Linux Binary Compatible Unikernels

How your Application runs on Unikraft

Simon Kuenzer

Project Founder & Lead Maintainer

CTO & Co-Founder
Unikraft GmbH
simon@unikraft.io
Unikraft: The Unikernel SDK
Unikernel Primer

- **Single purpose**: One application & one target platform
  - Flat and single address space
  - Only necessary kernel components
  - Small TCB and memory footprint
Current project focus: Linux Compatibility

- Our vision: Seamless application support
  → Most software is developed for Linux
  → Remove obstacles for running them on Unikraft
The 2 Approaches for Compatibility

Native (API-compatible)

Binary compatible (ABI-compatible)

musl

elfloader

syscall_shim

syscall mapping

posix-process

posix-user

vfscore

posix-mmap

Compat layer

Application
2

Loading ELF Binaries
Loading ELF Binaries

- Straight-forward process:
  1) Parse & load executable/loader
  2) Prepare entrance stack, jump to entrance
  3) Interact with system calls
Challenge PIE vs. Non-PIE Executables

- Non-PIE dictates AS-layout
  - Single AS $\rightarrow$ only one non-PIE app
  - Limits area where (uni-)kernel relies

![Diagram showing AS, Application space, and Kernel space]
Challenge PIE vs. Non-PIE Executables

- Non-PIE dictates AS-layout
  - Single AS → only **one** non-PIE app
  - Limits area where (uni-)kernel relies

- PIE provides AS-layout flexibility
  - Multiple apps in single AS possible
  - No AS-switch on context switches
Challenge PIE vs. Non-PIE Executables

- Non-PIE dictates AS-layout
  - Single AS → only one non-PIE app
  - Limits area where (uni-)kernel relies

- PIE provides AS-layout flexibility
  - Multiple apps in single AS possible
  - No AS-switch on context switches
  - Opportunity:
    *Full-stack ASLR with max. entropy*
Challenge PIE vs. Non-PIE Executables

- Non-PIE dictates AS-layout
  - Single AS → only one non-PIE app
  - Limits area where (uni-)kernel relies

- PIE provides AS-layout flexibility
  - Multiple apps in single AS possible
  - No AS-switch on context switches
  - Opportunity: Full-stack ASLR with max. entropy

Go binaries still commonly built without PIE for Linux
Interesting read: https://rain-1.github.io/golang-aslr.html

Major distros moved to PIE for security hardening with ASLR ~5-20 years ago
https://isopenbsdsecu.re/mitigations/pie/
https://wiki.debian.org/Hardening/PIEByDefaultTransition
Challenge PIE vs. Non-PIE Executables

- **Non-PIE dictates AS-layout**
  - Single AS → only one non-PIE app
  - Limits area where (uni-)kernel relies

- **PIE provides AS-layout flexibility**
  - Multiple apps in single AS possible
  - No AS-switch on context switches
  - *Opportunity: Full-stack ASLR with max. entropy*

---

Go binaries still commonly built without PIE for Linux
Interesting read: [https://rain-1.github.io/golang-aslr.html](https://rain-1.github.io/golang-aslr.html)

Major distros moved to PIE for security hardening with ASLR ~5-20 years ago
[https://isopenbsdsecure.re/mitigations/pie/](https://isopenbsdsecure.re/mitigations/pie/)
[https://wiki.debian.org/Hardening/PIEByDefaultTransition](https://wiki.debian.org/Hardening/PIEByDefaultTransition)

---

AS

Application space

Kernel space

---
3

System Calls
System Call Trap Handler

syscall → Switch to auxiliary stack → Save extended registers (FPU, SSE, …) → Save & switch to TLS register → Handler function → Restore TLS register → Restore extended registers → Switch to application stack → jmp

*here: x86_64
System Call Trap Handler

- Special instruction
  - Takes care of protection domain switch (that we do not need)
- x86_64: `jmp` instead of `sysret` because of implicit privilege mode change to ring 3 [1]

