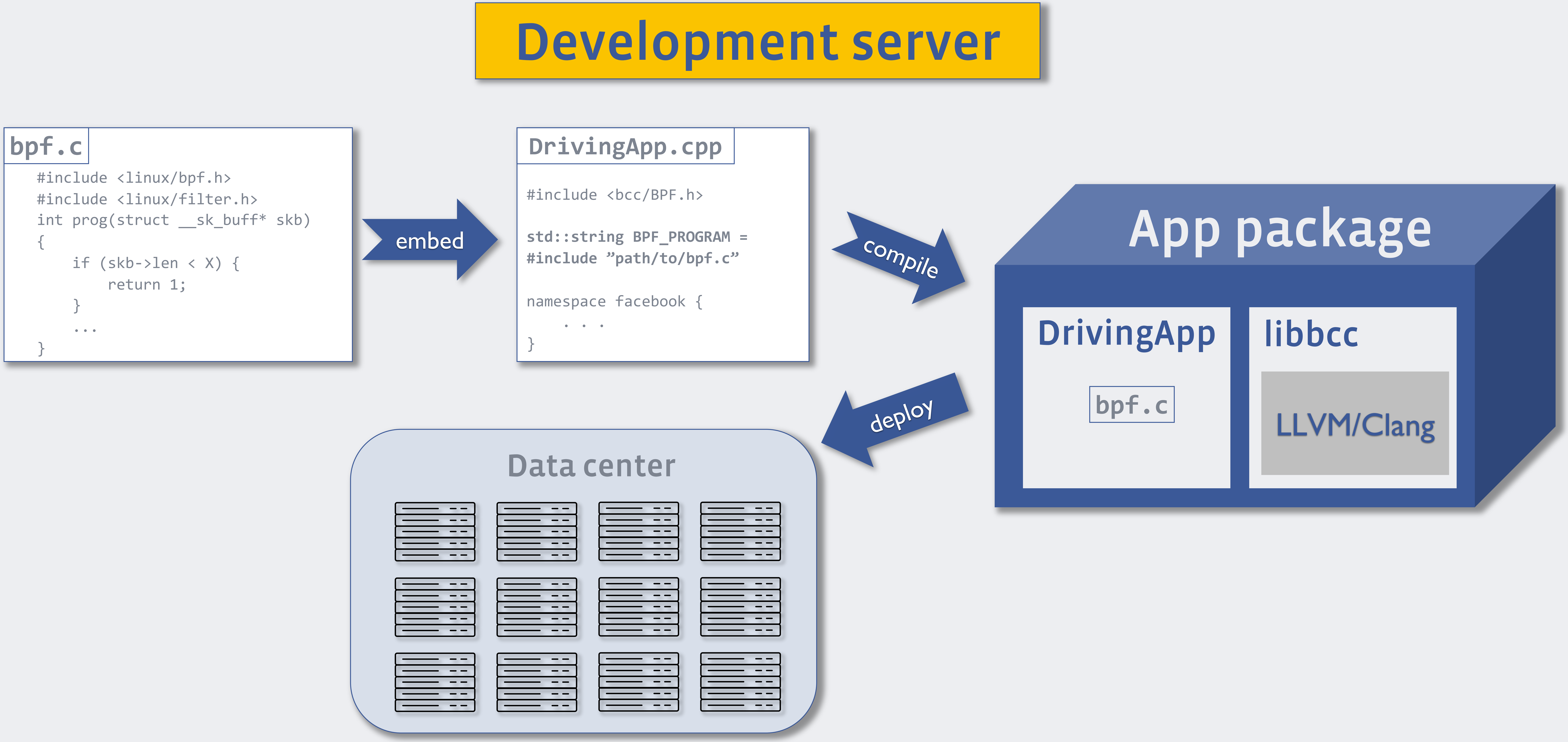
facebook

BPF CO-RE

(Compile Once – Run Everywhere)

Andrii Nakryiko

Developing BPF application (today)



```
#include <linux/bpf.h>
#include <linux/filter.h>
int prog(struct __sk_buff* skb)
{
    if (skb->len < X) {
        return 1;
    }
    ...
}
```

