CS202 – Advanced Operating Systems

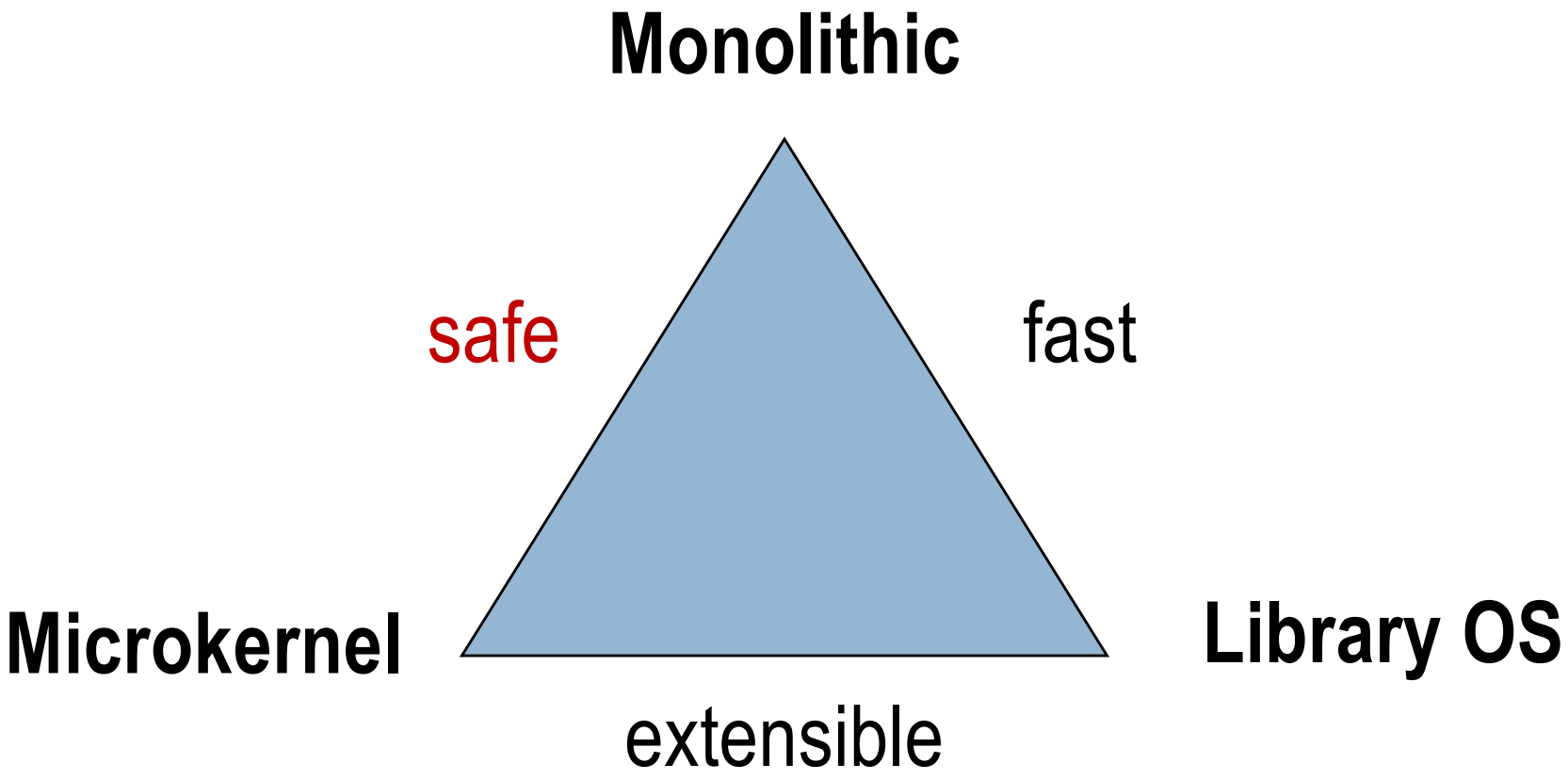
Isolation within Monolithic Kernels

January 29, 2025

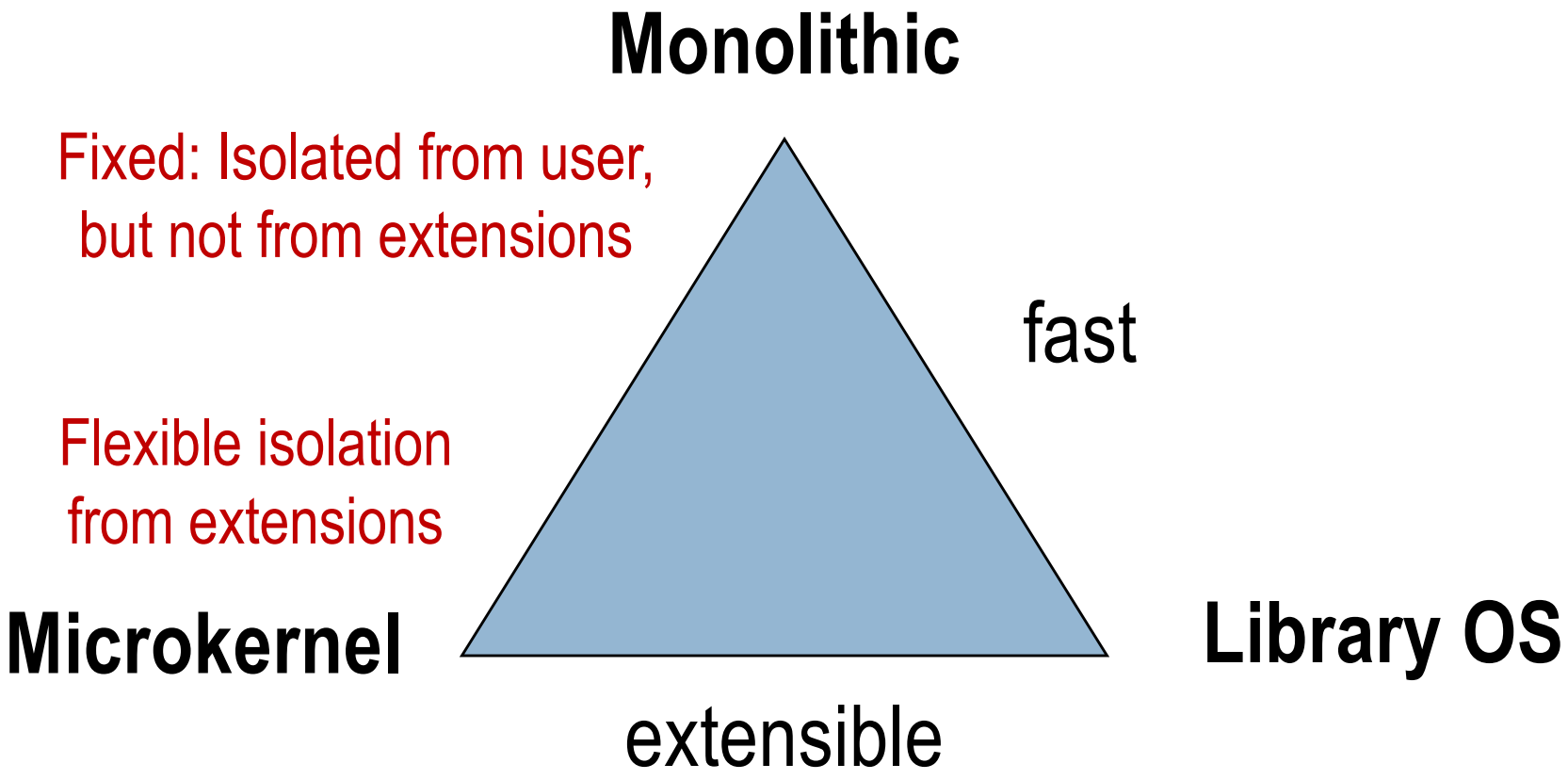
Check your understanding

- True or False: a non-preemptive CPU scheduler be invoked on every mode switch (i.e., trap or interrupt)
  - No, cannot preempt a running process until it gives up the CPU (I/O)
- True or False: we should schedule CPU-bound processes by giving them a higher priority because they will use the CPU
  - No, we typically want to bias higher priorities toward I/O bound processes since they will be more responsive and get out of the way
