SUCCESS] [Unikernel created in Folder: 'build/']
```

Figure 2: The current toolchain automatic build tool

References

[1] "The go programming language." [Online]. Available: https://golang.org

[2] "Java." [Online]. Available: https://www.java.com/com/download/

[3] "Python." [Online]. Available: https://www.python.org

[4] J. Lakos, *Large-scale C++ Software Design*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996.

[5] C. Patachia, E. Slusanschi, F. Huici, G. Bosson, G. Carrozzo, J. Martín, J. Bromell, J. Guijarro, M. Rapoport, R. Stoenescu, and R. Deaconescu, "D2.1 requirements," Apr. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2783992

[6] "Clang static analyzer." [Online]. Available: https://clang-analyzer.llvm.org/

[7] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A Binary-Compatible Unikernel," p. 15, 2019.

[8] "Sqlite." [Online]. Available: https://www.sqlite.org

[9] "Debian popularity contest." [Online]. Available: https://popcon.debian.org

[10] "Gnu operating system - automake." [Online]. Available: https://www.gnu.org/software/automake/

[11] "mini_httpd - small http server." [Online]. Available: https://acme.com/software/mini_httpd

[12] "Mysql." [Online]. Available: https://www.mysql.com

[13] "Cmake." [Online]. Available: https://cmake.org

[14] "Scons: A software construction tool." [Online]. Available: https://scons.org