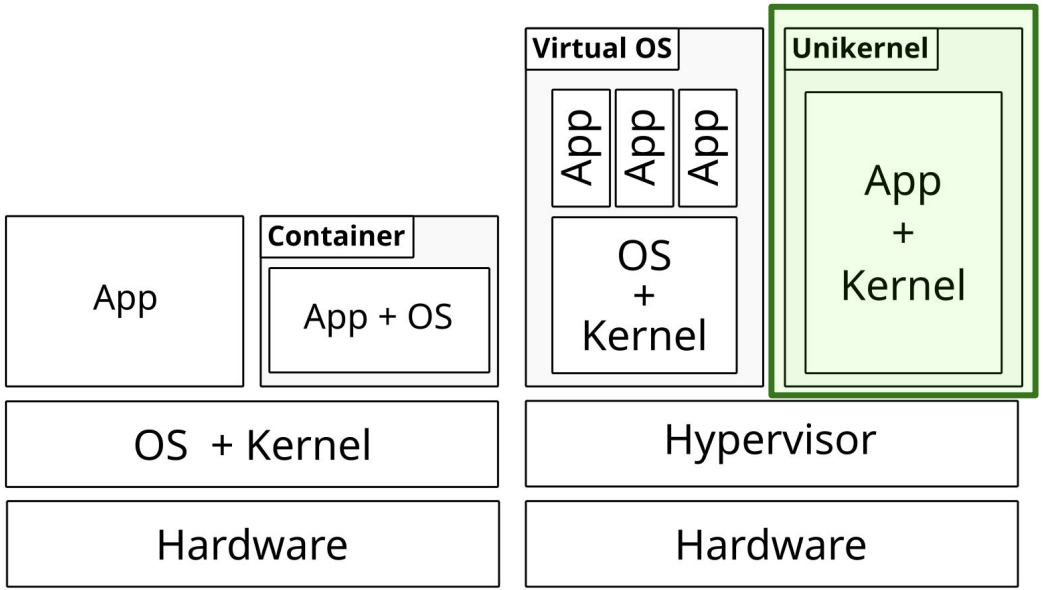
Extending unikernels with a language runtime

Vanda Hendrychová
Advisor: Dr. Masanori Misono
Chair of Distributed Systems and Operating Systems
https://dse.in.tum.de/

15.11.2022 – 22.5.2023

Unikernels: run your apps directly on the hypervisor



Unikernels

- Specialized OS images
- Single process
- Single address space

➔ **short boot time**

➔ **high performance**

➔ **small image size**

Why aren't unikernels more popular?

- difficult to inspect and debug
- including external tools bloats the VM image + requires recompilation

Can we load external applications at runtime?

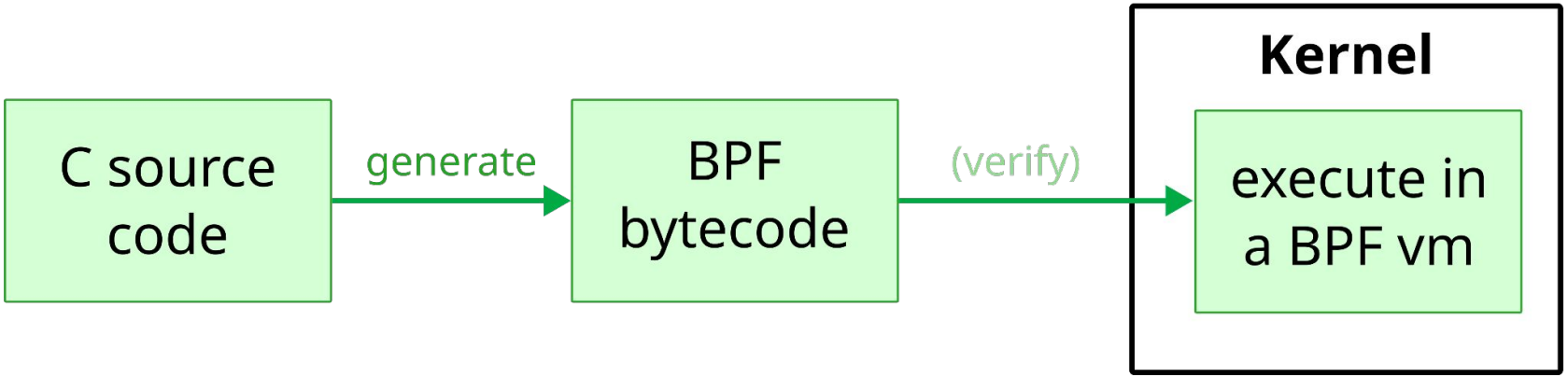
Project goal: Safely load binary programs into a unikernel at runtime

Requirements:

- **Safety:** Injected code doesn't crash, code finishes in finite time
- **Performance:** Performance impact on the main application should be minimal.
- **Sandboxing:** Injected code doesn't modify application memory in undesirable ways.