- How is scheduling related information stored?
  - In queues of Process Control Blocks for each state (running, ready)

OS Structures and Their Impacts



OS Structures and Their Impacts



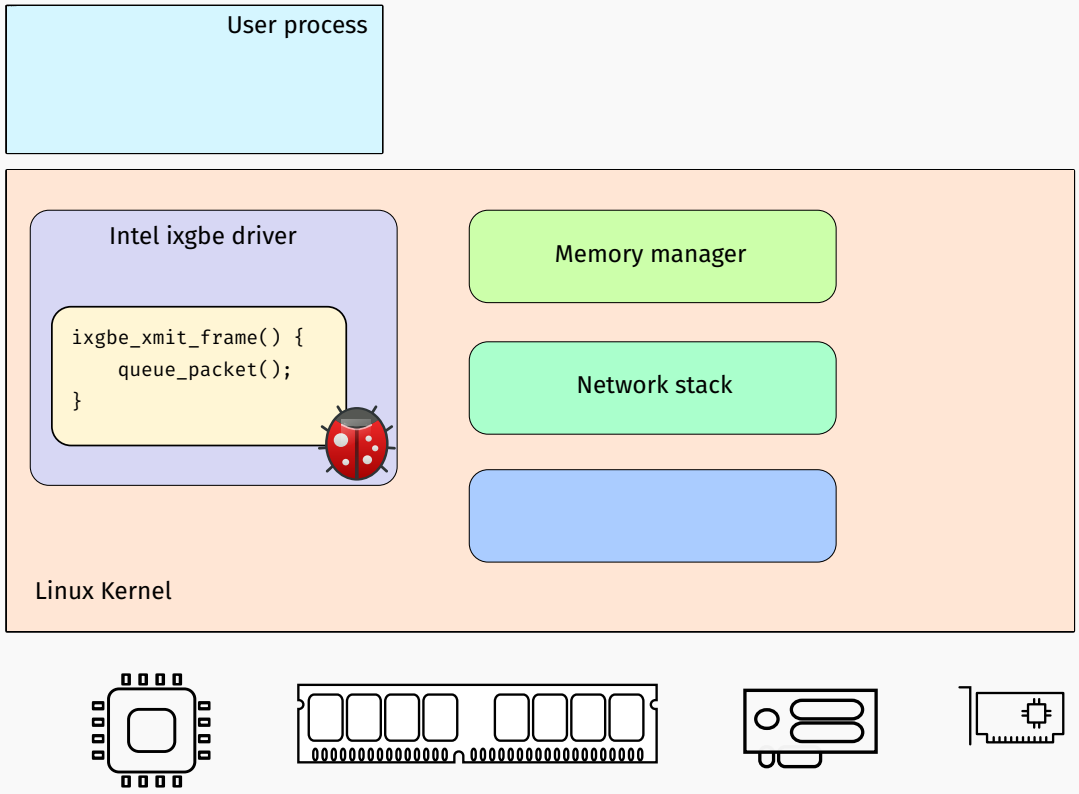
Kernel Modules in Monolithic

- Code extensions for monolithic kernels
  - A piece of code that can be dynamically loaded and unloaded into the running kernel to extend its functionality without requiring a system reboot
  - Commonly used to support the addition of hardware devices on demand
- Extends the code running in kernel space