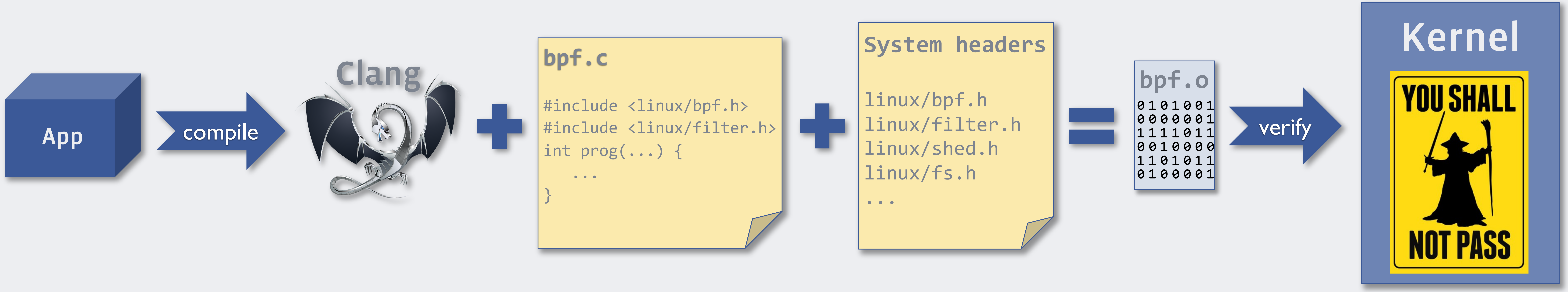
```
#include <bcc/BPF.h>

std::string BPF_PROGRAM =
#include ”path/to/bpf.c”

namespace facebook {
    . . .
}
```

Developing BPF application (today)

Production server



bpf.c

```
#include <linux/bpf.h>
#include <linux/filter.h>
int prog(...) {
   ...
}
```

System headers

```
linux/bpf.h
linux/filter.h
linux/shed.h
linux/fs.h
...
```

bpf.o

```
0101001
0000001
1111011
0010000
1101011
0100001
```

Developing BPF application (today)

Problem:

“On the fly” compilation

“On the fly” BPF compilation

Why?

- Accessing kernel structs (e.g., task_struct or sk_buff)
- Memory layout **changes** between versions/configurations
- BPF code needs to be compiled w/ **fixed** offsets/sizes

“On the fly” BPF compilation

Problems

1. Every prod machine **needs kernel headers**
2. LLVM/Clang is **big and heavy**
3. **Testing is a pain**

“On the fly” BPF compilation

Problems

Every prod machine **needs kernel headers**

- `kernel-devel` package required
- `kernel-devel` is missing internal headers
- custom one-off kernels are a pain
- `kernel-devel` can get out of sync

