Runikraft 小组　调研报告

吴骏东　　张子辰　　蓝俊玮　　郭耸霄　　陈建绿

2022 年 3 月 29 日

目录

1　项目简介

Runikraft 是用 Rust 语言编写的能在 RISC-V + KVM 上运行 unikernel。它参考 Unikraft 的架构，在继承 Unikraft 的高效、可定制、兼容性好、安全的同时，进一步简化了构建系统镜像的流程，加入了 RISC-V 支持，并且用 Rust 语言提供了更强的内核安全保证。

2 项目背景

2.1 Rust 语言

by 郭耸霄

Rust 是由 Mozilla 研究室主导开发的一门现代系统编程语言，自 2015 年 5 月发布 1.0 之后，一直以每 6 周一个小版本的开发进度稳定向前推进。语言设计上跟 C++ 一样强调零开销抽象和 RAII。拥有极小的运行时和高效的 C 绑定，使其运行效率与 C/C++ 一个级别，非常适合对性能要求较高的系统编程领域。利用强大的类型系统和独特的生命周期管理实现了编译期内存管理，保证内存安全和线程安全的同时使编译后的程序运行速度极快，Rust 还提供函数式编程语言的模式匹配和类型推导，让程序写起来更简洁优雅。[5] 总地来说，Rust 是一门赋予每个人构建可靠且高效软件能力的语言。[4] 它具有以下三个方面的优越性。

2.1.1 高性能（*Performance*）

Rust 速度惊人且内存利用率极高。由于没有运行时和垃圾回收，它能够胜任对性能要求特别高的服务，可以在嵌入式设备上运行，还能轻松和其他语言集成。[4]

- 可执行文件：
  Rust 是编译语言，这意味着程序直接转换为可执行的机器代码，因此可以将程序作为单个二进制文件进行部署；与 Python 和 Ruby 等解释型语言不同，无需随程序一起分发解释器，大量库和依赖项，这是一大优势。与解释型语言相比，Rust 程序非常快。[7]
- 对动态类型语言与静态类型的平衡
  动态类型语言在调试、运行时具有不确定性，而静态类型的语言减少了程序理解的开销和动态运行的不确定性，但并不是所有静态类型系统都是高效的。Rust 的静态类型尽最大努力避免程序员的麻烦，同时鼓励长期的可维护性。一些静态类型的语言给程序员带来了沉重的负担，要求他们多次重复变量的类型，这阻碍了可读性和重构。其他静态类型的语言允许在全局进行数据类型推断。虽然在最初的开发过程中很方便，但是这会降低编译器在类型不再匹配时提供有用的错误信息的能力。Rust 可以从这两种样式中学习，并要求顶层项具有显式类型，同时允许在函数体内部进行类型推断。[6]
- 解决垃圾回收问题
  Rust 可以选择将数据存储在堆栈上还是堆上，并在编译时确定何时不再需要内存并可以对其进行清理。这样可以有效利用内存，并实现更高性能的内存访问。Rust 项目非常适合被其他编程语言通过外部功能接口用作库。这使现有项目可以用快速的 Rust 代码替换对性能至关重要的代码，而不会产生其他系统编程语言固有的内存安全风险。[6]

2.1.2 可靠性（*Reliability*）

Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，让您在编译期就能够消除各种各样的错误。[4]

- 处理系统级编程
  与其他系统级编程语言（例如 C 或 C++）相比，Rust 可以提供的最大好处是借阅检查器。这是编译器的一部分，负责确保引用不会超出引用的数据寿命，并有助于消除由于内存不安全而导致的所有类型的错误。与许多现有的系统编程语言不同，Rust 不需要你将所有时间都花在细节上。当安全的 Rust 无法表达某些概念时，http://cliffle.com/p/dangerust/可以使用不安全的 Rust。这样可以释放一些额外的功能，但作为交换，程序员现在有责任确保代码真正安全。然后，可以将这种不安全的代码包装在更高级别的抽象中，以确保抽象的所有使用都是安全的。[6]
- Concurrent programming made easier
  Rust makes it easier to write concurrent programs by preventing data races at compile time. Rust language can check if we are performing any incorrect operations and inform us at

compile time.[9]

2.1.3 生产力（Productivity）

Rust 拥有出色的文档、友好的编译器和清晰的错误提示信息，还集成了一流的工具——包管理器和构建工具，智能地自动补全和类型检验的多编辑器支持，以及自动格式化代码等等。[4]

- Cargo 包管理器
  Rust 由于有 Cargo 这样一个非常出色的包管理工具，周边的第三方库发展非常迅速，各个领域都有比较成熟的库，比如 HTTP 库有 Hyper，异步 IO 库有 Tokio, mio 等，基本上构建后端应用必须的库 Rust 都已经比较齐备。总体来说，现阶段 Rust 定位的方向还是高性能服务器端程序开发，另外类型系统和语法层面上的创新也使得其可以作为开发 DSL 的利器。[5]

2.1.4 Rust 语言的兼容性

The Rust language is fast evolving, and because of this certain compatibility issues can arise, despite efforts to ensure forwards-compatibility wherever possible.

Rust, like many programming languages, has the concept of "keywords". These identifiers mean something to the language, and so you cannot use them in places like variable names, function names, and other places. Raw identifiers let you use keywords where they would not normally be allowed. This is particularly useful when Rust introduces new keywords, and a library using an older edition of Rust has a variable or function with the same name as a keyword introduced in a newer edition.[11]

2.1.5 相比其他语言 Rust 的优势

- Go：Rust 语言表达能力更强，性能更高，同时线程安全方面 Rust 也更强，不容易写出错误的代码，包管理 Rust 也更好，Go 虽然在 1.10 版本后提供了包管理，但是目前还比不上 Rust 的。
- C++：与 C++ 相比，Rust 的性能相差无几，但是在安全性方面会更优，特别是使用第三方库时，Rust 的严格要求会让第三方库的质量明显高很多。语言本身的学习，Rust 的前中期学习曲线会更陡峭，但是对于未来使用场景和生态的学习，C++ 会更难、更复杂。
- Java：除了极少部分纯粹的数字计算性能，Rust 的性能是全面领先于 Java 的，同时 Rust 占用内存小的多，因此实现同等规模的服务，Rust 所需的硬件成本会显著降低。
- Python：性能自然是 Rust 完胜，同时 Rust 对运行环境要求较低，这两点差不多就足够抉择了，因为 python 和 rust 的彼此适用面其实不太冲突。[8]

2.2 RISC-V 指令集架构

by 张子辰

这一部分解释我们为什么要选择 RISC-V 架构，并且说明 RISC-V 相比所谓的“成熟”“生态好”的 ARMv8 没有任何劣势。

2.2.1 RISC-V 的特点

RISC-V(“RISC five”) 的目标是成为一个通用的指令集架构 (ISA):[12]

- 它要能适应包括从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器。

- 它应该能兼容各种流行的软件栈和编程语言。
- 它应该适应所有实现技术，包括现场可编程门阵列 (FPGA) 、专用集成电路 (ASIC) 、全定制芯片，甚至未来的设备技术。
- 它应该对所有微体系结构样式都有效: 例如微编码或硬连线控制; 顺序或乱序执行流水线; 单发射或超标量等等。
- 它应该支持广泛的专业化，成为定制加速器的基础，因为随着摩尔定律的消退，加速器的重要性日益提高。
- 它应该是稳定的，基础的指令集架构不应该改变。更重要的是，它不能像以前的专有指令集架构一样被弃用，例如 AMD Am29000、Digital Alpha、Digital VAX、Hewlett Packard PA-RISC、Intel i860、Intel i960、Motorola 88000、以及 Zilog Z8000。

RISC-V 的不同寻常不仅在于它是一个最近诞生的指令集架构 (它诞生于最近十年，而大多数其他指令集都诞生于 20 世纪 70 到 80 年代)，而且在于它是一个开源的指令集架构。与几乎所有的旧架构不同，它的未来不受任何单一公司的浮沉或一时兴起的决定的影响 (这一点让许多过去的指令集架构都遭了殃)。它属于一个开放的，非营利性质的基金会。RISC-V 基金会的目标是保持 RISC-V 的稳定性，仅仅出于技术原因缓慢而谨慎地发展它，并力图让它之于硬件如同 Linux 之于操作系统一样受欢迎。

RISC-V 的另一个不同寻常之处在于和几乎所有以往的 ISA 不同，它是*模块化*的。它的核心是一个名为 RV32I 的基础 ISA，运行一个完整的软件栈。RV32I 是固定的，永远不会改变。这为编译器编写者，操作系统开发人员和汇编语言程序员提供了稳定的目标。模块化来源于可选的标准扩展，根据应用程序的需要，硬件可以包含或不包含这些扩展。这种模块化特性使得 RISC-V 具有了袖珍化、低能耗的特点，而这对于嵌入式应用可能至关重要。RISC-V 编译器得知当前硬件包含哪些扩展后，便可以生成当前硬件条件下的最佳代码。惯例是把代表扩展的字母附加到指令集名称之后作为指示。例如，RV32IMFD 将乘法 (RV32M)，单精度浮点 (RV32F) 和双精度浮点 (RV32D) 的扩展添加到了基础指令集 (RV32I) 中。

2.2.2 RISC-V 的支持

自诞生以来，RISC-V 就得到了 Google、华为、IBM、Microsoft、삼성等著名企业[12] 和众多自由软件开发者的支持，所以 RISC-V 发展迅猛。尽管历史不长，RISC-V 目前已经得到相当完善的软件和硬件方面的支持。在软件方面，GCC[14] 和 LLVM[15] 都支持 RISC-V 架构，这一味着以 LLVM 为后端的 rustc 也支持 RISC-V 架构。Linux 内核自 4.15 版开设支持 RISC-V 架构，到 2020 年底，Linux 5.10 已经支持 RISC-V 架构的 UEFI 引导。[16] 在 Debian 发行版中，已经有超过 95% 的软件包支持 RISC-V 架构。[17]

