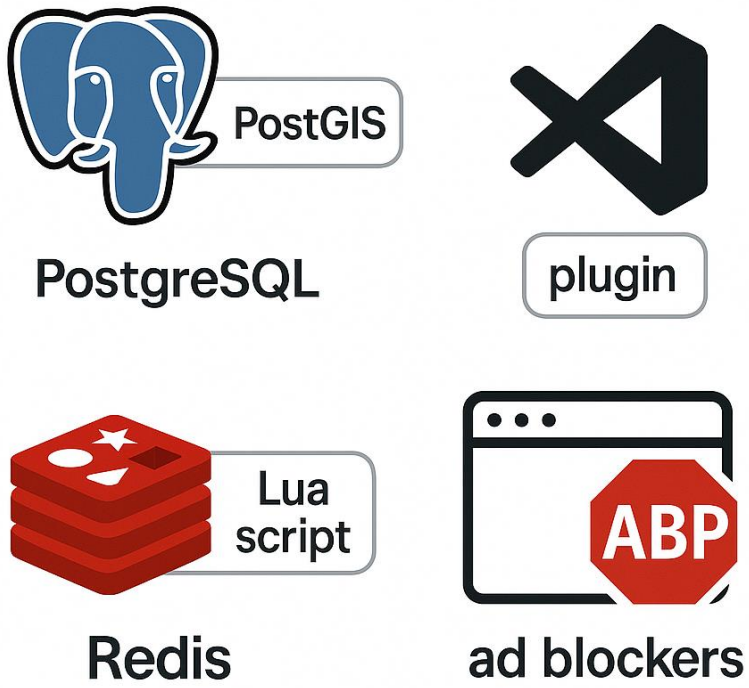
Extending Applications Safely and Efficiently

Yusheng Zheng[1] • Tong Yu[2] • Yiwei Yang[1] • Yanpeng Hu[3]
Xiaozheng Lai[4] • Dan Williams[5] • Andi Quinn[1]

[1]UC Santa Cruz [2]eunomia-bpf Community [3]ShanghaiTech University
[4]South China University of Technology [5]Virginia Tech

Extensions are everywhere



What are extensions?

- Customize software without modifying source code

Why do we need them?

- Different deployments, different needs

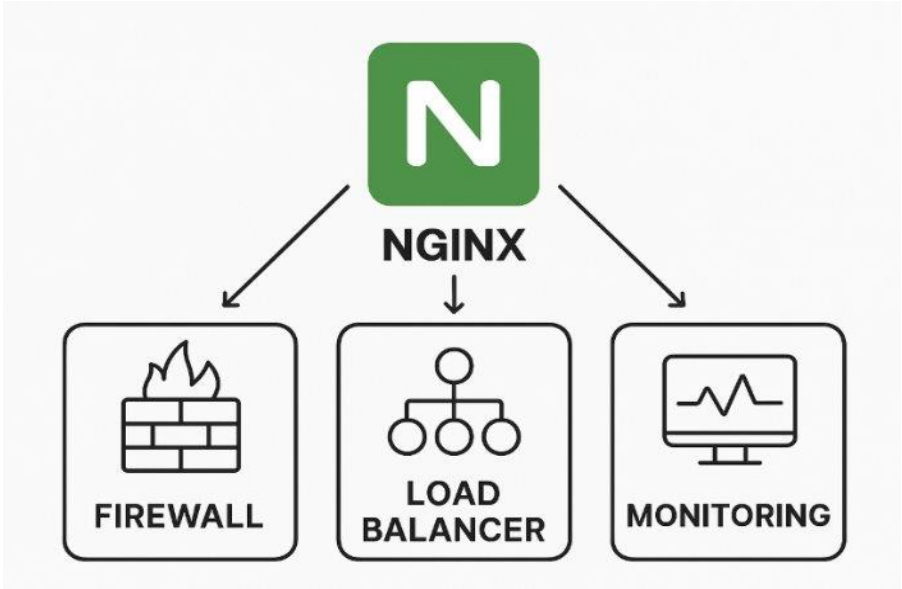
Nginx firewall example

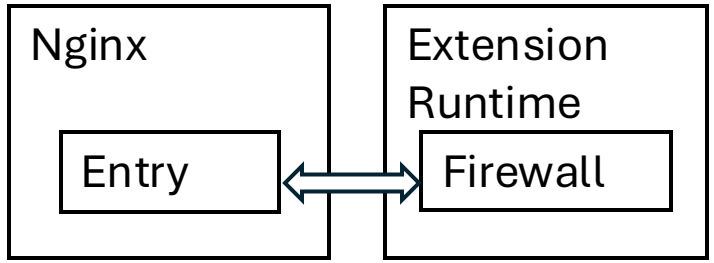
Before deployment, user:

- Writes firewall using nginx APIs
- Associates firewall with request processing extension entry.

During runtime, Nginx:

- Jumps to firewall when reaching request processing entry.
- Executes firewall in the extension runtime execution context.





Extension Problems & requirements

Real-world safety violations:

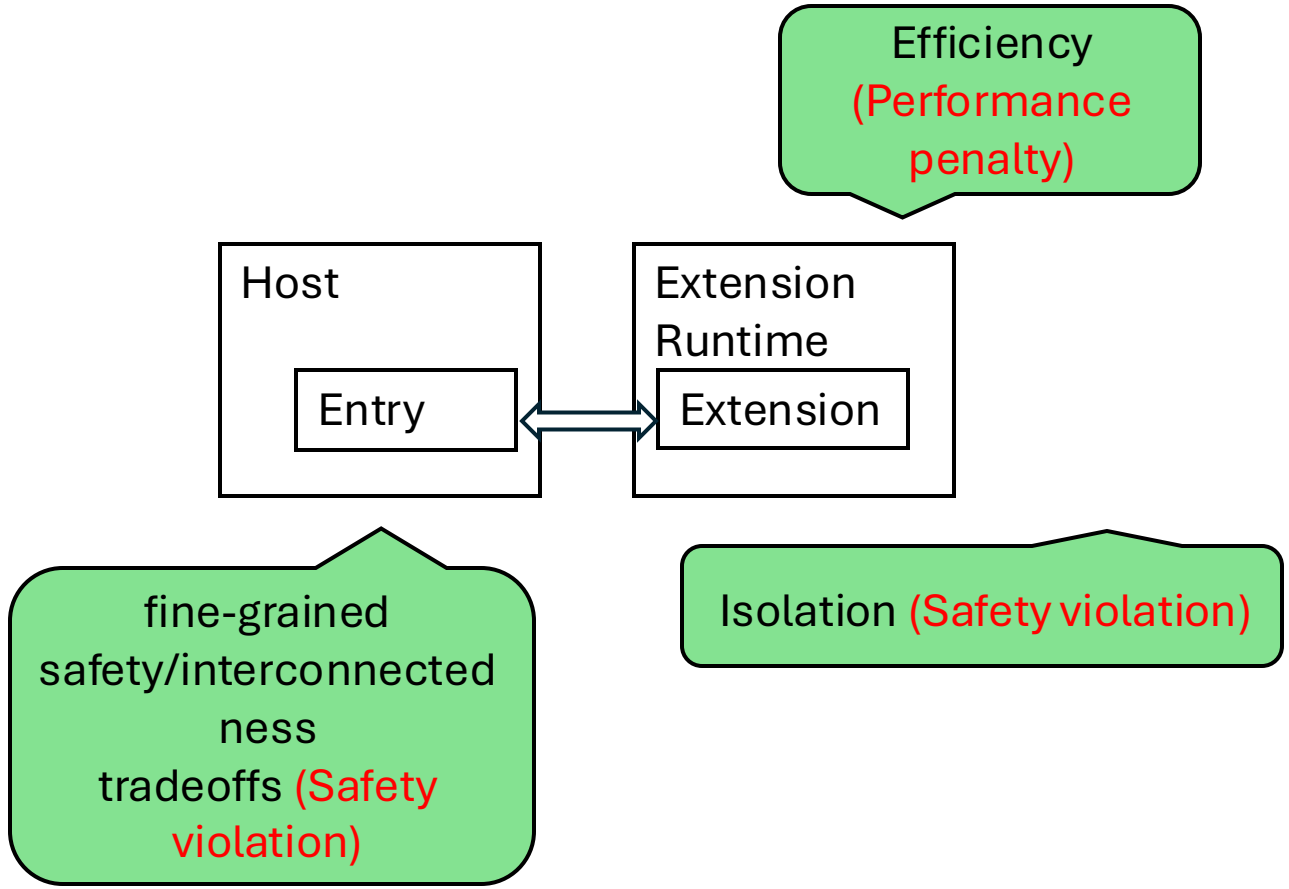
- Bilibili CDN outage, Apache buffer overflow, Redis RCE

Performance penalty:

- WebAssembly/Lua impose 10-15% overhead

Requirements:

- Fine-grained safety and interconnectedness trade-offs
- Isolation
- Efficiency



State-of-the-Art Falls Short

- Dynamic loading: efficiency but no isolation or fine-grained safety-interconnectedness policies (LD_PRELOAD, DBI tools)
- Software Fault Isolation: safety with 10–15 % performance penalty (XFI [OSDI 06], NaCL [SOSP 09], RL-Box [USENIX Security 20], Wasm and Lua)
- Subprocess: strong isolation but high IPC overhead (Wedge [NSDI 08], Shreds [IEEE SP 16], lwC [OSDI 16], and Orbit [OSDI 22])
- Kernel eBPF uprobes: isolation at micro second-level trap cost, low efficiency

Contributions

Extension Interface model (EIM)

Navigate fine-grained safety/interconectedness trade-offs for extensions

Bpftime runtime

Efficient support for EIM and isolation through userspace eBPF runtime

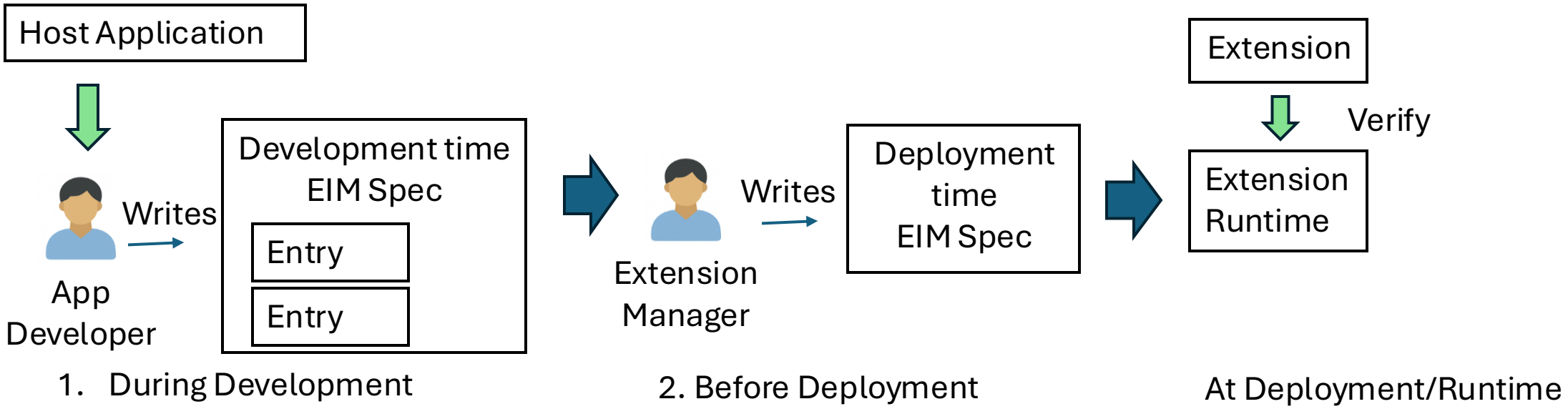
- Up to 6x less overhead than current state-of-the-art!

Outline

- Background & motivation: Extensions
- → **Extension Interface Model (EIM)**: Fine-grained Interface
- bpftime Runtime: safety & performance
- Evaluation

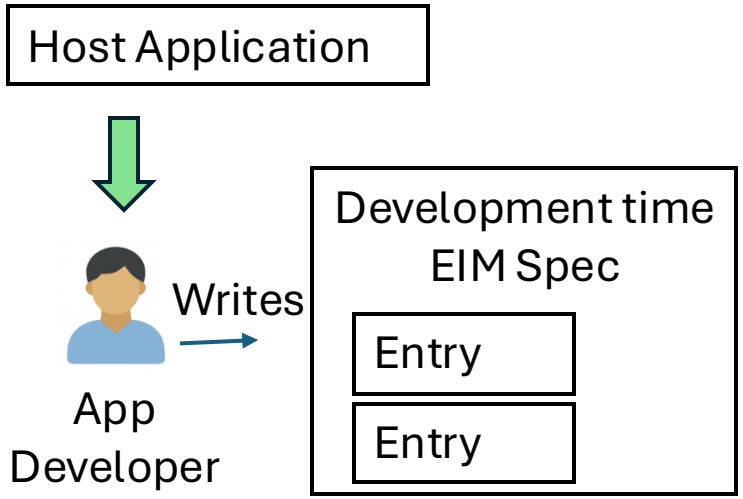
EIM: Extension Interface Model

- Goal: enable fine-grained safety/interconnectedness trade-offs
- Challenge: supporting per deployment tradeoffs
- Solution:
  - Two-phase specification (Development Time and Deployment Time)
  - Model all resources as capabilities



EIM: Development Time Specification

- Developers annotate code for capabilities
- Automatically extracted into capability manifest



1. During Development

```
EIM_STATE_DEFINE(readPid, read, ngx_pid);
EIM_HFUNC_DEFINE_WITH_CONSTRAINTS(
    nginxTime,
    HF_RET_POSITIVE);
EIM_EXTENSION_ENTRY_DEFINE(
    processBegin,
    ngx_http_process_request,
    int,
    struct Request *);
```