“On the fly” BPF compilation

Problems

LLVM/Clang is **big and heavy**

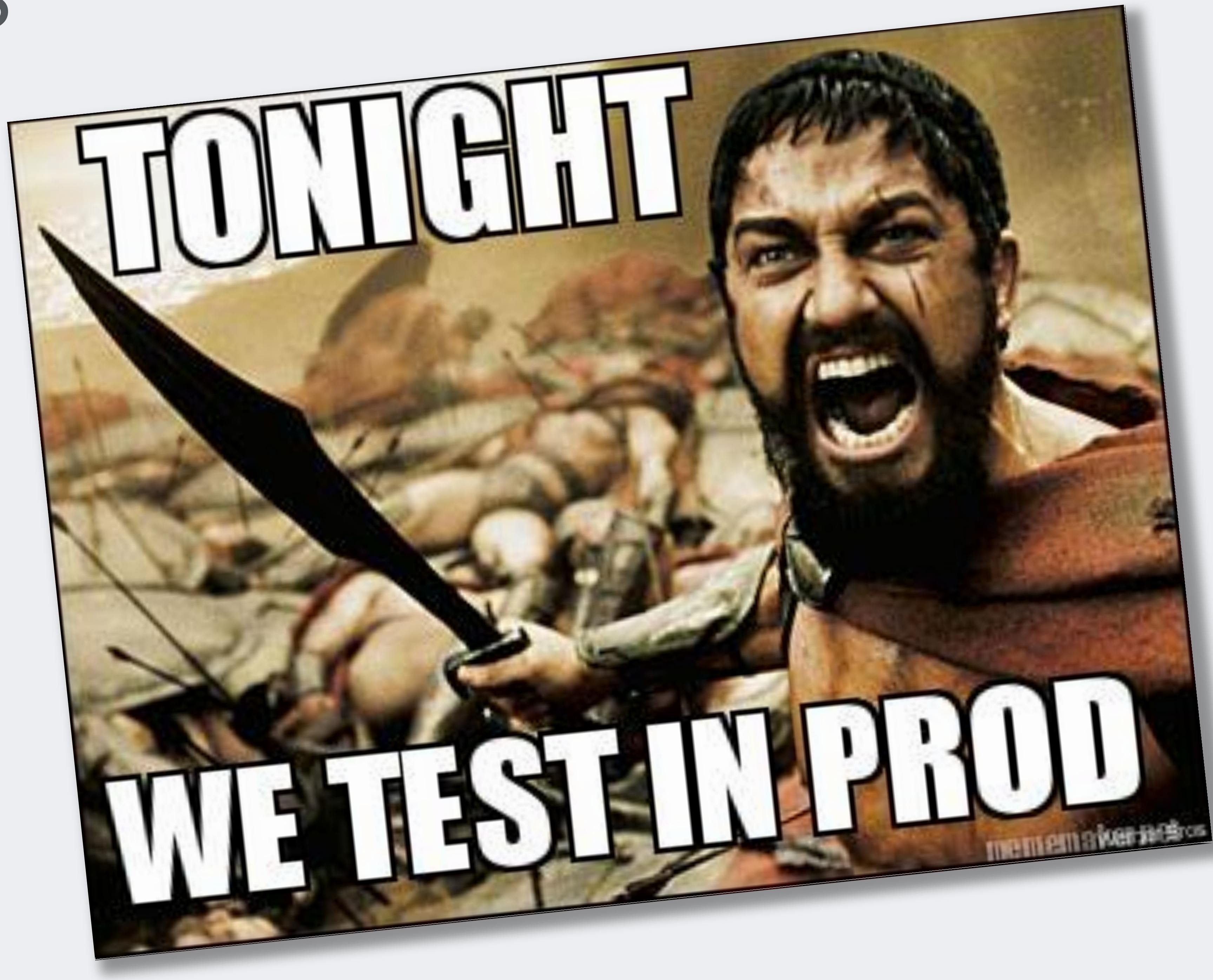
- libbcc.so > **100MB**
- compilation is a heavy-weight process
    - can use lots of memory and CPU
    - on busy machine can tip over prod workload

“On the fly” BPF compilation

Problems

Testing is a pain

- variety of kernel versions/configurations
- “works on my machine” means nothing
- Problem is detected only at run time



Can we compile once?

Then run same binary everywhere?

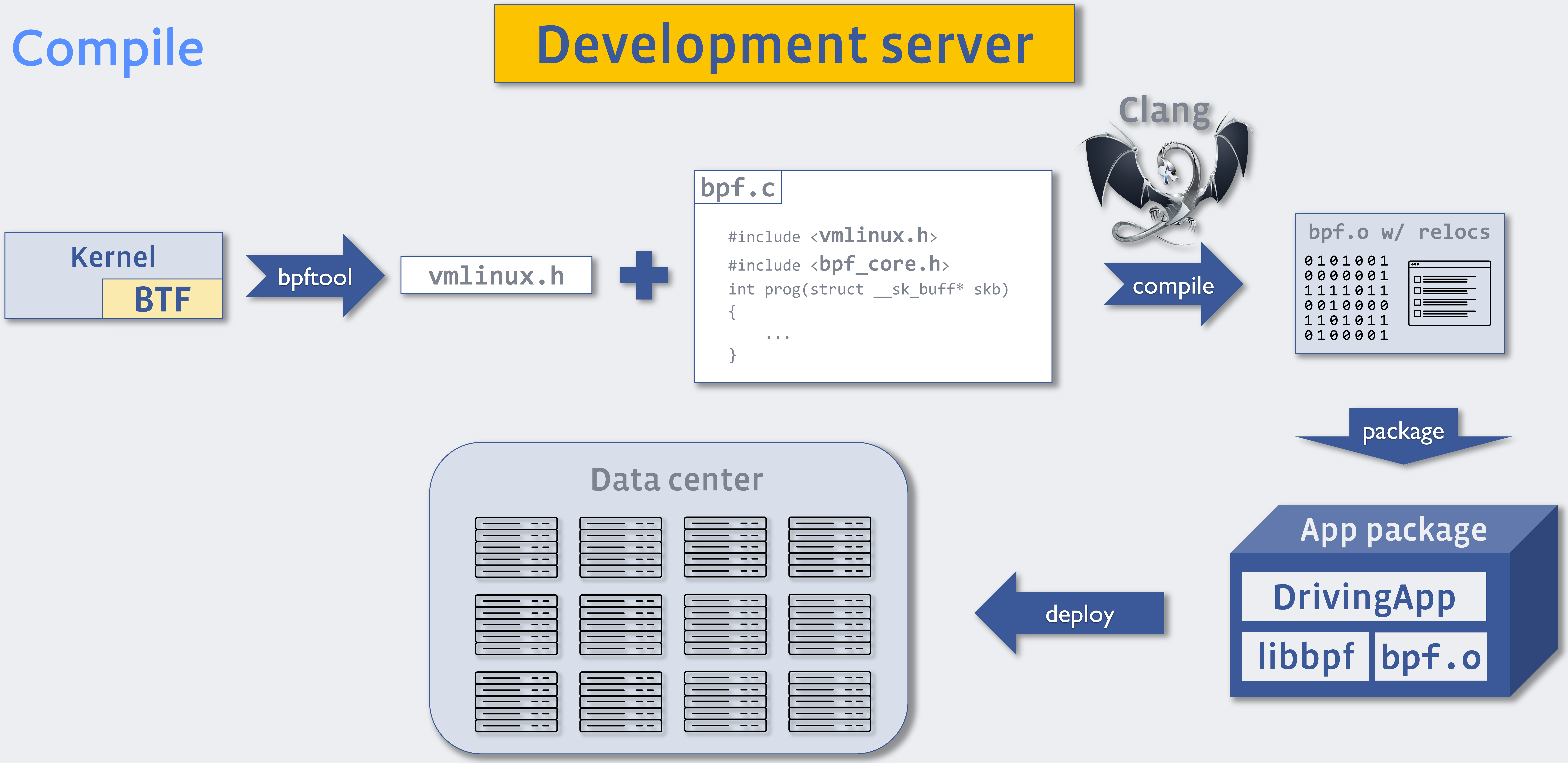
BPF CO-RE
(Compile Once – Run Everywhere)