System Call Trap Handler

- Needed to be compliant with Linux ABI: The system call handler must not require a userland stack
- In reality: Only needed for apps where userland stack is too small (e.g., go)
System Call Trap Handler

- Needed if we compile Unikraft with full CPU features utilization
System Call Trap Handler

- TLS used as TCB in Unikraft
  - Compartmentalization of library implementations (no central TCB structure definition needed)
System Call Trap Handler

- syscall
- Switch to auxiliary stack
- Save extended registers (FPU, SSE, ...)
- Save & switch to TLS register
- Handler function
- Restore TLS register
- Restore extended registers
- Switch to application stack
- jmp

■ Actual system call handler function
System Call Trap Handler

- syscall
- Switch to auxiliary stack
- Save extended registers (FPU, SSE, ...)
- Save & switch to TLS register
- Handler function
- Restore TLS register
- Restore extended registers
- Switch to application stack
- jmp

- Actual system call handler function

Is there a more direct approach?
vDSO and \texttt{\_\_kernel\_vsyscall()}

- vDSO\cite{vDSO} in Unikraft is a symbol lookup table only
  - Within single-AS/single-protection domain we can directly execute kernel functions

---

vDSO and \texttt{\_\_kernel\_vsyscall}()

- vDSO\textsuperscript{[1]} in Unikraft is a symbol lookup table only
  - Within single-AS/single-protection domain we can directly execute kernel functions
- Resurrect \texttt{\_\_kernel\_vsyscall}()
  - *Origin* i386: Switch between int\_0x80/sysenter/syscall depending on CPU \textsuperscript{[1]}
  - Idea: Use this mechanism to enter Unikraft
    - Normal function call
    - No trap, interrupt or privilege domain change
    - No need to save & restore extended context \textsuperscript{[2]}

\textsuperscript{[1]} \url{https://man7.org/linux/man-pages/man7/vdso.7.html}
\textsuperscript{[2]} System V Application Binary Interface, 3.2.1 Registers, \url{https://gitlab.com/x86-psABIs/x86-64-ABI}
vDSO and __kernel_vsyscall()

- vDSO[1] in Unikraft is a symbol lookup table only
  - Within single-AS/single-protection domain we can directly execute kernel functions
- Resurrect __kernel_vsyscall()
  - Origin i386: Switch between int_0x80/sysenter/syscall depending on CPU [1]
  - Idea: Use this mechanism to enter Unikraft
    - Normal function call
    - No trap, interrupt or privilege domain change
    - No need to save & restore extended context [2]
  - Patch application’s libc.so
    - Most syscalls done via libc wrappers

---

System Call Trap Handler

1. syscall
2. Switch to auxiliary stack
3. Save extended registers (FPU, SSE, …)
4. Save & switch to TLS register
5. Handler function
6. Restore TLS register
7. Restore extended registers
8. Switch to application stack
9. jmp
System Call Trap Handler

Function call __kernel_vsyscall()
4

The fork Dilemma
The **fork Dilemma**

- *fork* traditionally used for
  - a) Creating worker processes
  - b) Instantiating new applications with *fork* + *exec*

![Diagram showing fork and exec processes]

Low addr

Parent AS

Stack

Child AS

Stack

Low addr

High addr

fork (copy, CoW)

exec

1

2
The fork Dilemma

- fork traditionally used for
  a) Creating worker processes
  b) Instantiating new applications with fork + exec

\[ \text{Low addr} \quad \text{Parent AS} \quad \text{Stack} \quad \text{Child AS} \quad \text{Stack} \quad \text{High addr} \]

\( \text{fork (copy, CoW)} \)

\( 1 \)

\( 2 \)

\( \text{exec} \)

\( \rightarrow \) Issue: Mechanism relies on per-process ASes
fork in a Unikernel