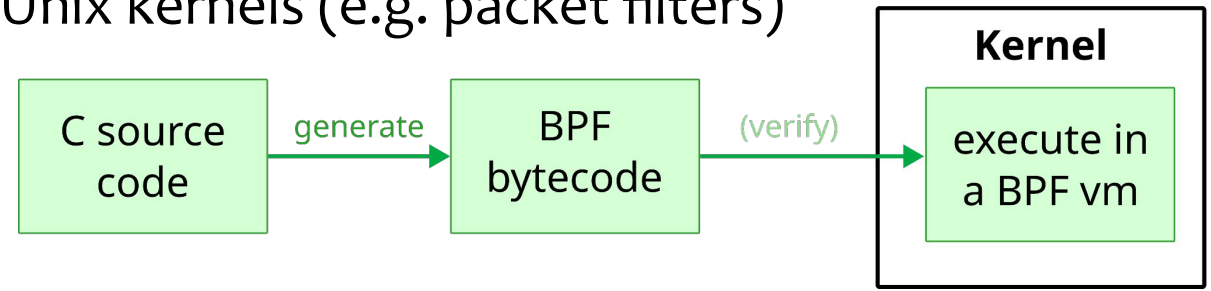
The BPF language runtime provides a potential solution

BPF: simple instruction set designed for safely executing dynamically loaded code in Unix kernels (e.g. packet filters)



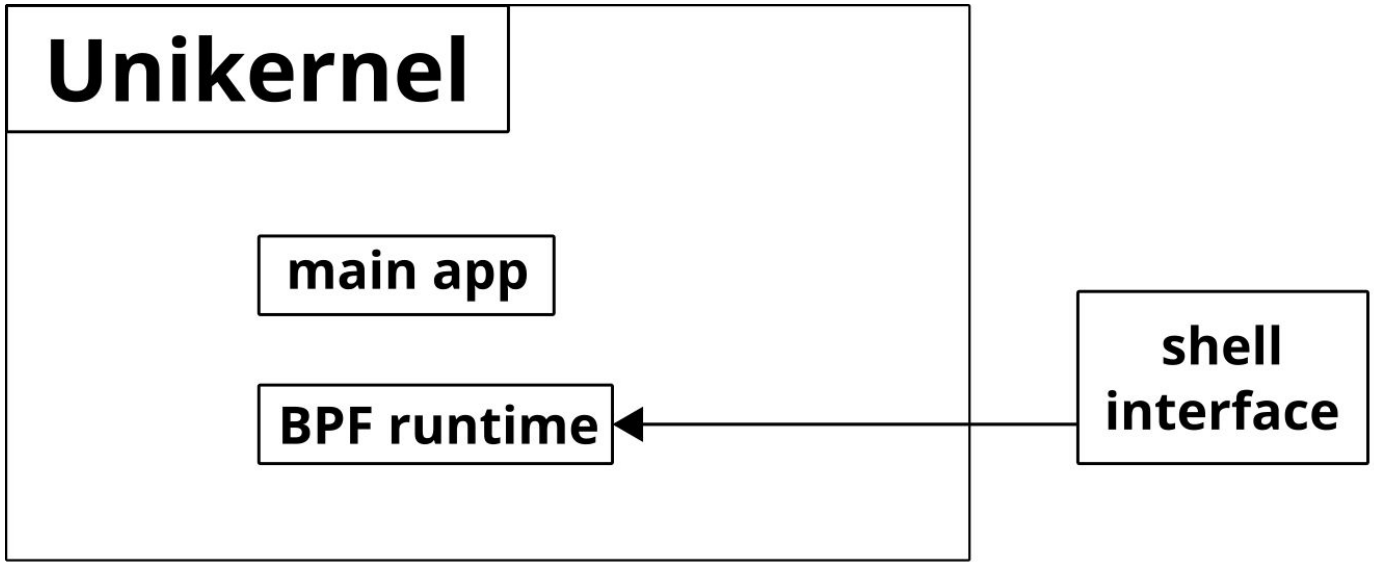
The BPF language runtime provides a potential solution

BPF: simple instruction set designed for safely executing dynamically loaded code in Unix kernels (e.g. packet filters)



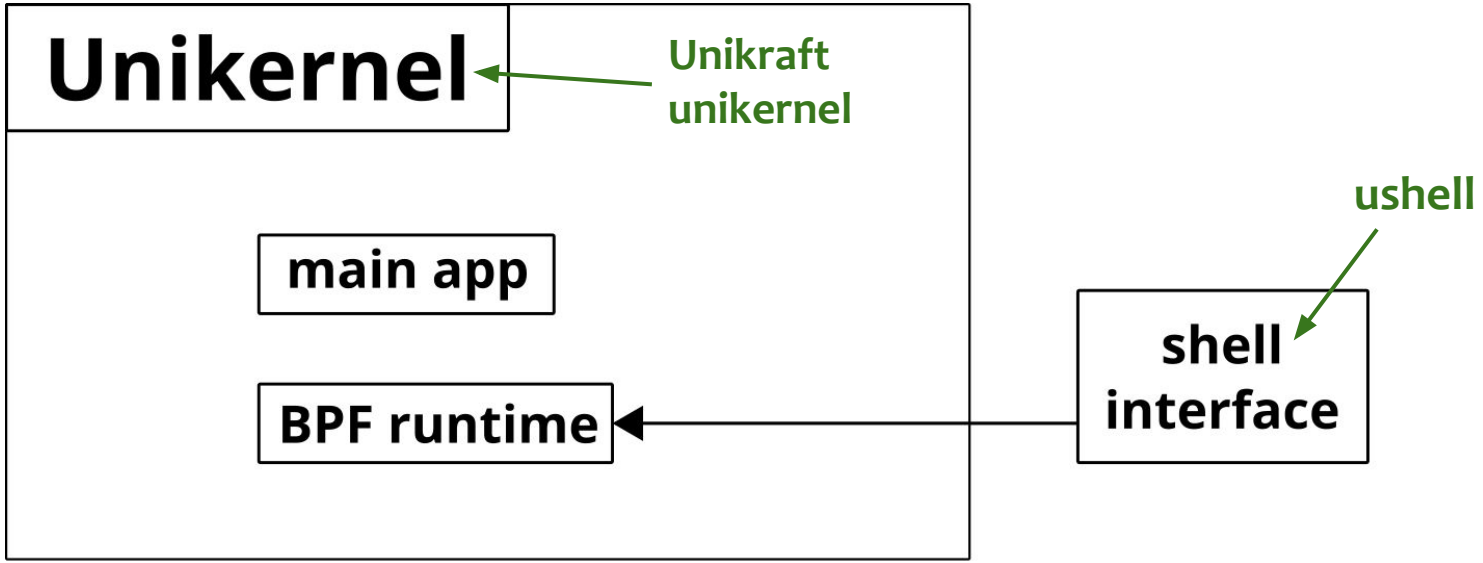
- ✓ **Safety:** BPF code can be verified + runs in a virtual machine
- ✓ **Performance:** BPF code can be compiled just-in-time or beforehand
- ✓ **Sandboxing:** BPF VMs have limited memory access