EIM: Deployment Time Specification

- YAML policies specify safety/interconnectedness tra deoffs
- Compact policies (avg of 30 lines in evaluation).

```
1 Extension_Class(
2  name = "observeProcessBegin",
3  extension_entry = "processBegin",
4  allowed = {instructions<inf, nginxTime,
       readPid, read(r)})
5 Extension_Class(
6  name = "updateResponse",
7  extension_entry = "updateResponseContent"
8  allowed = {instructions<inf, read(r),
       write(r)})
```

Writes

Deployment time EIM Spec

Extension Manager

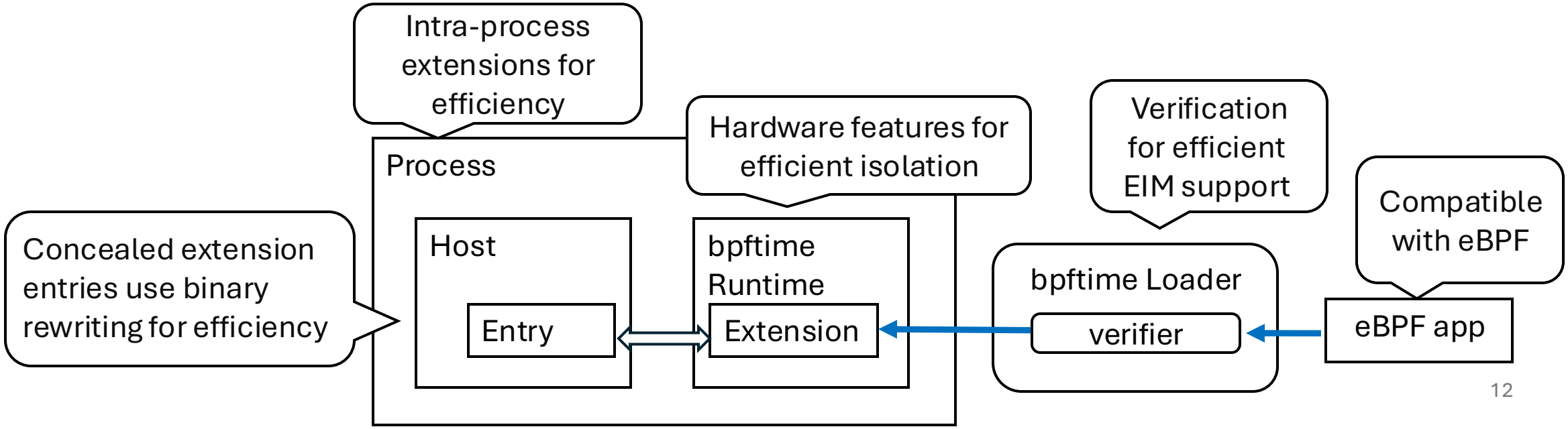
2. Before Deployment

Outline

- → Background & motivation: Extensions
- Extension Interface Model (EIM): Fine-grained Interface
- → **bpftime Runtime**: safety & performance
- Evaluation

bpftime: userspace eBPF extension framework

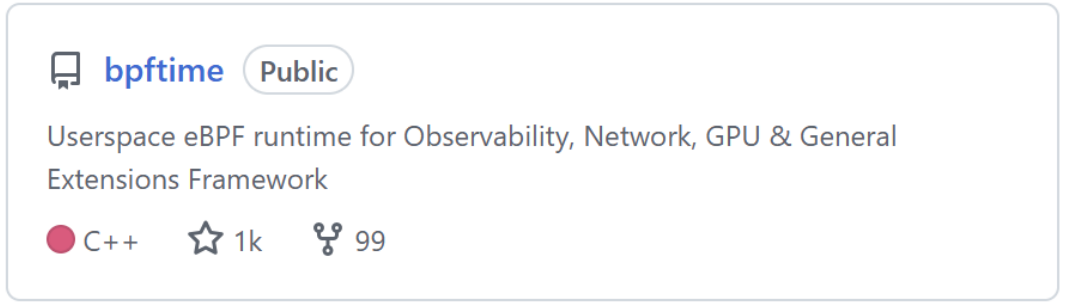
- Goal: efficiently support EIM and isolation
- Challenge: Existing extension runtimes use heavyweight safety & isolation techniques
- Solution:
  - build new design that exploits eBPF-style verification, binary rewriting, and hardware features to enable efficient intra-process extensions



Outline

- Background & motivation: Extensions
- Extension Interface Model (EIM): Fine-grained Interface
- bpftime Runtime: safety & performance
- → **Evaluation**

Six Real-World Use Cases



GitHub: https://github.com/eunomia-bpf/bpftime

Customization

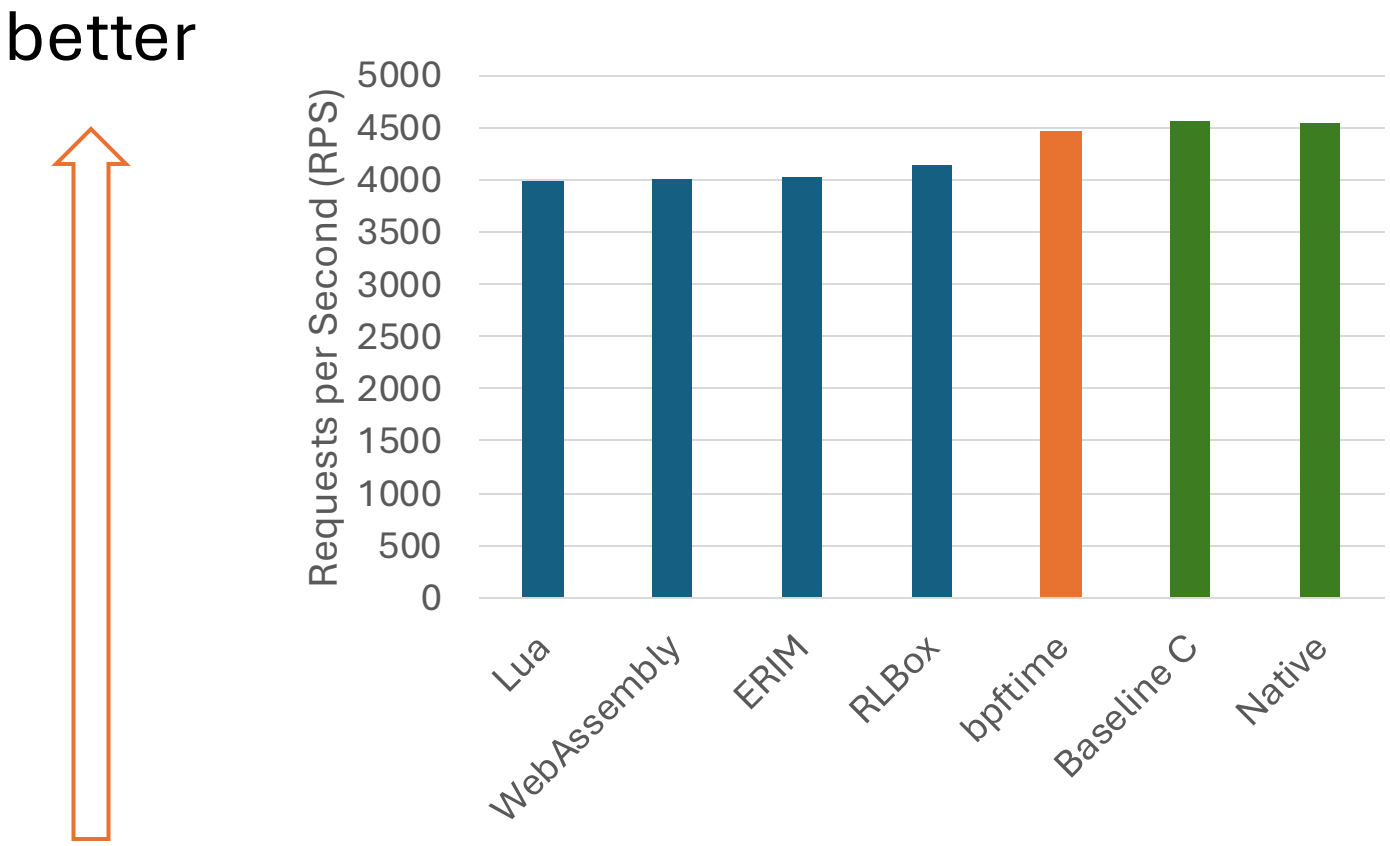
- Nginx Firewall
- Redis Durability
- FUSE Metadata Cache

Observability

- DeepFlow
- Syscount
- Sslsniff

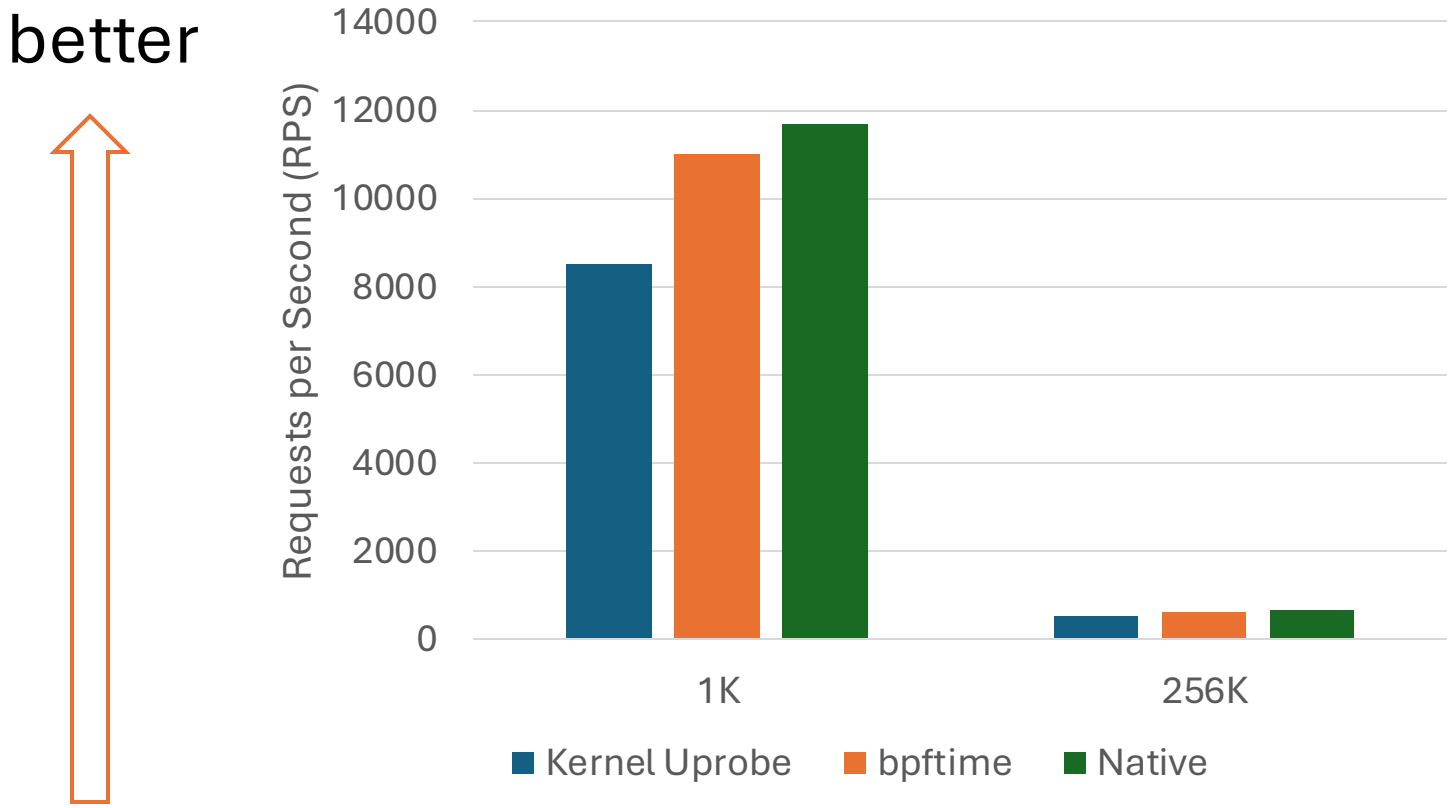
Customization: Nginx firewall

- 5× to 6× less overhead than lua or WebAssembly



Observability: sslsniff

- Maximum 21% less overhead than kernel eBPF



Contributions

Questions?