Device Drivers

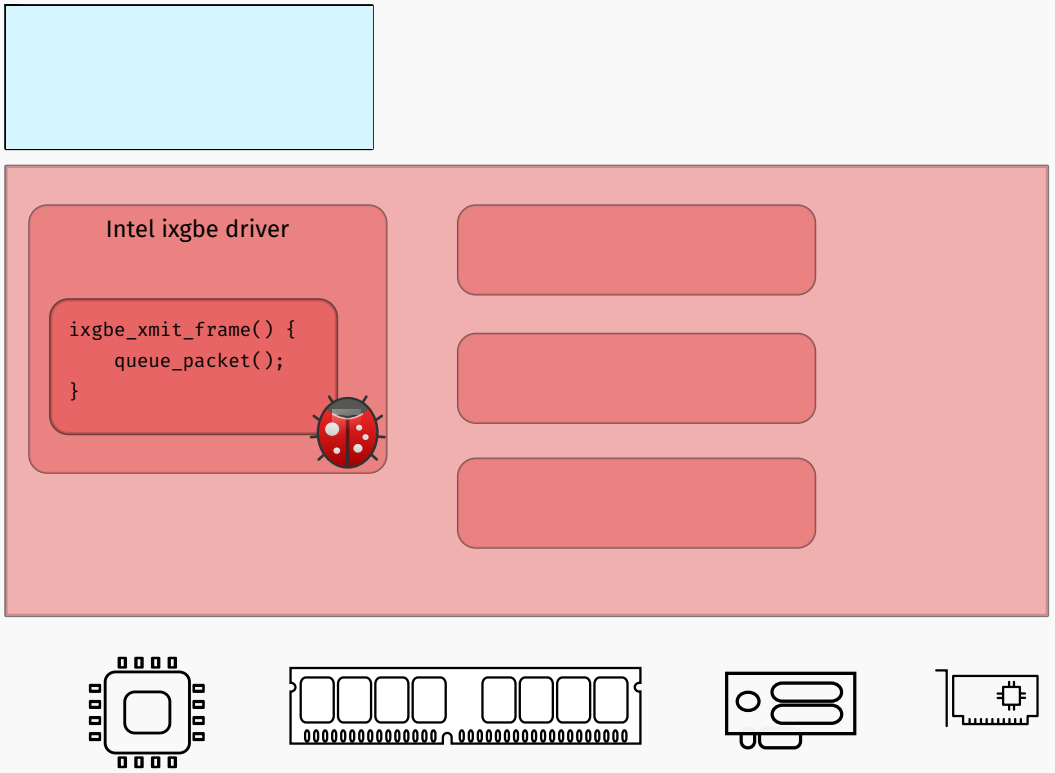
- Each type of peripheral device has a driver
  - Network cards
  - Storage devices
  - USB devices
  - GPS and gyroscope (mobile devices)
  - Etc.
- Custom code written by device vendor typically

Device Drivers Can Be Buggy



- Kernel extensions (fs, network stacks, drivers) run in the same address space
- **Vulnerability in a single component propagates to the entire kernel**