- Single AS: Child must be located at different address range as the parent
  - Copy&Patching hardly possible without compiler support, e.g.,
    - return addresses on the stack
    - absolute pointers
  → Worker processes cannot be created this way 😞


luckily, recent software prefer multi-thread model instead

**fork in a Unikernel**

- Single AS: Child must be located at different address range as the parent
  - Copy&Patching hardly possible without compiler support, e.g.,
    - return addresses on the stack
    - absolute pointers
  → Worker processes cannot be created this way 😢
  *luckily, recent software prefer multi-thread model instead*

→ Instantiating new application (fork+exec)
  - A PIE application can be loaded to any address
  - In principle multi-process with single-AS should work

---

**fork in a Unikernel**

- Single AS: Child must be located at different address range as the parent
  - Copy&Patching hardly possible without compiler support, e.g.,
    - return addresses on the stack
    - absolute pointers
  → Worker processes cannot be created this way 😞
  
  *luckily, recent software prefer multi-thread model instead*

→ Instantiating new application (`fork+exec`)
  - A PIE application can be loaded to any address
  - In principle multi-process with single-AS should work

A Solution: `vfork+exec`

- `vfork [1]`: Shares memory and stack with parent
  - No MMU required → we can keep single AS
  - Parent is suspended until child exits or calls `exec`
- `exec`: will drop current memory image and launch a new one from executable
  - PIE executable loaded to different base address and executed (elfloader)

A Solution: vfork+exec

- vfork [1]: Shares memory and stack with parent
  - No MMU required → we can keep single AS
  - Parent is suspended until child exits or calls exec
- exec: will drop current memory image and launch a new one from executable
  - PIE executable loaded to different base address and executed (elfloader)

→ Outlook/Trial: Translate fork+exec to vfork+exec

Risk of Bloat due to Linux Compatibility
Risk of Bloat due to Linux Compatibility

- Network interfaces and routing (getifaddrs() and co.)
  - Need complex subsystem in between: netlink sockets
  - Alternative: Provide functions directly via the vDSO (trade-off: libc patching)

Risk of Bloat due to Linux Compatibility

- Network interfaces and routing (getifaddr() and co.)
  - Need complex subsystem in between: netlink sockets
  - Alternative: Provide functions directly via the vDSO (trade-off: libc patching)

- Applications relying on specific Linux behaviors
  - For example: Preemptive scheduling:
    - e.g., frankenphp, mysql, initialize thread pools with busy waiting

Risk of Bloat due to Linux Compatibility

- Network interfaces and routing (getifaddr() and co.)
  - Need complex subsystem in between: netlink sockets
  - Alternative: Provide functions directly via the vDSO (trade-off: libc patching)

- Applications relying on specific Linux behaviors
  - For example: Preemptive scheduling:
    - e.g., frankenphp, mysql, initialize thread pools with busy waiting

- System call stubbing [1]:
  - Not all system calls need a full implementation
    - A number of syscalls can be stubbed (fake-it) but application dependent

Risk of Bloat due to Linux Compatibility

- Network interfaces and routing (getifadd() and co.)
  - Need complex subsystem in between: netlink sockets
  - Alternative: Provide functions directly via the vDSO (trade-off: libc patching)

- Applications relying on specific Linux behaviors
  - For example: Preemptive scheduling:
    - e.g., frankenphp, mysql, initialize thread pools with busy waiting

- System call stubbing [1]:
  - Not all system calls need a full implementation
    - A number of syscalls can be stubbed (fake-it) but application dependent

- Filesystem Hierarchy Standard [2]:
  - Specific files and file systems (e.g., /proc, /etc) at expected places and behavior
    
    Many of them can resolved by placing files with meaningful content in the VFS

Join us!

- OSS project
  unikraft.org
- Get started with kraftkit
  github.com/unikraft/kraftkit
- Code & Contributing
  github.com/unikraft

Follow us on
- Twitter: @UnikraftSDK
- LinkedIn: https://linkedin.com/company/unikraft-sdk
Thank you!