Extension Interface model (EIM)

Navigate fine-grained safety/interconectedness trade-offs for extensions

Bpftime runtime

Efficient support for EIM and isolation through userspace eBPF runtime

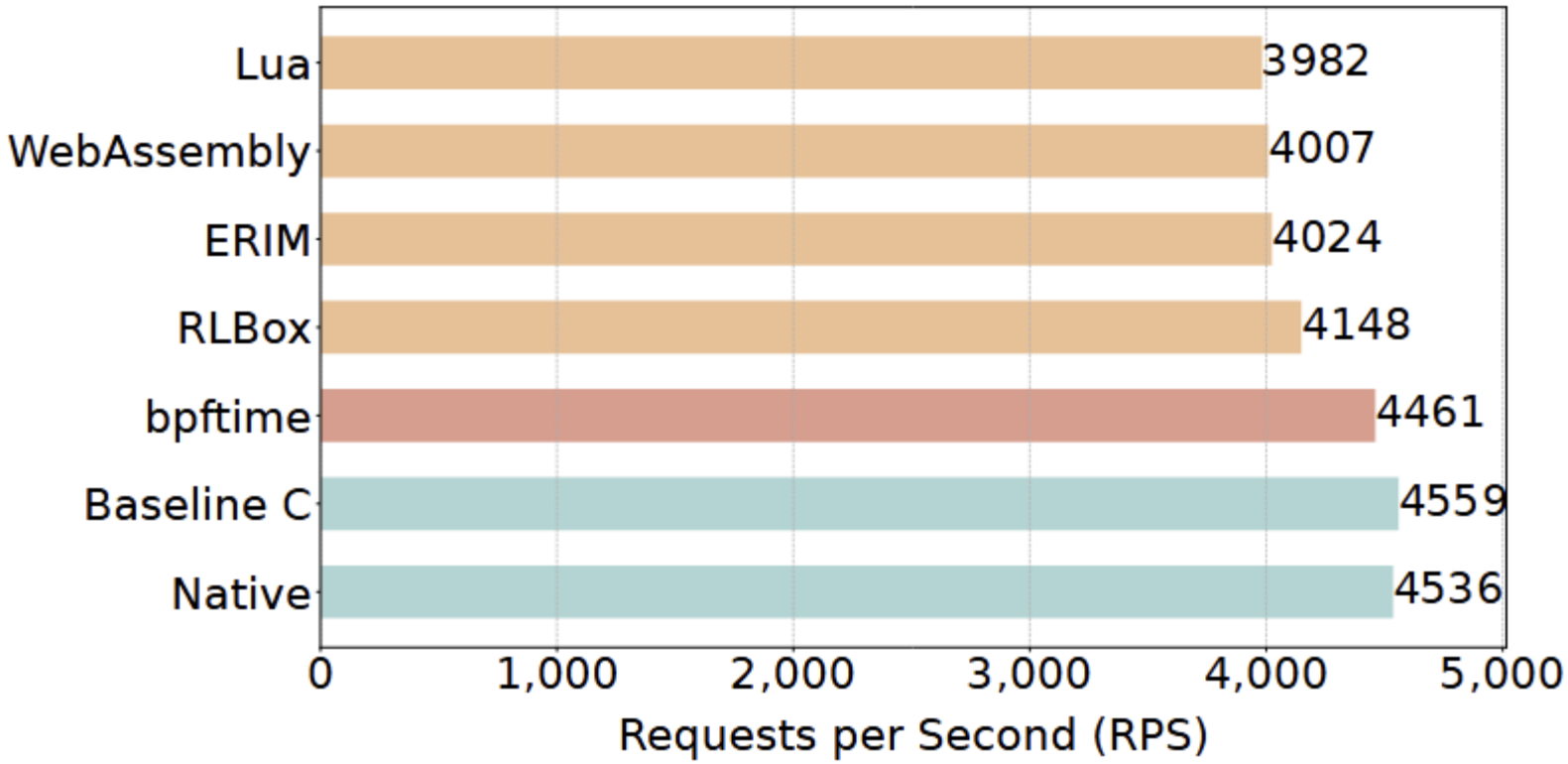
- Up to 6x less overhead than current state-of-the-art!

```
bpftime load ./example/malloc/malloc
bpftime start nginx -c ./nginx.conf
```

Backup

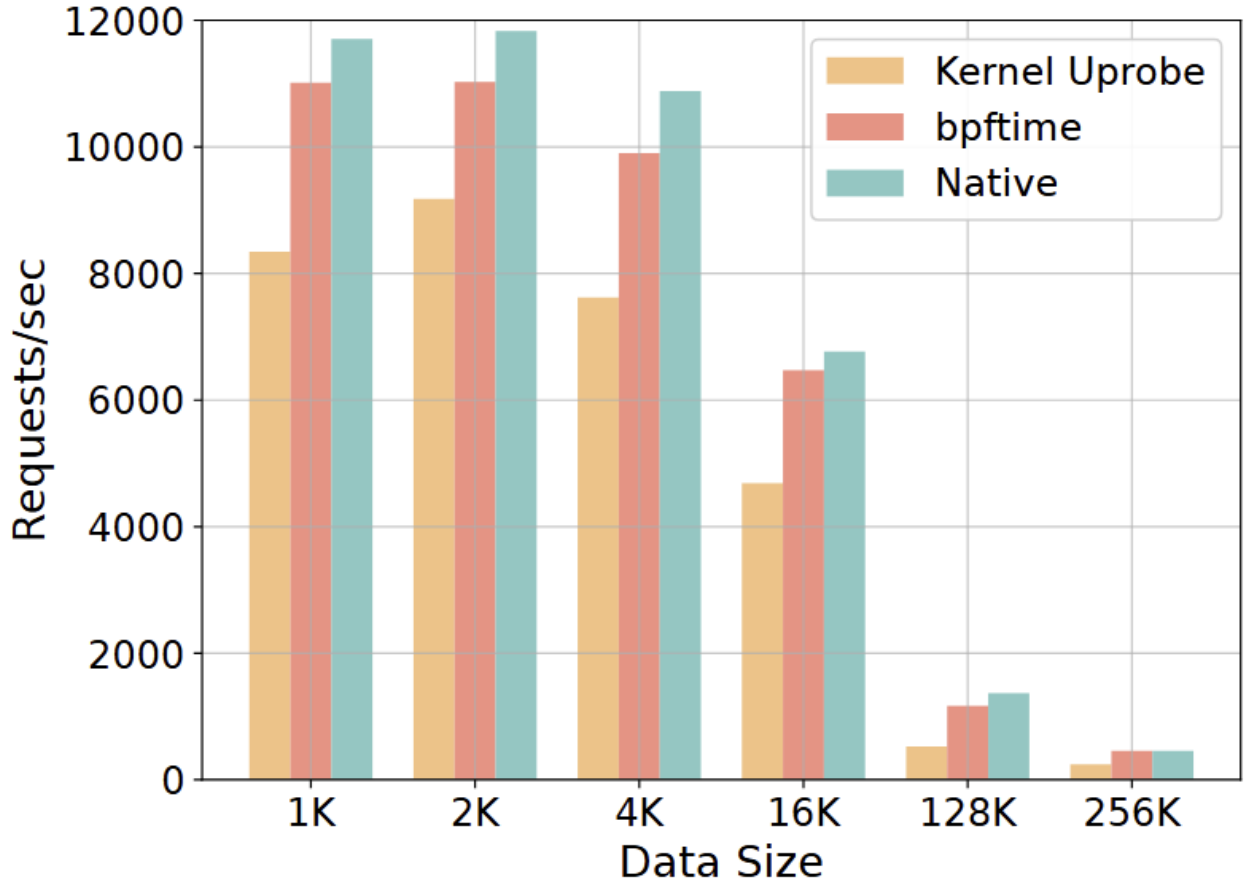
Customization: Nginx firewall

- 5× to 6× improvement
- Less is better

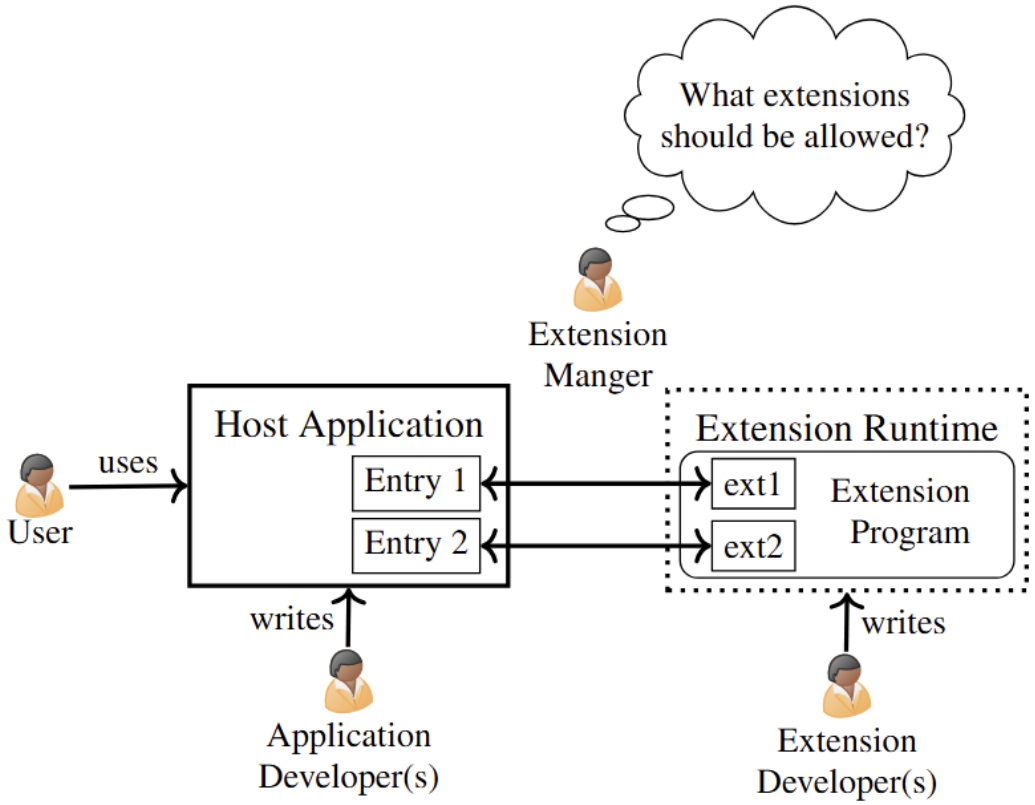


Observability: sslsniff

- 21% less overhead than kernel eBPF



Four Roles in an Extension Ecosystem



Micro-Benchmark

Compare with eBPF:

- **Uprobe Dispatch**: 2.56 µs → 190 ns (14× faster)
- **Syscall Tracepoint**: 151 ns → 232 ns (1.5× slower)
- **Memory access** (Table 3): user-space read/write 2 ns vs 20 ns (10× faster)
- **Overall**: average 1.5× faster than ubpf/rbpf (Figure 11)

Extensions have issues

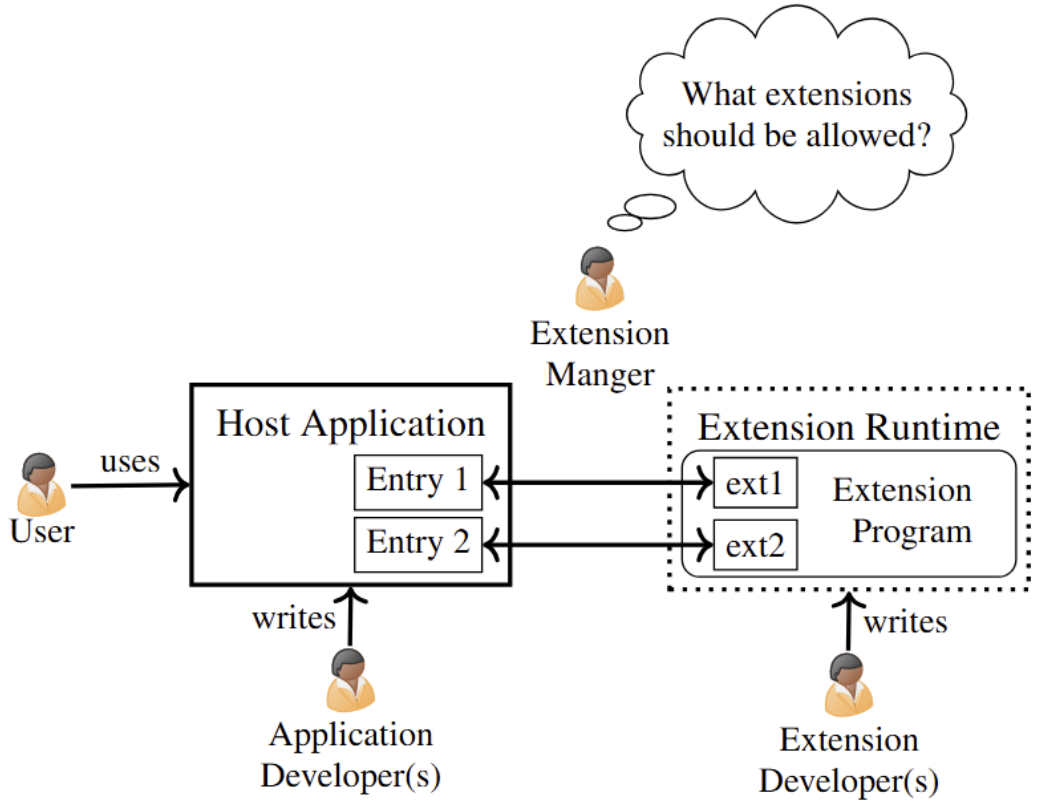
- Example issues caused by extension safety violations

Bug	Software	Summary
Bilibili [73]	Nginx	Livelock (infinite loop) in an extension caused production outage.
CVE-2021-44790 [47]	Apache	Buffer overflow in httpd's lua module causes application to crash.
CVE-2024-31449 [42]	Redis	Stack overflow in Lua script leads to arbitrary remote code execution.

- The performance penalty of existing approaches

Nginx example

- d



Get started

- Use uprobe to monitor userspace malloc function
- Try eBPF in GitHub codespace!**(Unprivilidged container)**

To get started, you can build and run a libbpf based eBPF program starts with `bpftime` cli:

```
make -C example/malloc # Build the eBPF program example
export PATH=$PATH:~/.bpftime/
bpftime load ./example/malloc/malloc
```

In another shell, Run the target program with eBPF inside:

```
$ bpftime start ./example/malloc/victim
Hello malloc!
malloc called from pid 250215
continue malloc...
malloc called from pid 250215
```

Loader & Runtime Workflow

- **Intercept** standard eBPF syscalls from libbpf/bcc.
- **Parse** EIM manifests and DWARF/BTF to generate constraints.
- **Verify** byte-code via kernel's eBPF verifier with added assertions.
- **JIT-Compile** verified byte-code into native x86.
- **Inject** user-runtime via `ptrace` + Frida + Capstone trampolines.
- **Execute** extension: flip MPK key → jump to code → flip back → resume.