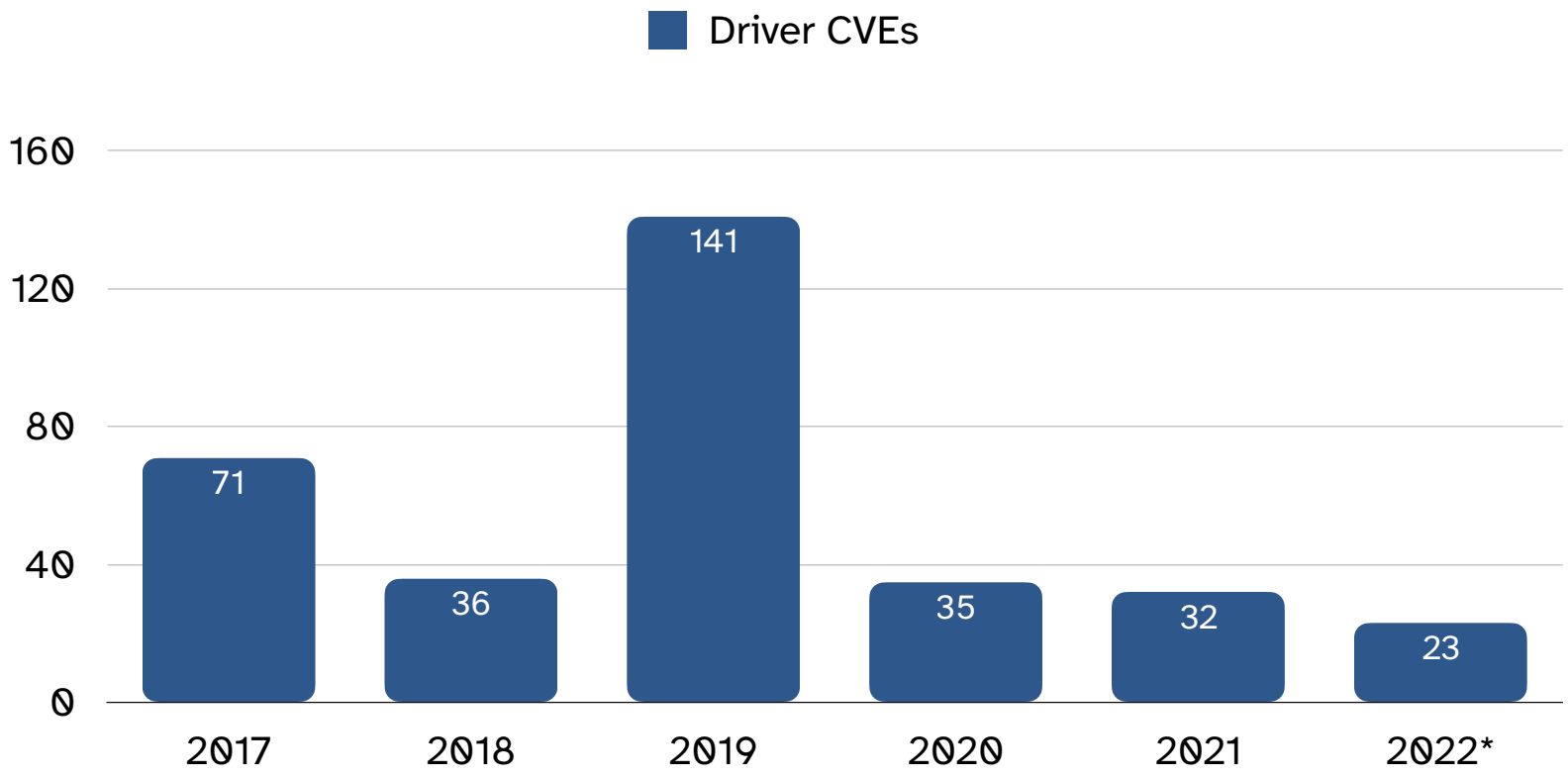
Which Can Impact the Entire Kernel



- Kernel extensions (fs, network stacks, drivers) run in the same address space
- **Vulnerability in a single component propagates to the entire kernel**

Device Driver Vulnerbilities

- 16-50 % of all Linux kernel CVEs

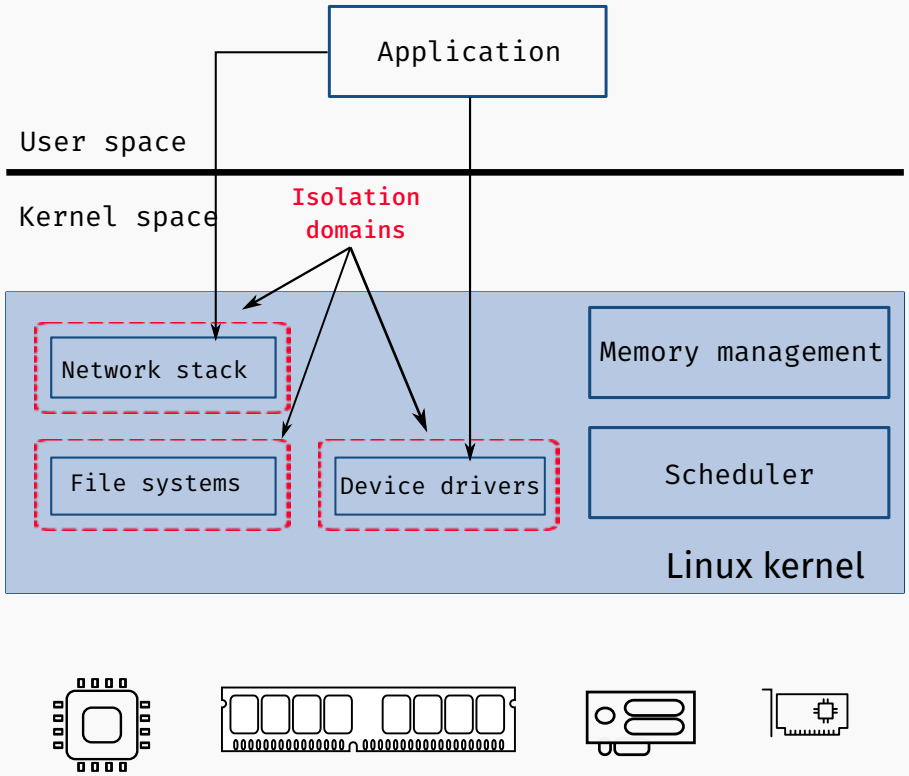


Isolation Approaches

- Three distinct approaches to isolating code within monolithic operating systems
- Hardware-supported Isolation (e.g., LXDs)
  - Configure hardware to restrict memory access
- Software Fault Isolation (SFI)
  - Add code to limit the memory accessible to confined instructions
- Verification (e.g., eBPF)
  - Verify that an extension will only access expected memory region

HARDWARE-SUPPORTED ISOLATION (LXDS)

Isolate Drivers/Etc from the Kernel



Split monolithic kernel into isolated components

- to confine faults
- to improve reliability

LXDs Approach

- Goal: Ensure that all memory references within an isolated component are limited to that component's memory region
- What is an easy way (conceptually at least) to do that?