在硬件方面，目前已有相当多的 RISC-V 架构的处理器，比如：[13]

- The BeagleV, a low-cost single board computer than can run Linux
- The HiFive1 Rev B microcontroller
- The Seeed Studio Perf-V based on a Xilinx Artix-7 RISC-V FPGA
- The SparkFun RED-V RedBoard, a microcontroller in an Arduino form factor
- The LoFive RISC-V SoC evaluation kit

2.2.3 RISC-V 的优势

RISC-V 在设计时考虑了成本、简洁性、性能、架构和具体实现的分离、提升空间、程序大小和易于编程/编译/链接七个因素，并且充分从过去的错误中吸取了教训：[12]

	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
成本	必须支持整数乘除法	必须支持整数乘除法	8 位以及 16 位操作、必须支持整数乘除法	无 8 位、16 位操作、可选的整数乘除法支持 (RV32M)
简洁性	无零寄存器、条件指令执行、复杂的寻址模式、栈操作指令 (push/pop)、算术/逻辑指令中存在移位	立即数支持零扩展及符号扩展、一些算术指令会造成溢出异常	无零寄存器、复杂的过程调用指令 (enter/leave)、栈指令 (push/pop)、复杂寻址模式、循环指令	寄存器 x0 专门用于存放常数 0、立即数只进行符号扩展、一种数据寻址模式、没有条件执行、没有复杂的函数调用指令以及栈指令、算术指令不抛异常、使用单独的移位指令来处理移位操作
性能	分支指令使用条件码、在不同格式的指令中，源和目的寄存器的位置不同、加载多个计算得到的立即数、PC 是一个通用寄存器	在不同格式的指令中，源和目的寄存器的位置不同	分支指令使用条件码、每个指令中最多只能使用两个寄存器	使用同一条指令实现比较及跳转 (不使用条件码)、每条指令三个寄存器、不能一次 load 多个数据、不同指令格式中，源及目的寄存器字段位置固定、立即数是常数 (不是由计算得出的)、PC 不是通用寄存器
架构和具体实现的分离	将 PC 像普通寄存器一样读写，这样暴露了流水线长度	分支指令延迟槽、Load 指令延迟槽、乘除法使用单独的 HI，LO 寄存器	寄存器不是通用的 (AX，CX，DX，DI，SI 有特殊用途)	不是通用寄存器分支指令没有延迟槽、Load 指令无延迟槽、通用寄存器
提升空间	有限的指令码空间	有限的指令码空间		大量可用的指令码空间位指令位
程序大小	仅有 32bit 指令 (Thumb-2 是一个独立的 ISA)	仅 32bit 指令 (microMIPS 是一个独立的 ISA)	指令长度可用是不同字节，但这是一个很不好的选择	32 位指令 +16 位 RV32C 扩展
易于编程/编译/链接	仅 15 个寄存器、内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	仅 8 个通用寄存器、内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	31 个寄存器、数据可用不对齐、PC 相对的数据寻址模式、对称的数据寻址模式、定义在架构中的性能计数器

而且，从学习成本上看，RISC-V 相比 ARMv8 和 AMD64 也有明显的优势：RISC-V 的

“规范”只有 4 卷，238[18]+155[19]+141[20]+80[21]=614 页，AMD64 的“程序员手册”有 5 卷，3273 页[22]①，ARMv8 的“参考手册”更是长达 11 卷，11530 页[24]。按两分钟读一页，一天工作八小时，一周工作五天计算，读完 RISC-V 的手册需要不到三天，读完 AMD64 的手册需要将近三周，而读完 AMDv8 的手册需要两个多月。

2.3 Unikernel

Unikernel 是专一用途的、单地址空间的轻量操作系统。Unikernels 在虚拟机上运行时，能够提供比传统的容器更短的启动时间、更高的运行效率和更强的隔离性，因此 unikernels 通常被用在云计算领域。[1] 然而，为了追求轻量性，unikernels 裁剪了传统的操作系统的众多组件，因此 unikernels 无法提供许多常用的库的应用程序接口，所以为了将现有的程序移植到某个 unikernel 平台，开发者不得不根据该 unikernel 的 API 重构程序。[2] 此外，为了轻量、快速，unikernels 删去的许多基本的并且不会影响性能的安全措施，这导致 unikernels 相比容器更容易受到用户程序的安全漏洞的影响。[3]

下面简要介绍我们小组详细调研的 ClickOS、MirageOS、IncludeOS、Rusty-Hermit、Rumprun 和 Unikraft 等六个目前仍然在维护的 unikernel 项目。我们假定读者对 unikernel 已经有了基本的认知，因此我们略去了 unikernels 都具有的特性，比如启动时间短、镜像小、延迟低、用在云计算领域。

2.3.1 *ClickOS* *by 蓝俊玮*

ClickOS 是一个基于 Xen 的高性能的虚拟化软件中间盒平台。为了达到高性能，ClickOS 对 Xen 的 I/O 子系统实现了广泛的翻修，包括对后端交换机、虚拟网络设备和后端前端驱动程序。这些更改使 ClickOS 能够显著加快中间盒运行时的网络连接。[25]

ClickOS 虚拟机只有 5MiB ，启动仅需要大约 30ms，而且延迟只有 45μs。ClickOS 实现了广泛的中间盒，包括防火墙、运营商级 NAT 和负载均衡器，并证明 ClickOS 可以每秒处理数百万个数据包，达到生产级性能。[25][26]

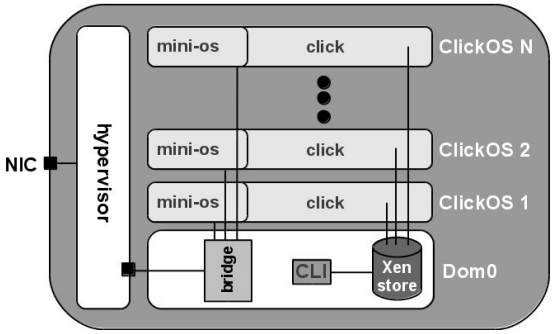

图 1: ClickOS 的架构[26]

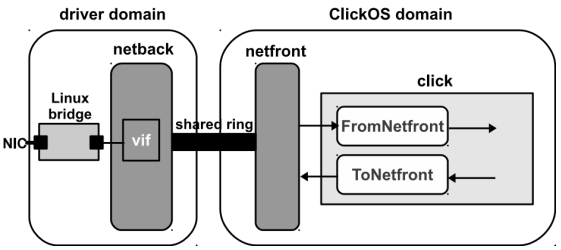

图 2: Basic ClickOS networking in Xen[25]

2.3.2 *MirageOS* *by 蓝俊玮*

MirageOS 是一个用 OCaml 语言编写的，用于在各种云计算和移动平台构建安全、高性能网络应用程序的库操作系统。它可以将大型服务器划分为很多更小的虚拟机，使得服务器具有更强的拓展性和安全性。其代码可以在 Linux 、Mac OS 等系统中开发，然后编译成一个完全独立的、专门的内核，可以在 Xen、KVM hypervisors 或轻量级 hypervisors 下运行。MirageOS 已经发展成为一个由近 100 个开放源码库组成的成熟库，实现了一系列广泛的功能，并且正开始与 Citrix XenServer 等商业产品集成。MirageOS 将 Xen hypervisors 当成一个稳定的硬件

① 这是 AMD 提供的手册，Intel 提供的手册更长，有 4 卷，4830 页[23]

平台，让我们可以专注于实施高性能协议，没必要为支持传统操作系统里面的成千上万个设备驱动程序而操心。[27]

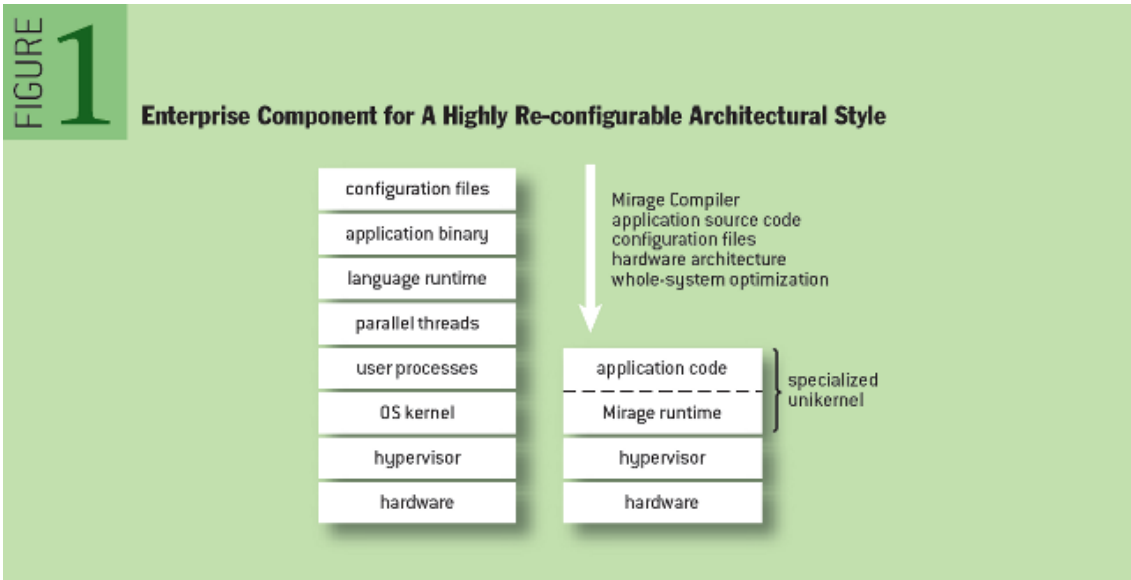

图 3: MirageOS 的架构

MirageOS 的亮点是输入应用程序的所有源代码依赖项都被显式跟踪，包括实现内核功能所需的所有库。MirageOS 包括一个构建系统，该系统内部使用一个 SAT 解算器 (使用 OPAM 包管理器，使用 Mancoosi 项目的解算器) 从一个已发布的在线包集中搜索兼容的模块实现。由于 OCaml 的静态类型检查，在编译时将捕获接口中的任何不匹配。