Efficient Safety & Isolation

- **The eBPF compatibility challenge:**
  - Linux eBPF has tightly coupled components (compilers, runtime, kernel)
  - Prior user eBPF failed by re-implementing entire stack
  - **bpftime solution**: Interpose on eBPF syscalls only
- **Key design principles:**
  - Lightweight EIM enforcement
  - Concealed extension entries: 10× faster uprobe

Contribution

- **Extension Interface Model (EIM)**: Fine-grained capability control
- **bpftime Runtime**: Kernel-grade safety with library-grade performance

State-of-the-Art Falls Short

Approach	Safety	Isolation	Efficiency	Fine-Grained Control
Dynamic Loading	✗	✗	✓	✗
SFI (Wasm, Lua)	Limited	✓	✗ (10-15% overhead)	✗
Subprocess	✓	✓	✗ (context switches)	Limited
eBPF uprobes	✓	✓	✗ (kernel traps)	Limited

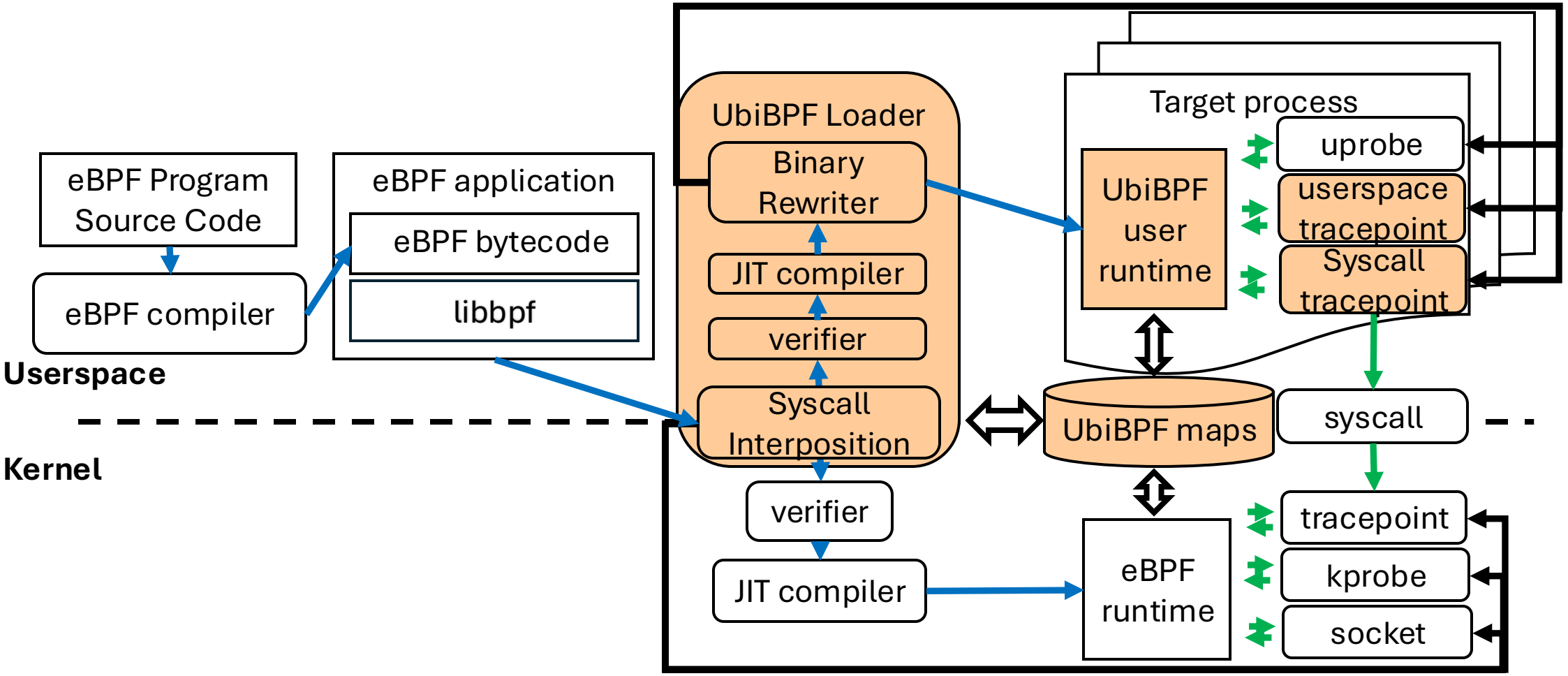
- No single framework satisfies all requirements

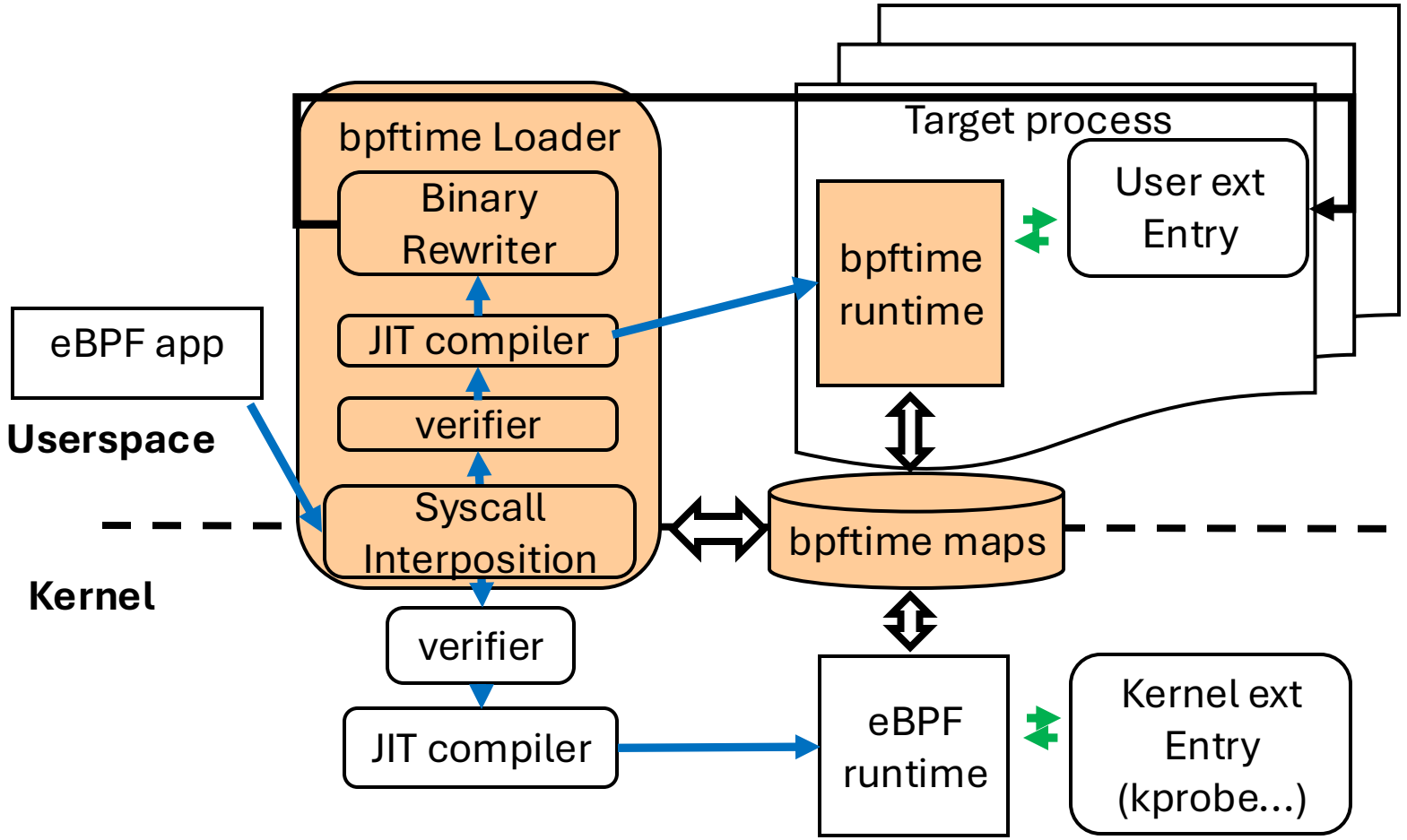
Summary of EIM

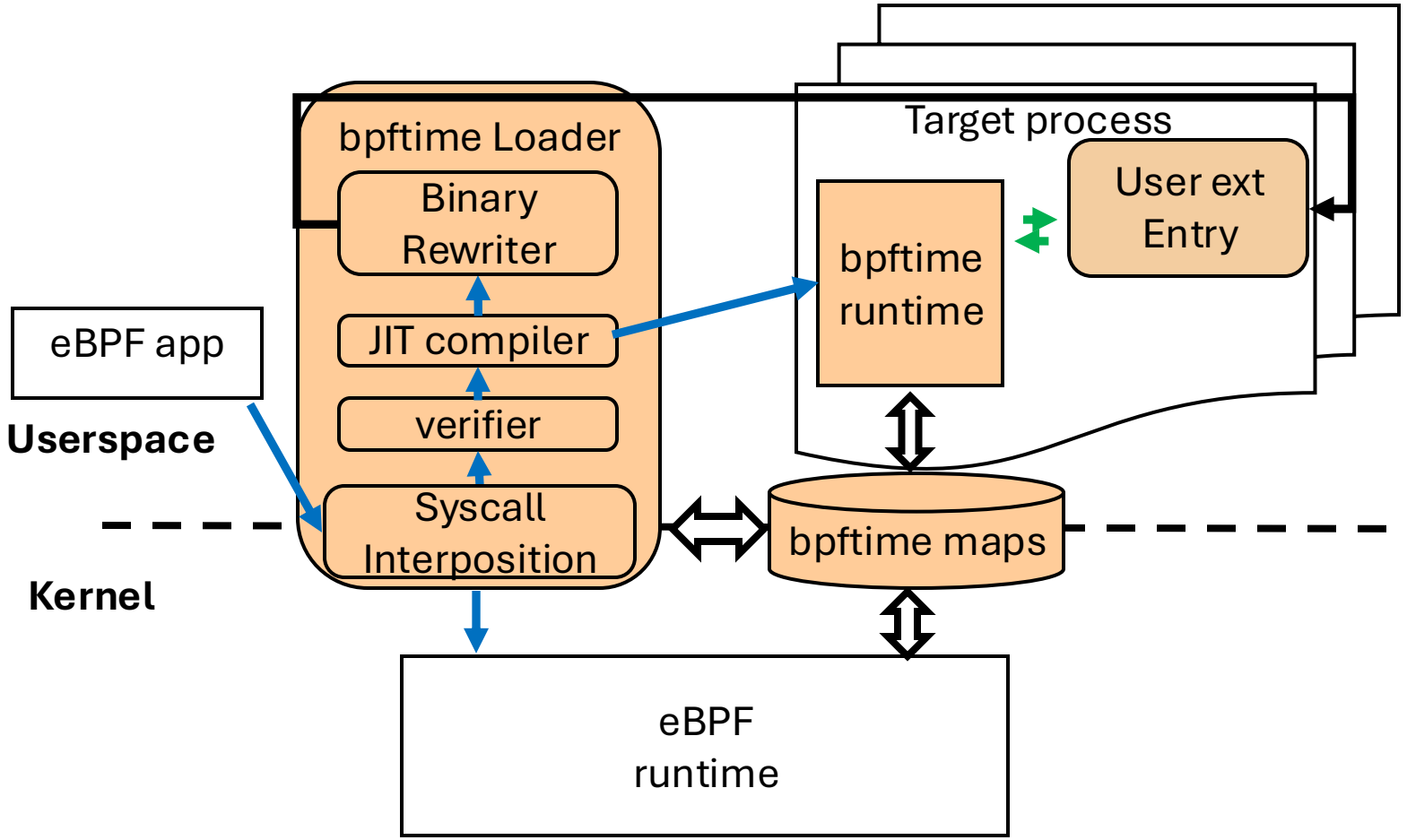
- Existing frameworks → no control OR coarse-grained bundles
- Treats safety and interconnectedness as independent dimensions
- **Example policies:**
  - Monitoring extension: read-only access to specific variables
  - Firewall extension: read/write for response modification