Overview

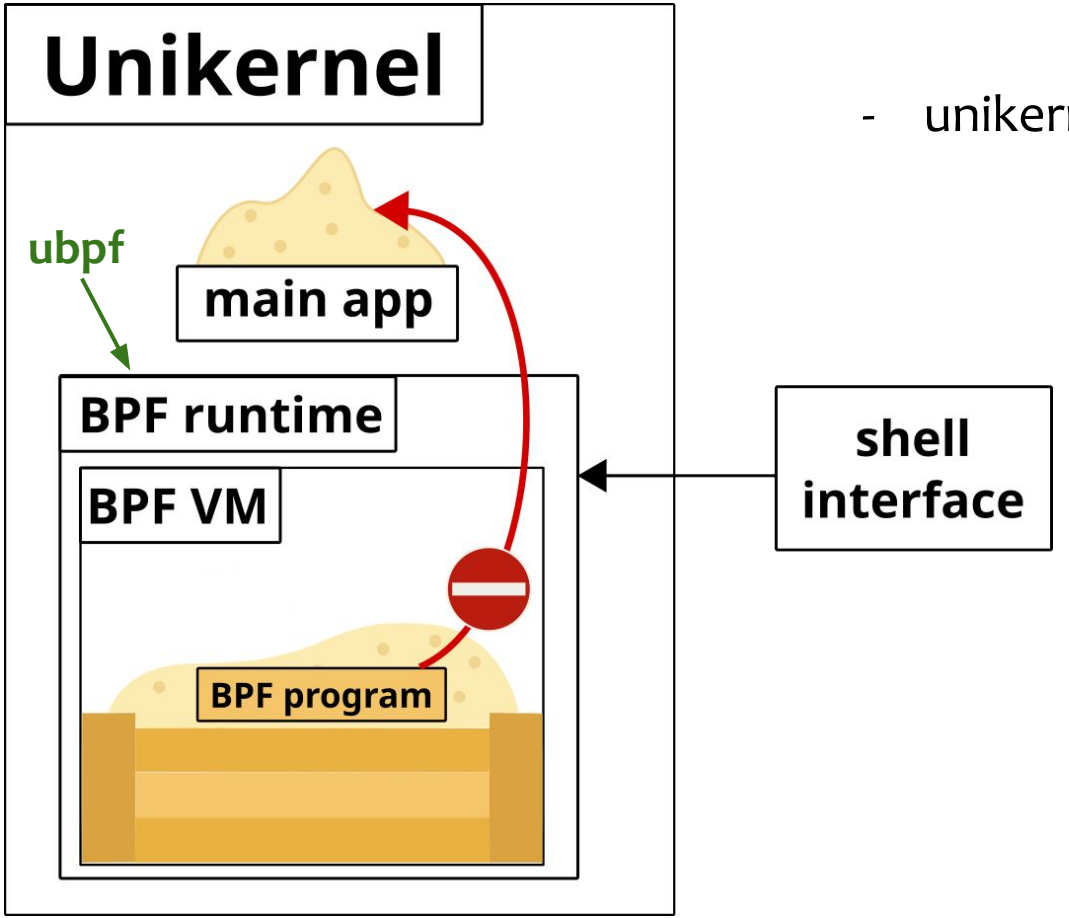


Design challenge #1: No common interface

- unikernels: no common (shell) interface, only framework-specific shells

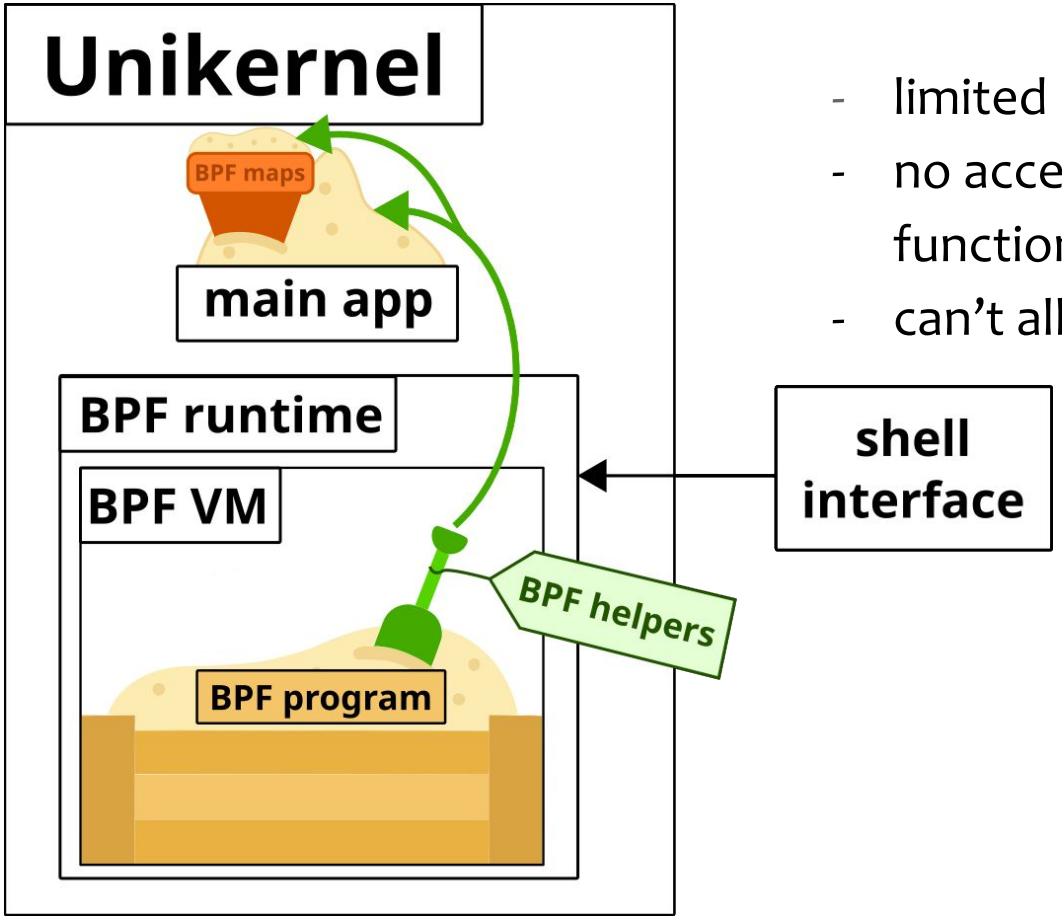


Design challenge #2: Lack of isolation



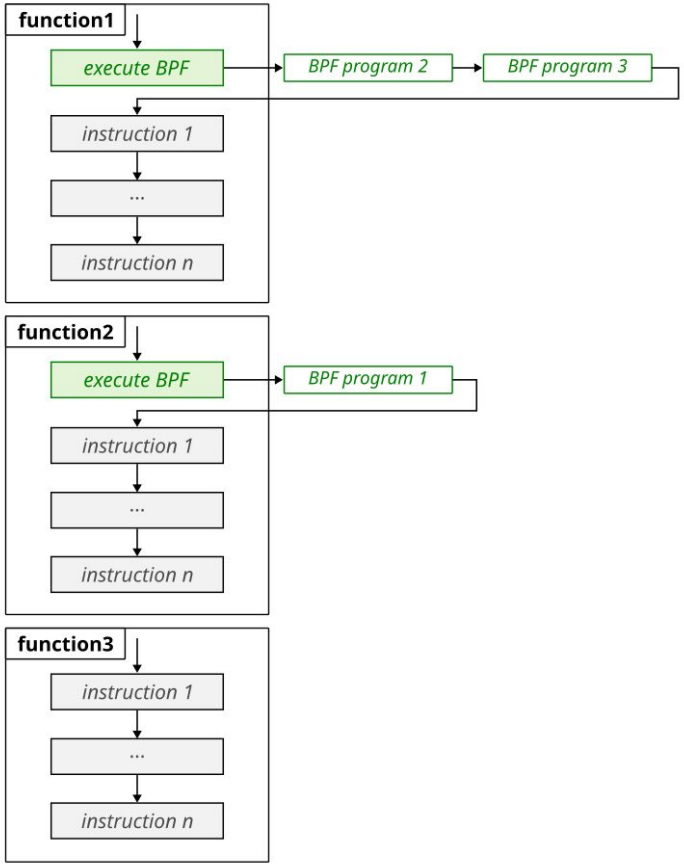
- unikernels: no process isolation

Design challenge #3: Limitations of BPF



- limited memory access
- no access to shared libraries or functions in the main app
- can’t allocate memory from BPF

Showcase application: Dynamic tracer



- trace functions by attaching custom binaries to their entrypoints
- load binaries at runtime
- read and write stats about function calls

Evaluation: Safety - Loops

Attached shell:

```
> bpf_exec loop_fail.bin
Failed to load code: infinite loop at PC 2
```

loop_fail.c: (compiled into loop_fail.bin)

```c
int bpf_prog(void *arg)
{
    while (1)
        ;
    return 0;
}
```

Simple infinite loops are **detected by the bounds checker**
-> BPF program execution fails