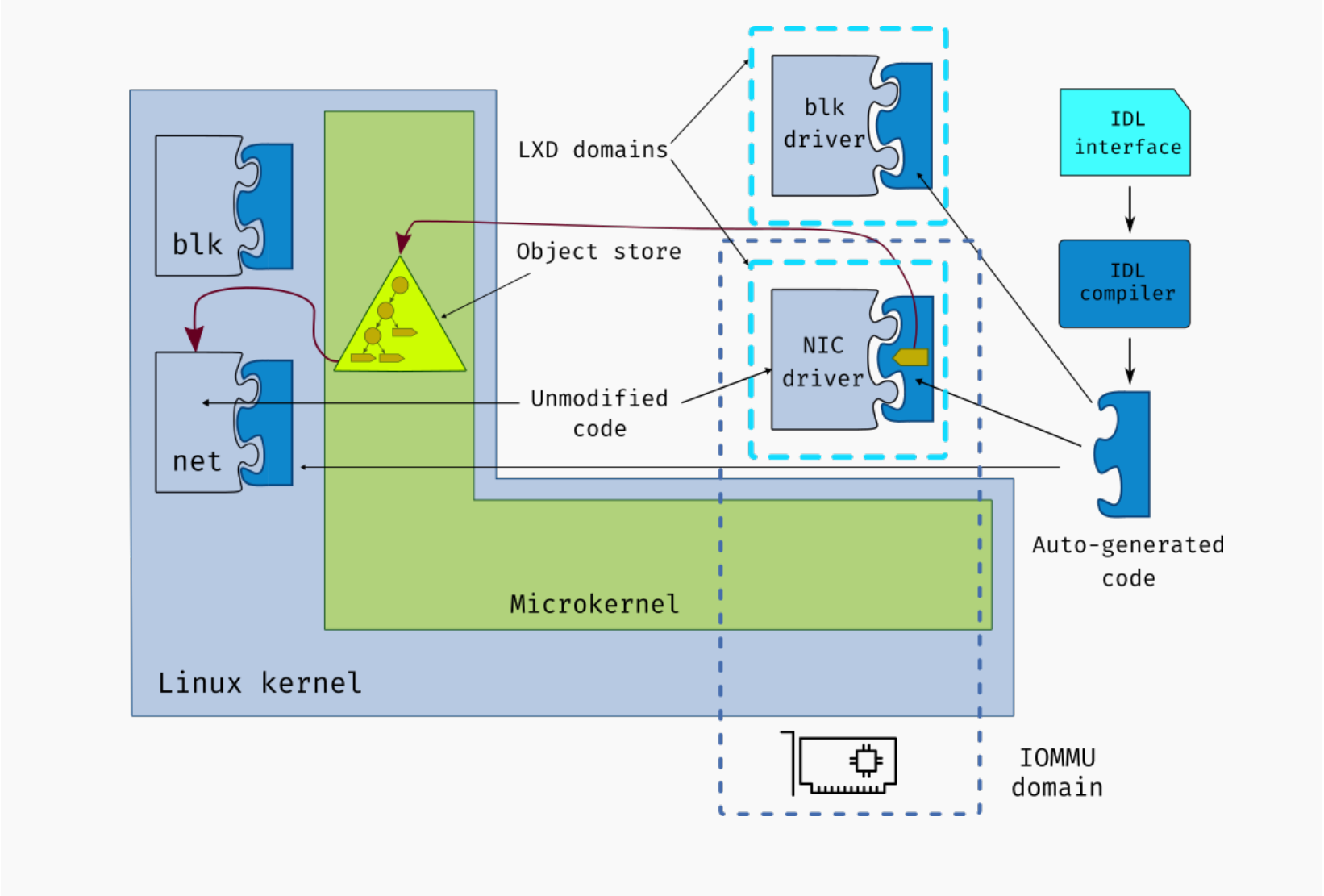
LXDs Approach

- Goal: Ensure that all memory references within an isolated component are limited to that component's memory region
- What is an easy way (conceptually at least) to do that?
  - Run the isolated components as separate processes
- Issues with doing that?

LXDs Approach

- Goal: Ensure that all memory references within an isolated component are limited to that component's memory region
- What is an easy way (conceptually at least) to do that?
  - Run the isolated components as separate processes
  - Without modifying the code of the isolated components
  - And run in kernel mode

Lightweight Execution Domains (LXDs)



LXDs Approach

- Monolithic: function call from kernel to driver
- Isolated: remote procedure call from kernel domain to driver domain
  - What needs to be done to transform function calls to calls between separate protection domains?

LXDs Approach

- Monolithic: function call from kernel to driver
- Isolated: remote procedure call from kernel domain to driver domain
  - What needs to be done to transform function calls to calls between separate protection domains?
  - Copying
    - Need to copy arguments from the caller to the callee
    - Need to copy the return values back
    - And copy any changes to the arguments made in the callee

LXDs Example

```
int register_netdev(struct net_device *dev);
```

```
/* Projections */
projection <struct net_device> net_device {
  ...
  /* [modifier] <data_type> <struct_member_name> */;
  [in] unsigned int flags;
  [in] unsigned long long hw_features;
  [in] unsigned long long features;
  ...
  projection net_device_ops [alloc(caller)] *netdev_ops;
};
```

To implement a kernel call to the driver's `register_netdev` function, we need to copy the structure `net_device` to the driver

LXDs Approach

- OS maxim: unnecessary copying is bad
  - How to reduce the amount of copying?

LXDs Approach

- OS maxim: unnecessary copying is bad
  - How to reduce the amount of copying?
    - Only copy the minimal necessary

LXDs Minimal Copying

```
int register_netdev(struct net_device *dev);

/* Projections */
projection <struct net_device> net_device {
  ...
  /* [modifier] <data_type> <struct_member_name> */;
  [in] unsigned int flags;
  [in] unsigned long long hw_features;
  [in] unsigned long long features;
  ...
  projection net_device_ops [alloc(caller)] *netdev_ops;
};
```

Only copy fields in `net_device` from the kernel that are actually read by the driver in performing `register_netdev` (the [in] case)

LXDs Projections

```
int register_netdev(struct net_device *dev);

/* Projections */
projection <struct net_device> net_device {
  ...
  /* [modifier] <data_type> <struct_member_name> */;
  [in] unsigned int flags;
  [in] unsigned long long hw_features;
  [in] unsigned long long features;
  ...
  projection net_device_ops [alloc(caller)] *netdev_ops;
};
```

The copying requirements are specified in "projections". Originally, produced manually, but later generated using automated kernel analysis (Huang et al, OSDI 2022)