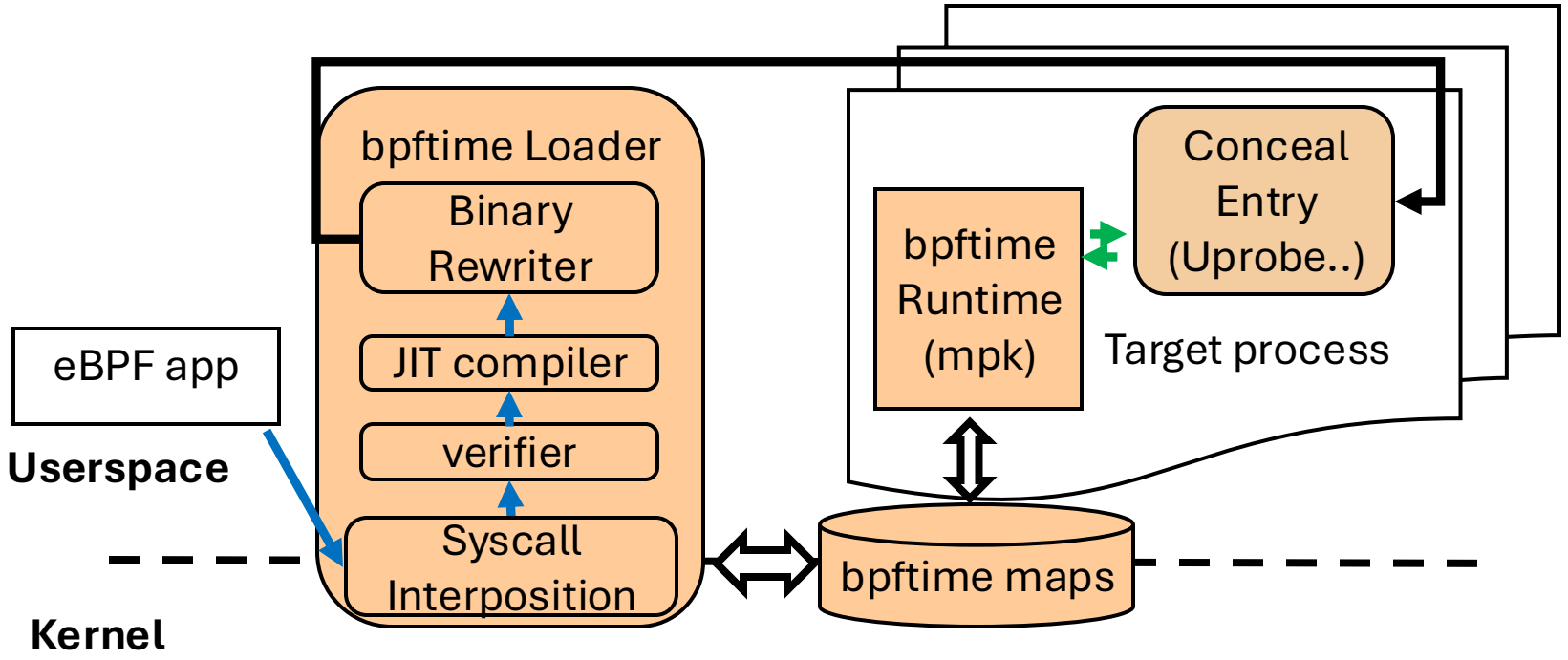
bpftime - Why We Need a New Runtime

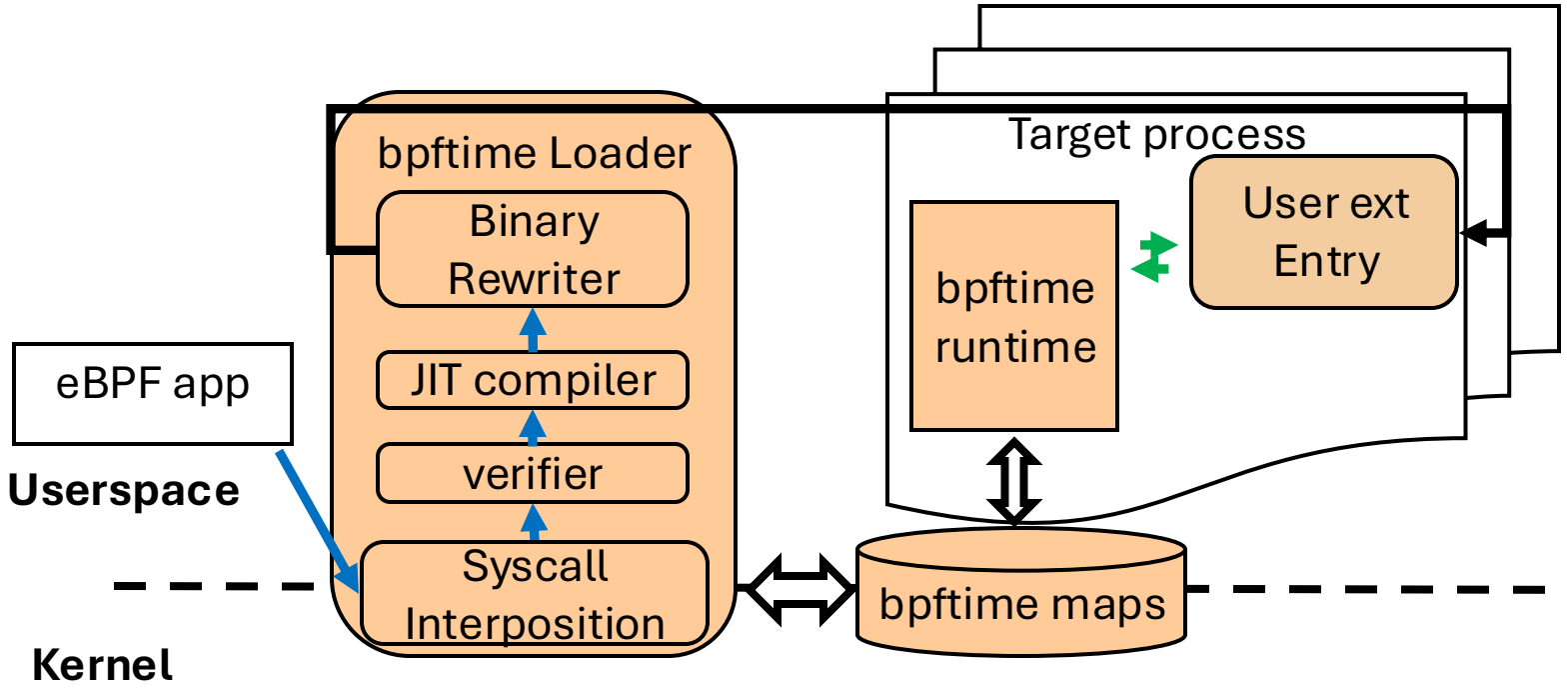
- Can't existing frameworks enforce EIM efficiently?
  - **WebAssembly/SFI**: 10-15% overhead, **Subprocess isolation**: Expensive switches, **Kernel eBPF uprobes**: Kernel traps
- A userspace extension framework in eBPF
  - **Compatibility and Work together with kernel eBPF extensions**
  - verification for safety
  - **Conceal for efficient**
  - **Mpk for isolation**

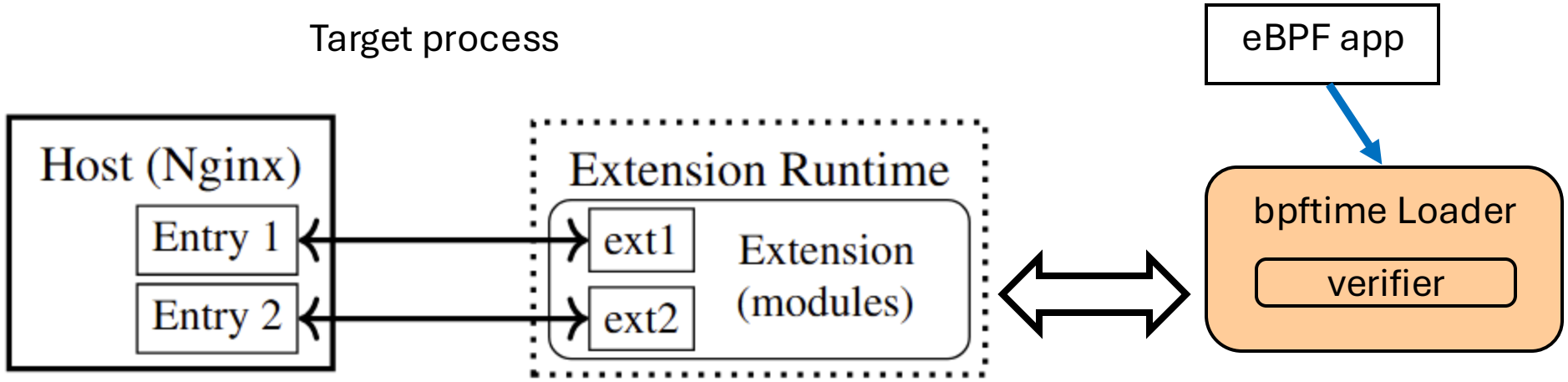

eBPF Program Source Code
eBPF compiler
eBPF application
eBPF bytecode
libbpf
Userspace
Kernel
UbiBPF Loader
Binary Rewriter
JIT compiler
verifier
Syscall Interposition
verifier
JIT compiler
Target process
uprobe
userspace tracepoint
Syscall tracepoint
UbiBPF user runtime
UbiBPF maps
syscall
tracepoint
kprobe
socket
eBPF runtime

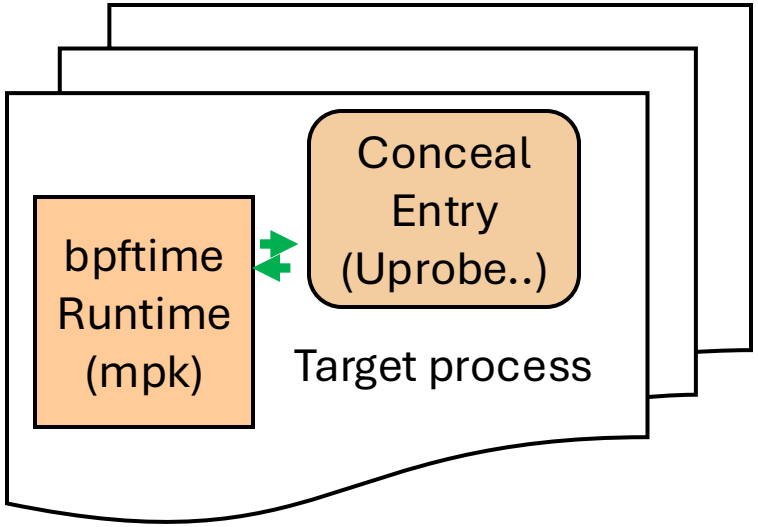

bpftime Loader
Binary Rewriter
JIT compiler
verifier
Syscall Interposition
verifier
JIT compiler
eBPF app
Userspace
Kernel
Target process
bpftime runtime
User ext Entry
bpftime maps
eBPF runtime
Kernel ext Entry (kprobe...)

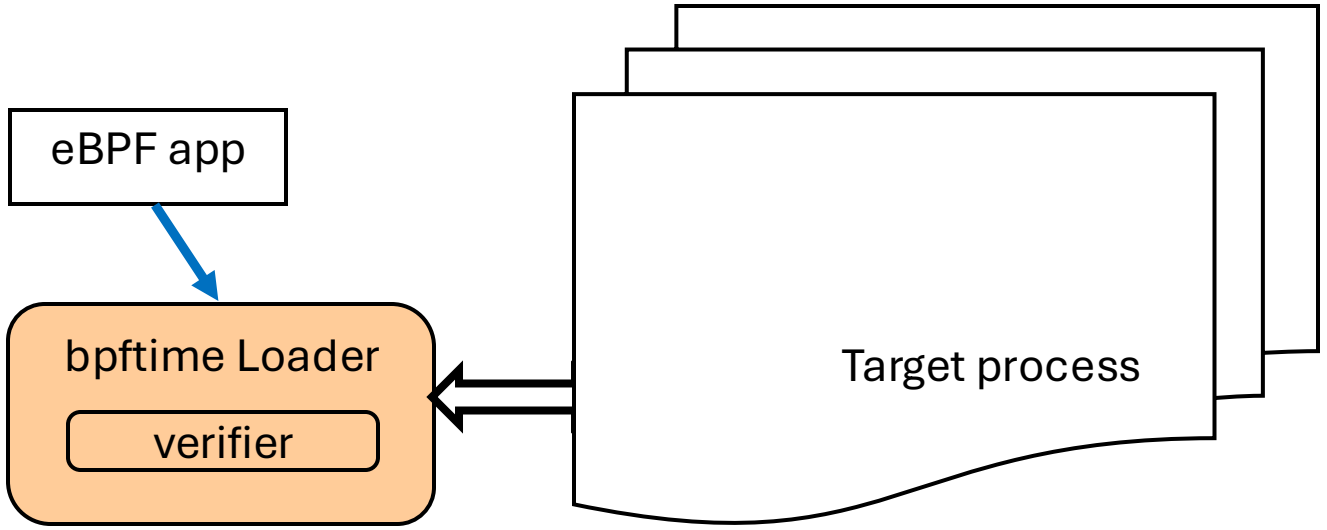

bpftime Loader
Binary Rewriter
JIT compiler
verifier
Syscall Interposition
eBPF app
Userspace
Kernel
Target process
bpftime runtime
User ext Entry
bpftime maps
eBPF runtime


bpftime Loader
Binary Rewriter
JIT compiler
verifier
Syscall Interposition
eBPF app
Userspace
Kernel
bpftime Runtime (mpk)
Conceal Entry (Uprobe..)
Target process
bpftime maps


bpftime Loader
Binary Rewriter
JIT compiler
verifier
Syscall Interposition
eBPF app
Userspace
Kernel
Target process
bpftime runtime
User ext Entry
bpftime maps


Target process
eBPF app
Host (Nginx)
Entry 1
Entry 2
Extension Runtime
ext1
ext2
Extension
(modules)
bpftime Loader
verifier


Conceal
Entry
(Uprobe..)
bpftime
Runtime
(mpk)
Target process


eBPF app
bpftime Loader
verifier
Target process

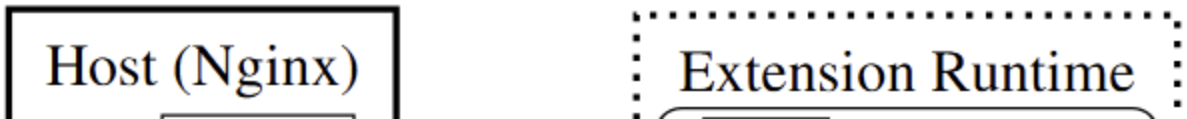
Nginx firewall example

User wants to have a firewall to block malicious requests

User write custom firewall logic using nginx helper functions

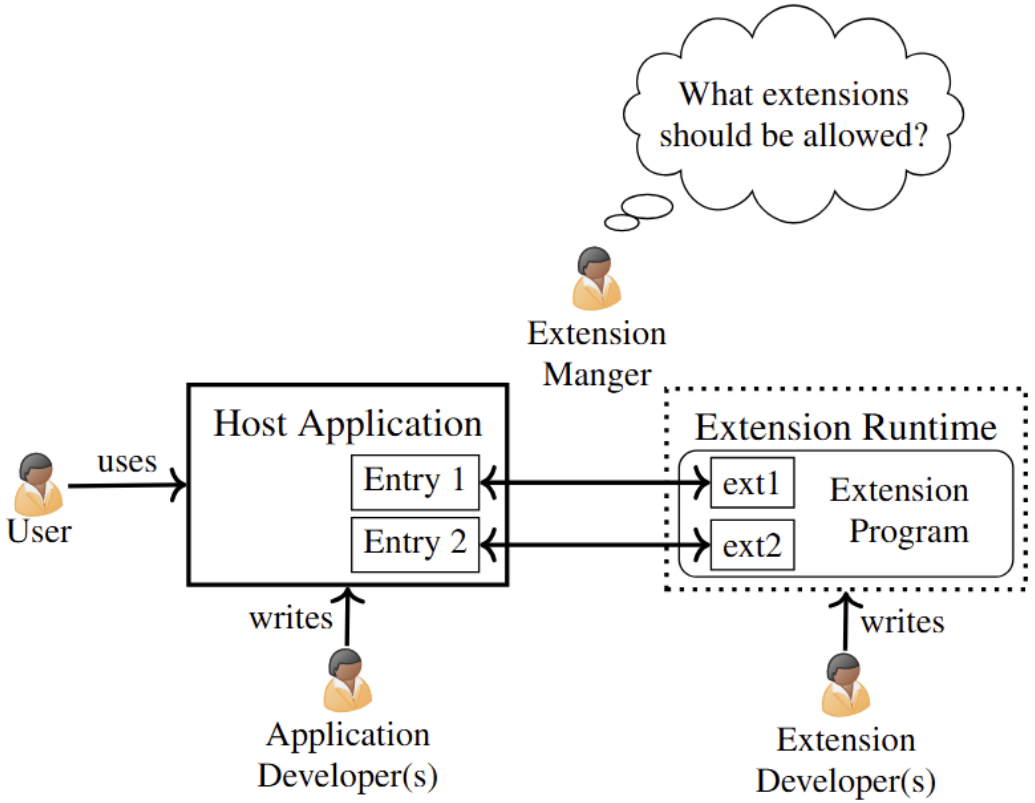
Load their extension at an extension entry for request pr