More complex loops: easy to limit the number of instructions

Evaluation: Sandboxing - OOB memory access

Unikernel console:

```
body of function myfun
uBPF error: out of bounds memory
load at PC 4, addr 0x0, size 4
mem 0x0/zd stack 0x0/520223744
body of function myfun
body of function myfun
```

execution of the main application continues

Attached shell:

```
> bpf_exec oob.bin
BPF program execution failed.
```

oob.c: (compiled into oob.bin)

```
int bpf_prog(void *arg)
{
    int *p = 0;
    return *p;
}
```

Evaluation: Runtime of BPF programs

Attached BPF program	Median runtime (ns)
no attached program	19
`noop.bin`	119
`count.bin`	619
`get_count.bin`	7798
`notify.bin`	2892427

~100ns to find attached programs and execute a program in a BPF vm

~500 ns to access BPF maps through helpers

noop: does nothing
count: collects statistics into BPF maps
get_count: searches debug symbols and accesses BPF maps
notify: prints into the console

Summary

Unikernels:

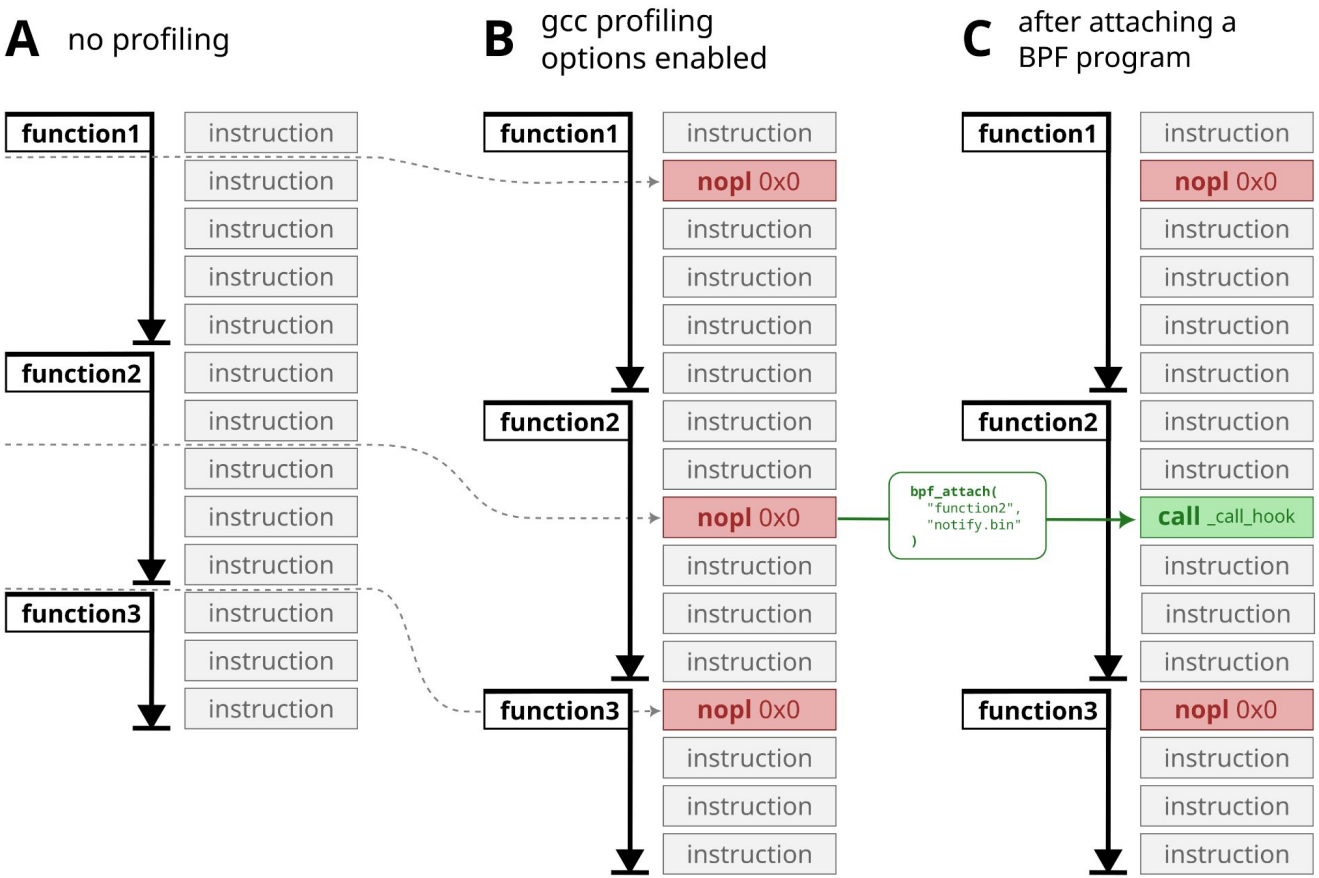
- difficult to inspect and debug
- including external tools bloats the VM image + requires recompilation

Extending unikernels with a BPF language runtime:

- allows loading binaries at runtime and safely executing them
- dynamic tracer simplifies debugging