Goals

- No kernel headers
- No “on the fly” compilation
- Upfront validation against prod kernels

BPF CO-RE flow

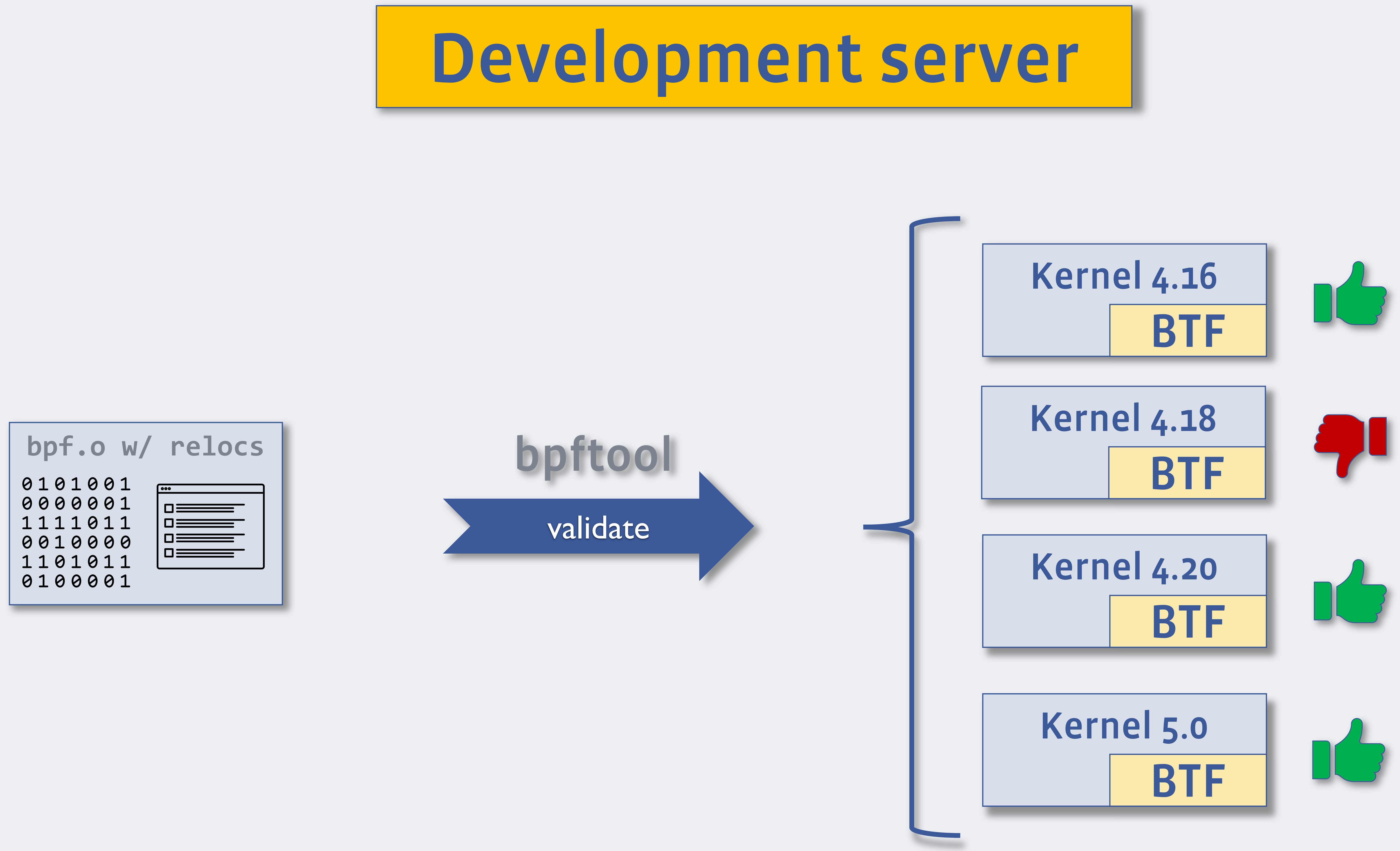
Compile



```
#include <vmlinux.h>
#include <bpf_core.h>
int prog(struct __sk_buff* skb)
{
    ...
}
```