Extension execution model: Thread → Extension entry → Jump to extension → Execute by extension runtime → Return to host



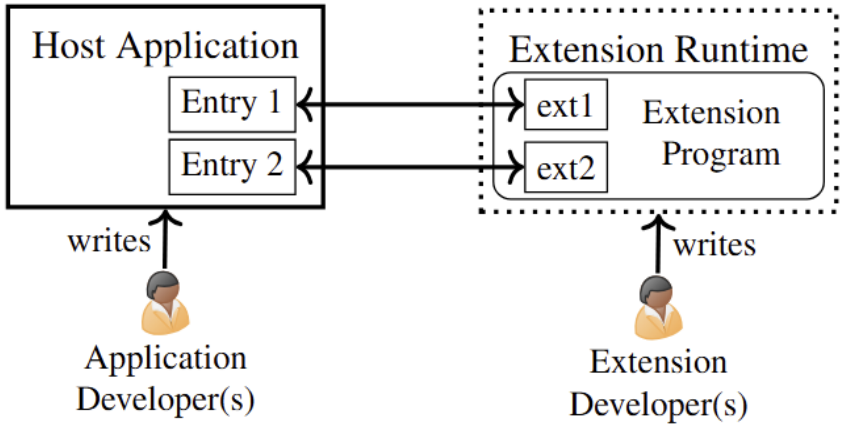
EIM: Extension Interface Model

- Solution to nav fine-grained safety-interconnectedness trade-offs
- Two-Phase Specification
  - Development-Time (by Developer)
  - Deployment-Time (by Manager)
- Capabilities as Resources



EIM: Development-Time Specification

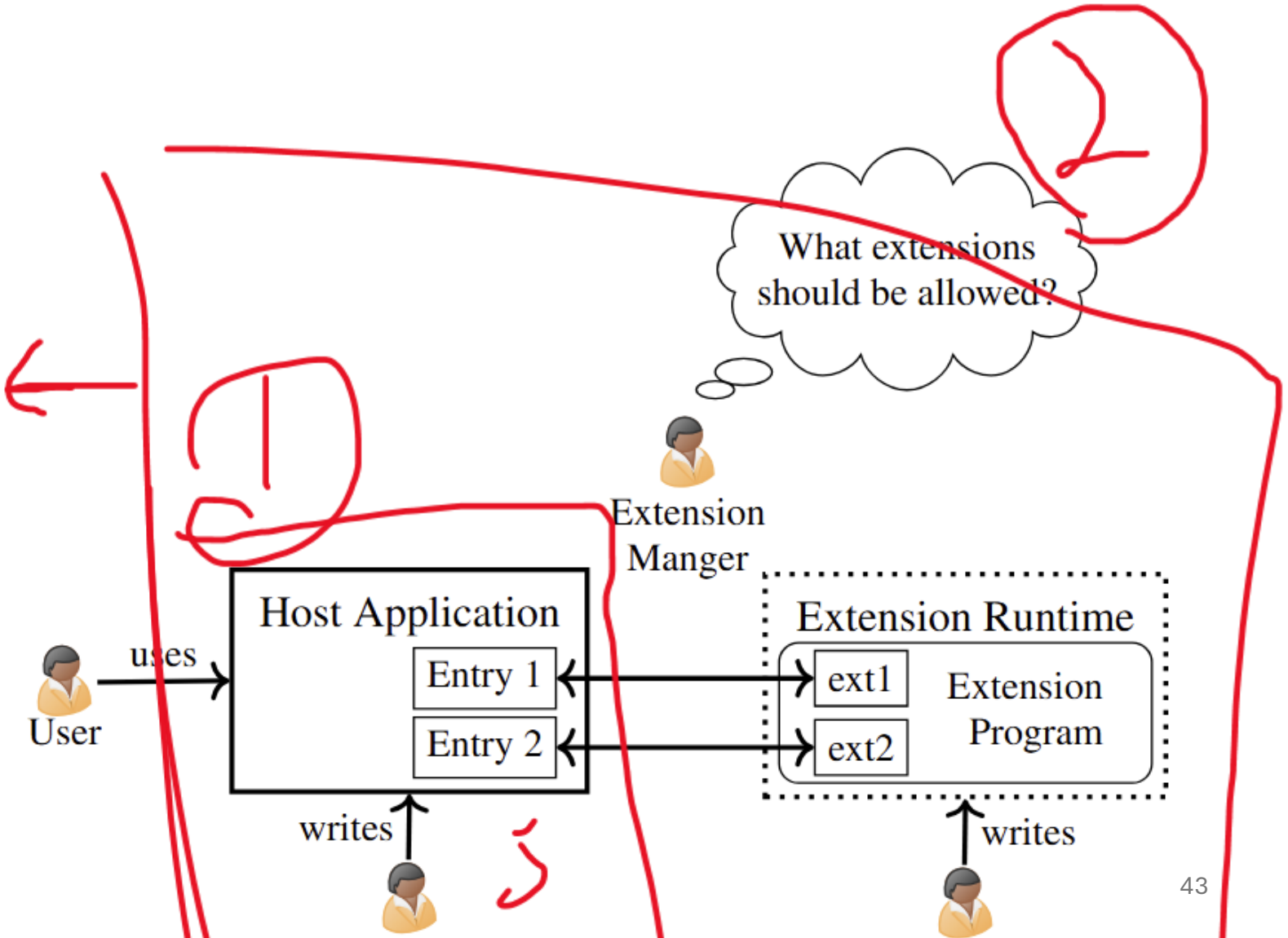
- Developers annotate code for capabilities
- Automatically extracted into capability manifest




```
EIM_STATE_DEFINE(readPid, read, ngx_pid);
EIM_HFUNC_DEFINE_WITH_CONSTRAINTS(
    nginxTime,
    HF_RET_POSITIVE);
EIM_EXTENSION_ENTRY_DEFINE(
    processBegin,
    ngx_http_process_request,
    int,
    struct Request *);
```

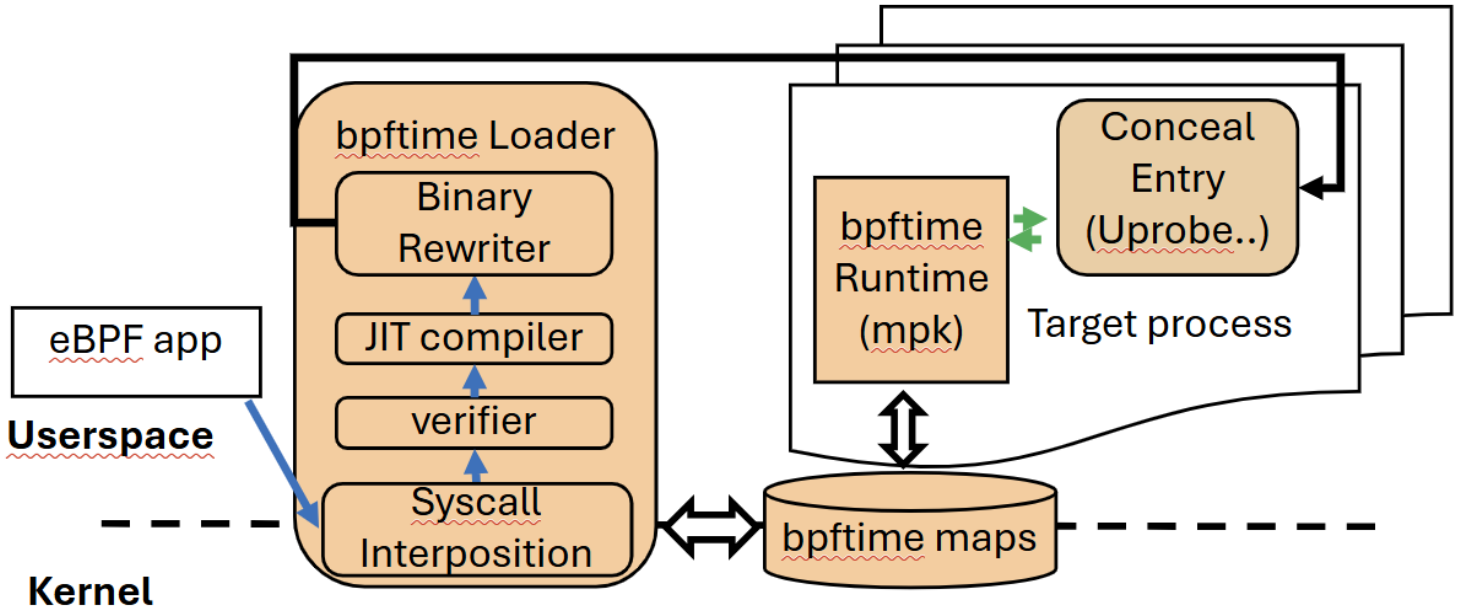
EIM: Extension Interface Model

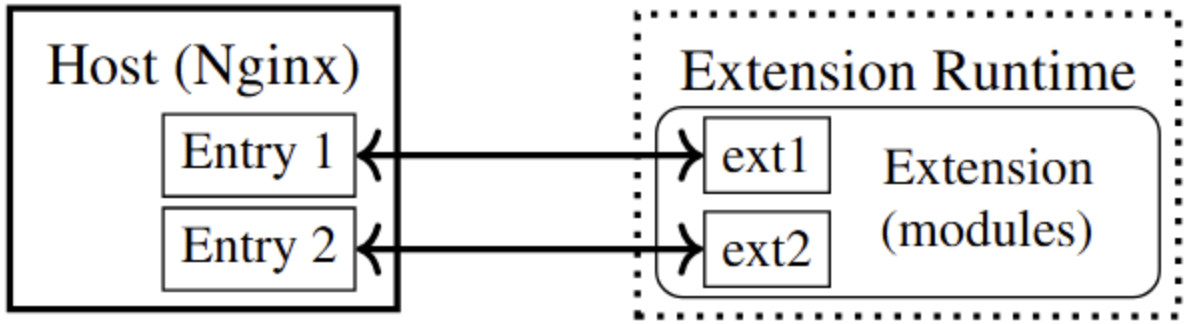
- Solution to nav fine-
  safety-interconnect
  trade-offs
- Two-Phase Specific
  - Development-Time
    Developer)
  - Deployment-Time (
    Manager)
- Capabilities as Resc

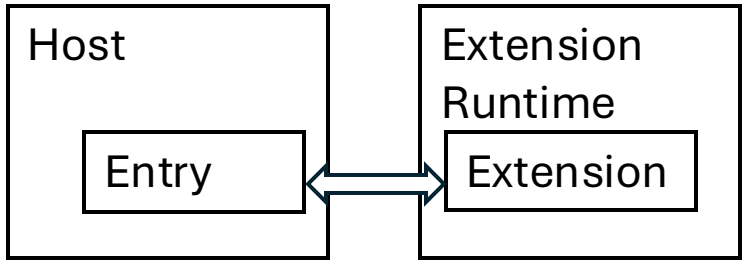


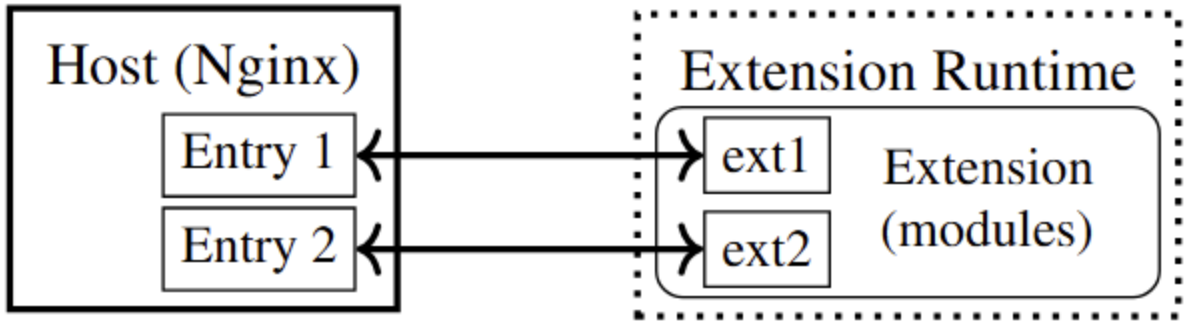
bpftime: userspace eBPF extension framework

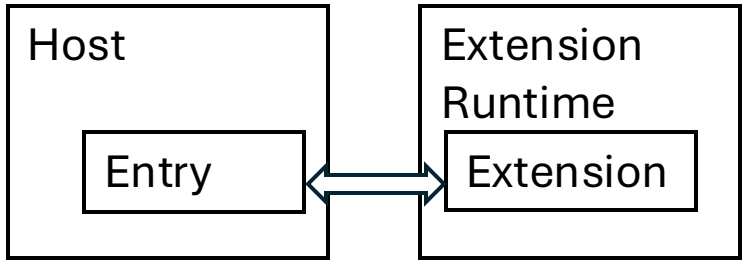
- Challenge for compatibility and efficiency:
  - eBPF: tightly coupled components
  - Bpftime: Intercept syscalls & Share memory maps