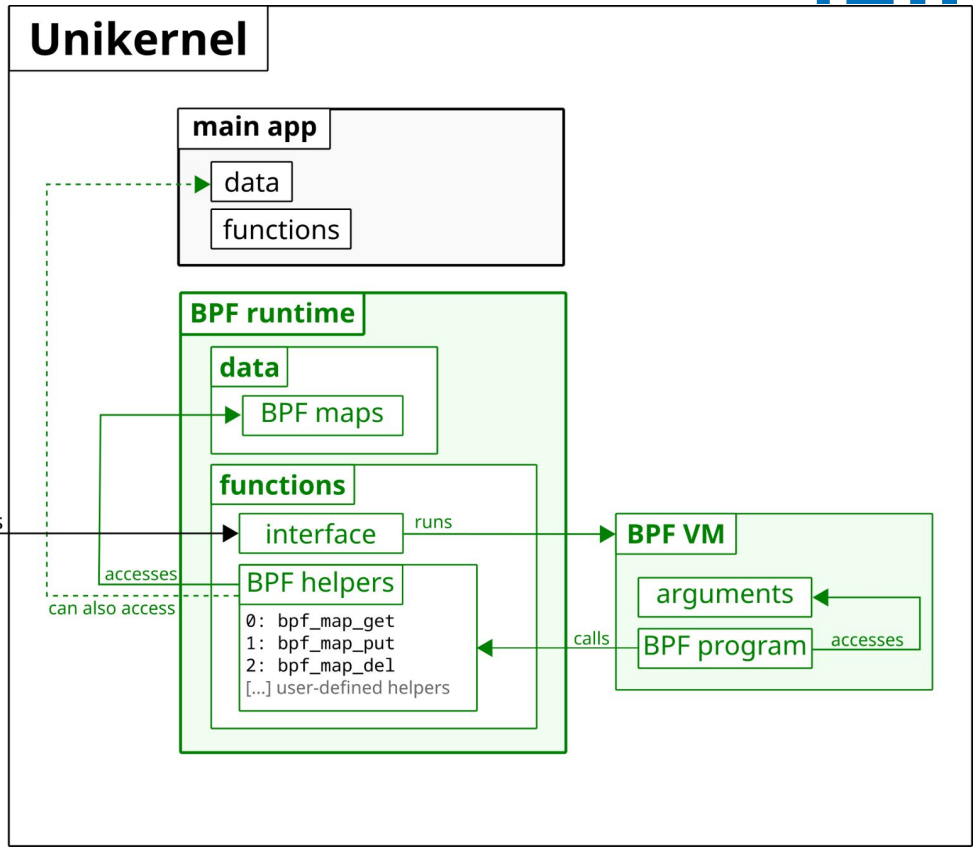
Backup

Showcase application: Dynamic tracer



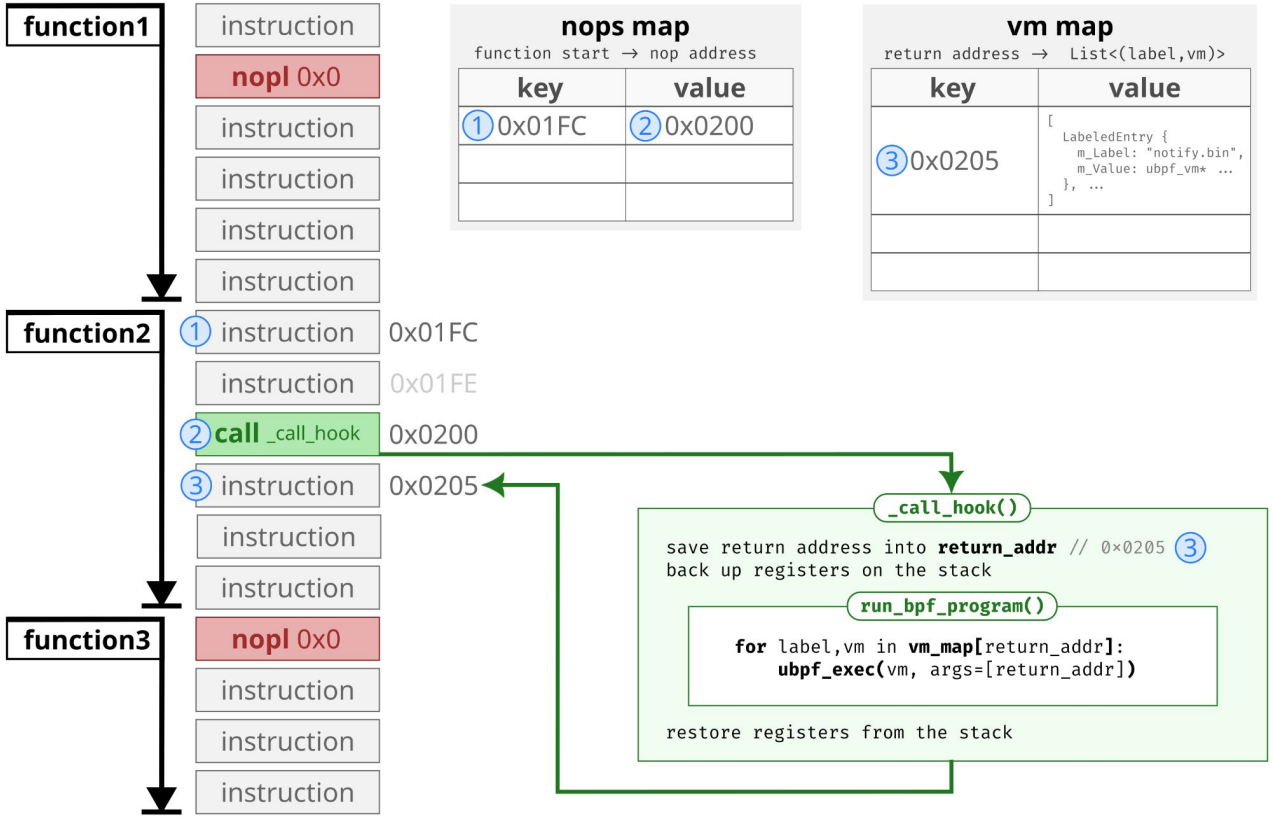
Implementation

- **Unikraft:** unikernel framework
- **ushell:** shell interface ✓
- **ubpf:** BPF runtime with integrated bounds checker isolating the application memory ✓