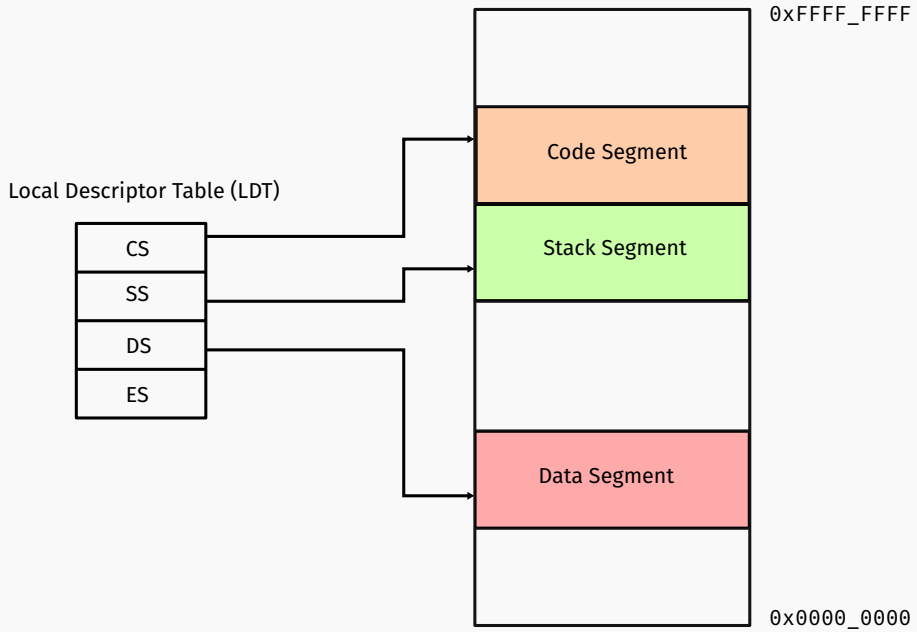
LXDs Isolation

- Goal: Create multiple protection domains within the kernel that can be switched among efficiently
  - What is the main task in switching between the kernel and an isolated driver and back?

LXDs Isolation

- Goal: Create multiple protection domains within the kernel that can be switched among efficiently
  - What is the main task in switching between the kernel and an isolated driver and back?
    - Changing memory permissions
    - How can we do that efficiently?

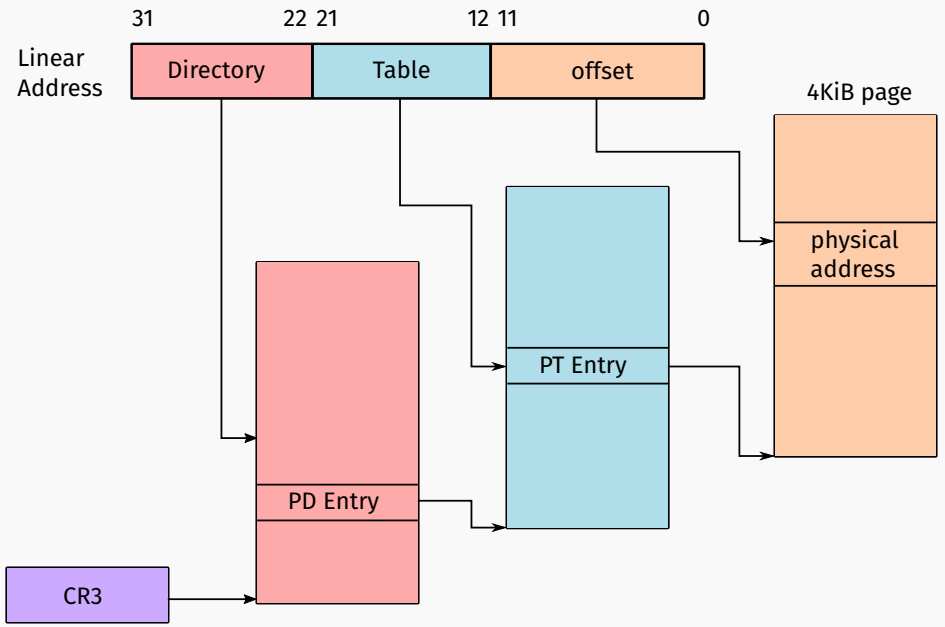
Old Solution: Segmentation (used by L4)



- Call-reply invocation (46 cycles)[4]
- Deprecated in x64

[4] Liedtke, J. (1995). Improved address-space switching on Pentium processors by transparently multiplexing user address spaces

Base Solution: Page Table Switching (seL4)



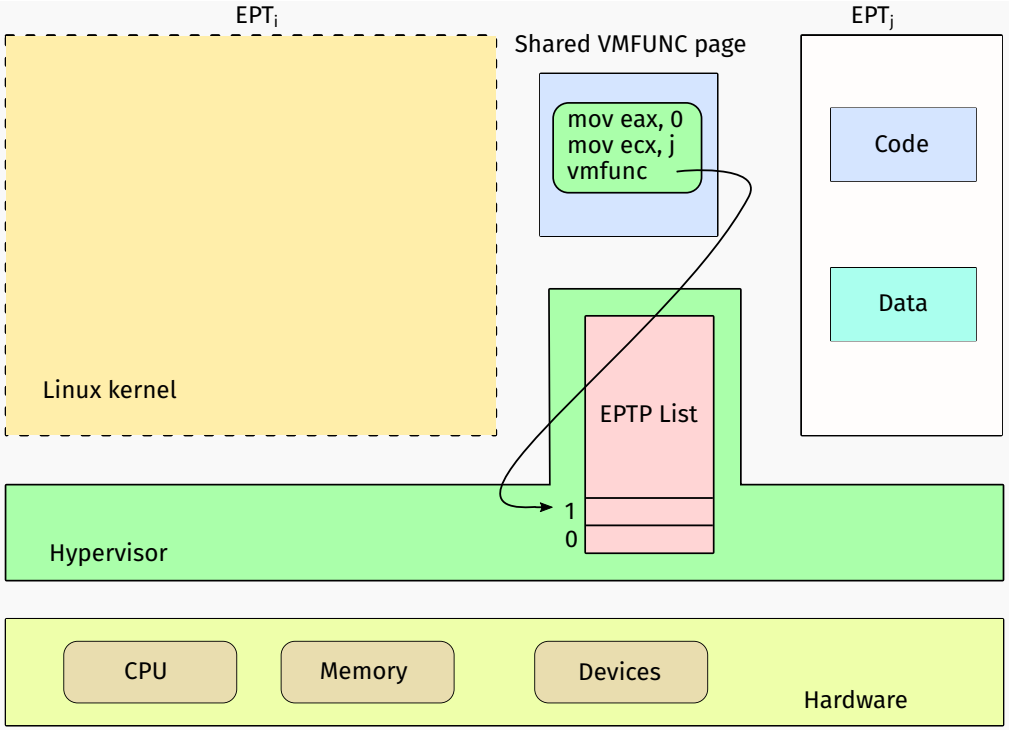
- Page-table switching (952 cycles)[5]