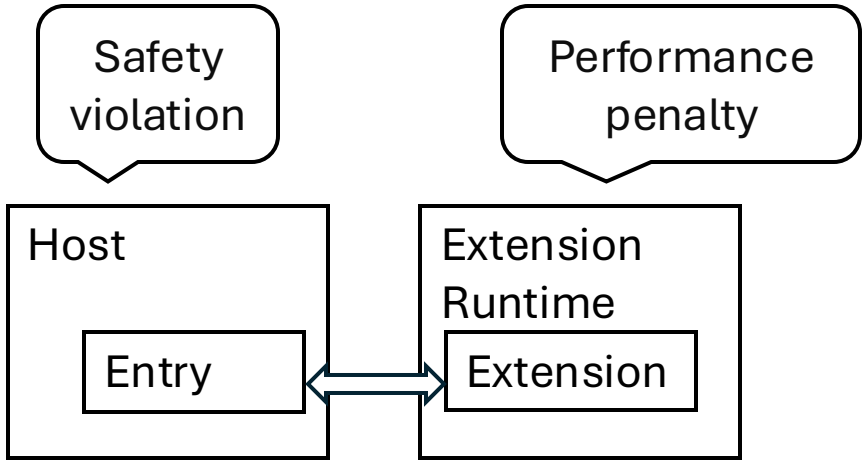


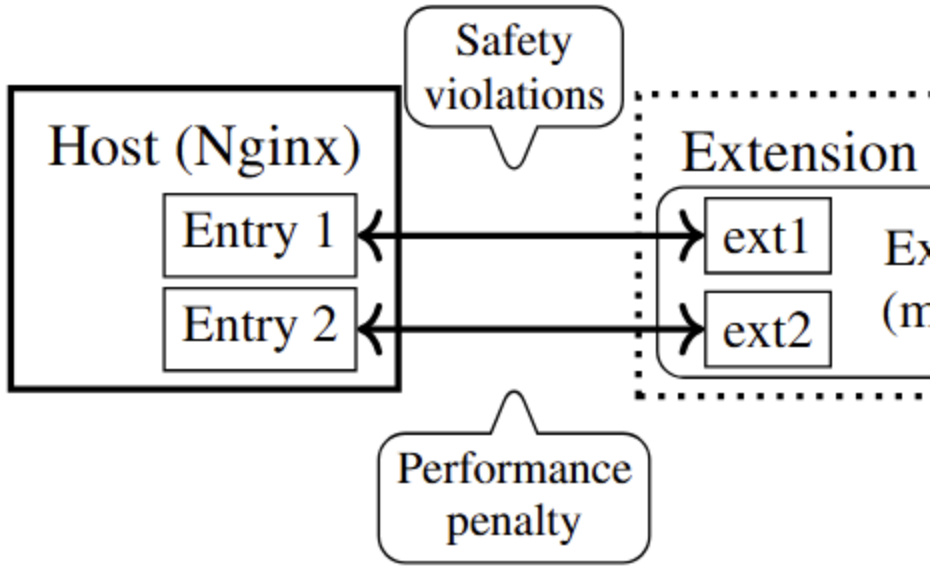

Host (Nginx)
Entry 1
Entry 2
Extension Runtime
ext1
ext2
Extension
(modules)

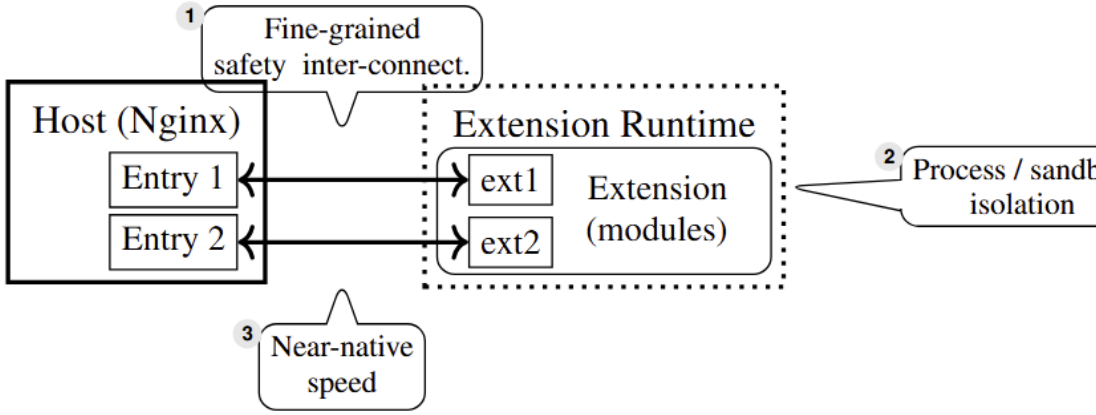

Host
Entry
Extension
Runtime
Extension


Host (Nginx)
Entry 1
Entry 2
Extension Runtime
ext1
ext2
Extension
(modules)


Host
Entry
Extension
Runtime
Extension

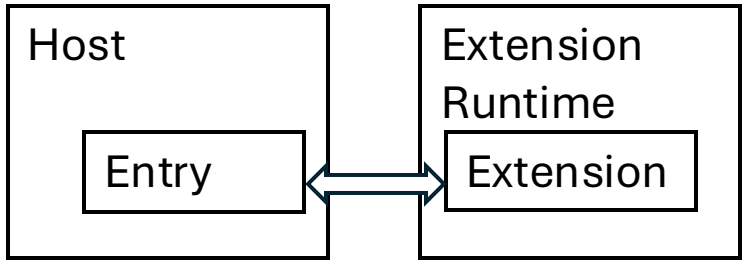

Safety violation
Performance penalty
Host
Entry
Extension Runtime
Extension


Safety violations
Host (Nginx)
Entry 1
Entry 2
Extension
ext1
ext2
Performance penalty



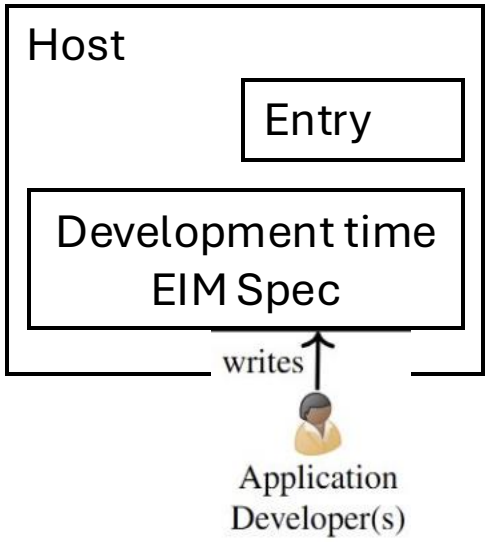
fine-grained safety and interconnectedness trade-offs

efficiency

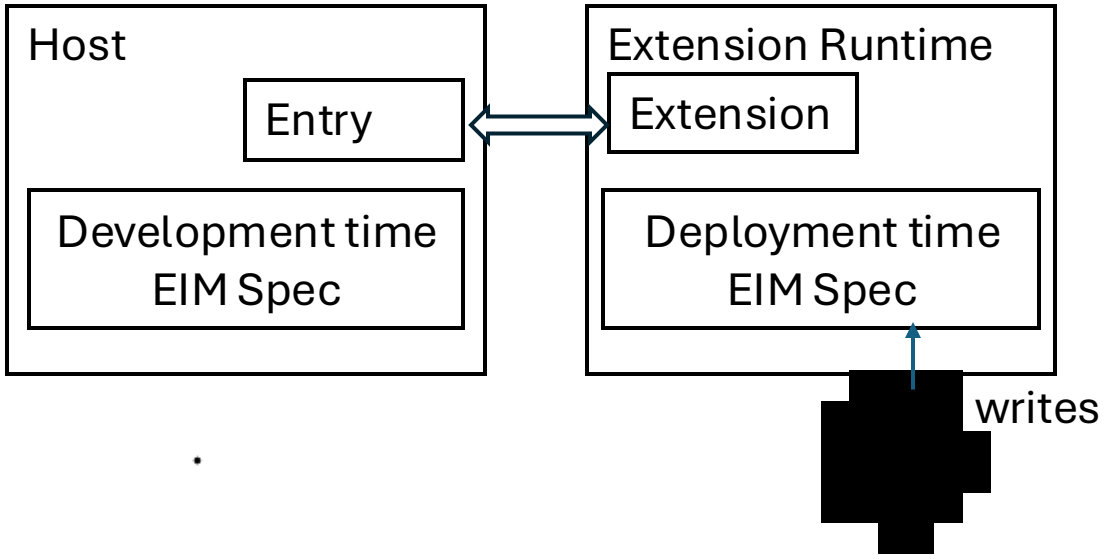


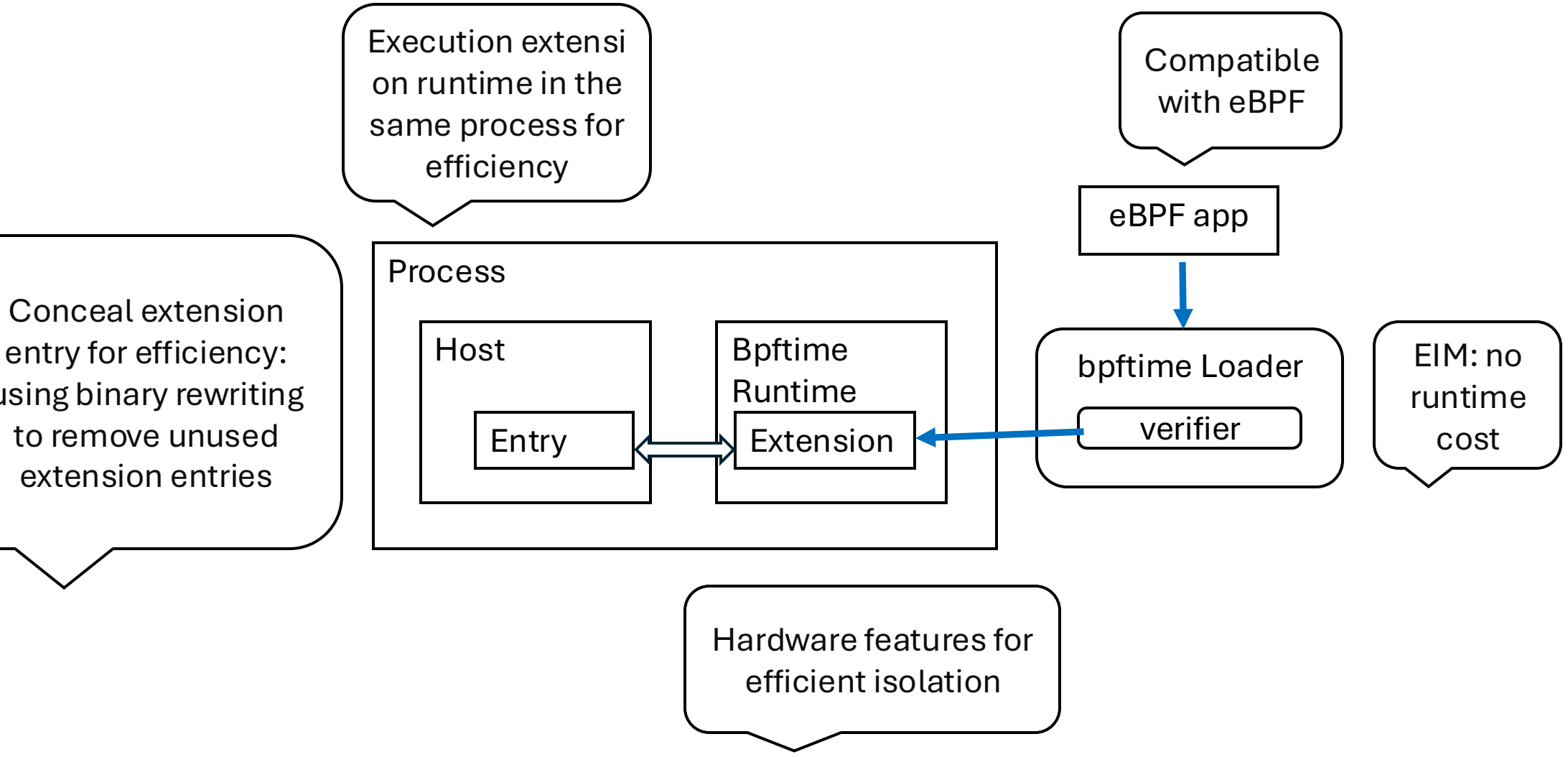
Isolation

Stage 1: Development



Stage 2: Deployment

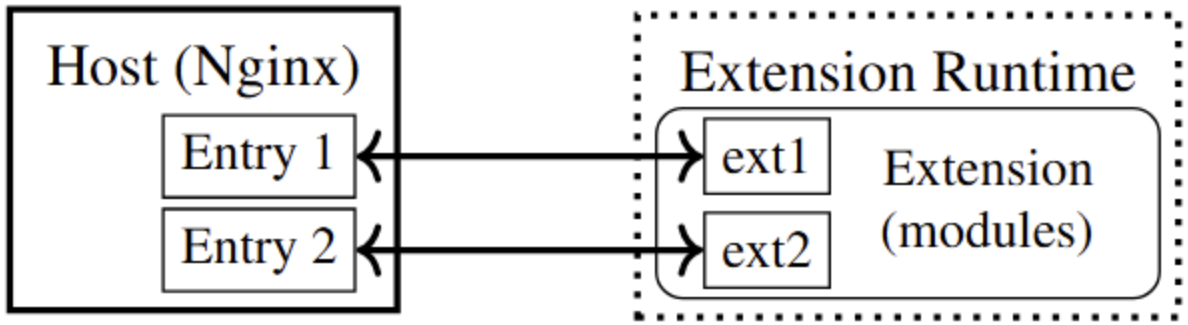


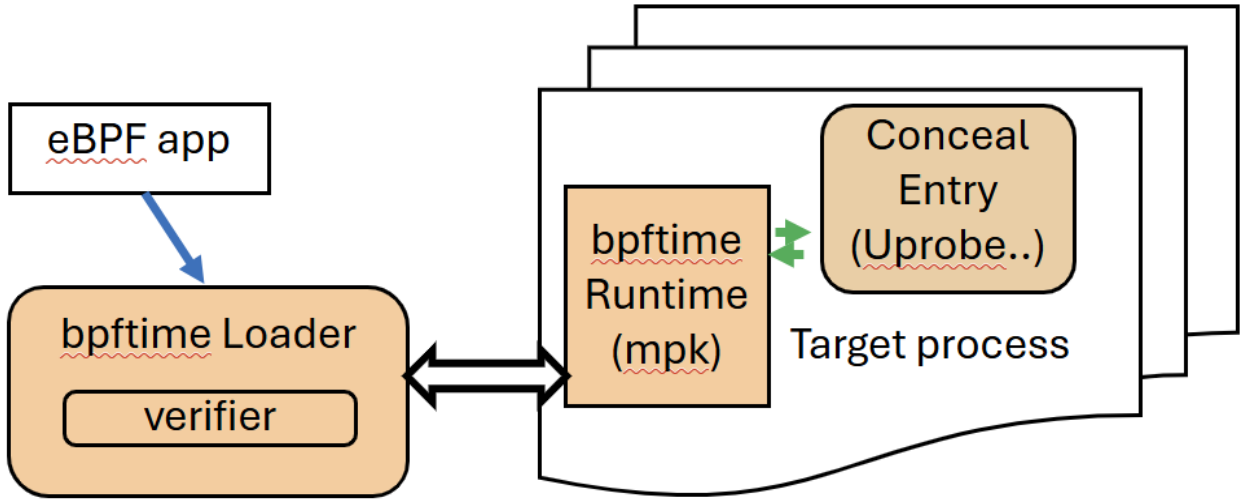

Execution extensi on runtime in the same process for efficiency
Compatible with eBPF
eBPF app
Process
Host
Entry
Bpftime Runtime
Extension
bpftime Loader
verifier
EIM: no runtime cost
Conceal extension entry for efficiency: using binary rewriting to remove unused extension entries
Hardware features for efficient isolation