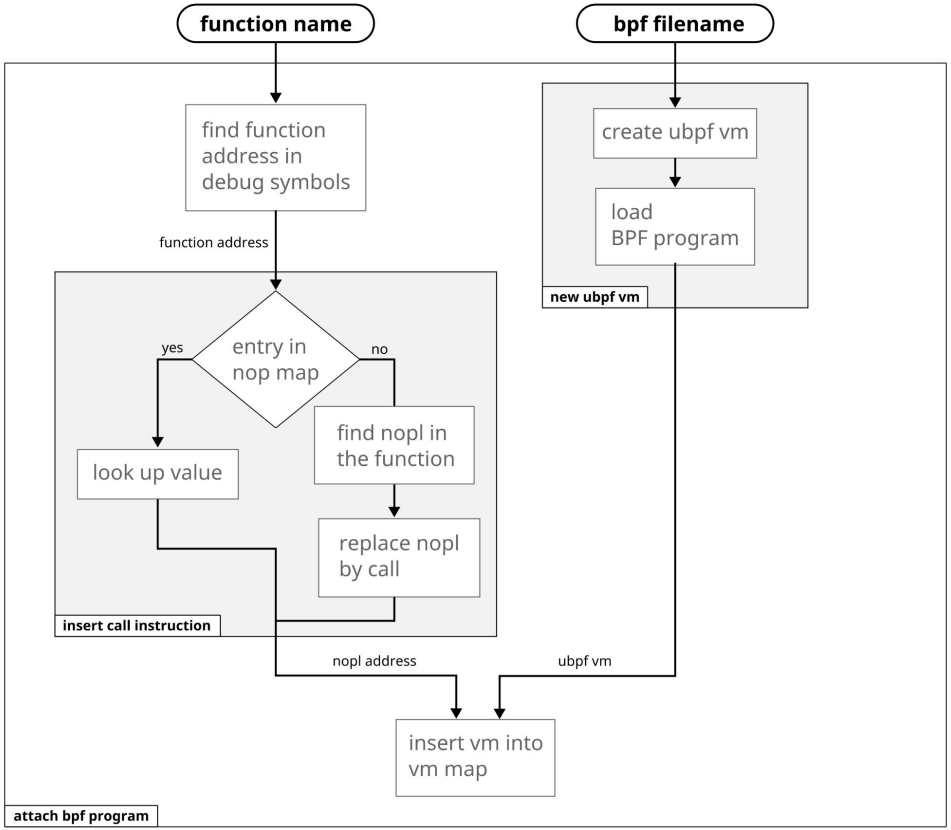


- **BPF maps and BPF helpers:** extend BPF runtime with trusted statically included functions and storage ✓

Showcase application: Dynamic tracer



Showcase application: Dynamic tracer



The BPF runtime and dynamic tracer are easy to use

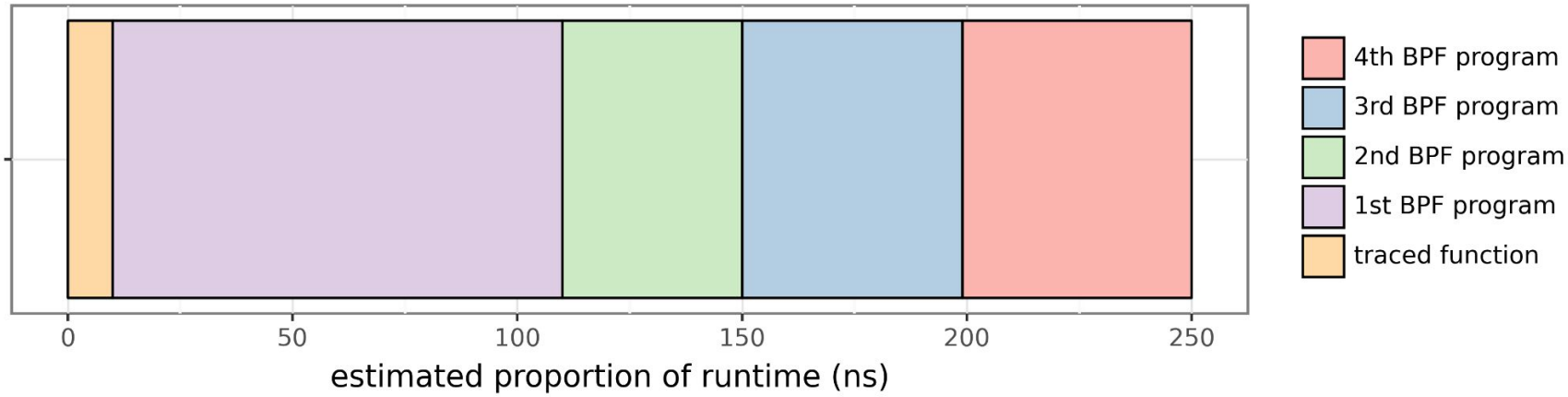
Unikernel console:

```
body of function myfun
body of function myfun
Notify: myfun
body of function myfun
Notify: myfun
body of function myfun
body of function myfun
body of function myfun
```