在 MirageOS 中，OCaml 编译器接收整个内核代码的源代码，并将其链接到一个独立的本机代码对象文件。它链接到提供引导支持和垃圾回收器的最小运行时。没有抢占式线程，内核是通过一个 I/O 循环轮询 Xen 设备的事件驱动的。[27]

2.3.3 IncludeOS[29] *by 蓝俊玮*

IncludeOS 是一个为开发基于 unikernel 的应用程序而创建 C++ API 的项目。[28]

当 IncludeOS 映像引导时，它通过设置内存、运行全局构造函数、注册驱动程序和中断处理程序来初始化操作系统。IncludeOS 不支持虚拟内存，应用程序和 unikernel 库使用单个地址空间。因此，没有系统调用或用户空间的概念; 所有操作系统服务都通过对库的简单函数调用来调用，并且都以特权模式运行。

IncludeOS 有如下优点：

- IncludeOS 中目前没有进程抢占，所以操作系统的行为非常静态。只要机器本身是可预测的，延迟也将是完全可预测的。因此，在裸机硬件上，IncludeOS 可被视为低延迟，可预测的操作系统。
- 对网络的支持很好，与 Linux 相比表现出色。
- IncludeOS 系统作为一个整体进行编译和优化。在编译器和连接器阶段，优化器可以更多地了解整个系统正在做什么，并且有可能进一步优化。
- IncludeOS 中的所有 IRQ 处理程序将简单地 (原子地) 更新计数器，并在有时间时将进一步的处理推迟到主事件循环。这消除了对上下文切换的需要，同时也消除了与并发相关的问题，如竞争条件。通过使所有 I/O 都是异步的，CPU 保持忙碌，这样就不会发生阻塞。

IncludeOS 有如下缺点：

- IncludeOS 不实现所有 POSIX。开发人员认为，只有在需要时才会实现 POSIX 的某些部分。开发人员不太可能将完全遵守 POSIX 作为一个目标。

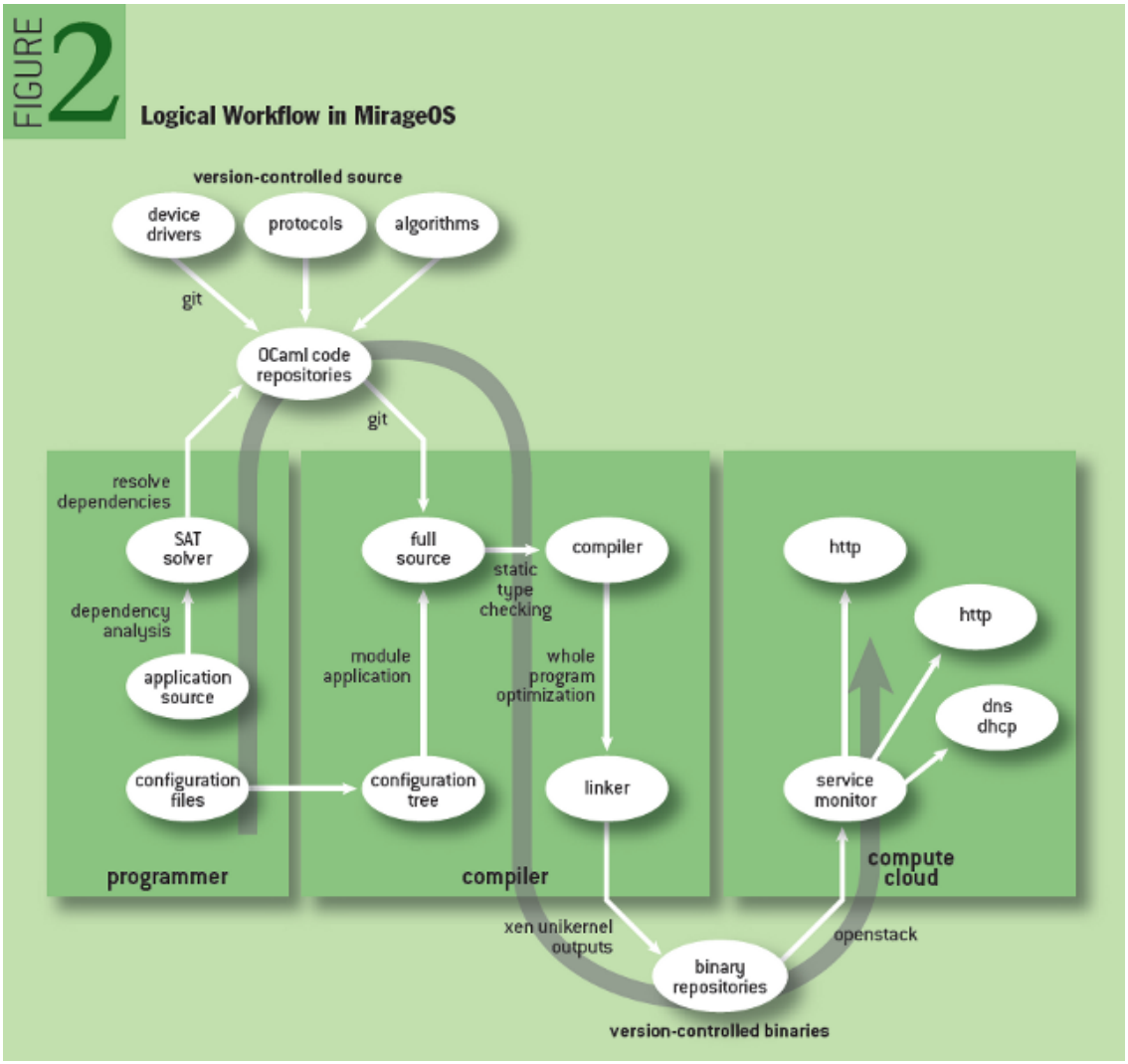

图 4: MirageOS 的逻辑工作流

- 与 MirageOS 一样，IncludeOS 中没有实现阻塞调用，因为当前的事件循环模型是使用它的最佳方式。
- IncludeOS 目前还缺少可写的文件系统。

2.3.4 *RustyHermit*[30][31][32][33] *by 陈建绿*

RustyHermit（Github）是一个基于 Rust 的、轻量级的 Unikernel。它用 Rust 语言完全改写了 RWTH Aachen University 开发的研究项目 HermitCore。

> HermitCore 最初是用 C 语言编写的，是一种针对高性能和云计算的可伸缩和可预测的运行时的 Unikernel。

该项目完全使用 Rust 语言开发，**Rust 的所有权模型保证了它的内存/线程安全**，并且让开发者能够在编译时就消除许多种 bug。因此，与通用编程语言相比，使用 Rust 进行内核开发会留下更少的漏洞，得到更加安全的内核。

开发者**扩展了 Rust 工具链**以至于 RustyHermit 的 build 过程与 Rust 通常的工作流程相似。使用 Rust runtime 而且不直接使用 OS 服务的 Rust 应用程序能够直接在 RustyHermit 上运行而不需要修改。因此，**原则上，每一个现有的 Rust 应用程序都可以建立在 RustyHermit 之上**。

RustyHermit 中**优化实现了网络栈**。它使用 smoltcp (Rust 语言编写) 作为它的网络栈，使用 Virtio (KVM 的准虚拟化驱动程序，广泛应用于虚拟化 Linux 环境中) 作为客户机和主机操作系统之间的接口。将 RustyHermit 和 Linux 分别作为客户端运行在基于 Linux 的主机系统上的虚拟机中，以信息的比特数作为自变量，吞吐量/Mbps 作为因变量，进行测试并绘图，结果如下：[36]

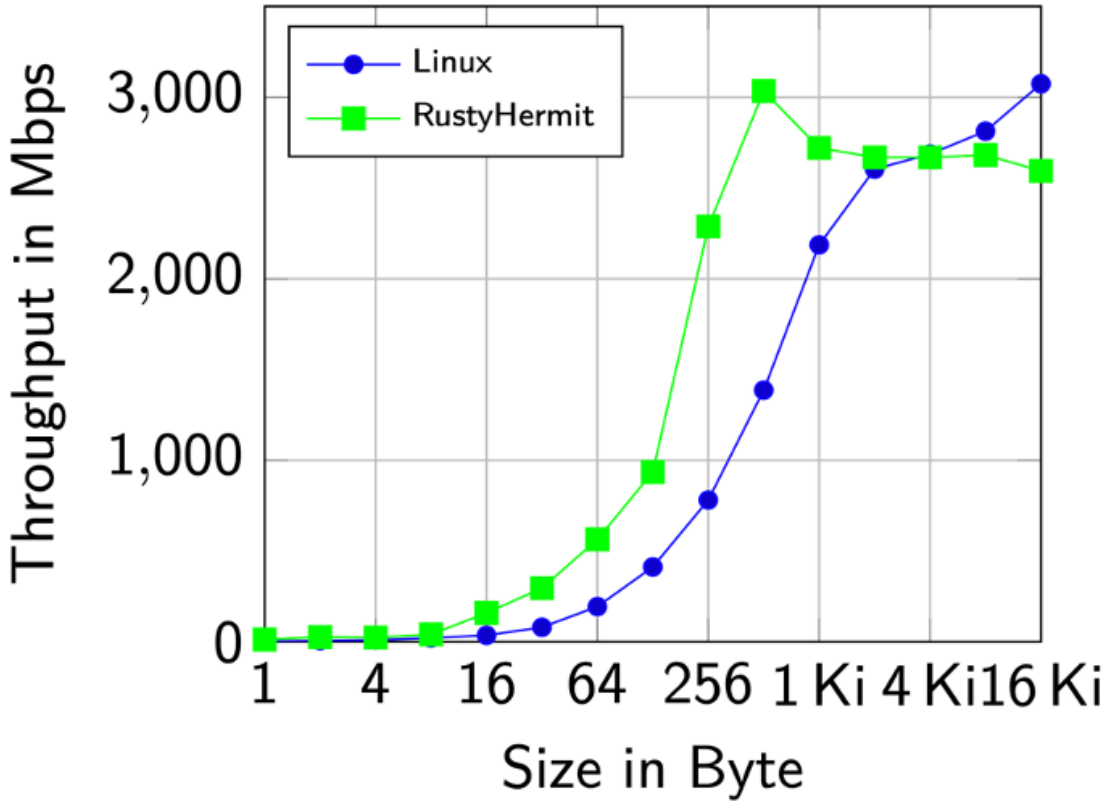

图 5

由结果图可以看出，**RustyHermit 在信息比特数较小时吞吐量明显比 Linux 更快**。

RustyHermit 也是一个用来评估操作系统新的设计的研究项目。

有 Sung, Olivier, Lankes and Ravindran[34] 提出了**一个修改版本的 RustyHermit**，该版本提供了一个**使用 Intel MPK**(Memory Protection Keys) **进行内部隔离**的 Unikernel。这篇论文的摘要翻译如下：