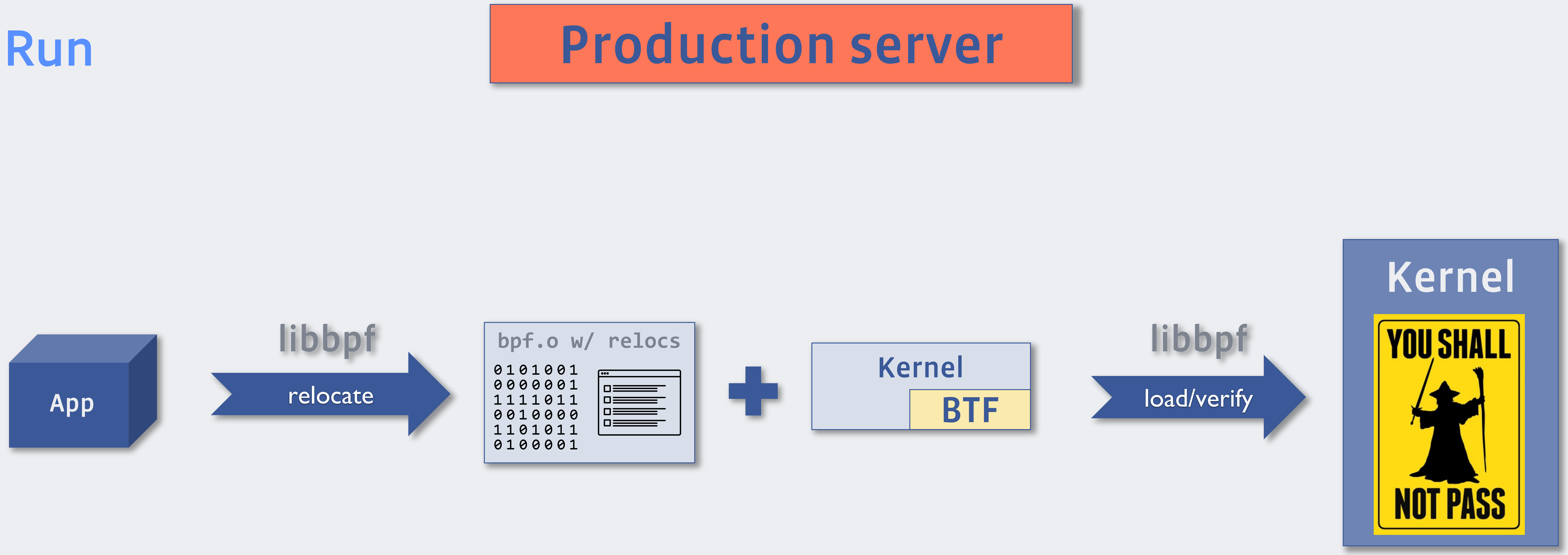
BPF CO-RE flow

Test



BPF CO-RE flow

Run



BPF CO-RE

Overview

- Self-describing kernel (BTF)
- Clang w/ emitted relocations
- Libbpf as relocating loader
- Tooling for testing