[5] https://sel4.systems/About/Performance/

New Hardware

- Recent hardware features like extended page tables and memory protection keys provide more efficient ways to changing memory configurations
  - LXDs use extended page table switching
  - Idea: have a set of memory configurations (page tables) and can switch among them without any other tasks

New Solution: Extended Page Table Switching



- Overhead (109 - 147 cycles)
  - Null syscall (150 cycles)
- Requires hypervisor

Ensure EPT Switching Is Secure

- See optional paper
  - Narayanan et al. Lightweight Kernel Isolation. VEE ’20 (Best Paper Award)
- Warning: low-level

SOFTWARE FAULT ISOLATION

Software Fault Isolation

- What is meant by this term?
- Fault: a corrupted pointer (memory reference) is used to access unauthorized memory
  - Also, happens to be a fault in the software
- Software Isolation: However, SFI is named for the fact that the isolation is implemented by changing the software
  - Could be called software-based fault isolation

Software Fault Isolation: Goal

- Alternative to hardware-based fault isolation.
  - Traditional context switches are slow
  - Some code we want to isolate is small
  - Or run infrequently
- Trade-off:
  - “substantially faster communication between fault domains, at a cost of slightly increased execution time for distrusted modules.”

SFI – Memory Isolation

- "substantially faster communication between fault domains, at a cost of slightly increased execution time for distrusted modules."
- What's the cost?
  - Ensure that any memory access by a module being isolated is limited to a prescribed memory region
  - Cost: Requires runtime checks

SFI – Memory Isolation

- Ensure that any memory access by a module being isolated can only access memory within a prescribed memory region
- Two approaches
  - Segment matching: reject every memory access using an address outside the allowed memory segment
  - Address sandboxing: limit every memory access to be within the allowed memory segment

SFI – Segment Matching

- Compare the memory address to be accessed to the expected memory segment (segmentation)
- Intuition (paging):
  - Check that address is within range by checking that the expected address bits are set

```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Figure 1: Assembly pseudo code for segment matching.

SFI – Segment Matching

- Intuition (paging):
  - Check that address is within range by checking that the expected address bits are set
- E.g., suppose that all the pages with bits 14-31 set to 1 are in the SFI region
  - Can check that all those bits are set – how many pages?

```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Figure 1: Assembly pseudo code for segment matching.

SFI – Address Sandboxing

- Limit the address to the expected memory segment
  - Remove any address bits outside the expected region
  - Set the required address bits – may create invalid address, but it will be within the expected region
  - E.g., Set bits 14-31 to 1 before the memory operation
    - Faster than the check

```
dedicated-reg ⇐ target-reg&and-mask-reg
      Use dedicated register and-mask-reg
      to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
      Use dedicated register segment-reg
      to set segment identifier bits.
store instruction uses dedicated-reg
```

Figure 2: Assembly pseudo code to sandbox address in `target-reg`.

SFI – Address Sandboxing

- Limit the address to the expected memory segment
  - Remove any address bits outside the expected region
  - Mask the requested address – may create invalid address, but it will be within the expected region

```
dedicated-reg ⇐ target-reg&and-mask-reg
      Use dedicated register and-mask-reg
      to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
      Use dedicated register segment-reg
      to set segment identifier bits.
store instruction uses dedicated-reg
```

Figure 2: Assembly pseudo code to sandbox address in `target-reg`.

SFI – Cheaper Domain Crossing

- No change in the page tables is necessary
  - SFI grants the isolated code a subset of the memory accessible to the process at large
  - Untrusted memory operations are restricted by software

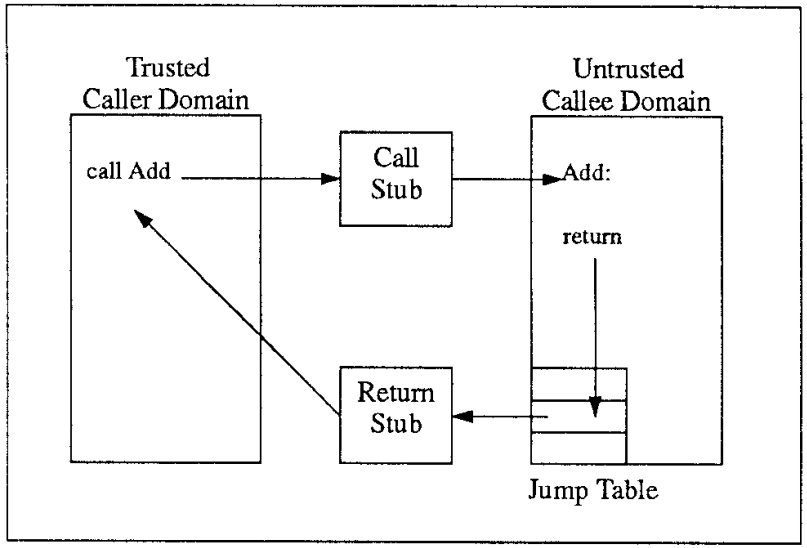


Figure 4: Major components of a cross-fault-domain RPC.