这篇论文中的内容可以作为我们实现 runikraft 的参考。[35]

2.3.5 Rumprun

by 陈建绿

Rumprun unikernel 是在 rump kernels 的基础上开发的。Rumprun 不仅可以在像 KVM 和 Xen 这样的管理程序上工作，还可以在裸金属上工作。无论有没有 POSIX-y 接口，Rumprun 都可以正常使用。如果有 POSIX-y 接口，Rumprun 则允许现有的、未经修改的 POSIX 应用程序开箱即用；如果没有 POSIX-y 接口，Rumprun 则允许构建高度自定义的解决方案，并且占用的空间最小。

Rumprun unikernel 支持用 C、C++ 、Erlang、Go、Java、Javascript (node.js)、Python、Ruby 和 Rust 等语言编写的应用程序。

在 rumprun-packages repository 中可以找到用于 Rumprun 的现成软件包，比如 `LevelDB`, `Memcached`, `nanomsg`, `Nginx` 和 `Redis`。

1. Rump kernels 的相关介绍 [37][38]

Rump kernels 的组件来自未经修改的 NetBSD，由此开发者提供了一个 POSIX-y API。Rump Kernel 项目以一种可用于构建轻量级、特殊用途虚拟机的形式提供了 NetBSD 的模块化驱动程序。因为开发者没有做会将错误引入到应用程序运行时 (application runtime)、libc 或驱动程序中的移植工作，所以程序可以很稳定地工作。下面这张图片阐述了 Anykernel、Rump kernel 和 Rumprun Unikernel 的关系：

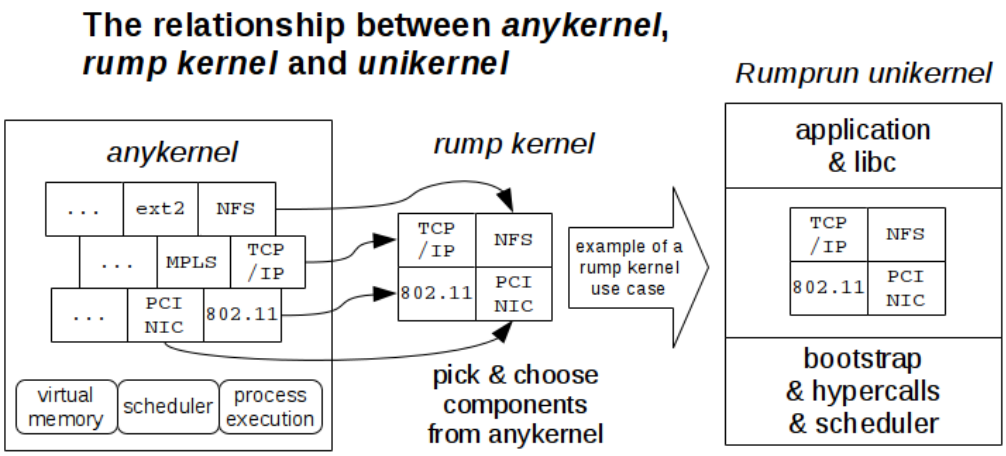

图 6

> “Anykernel” 概念指的是一种与架构无关的驱动程序方法，在这种方法中，驱动程序既可以编译到宏内核中，也可以作为用户空间进程运行，具有微内核风格，并且不需要修改代码。
> ——维基百科

2. Rumprun 的相关介绍 [39][40]

目前已经有很多 Unikernel 项目，它们的实现方式大致可以分为两种：

- 全新的方式 (Clean-slate)：在构建单一用途的操作系统的假设下，自由地使用现代工具来进行构建，比如模块化 (modularity)、声明性代码 (declarative code)、避开样板文件 (avoiding boilerplate) 等。并且从头开始思考操作系统和应用程序层的实现，使用高级语言进行系统库的编写，从而使得实现更加可掌控，得到的系统库质量更高。
- 传统的方式 (Legacy)：在不进行修改或只进行一些小的修改的前提下，运行现有的软件。这通常通过将现有的操作系统代码库重构到库操作系统中来实现。

用 OCmal 语言编写的 MirageOS Unikernel 就是使用 Clean-slate 方式实现的，而用 C 语言编写的 Rumprun Unikernel 则是使用 Legacy 方式实现的。

Rumprun 可用于将几乎任何与 POSIX 兼容的程序转换为一个可工作的 Unikernel。使用 Rumprun，理论上可以将 Linux 或者类 Unix 系统上的大部分程序编译成 Unikernel。Rumprun 以开发 NetBSD 内核中的驱动程序并在用户空间中进行测试的需求为出发点，主要的工作是重构这个代码库，使其看起来像一个库操作系统。

下面是 Rumprun 的架构图：

Rumprun: unikernel based on rump kernels

- from rump/NetBSD
  - rump kernel & drivers
  - (mostly) unmodified libc
- our own
  - platform-specific bootstrapping
  - "bare-metal" hypercall implementation
    - thread scheduler
    - memory allocator
    - console output

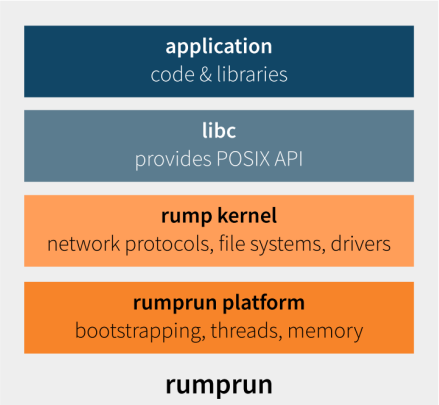

图 7

Rumprun 也有一些限制：

- single address-space
  - no processes
  - no virtual memory
  - no signals
- toolchain
  - still experimental
- threading
  - cooperative
  - single-core
    - need to spawn multiple unikernels to use multiple cores

下图是 Rumprun 的一个工作流程示例图：

Rumprun **workflow**

step 1: cross-compile
- compile against NetBSD's libc
- support for autotools & cmake

step 2: bake
- choose hypervisor, drivers & subsystems

step 3: launch
- mount points for block devices
- configure network
- environment variables, main args

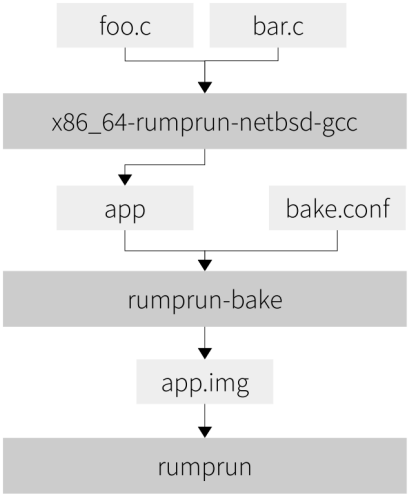

图 8

我在调研的过程中发现，rump kernel 的好多官方文档都会重定向到 https://rumpkernel.org这个网址，而这个网址目前只有一些 IT News，并非和 rump kernel 相关的内容，所以*猜测该项目目前已经无人维护*。

2.3.6 Nanos

by 陈建绿

我在调研过程中发现一个比较新的正在开发中的 Unikernel：Nanos(Github)。下面是它的一些介绍：

- Nanos 是一个新的内核，旨在虚拟化环境中运行一个且仅有一个应用程序。与 Windows 或 Linux 等通用操作系统相比，它有几个限制——即它是一个单进程系统，不支持运行多个程序，也不具备通过 ssh 进行用户或远程管理的概念。
- Nanos 的目标是成为一个比 Linux 安全得多的系统。它做到这一点的几个依赖：**没有用户的概念，每个虚拟机只运行一个进程，限制每个虚拟机中包含的代码数量**。
- Nanos 并不打算在裸金属上运行，所以开发者努力使**其内核尽可能简单**。

这也许会对我们的项目有所帮助。

2.3.7 Unikraft

by 张子辰

Unikraft 是一个比较新的 unikernel。可以说，它在设计时就充分考虑了现有的 unikernels 的优缺点。它在保持 unikernel 的极简化、高效的同时，兼容了完整的 POSIX 兼容层，使开发者可以轻松地将现有的为 Linux 编写的代码移植到 unikernel 上。Unikraft 由若干低耦合的模块组成，内存分配器、调度器、网络栈、引导代码都是独立的微型库。Unikraft 的 API 即为微型库本身，这意味着可以在生成时轻松地添加或移除 APIs。[2]