BPF CO-RE

Self-describing kernel

- *Deduplicated* BTF information
  - **compact** (no need to strip it out: 2MB vs 177MB of DWARF)
  - describes **all kernel types** (size, layout, etc)
  - always **in sync w/ kernel**
  - lossless BTF to **C conversion**
- Available today:
  - CONFIG_DEBUG_INFO_BTF=y (needs pahole >= v1.13)

BPF CO-RE Challenges

- **Struct layout changes**
- Version- / config-specific fields (logic in general)
- #define macros
- Unrelocatable sizeof()

Field offset relocation

```
#include <linux/sched.h>
#include <linux/types.h>



int on_event(void* ctx) {
    struct task_struct *task;
    u64 read_bytes;
    task = (void *)bpf_get_current_task();

      bpf_probe_read(
          &read_bytes,
          sizeof(u64),
          &task->ioac.read_bytes);

    return 0;
}
```

```
0: (85) call bpf_get_current_task
1: (07) r0 += 1952
2: (bf) r1 = r10
3: (07) r1 += -8
4: (b7) r2 = 8
5: (bf) r3 = r0
6: (85) call bpf_probe_read
7: (b7) r0 = 0
8: (95) exit
```

```
Field reloc:
  - insn: #1
  - type: struct task_struct
  - accessor: 30:4
```

BPF CO-RE Challenges

- Struct layout changes
- **Kernel version- / config-specific logic**
- #define macros
- Unrelocatable sizeof()

Extern relocation

```
#include <linux/sched.h>
#include <linux/types.h>
/* relies on /proc/config.gz */
extern bool CONFIG_IO_TASK_ACCOUNTING;

int on_event(void* ctx) {
    struct task_struct *task;
    u64 read_bytes;
    task = (void *)bpf_get_current_task();
    if (CONFIG_IO_TASK_ACCOUNTING) {
        return bpf_probe_read(
            &read_bytes,
            sizeof(u64),
            &task->ioac.read_bytes);
    }
    return 0;
}
```

```
 0: (85) call bpf_get_current_task
 1: (b7) r1 = XXX
 2: (15) if r1 == 0x0 goto pc+6
 3: (07) r0 += 1952
 4: (bf) r1 = r10
 5: (07) r1 += -8
 6: (b7) r2 = 8
 7: (bf) r3 = r0
 8: (85) call bpf_probe_read
 9: (b7) r0 = 0
10: (95) exit

Extern reloc:
  - insn: #1
  - name: CONFIG_TASK_IO_ACCOUNTING
  - type: bool

Field reloc:
  - insn: #3
  - type: struct task_struct
  - accessor: 30:4
```

Uncommon/experimental fields

```
struct task_struct___custom {
    u64 experimental;
};

int on_event(void* ctx) {
    struct task_struct *task, *exp_task;
    u64 value = 0;
    task = (void *)bpf_get_current_task();

    exp_task = (struct task_struct___custom *)task;
    bpf_probe_read(&value, sizeof(u64), &exp_task->experimental);

    return 0;
}
```

BPF CO-RE Challenges

- Struct layout changes
- Kernel version- / config-specific logic
- **#define macros**
- Unrelocatable sizeof()

#define macros

- Constants, flags, etc...
- DWARF doesn’t record #defines, so doesn’t BTF
- Copy/paste whatever you need?
- bpf_core.h can provide commonly-needed stuff

BPF CO-RE Challenges

- Struct layout changes
- Kernel version- / config-specific logic
- #define macros
- **Unrelocatable sizeof()**

Unrelocatable sizeof()

```
struct task_struct *task;
struct task_io_accounting io_acc;

task = (void *)bpf_get_current_task();

bpf_probe_read(&io_add, sizeof(struct task_io_accounting), &task->ioac);

// accessing fields on the stack is faster than
// bpf_probe_read()'ing them individually
io_acc.io_read_bytes;
io_acc.io_write_bytes;
io_acc.rchar;
io_acc.wchar;
```

Not relocatable

Unrelocatable sizeof()

```
struct task_struct *task;
struct task_io_accounting io_acc;

task = (void *)bpf_get_current_task();

io_acc = __builtin_bpf_read_field(&task, ioac);
```

Abstracts bitfield access?..

Maybe relocatable?

Questions?

facebook