SFI – Limitations

- Performance
  - If a lot of code must be run in the isolated mode, then the overhead can become significant
  - SFI is typically used only to control write operations rather than read+write
- Compatibility
  - If the isolated code uses dynamically linked libraries, then restrictions may be circumvented
- Robustness
  - The defense may be circumvented should program execution by hijacked (need control-flow integrity)

EBPF

Extended Berkeley Packet Filters (eBPF)

- eBPF allows a user-space processes to supply filter programs that can be loaded and run in kernel mode
- Goal: “eBPF makes the Linux kernel dynamically programmable at runtime, while ensuring its runtime integrity remains intact.”
  - Gbadamosi et al. - The eBPF Runtime in the Linux Kernel
- The eBPF verifier validates filter programs
  - Filter programs are limited to a prescribed memory region
  - Guarantee is similar to SFI, but no runtime checks

Extended Berkeley Packet Filters (eBPF)

- Approach
  - Filter program is defined in user-space
  - And run in the kernel
  - Yet the kernel is protected (goal – not always true)
  - Security depends on the correctness of the eBPF verifier
- Performance
  - eBPF programs are JIT-compiled after validation
  - For native performance

Extended Berkeley Packet Filters (eBPF)

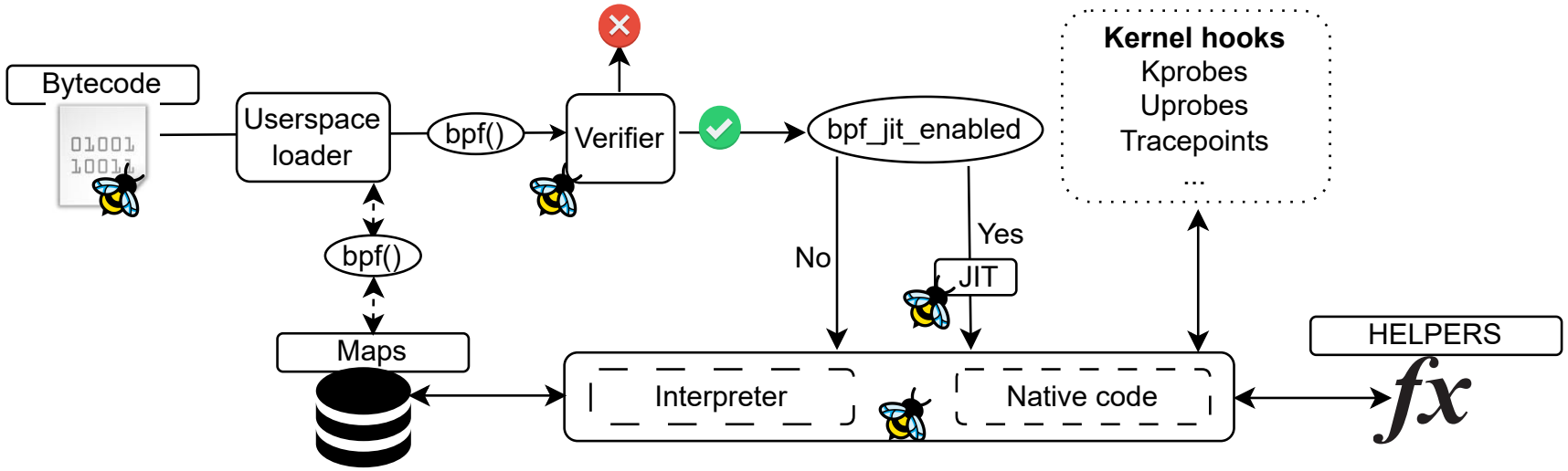


Figure 1: An overview of the eBPF key components and their correlation based on [24].

eBPF Probes

- eBPF programs are often triggered by the placement of probes
  - Linux has predefined hook points throughout the kernel
    - Events include system calls, function entry and exit, network sockets, tracepoints, etc. (Sect 4.2.3)
- Can also create custom hook points
  - Kernel probes (kprobes) or user probes (uprobes)
  - Attach eBPF programs to almost any position inside the kernel or user applications.

eBPF Maps

- eBPF maps are data structures for transferring data between the filter program and the kernel
- Store state across eBPF filter programs
  - Only way for an eBPF program to communicate with other eBPF programs and/or user-space.
  - Warning: “eBPF maps are not built with functionality guaranteeing integrity, which means … [must check] … that data is not overwritten by accident”
- Kernel also uses eBPF maps to convey information to-from filter programs

eBPF Verifier

- eBPF verifier is responsible for ensuring that only compliant filter programs are loaded
  - Checks a set of rules to aimed at ensuring the safety and stability of the kernel
  - E.g., limiting the memory that can be accessed when running a filter program in the kernel
- Limit the functionality of filter programs
  - E.g., type checking of operations, a stack limit of 512 bytes, no signed division, and the absence of loops
- eBPF verifier is complex – 20K SLOC

eBPF Verifier and Security

- Have found that the eBPF verifier may be bypassed
  - From paper: “For example, CVE-2017-16995 describes a way to read and write kernel memory and bypass the eBPF verifier”
- Researchers have applied fuzz testing to eBPF using syzbot
  - Which has produced many eBPF programs that bypass the eBPF verifier and crash the Linux kernel
  - How to protect the Linux kernel from eBPF programs is an ongoing issue

eBPF – For Project 3

- Your team may propose a project for eBPF
  - In lieu of Project 3 (File Systems)
- Project areas (examples)
  - Monitoring – new or building on existing
  - Security – circumvent eBPF verifier
  - Defense – augment verification

Conclusions

- Extending operating systems functionality is important for monolithic kernels
  - But, the challenge is to determine how to enable kernel extension securely within the monolithic architecture
- We discuss three approaches: software isolation, hardware isolation, and software verification
  - eBPF is adopted in the kernel, but has security concerns
  - SFI and hardware isolation are options, but have their own issues
- No single tool is likely to solve all problems

Questions