Unikraft 的设计者指出，现有的 unikernels 普遍存在以下问题：

- 编译它们并让它们达到高效率需要大量专业的工作，而且这些工作通常需要对每个目标应用程序重做。
- 它们通常不是 POSIX 兼容的，需要移植程序和语言环境。

Unikraft 遵从如下设计原则：

- 内核应该是完全模块化的，以便允许 unikernel 被彻底而轻松的定制。在 Unikraft 中，内存分配器、调度器、网络栈、引导程序等系统原件都是独立的微型库。
- 内核应该提供注重效率、良定义的 API，而且允许用户轻松地为了满足自己的程序的性能要求选择、组装它们。在 Unikraft 中，这些 APIs 就是微型库本身，这意味着它们可以轻松地在构建时增减，而且提供更多的微型库即可拓展它们功能。

目前，Unikraft 已经 SQLite, nginx, Redis 等程序，C/C++, Go, Python, Ruby, Web Assembly and Lua 等编程语言或运行环境。

在架构方面，Unikraft 融合了宏内核的单地址空间带来的高效性和微内核的模块化带来的可拓展性。OS 的功能被分割成若干细粒子度的组件，而各个组件之间通过良定义的 APIs 通信。Unikraft 用精心设计的 APIs 和静态链接获得高效率，而不是为了效率破坏 API 的边界。Unikraft 大致分为两部分：

- **微型库**：微型库是实现一部分的 Unikraft 的 APIs 的软件组件，Unikraft 的作者有意将它们分割到了不同的库中，并尽可能降低它们之间的依赖。实现相同 APIs 的微型库可以相互替换。比如，Unikraft 内核就提供了多种实现 `ukalloc` 接口的内存分配器。
- **构建系统**：它为用户提供基于 Kconfig 的配置菜单①，用户可用它选择要用哪些微型库，要为哪个平台和哪个 CPU 架构构建。

图 9 展示了 Unikraft 的架构。使用不同层次的 APIs 和替换 API 实现的能力给开发者提供了多种优化可能。首先，未经修改的程序（如用 C 语言写的 Hello World 和 nginx）可以使用 `musl`（图 9 的①）或 `nolibc` 提供的 POSIX 兼容层，并自动获得低启动时间、低内存消耗和更高的吞吐量，因为在 Unikraft 中，系统调用是高效的函数调用。

① 在实验 1 中，我们在定制 Linux 内核时看到的就是 Kconfig 菜单。

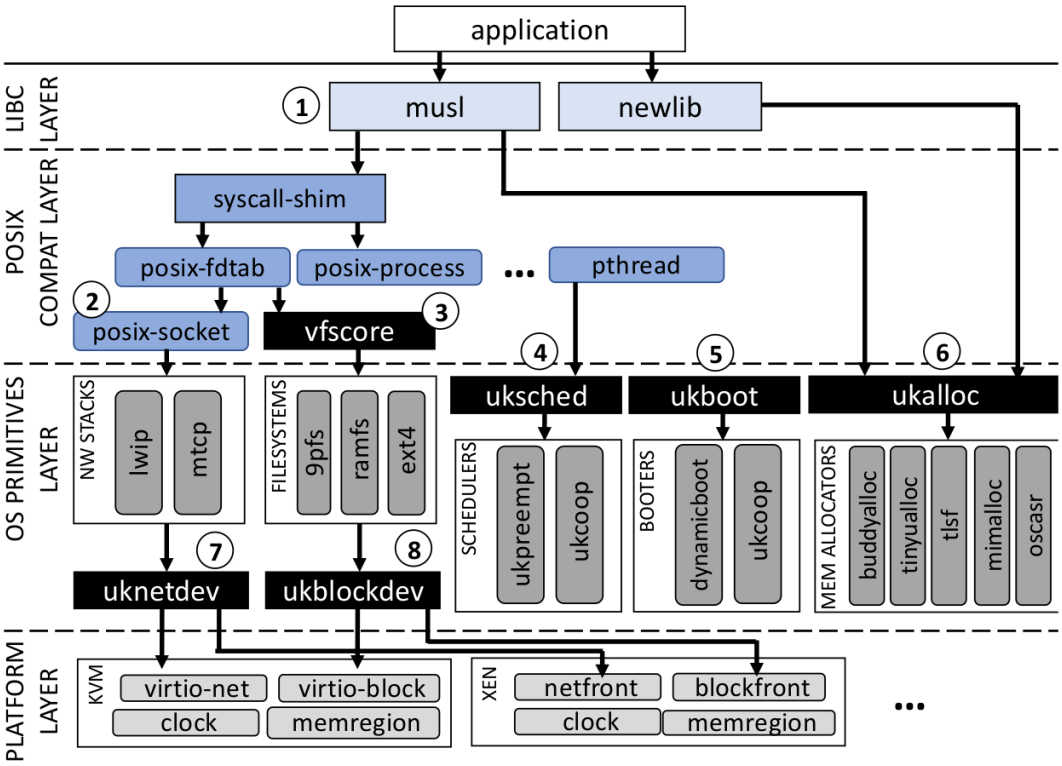

图 9: Unikraft 的架构（黑色框内的是 APIs）允许用户程序接入不同层次的 APIs，也允许用户选择不同的 API 实现。

类似地，程序的开发者可以轻松选择合适的内存分配器（⑥）以达到最高效率，甚至在同一个 unikernel 中使用多种分配器（如为引导程序选择简单、快速的内存分配器，并为程序本身选用默认分配器）。

关注快速引导的开发者也可以使用自己的遵守 `ukboot` API 的引导代码（⑤）。

对于网络密集型程序，开发者可以使用标准的套接字接口（②），或者使用更底层、更高效的 `uknetdev` API（⑦）以便大幅提高吞吐量。

类似地，数据库这样的硬盘密集型程序可以使用标志的 `vfscore` 微型库（③），或者用 `ukblock` API 提高吞吐量（⑧）。

调度器也是可以插拔的（④），而且每个 CPU 核可以运行不同的调度器。

与其他 unikernels 相比，Unikraft 的系统镜像更小、运行所需内存更小、吞吐量更大：

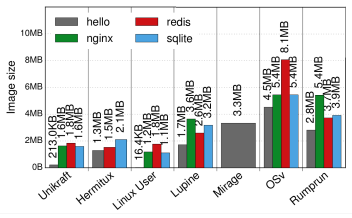

图 10

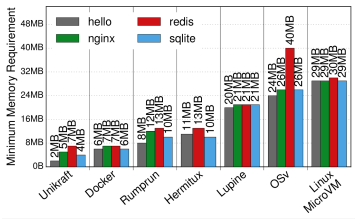

图 11

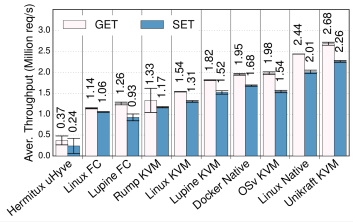

图 12

Unikraft 在提升效率的同时兼顾了安全性。根据 NCC Group[3]，虽然 unikernel 相比容器体积更小、隔离性更好，但是由于不存在内核态-用户态隔离，且缺乏 W^X、stack canary 等安全特性，unikernel 其实比传统的容器更不安全。也就是说，攻击者可以利用 unikernel 上的程序的漏洞，控制 unikernel 所在的虚拟机，进而获取本不应该拥有的权限。以 unikernel 的传统应用领域云计算为例，如果某个 unikernel 负责处理用户的敏感数据——它通过网络获取用

户的数据，然后将计算结果通过网络发回，则它一定拥有读用户数据的权限。那么，一旦这个 unikernel 存在安全漏洞，攻击者虽然不能控制 unikernel 所在的宿主机，但足够窃取用户的数据。所以，我们不能片面地把安全性与隔离性等同。而且我们不能片面地认为使用 Rust 这样的安全的程序设计语言就能保证安全[42]，因为完整的 unikernel 上不只包含安全的系统代码（或者说库代码），还包含可能不安全的用户代码，而后者可以导致整个系统不安全。因此，要实现安全的 unikernel，不能仅仅依靠安全的程序设计语言，而需要额外的安全特性。

NCC Group 提到的安全特性：

- 地址空间布局随机化（ASLR）：将数据、函数的位置随机放在内存段中，这样将无法通过跳转到特定地址完成攻击。
- 分页保护
  - W^X 政策：一段内存空间不能同时拥有执行权和写入权；
  - 内部数据加固：防止程序读取与程序本身无关的数据，比如全局偏移表；
  - 保护页：在不同数据段之间（比如.text 段与.data）放置不可读写的页面；
  - 空页面漏洞：`malloc` 函数可能返回 `nullptr`，而在 unikernel 上，`0x0` 可能是有效地址，访问 0 不会引发 segmentation fault。
- 栈保护标志（stack canary，典故：金丝雀曾经被用来检查煤矿的有毒气体[41]）：在栈的返回地址后加上一个随机的整型变量（canary），执行 `ret` 前检查 canary 是否被修改。
- 堆加固：堆的结构通常是双向链表，链表的元数据（如指针）通常与数据块相邻，堆上的溢出可以改变这些数据块，导致内存的分配、释放算法出错，通常的加固方法是为元数据增加校验值。
- 熵和随机数生成器：通常的 unikernel 缺乏足够产生密码学安全的伪随机数的硬件熵，这导致 unikernel 上生成的伪随机数的质量差，解决方法有使用 CPU 的专用随机数指令（如 x86 的 `rdrand`）
- 标准库加固
  - `printf` 的 `%n` 格式符：它将已经输出的字符的数量写入对应的参数指向的地址（没错，它不输出任何内容，而是写入内容），攻击者可以利用自定义的格式串和 `%n` 实现攻击，所以不应该支持它；
  - 自定义格式符：它可能可以扩大攻击面；
  - `_FORTIFY_SOURCE` 宏：定义它会导致一些函数执行轻量级的缓冲区溢出检查。