Attached shell:

```
> bpf_attach myfun count.bin
Program was attached successfully.
> bpf_attach myfun notify.bin
Program was attached successfully.
> bpf_list myfun
myfun:
 - count.bin
 - notify.bin
> bpf_detach myfun notify.bin
Program was detached successfully.
> bpf_exec get_count.bin myfun
The program returned: 4
```

Evaluation: Attaching multiple BPF programs to one function



Showcase application: Dynamic tracer

```
#include <stdint.h>

// bpf map helpers
#define bpf_map_get ((uint64_t(*)(uint64_t key1, uint64_t key2))0)
#define bpf_map_put ((void (*)(uint64_t key1, uint64_t key2,\
                                                uint64_t value))1)
#define bpf_map_del ((void (*)(uint64_t key1, uint64_t key2))2)

// tracer helpers
#define bpf_notify ((void (*)(uint64_t function_address))3)
#define bpf_get_ret_addr ((uint64_t(*)(const char *function_name))4)
```

The BPF runtime and dynamic tracer are easy to use

```
#include "config.h"

int bpf_prog(void *arg)
{
    struct UbpfTracerCtx *ctx = arg;
    bpf_notify(ctx->traced_function_address);

    return 0;
}
```

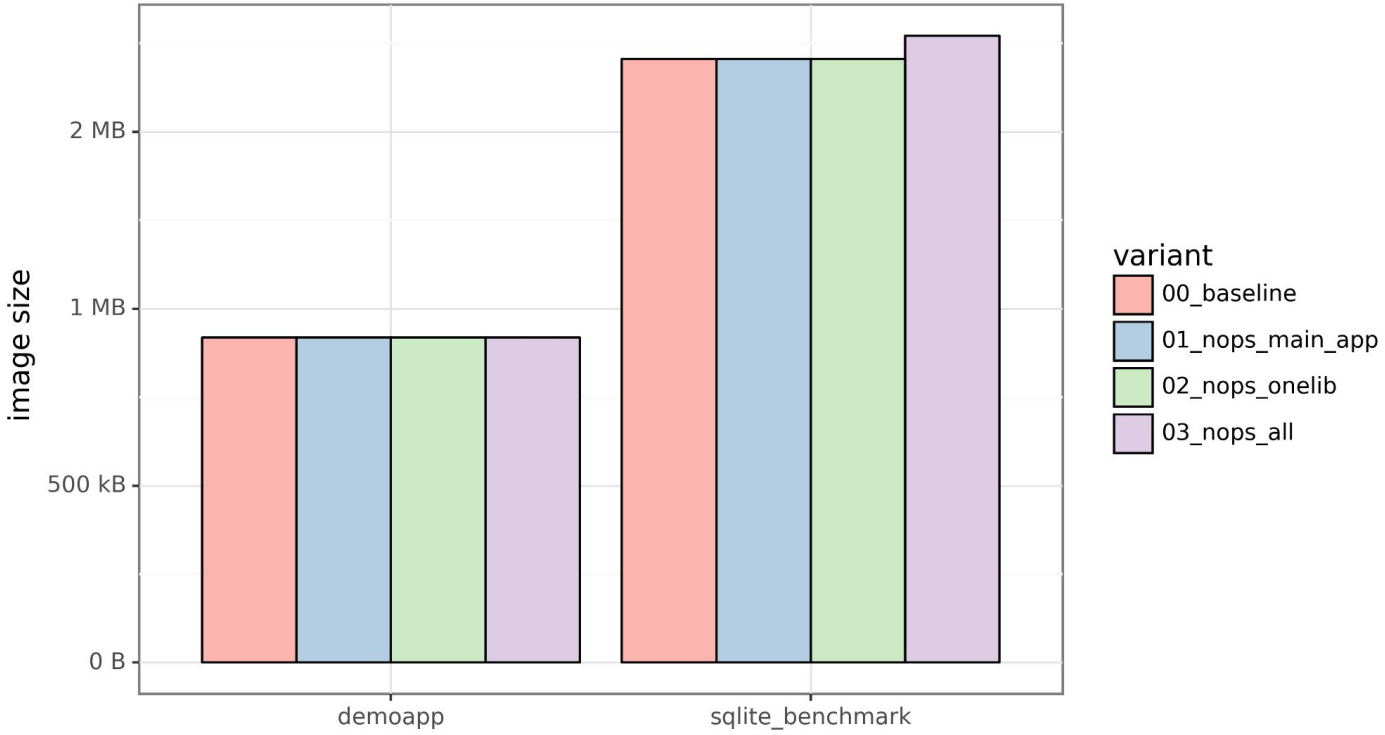
Showcase application: Dynamic tracer

```c
#include "bpf_helpers.h"
#include "config.h"

int bpf_prog(void *arg) {
    if (!arg) {
        return -1;
    }
    uint64_t addr = bpf_get_ret_addr((const char *)arg);
    if (addr == 0) {
        return -1;
    }
    uint64_t count = bpf_map_get(addr, COUNT_KEY);
    if (count == UINT64_MAX) {
        count = 0;
    }

    return count;
}
```

Evaluation: Impact of including nops on image size



Evaluation: Impact of including nops on performance