NCC Group 只测试了 rumprun 和 includeOS 两个 unikernels，并发现它们几乎没有实现任何安全特性。

尽管 Unikraft 使用 C 语言实现，但它支持（或计划支持）以下安全特性[42]：

Security feature	Status	Targets
Stack Smashing Protection (SP)	Upstream	`ARCH_ARM_64 || ARCH_X86_64`
Undefined Behavior Sanitization (UBSAN)	Upstream	any
Rust internal libraries in Unikraft	Upstream	`ARCH_X86_64`
ARM Pointer authentication (PAuth)	Under review	`ARCH_ARM_64 || ARCH_ARM_32`
ARM Branch Target Identification (BTI)	Under review	`ARCH_ARM_64`
Kernel Address Sanitizer (KASAN)	Under review	`PLAT_KVM && ARCH_X86_64`
Position Independent Executables (PIE)	Under review	`PLAT_KVM && ARCH_X86_64`
True Random Number Generator	Under review	`ARCH_X86_64`
ARM Memory Tagging Extension (MTE)	Work-in-progress	ARM
Intel Control-flow Enforcement Technology (CET)	Planned	`ARCH_X86_64`
Shadow stack	Planned	any
`FORTIFY_SOURCE`	Planned	any
ARM Speculation Barrier (SB)	Planned	`ARCH_ARM_64`

总的来说，Unikraft 是我们发现的最好的 unikernel 项目，所以我们的项目将主要参考它。

3 立项依据

by 张子辰

我们小组计划仿照 Unikraft 的架构，用 Rust 语言编写能在 RISC-V 架构 + KVM 平台上运行的 unikernel——Runikraft。Runikraft 的核心代码使用 Rust 编写，但允许用户代码使用任何语言编写——只要它能够被编译成入口为 `main` 的目标代码。Runikraft 强调构建系统镜像的简洁，用户只需要修改现有的项目的编译参数就可以构建基于 Runikraft 的系统镜像，而不必使用专用的工具链，更不需要重构代码。Runikraft 是 POSIX 兼容的，所以它将支持内存管理、进程调度，甚至磁盘管理和进程通信。不过，这些功能都是可选的且可拓展的，如果用户不需要某项功能，他可以不将相关模块打包进系统镜像中，如果用户能够提供某些功能的更好实现，他可以用自己的实现替换原有的模块，甚至 POSIX 兼容层本身也是可选的，如果用户愿意为了效率重构代码，他也可以直接用 Runikraft 的专用 API。Runikraft 可以支持多进程，因为我们认为，将若干密切管理的程序打包到一个镜像会提高效率。与 Unikraft 一样，Runikraft 在注重效率的同时兼顾安全性。我们计划实现 ASLR、W^X 政策、保护页、stack canary 四项安全技术。

以往的 unikernel 项目的不足之处可以概况为（并不每个 unikernel 都有所有缺点）：

- 无法兼顾效率和兼容性；
- 系统内的组件耦合度过高，系统不易裁剪或拓展；
- 需要使用专用的工具构建系统镜像；
- 将安全性与隔离性等同，忽视了单个 unikernel 虚拟机的安全；
- 不支持 RISC-V 架构；
- 核心代码使用不安全的程序设计语言编写。

而我们的项目将不具有以上缺点。

如果时间允许，我们还会尝试：

1. 支持更多架构，比如目前流行的 AMD64 和 ARMv8；
2. 支持在裸机上运行，虽然 unikernel 为云计算诞生，但这并不代表它只适合云计算领域，事实上，任何专一用途的设备上的系统都可以是 unikernel，而且 unikernels 理论上可以具有比现有的实时系统更高效率；
3. 支持调试，zos 小组[5.1.5] 曾做过相关研究；
4. 移植更多库。

我们考虑过但最终不打算实现与 Linux 的二进制兼容，即 unipanic[5.1.1] 小组的研究，因为我们认为不会出现需要移植无法获得源代码的程序的情况：

- 如果源代码因著作权问题无法获取，那移植二进制文件也会侵犯著作权；
- 如果源代码因软件无人维护无法获取，那这样的过时软件本身就不应该被继续使用。

在系统架构方面，我们将主要参考 Unikraft，并少量参考 MirageOS 和 RustyHermit；在技术方面，我们将参考 Chen and Wu 的 *rCore Tutorial Book*[43]。

4 前瞻性/重要性分析

by 张子辰

Rust 和 RISC-V 都是新兴事物，并且它们都是在吸取旧事物的教训的基础上诞生的，而且，实践表明，两者都正在经历蓬勃的发展，并正在分别逐步取代旧事物。而 unikernel 本身也是比较新颖的操作系统结构，它在云计算领域正在逐步取代传统的容器，并且有在嵌入式领域取代传统的嵌入式实时系统的潜能。因此，用 Rust 在 RISC-V 上开发 unikernel 顺应了历史的趋势。

此外，我们在开发的 Runikraft 将努力避免现有的 unikernels 的缺点，集中现有的 unikernels 的优点，所以，Runikraft 在 unikernel 领域内部也属于新事物，它将能促进 unikernel 的发展。

5 相关工作

5.1 往年项目

by 吴骏东

5.1.1 x-unipanic 小组

1. 项目简介

该项目旨在已有项目的基础上，小组希望在保持 Unikernel 现有优势（高效、安全、轻量）的前提下，改善 Unikernel 对二进制程序的支持，做出可以即时打包、分发的 Unikernel。目前致力于提供二进制兼容性的 Unikernel 项目 **HermiTux** 仍有较大改进空间，因此该小组将改善 **HermiTux** 二进制兼容性作为立项目标。

关键词：UniKernel, 进程调度

参考项目：HermiTux

2. 项目可行性分析

- 目前的 Unikernel 实现均要求对应用的重构，在实际应用中无法获取程序源码、程序依赖未被支持等问题非常常见
- 将应用打包为 Unikernel 要求大量的专业知识，步骤繁琐
- HermiTux 设计了一个二进制分析工具，能够扫描一个可执行文件，并检测该程序可以进行的各种系统调用
- HermiTux 基于hermitcore 这一 Unikernel 架构做了二进制支持，并重写 syscall 以保证性能。
- HermiTux 的内核中实现了一个基本的 RAM 文件系统——MiniFS，从而在这方面消除了对主机的依赖。

3. 项目困难点分析

- 如果无法获得程序源代码，重新编译和链接将无从进行，也就不可能打包到 Unikernel。对二进制文件的逆向往往会因为编译过程中的剥离和混淆难以进行，因此用 unikernel 层进行拆解和重新链接是不合适的。
- 让 Unikernel 支持某种语言十分困难，Unikernel 通常只支持一小部分的内核特性和软件库。如果语言用到了不支持的内容，就需要重写应用，很多情况下这意味着应用完全不可能被移植。
- Unikernel 使用复杂的构建工具，将一些传统应用的大型构建基础架构（大量的 Makefile、autotools/cmake 环境）加入 Unikernel 工具链是十分麻烦。并且，unikernel 还缺乏一些开发工具，如调试器（debugger）和分析工具（profiler）。

4. 项目成果分析

该小组主要参照了 KylinX 和 Hermitux 这两个项目。KylinX 项目提供实现 fork 的思路；Hermitux 主要实现 Unikernel 的二进制支持，可以在 Hermitux 的源码上进行改动。主要的成果有：

1. 支持 fork。参照了 KylinX 实现 fork 的方式，通过复制 hypervisor 启动新的虚拟机作为子进程。但是这样实现的性能较低，增大了系统负担。
2. 优化重写 syscall。修改了 syscall 打包的判断方式，将向后打包扩展成向前打包，从而 100% 重写了 syscall 函数。保持 Unikernel 因没有系统调用而具有的优良运行速度。

5.1.2 x-KATA-Unikernel 小组

1. 项目简介

该项目利用 Unikernel 得天独厚的轻量和攻击面小的特性，结合虚拟化技术，为 FaaS（Function As A Service）场景下的云服务提出一种解决方案：从客户端提交代码，到云平台进行 Serverless 运算。采用 KVM 的虚拟机接口，在虚拟化环境中以 Unikernel 减少资源开销，达到空间的高效利用和速度的极限提升。

关键词：UniKernel, 虚拟化, 云计算

参考项目：Kata、gVisor、Firecracker 文档

2. 项目可行性分析

- Firecracker 是在 rust 众多 crates 基础上实现的 VMM。它拥有非常有限的设备模型，提供轻量级的服务并且暴露的攻击面极小，在 FaaS 场景下有极大的应用空间。但其本质上还是传统的虚拟机架构，不可避免地带来多层嵌套的性能损耗。
- Google 提出的 gVisor 解决方案，在容器的后端将所有的系统调用截断，凭借 gVisor 中用户程序来实现系统调用的 API。gVisor 极其轻量，隔离性相对不足。此外，其也面临着过多系统调用时无法忍受的上下文转换问题。并且，gVisor 采用了带有 GC 的 Go 语言编写，也有比较大的性能开销。
- Unikernel 的缺点可以被 kata Container 易于分发的优点改善，同时纳入 kubernetes 生态，使得 Unikernel 的应用更加广泛。
- KVM 是采用硬件虚拟化技术的全虚拟化解决方案。其优势有：依赖 Linux 内核的内存管理、存储和客户机镜像格式多样、支持实时迁移与状态保存、支持高性能 I/O 接口、性能极强等。

3. 项目困难点分析

- Unikernel 的迁移问题。虽然 Unikernel 的概念被提出很久，市面上也涌现很多 Unikernel 的具体实现，但要找到易于适配 KVM，并且功能齐全的 core，是一件比较困难的事情。[项目使用了 Nanos 解决]
- 虚拟机对象问题。缺少统一的方式定义虚拟机的各种可管理对象。[项目使用了 libvirt 相关工具解决]
- 人机交互问题。与客户端的交互需要将虚拟机内部的结果重定向到主机，此过程中对结果的保护和加密是十分重要的。但这需要较多的知识积累。

4. 项目成果分析

该小组针对当前常用的两种解决方案 Firecracker microVM 和 gVisor 进行了改造与借鉴，利用 Firecracker 基于 KVM 和 virIO 的架构获得优异的封装和性能提升，同时希望借鉴 gVisor 系统调用截断的方式，使其与 Unikernel 进行交互，取代 gVisor 中 sentry+gofer 的类内核架构，从而达到轻量高效的目的。相关成果如下：

1. 使用了支持多种语言环境的 Nanos 内核，以 KVM 作为 Unikernel 的载体。
2. 分离 Nanos 的编译编排工具 ops 中的 build 模块，对虚拟机进行硬件加速。
3. 封装 libvirt API ，从而可以更加方便地创建与管理虚拟机。
4. 使用 virt-viewer 工具实现了虚拟机可视化。

改造后的 Unikernel 在算法性能上相较于传统 Linux 提升了约 40%。后续还可以将 Nanos 进一步与 Firecracker 结合，microVM 与 Unikernel 的结合可以将性能发挥到极限。

5.1.3 x-orz 小组

1. 项目简介

该项目将一般网络程序中的任务看作各种（并发的）基本服务的组合，抽象出一些常用的服务并让每个 Unikernel 与一个服务相对应，构成 Unikernel 实例的集群。通过合理地编排调度 Unikernel 集群，将各种并发的服务组合起来，处理任务请求，从而充分利用多核/多 CPU 资源，提高系统性能，同时又不破坏 Unikernel 原有的轻量、安全的特性。

关键词：Unikernel, 云计算, 高性能计算
参考项目：Firecracker、x-Doudou

2. 项目可行性分析

- Unikernel 省去了上下文切换、进程管理、资源竞争等工作带来的开销，但这样无法充分利用多核尤其是多 CPU 的资源。单个 Unikernel 进程通常仅使用一个核。支持多核的 Unikernel 往往需要引入 OS 中有关进程管理、资源分配的复杂模块，这样便会破坏 Unikernel 的高精简度。
- 小规模的多进程任务可以将其修改为多线程从而装入同一个 Unikernel 。但大规模任务只能启动更多的 Unikernel 实例，从而造成相同模块的重复使用。
- 服务的拆分提高了系统容错性。因为一个 Unikernel 实例相当于一个虚拟服务器，它的崩溃不会影响整个任务的执行，调度系统只需要再创建/调度另一个提供同样服务的 Unikernel 即可。
- Firecracker 是一个由 AWS 开发的轻量级 Hypervisor，旨在加速他们的 Serverless 服务。其仅实现了五种必要的 I/O 设备：virtio-net、virtio-block、virtio-vsock、串口、键盘，而且它的的启动过程也更为简单，省去了实模式加载等步骤，有着显著的性能提升。

3. 项目困难点分析

- 相关 OSv 内核的管理工具大部分是为虚拟机或容器开发的,不容易保留原本 OSv+Firecracker 方案的优势（如冷启动时间）。

4. 项目成果分析

该小组参考研究了工业控制系统的结构。其大体的工作流程是在 Interface 部分利用传感器等采集信号，然后通过 Information Processing 部分进行信息的处理，最后在 Intelligence 部分对系统进行智能控制。该项目选取了其中的信息处理部分的一小部分，将应用进行解耦与模块化。将相对独立的功能封装进 Unikernel 运行，来发挥 Unikernel 快速，安全，轻量的优点，满足相应需求。相关成果如下：

1. 选用支持多种语言的 OSv 作为 Unikernel 内核，并在此基础上对 OSv 中相关参数进行了修改，从而提升了 CPU 性能。
2. 使用 Go 语言实现一个轻量的 OSv 管理工具 Uigniter，功能包括创建、启动、停止 OSv 实例。详细内容见Uigniter 文档。

Unikernel 的解决方案具有容器方案所没有的隔离性、安全性、多进程/线程方案所没有的低延迟、轻量性、高容错率与模块解耦的特性，在未来 IoT 互联领域有着相当不错的前景。

5.1.4 X-Doudou 小组

1. 项目简介

该项目设计并初步实现了一个面向开发人员和系统管理人员的平台 Cunik ，用于方便地构建、分发、运行、管理 Unikernel 应用。Cunik 的设计目标是克服 Unikernel 配置难、部署繁琐的缺点，同时发挥 Unikernel 隔离性好、性能优良的特点，使运维人员轻松地获益于 Unikernel 这一新兴的技术。

关键词：Unikernel 、虚拟化、容器化
参考项目：libvirt、Rumprun 、OSv

2. 项目可行性分析

- Unikernel 在保持了原有的安全性、隔离性、易部署性的前提下，还做到了在启动速度、运行速度、内存开销等方面全面胜过 Docker。Unikernel 可以在不同的硬件平台上用不同的方法实现不同的应用程序，现在 Unikernel 正运行在世界各地的研究实验室、服务器机房以及各种低功耗设备上。

- Cunik 向用户隐藏繁琐的细节，使用户可以轻松地构建、分发、获取和配置 Unikernel 应用，降低开发、部署和运维成本，并可以克服 Unikernel 开发难度高、分发部署困难、对系统管理人员要求高、对现有云计算架构改动大的缺点。借助 Unikernel 的优势，Cunik 可以使用户轻松获得显著的性能提升和更高的安全性、减小攻击面、降低资源占用。
- libvirt 提供了便捷且功能强大的虚拟机管理工具。可以基于 libvirt 构建 Cunik-engine 的 VM Backends 和 VM Hypervisor 部分从而方便管理虚拟机。

3. 项目困难点分析

- 需要基于现有的 Unikernel 应用重新开发所需要的平台。

4. 项目成果分析

该小组通过 Python 完成了 Cunik-engine 和 Cunik-cli 的设计，并手动制作了包含 nginx(Rumprun)、redis(Rumprun) 和 redis(OSv) 的本地镜像仓库，用 Cunik 成功运行了这三种应用。最终在 redis(OSV) 这个应用上取得了比 Linux 上的原生进程更高的性能。

Cunik-engine 架构如下：

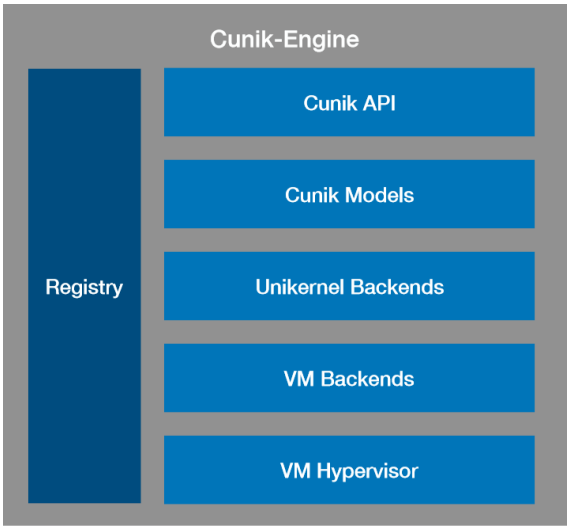

图 13

其具体细节可参考 X-Doudou 文档。调用 Cunik 后，程序会执行如下的内容：

1. 用户通过调用 Cunik API 中的 Creat、Run、Stop、Remove、Inspect 等 API 接口命令来启动 Cunik-engine。
2. Cunik-engine 在接受到命令后，首先会生成一个 Cunik Config，用于生成 Cunik Object。
3. 通过 Cunik Models，engine 会生成 Cunik Object，并加入到 Cunik Registry 中，或对已有 Cunik Object 进行运行状态的修改。
4. 然后，Unikernel Backends 会根据不同的 Cunik Object 选择不同的 Unikernel 实现方式。
5. 接下来，根据所选择的 Unikernel 实现方式，并在 Image Regsitry 中查询 Unikernel 应用的 image ，然后由 VM Backends 生成 VM Config。
6. VM Hypervisor 接收 VM Config 并选择合适的虚拟机来运行这个 Unikernel 应用。

该项目目前只实现了对 kvm/qemu 虚拟机、Rumprun 和 OSv 两种 Unikernel 实现的简单支持。可以改进的内容包括：

- 整理当前 Cunik-engine 的架构；
- 实现对更多虚拟机平台以及 Unikernel 实现的支持；
- 持续支持新的 Unikernel 实现，并加入更多方便镜像打包与应用部署的特性，使其能够满足生产环境的需要；

- 更好的交互体验：实现在用户发出 Request 后，自动为用户选择最合适的一系列 Cunik 应用，达成从前端到后端的一键式搭建服务。

5.1.5 X-zos 小组

1. 项目简介

该项目设计了一个利用系统自带的虚拟网卡，通过 socket 和多线程并发收发调试信息的日志式调试系统 Umonitor，为运维人员对 Unikernel 的调试和维护工作提出了更为轻松有效的解决方案。用户在使用时只需在每个需要调试的 Unikernel 里调用工具中的 `send_log()` 函数，将想要得到的调试信息传入函数，然后在主机的环境里面启动一个 host 端，host 端就能通过虚拟网卡接口接收到来自不同 Unikernel 的调试信息并整理保存。

关键词：Unikernel、调试

参考项目：Rumprun

2. 项目可行性分析

- 传统的调试手段在 Unikernel 上难以进行。包括：
  - 通过与其他进程通信来进行追踪和调试
    Unikernel 为了实现精简，而放弃了原有的很多功能，其中就包括多进程切换。没有了多进程，就无法利用与其他进程通信来进行 debug。
  - 编程过程中将信息输出在控制台或者文件中
    在实际运行 Unikernel 的时候是不会模拟显示器的，所以无法将调试信息输出到控制台。又因为 Unikernal 的文件系统做了很大的精简，没有 VFS，而且不同 Unikernal 的文件系统设计也不完全一样, 所以，我们如果将日志写入文件，就很难再将虚拟磁盘中的东西读出来。
- Unikernel 采用了比较原始的单地址空间方式，这可以简化了调试的难度。单地址空间有助于定位需要的信息在的位置，而并不会影响 Unikernel 的性能。

3. 项目困难点分析

- 可能实现的方案选择很多，包括文件 I/O，串口通信，网络通信等。如何选择最合适的方案需要一定的时间与试错成本。本实验最终选择了通过网络完成 unikernel 向 host 发送日志信息的过程。

4. 项目成果分析

该小组设计的 Umonitor 已经可以在 rumpkernel 的平台上通过对 Unikernel 源代码的修改，通过网络通信的方式将 Unikernel 中我们想要的调试信息输出到制定文件中，初期制定目标已经达到。项目的优势包括：

1. **并发性**：只需启动一个 host 端就能服务复数的 Unikernel 而无需多开，提高了效率；
2. **兼容性**：避开了不同的 Unikernel 的差异性，如使用的语言，内存空间，文件系统等的不同，选择了它们的共性，对 socket 的支持作为实现方法，几乎所有的 Unikernel 都能无难度地移植这个调试系统；
3. **高度可控可定制化**：直接在运行 Unikernel 的虚拟机的模拟 vga 输出界面打印调试信息会造成很大的切换和检索的麻烦，而重定向 vga 输出信息至某个文件会输出非常多 Unikernel 自带对调试无用或者不够清晰的信息，不能得到一个组织良好的日志文件。此外，多个 Unikernel 并发重定向在某些情况下可能造成输出混杂，不能正确地输出文件。利用 socket 传递自己想要的调试信息并组织保存，能够生成用户自己最需要的最有用的日志文件，提高调试的效率。

项目可以改进的方向包括：

1. Unikernel 的调试工具必然需要提供一个通用的接口以实现对不同种类 Unikernel 的支持。目前的 Umonitor 已经实现了能同时对多个 Unikernel 的调试，所以下一步的目标可以是实现对多种 Unikernel 的通用接口。使其够方便的支持现阶段较为成熟的 Unikernel 实现的同时也能够通过用户友好的配置界面对其他 Unikernel 进行支持。
2. Umonitor 在运行之后实际上仍然只能被动地接受被调试的 Unikernel 输出的调试信息，这样虽然能够在一次设置后找到对应的错误信息出现的位置，但想要在 Unikernel 运行中途添加调试信息输出或者更进一步的设置断点和逐句执行都还做不到。可以考虑添加交互式调试功能。

参考文献

[1] ——. *Unikernels: Rethinking Cloud Infrastructure*[Z/OL]. @unikernel [2022-02-18]. https://web.archive.org/web/20220218194213/http://unikernel.org/

[2] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu and Felipe Huici. *Unikraft: Fast, Specialized Unikernels the Easy Way*[J/OL]. EuroSys '21, 2021, April: 26–29 [2022-03-27]. https://dl.acm.org/doi/10.1145/3447786.3456248 doi:10.1145/3447786.3456248.

[3] Spencer Michaels and Jeff Dileo. *Assessing Unikernel Security* [R/OL]. Version 1.0. NCC Group, 2019: 4-10 [2022-03-27]. https://research.nccgroup.com/wp-content/uploads/2020/07/ncc_group-assessing_unikernel_security.pdf

[4] Rust Programming Language https://www.rust-lang.org/

[5] 我们为什么要选择小众语言 Rust 来开发软件? https://www.techug.com/post/why-we-choose-rust-to-dev.html

[6] What is Rust and why is it so popular? - Stack Overflow Blog https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

[7] 也许是最客观、全面的比较 Rust 与 Go：都想把 Rust 也学一下https://www.cnblogs.com/Chary/p/14097609.html

[8] 为什么要使用 Rust 语言? Rust 语言的优势在哪里?https://www.zhihu.com/question/393796866

[9] [Rust programming language - what is rust used for and why is so popular?]https://codilime.com/blog/why-is-rust-programming-language-so-popular/

[10] [Rust by the Numbers: The Rust Programming Language in 2021 – The New Stack]https://thenewstack.io/rust-by-the-numbers-the-rust-programming-language-in-2021/

[11] [Compatibility - Rust By Example (rust-lang.org)]https://doc.rust-lang.org/rust-by-example/compatibility.html

[12] David Patterson and Andrew Waterman. *RISC-V 手册：一本开源指令集的指南* [S]. 勾凌睿, 黄成, 刘志刚译. 2018: 13-21

[13] https://www.ics.com/blog/what-risc-v-and-why-it-important

[14] https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/RISC-V-Options.html#RISC-V-Options

[15] https://clang.llvm.org/docs/ClangCommandLineReference.html#riscv

[16] https://elinux.org/images/3/3d/Linux_riscv_elce2020.pdf

[17] https://wiki.debian.org/RISC-V

[18] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*[S/OL]. Editors Andrew Waterman and Krste Asanović. Version 2.1. RISC-V Foundation, 2019 (20191213) [2022-03-25]. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[19] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*[S/OL]. Editors Andrew Waterman, Krste Asanović and John Hauser. Version 1.9.1. RISC-V International, 2021 (20211204) [2022-03-27]. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[20] *RISC-V Debug Support* [S/OL]. Editors Ernie Edgar and Tim Newsome. Version 1.0.0-Stable. SiFive, Inc., 2022(20220209)[2022-03-27]. https://github.com/riscv/riscv-debug-spec/raw/b659d7dc7f578e1a2a76f9e62a5eec0f2d80045c/riscv-debug-stable.pdf

[21] *RISC-V Processor Trace*[S/OL]. Gajinder Panesar and Iain Robertson. Version 1.0. UltraSoC Technologies Ltd., 2020(20200320)[2022-03-27]. https://github.com/riscv/riscv-trace-spec/raw/e372bd36abc1b72ccbff31494a73a862367cbb29/riscv-trace-spec.pdf

[22] *AMD64 Architecture Programmer's Manual: Volumes 1-5*[S/OL]. Revision 4.04, Advanced Micro Devices, Inc., 2021 (202111) [2022-03-27]. https://www.amd.com/system/files/TechDocs/40332.pdf

[23] *Intel® 64 and IA-32 Architectures Software Developer' s Manual*[S/OL]. Order Number: 325462-076US, Intel Corporation, 2021 (202112) [2022-03-27]. https://cdrdv2.intel.com/v1/dl/getContent/671200

[24] Arm® Architecture Reference Manual for A-profile architecture[S/OL]. Version 21.0, Arm Limited, 2022 [2022-03-27]. https://developer.arm.com/documentation/ddi0487/latest

[25] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco and Felipe Huici. *ClickOS and the Art of Network Function Virtualization*[C/OL]. 11th USENIX Symposium on Networked Systems Design and Implementation. Seattle: USENIX Association. 2014: 459-473[2022-03-22]. http://cnp.neclab.eu/projects/clickos/clickos.pdf

[26] Joao Martins, Mohamed Ahmed, Costin Raiciu and Felipe Huici. *Enabling Fast, Dynamic Network Processing with ClickOS*[J/OL]. HotSDN'13, 2013, August: 67-72[2022-03-22]. http://cnp.neclab.eu/projects/clickos/clickos-workshop.pdf

[27] Anil Madhavapeddy and David J. Scott. *Unikernels: Rise of the Virtual Library Operating System*[J/OL]. ACM Queue, 2014, 11 [2022-03-22]. https://queue.acm.org/detail.cfm?id=2566628

[28] Bratterud et al. *IncludeOS: A minimal, resource efficient unikernel for cloud systems*[G/OL]. Adrian Colyer, The morning paper: a random walk through Computer Science research. 2016 (20160222) [2022-03-22]. https://blog.acolyer.org/2016/02/22/includeos

[29] Nur Hussein. *IncludeOS: a unikernel for C++ applications*[Z/OL]. 2017 (20170725) [2022-03-22]. https://lwn.net/Articles/728682/

[30] [rusty-hermit 0.3.10 doc]https://docs.rs/crate/rusty-hermit/0.3.10

[31] [Rust Runtime 与 ABI——知乎专栏]https://zhuanlan.zhihu.com/p/370897059

[32] [Rust 运行时 - Rust 参考 (rust-lang.org)]https://doc.rust-lang.org/reference/runtime.html

[33] [The 'RustyHermit' Unikernel——Rust OSDev]https://rust-osdev.com/showcase/rusty-hermit/

[34] [Intra-Unikernel Isolation with Intel Memory Protection Keys.pdf]../../references/Intra-UnikernelIsolationwithIntelMemoryProtectionKeys.pdf

[35] [MPK——Core API Doc]https://www.kernel.org/doc/html/latest/core-api/protection-keys.html#:~:text=Memory%20Protection%20Keys%20provides%20a%20mechanism%20for%20enforcing,to%20a%20%E2%80%9Cprotection%20key%E2%80%9D%2C%20giving%2016%20possible%20keys.

[36] [linux 内核那些事之 Memory protection keys(硬件原理)——CSDN 博客]https://blog.csdn.net/weixin_42730667/article/details/121386896

[37] [Rump kernel——Wikipedia (其中有介绍 Anykernel)]https://en.wikipedia.org/wiki/Rump_kernel

[38] [Xen on Rump Kernels and the Rumprun Unikernel——XenProject]https://xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/

[39] [All About Unikernels: Part 2, Two Different Approaches, MirageOS and Rumprun——Container Solutions blog]https://blog.container-solutions.com/all-about-unikernels-part-2-mirageos-and-rumprun

[40] [The Rumprun Unikernel]../../references/TheRumprunUnikernel.pdf

[41] https://en.wikipedia.org/wiki/Animal_sentinel#Historical_examples

[42] https://unikraft.org/docs/features/security/

[43] Yu Chen and Yifan Wu. *rCore Tutorial Book*[M/OL]. Version 3, 2022(20220102) [2022-03-27]. https://rcore-os.github.io/rCore-Tutorial-Book-v3/chapter0/0intro.html