bpftime: userspace eBPF extension framework

Provide efficient solution to enforce EIM and isolation

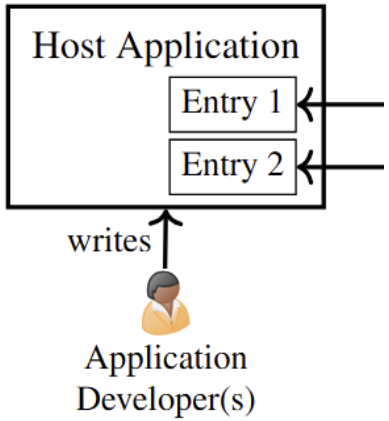
- Verification: EIM: no runtime cost
- Execution extension runtime in thesame process forefficiency
- Conceal extension entry for efficiency: using binary rewriting to remove unused extension entries
- Hardware features for efficient isolation
- Compatibility





EIM: Development-Time Specification

- Developers annotate code for capabilities
- Automatically extracted into capability manifest

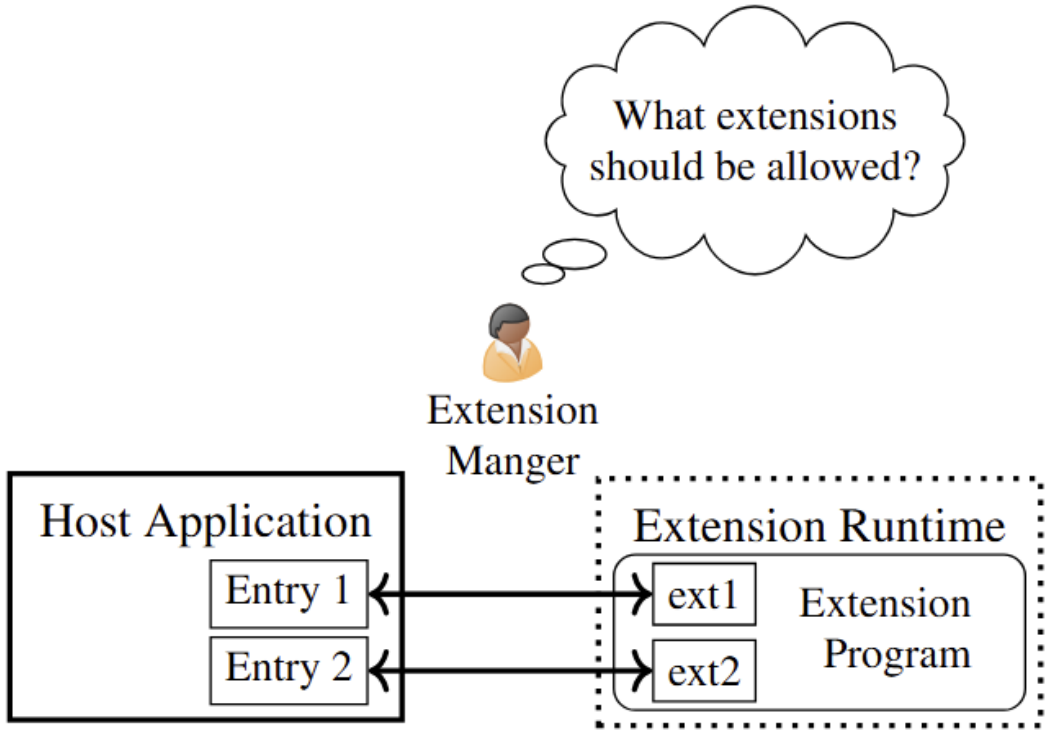



```
EIM_STATE_DEFINE(readPid, read, ngx_pid);
EIM_HFUNC_DEFINE_WITH_CONSTRAINTS(
    nginxTime,
    HF_RET_POSITIVE);
EIM_EXTENSION_ENTRY_DEFINE(
    processBegin,
    ngx_http_process_request,
    int,
    struct Request *);
```

EIM: Deployment-Time Specification

- Extension Manager write simple YAML policies to explore interconnectedness/safety trade-offs without recompiling

```
1 Extension_Class(
2  name = "observeProcessBegin",
3  extension_entry = "processBegin",
4  allowed = {instructions<inf, nginxTime,
       readPid, read(r)})
5 Extension_Class(
6  name = "updateResponse",
7  extension_entry = "updateResponseContent"
8  allowed = {instructions<inf, read(r),
       write(r)})
```



contribution

- Background & motivation: Extensions
- Extension Interface Model (EIM): Fine-grained Interface
- bpftime Runtime: safety & performance
- Evaluation
- ..... add more and make it a contribution

Q & A?

- GitHub repo:
- Get started:

```
bpftime load ./example/malloc/malloc
bpftime start nginx -c ./nginx.conf
```

Contribution

- **Extension Interface Model (EIM)**: Solution to navigate fine-grained safety-interconnectedness trade-off & Two-Phase Specification: Development-Time (by Developer) Deployment-Time (by Manager)
- **bpftime Runtime**: An userspace eBPF runtime implemented EIM with isolation and efficiency

- Evaluation: 6 usecases and Up to 6x less overhead

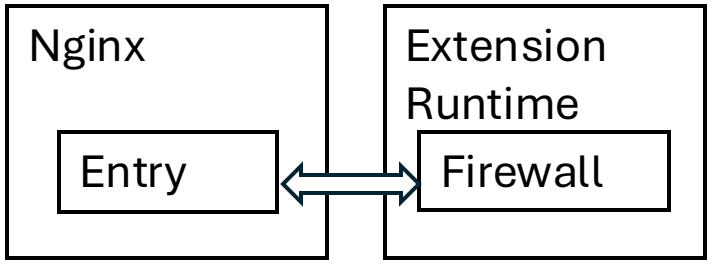
Nginx firewall example

Offline:

- Writes custom firewall logic using nginx helper functions
- Loads their extension at an extension entry for request processing

At runtime

- Nginx jumps to the extension runtime when reaches the extension entry
- Extension runtime execute the extension entry and return to Nginx



Six Real-World Use Cases

- **Nginx Firewall**
- **Redis Durability**
- **FUSE Metadata Cache**
- **DeepFlow**
- **Syscount**
- **Sslsniff**

(grounp and figture?)

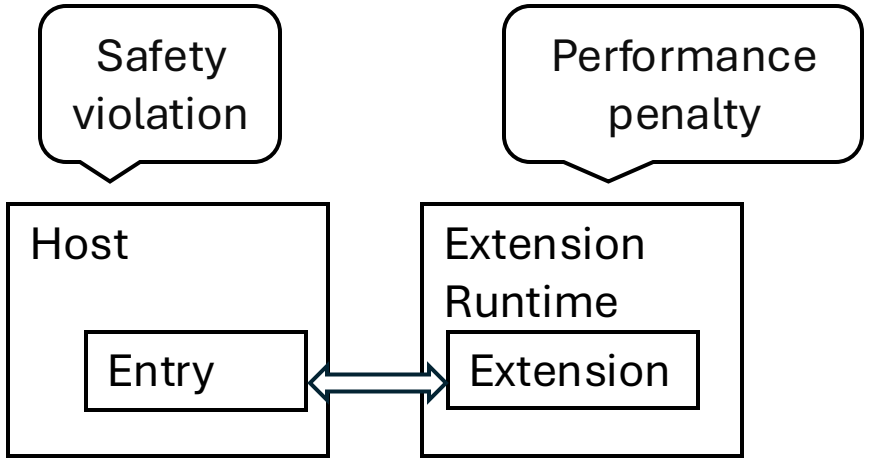
GitHub: https://github.com/eunomia-bpf/bpftime

Extension Problems

- **Real-world safety violations:**

Bilibili CDN outage, Apache buffer overflow, Redis RCE

- **Performance penalty**: WebAssembly/Lua impose 10-15% overhead



contribution

- Background & motivation: Extensions
- Extension Interface Model (EIM): Fine-grained Interface
- bpftime Runtime: safety & performance
- Evaluation
- ..... add more and make it a contribution

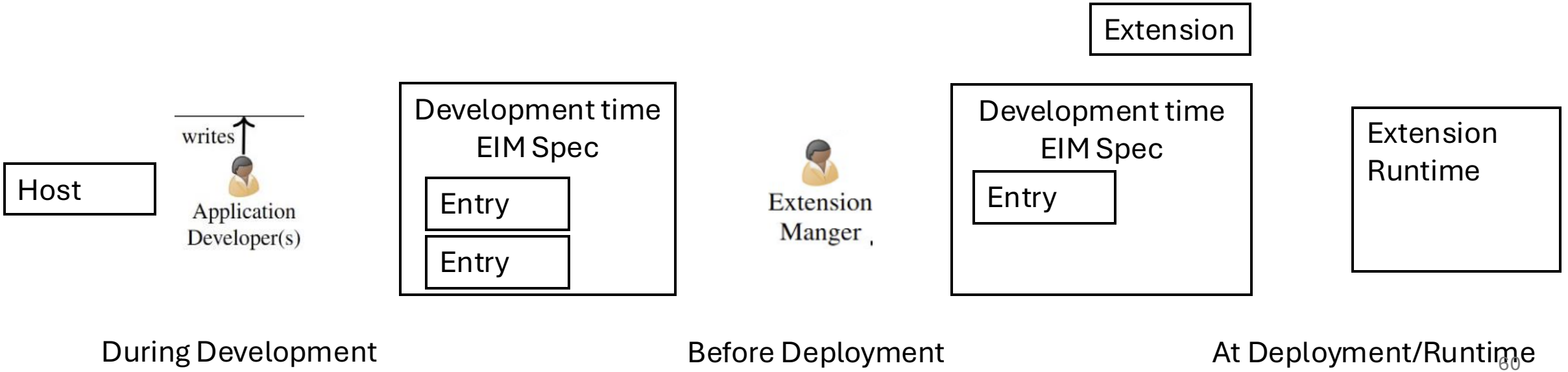
Q & A?

- GitHub repo:
- Get started:

```
bpftime load ./example/malloc/malloc
bpftime start nginx -c ./nginx.conf
```

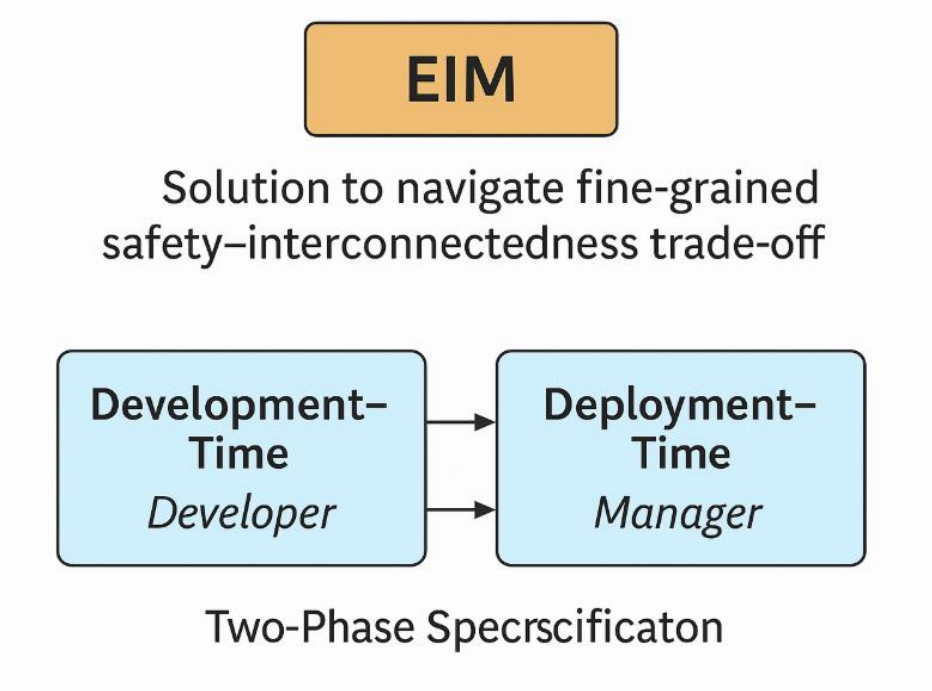
EIM: Extension Interface Model

- Goal: enable fine-grained safety/interconnectedness trade-offs
- Challenge: supporting per deployment tradeoffs
- Solution:
  - Two-Phase Specification (Development-Time and deployment-Time)
  - Model all resources as capabilities

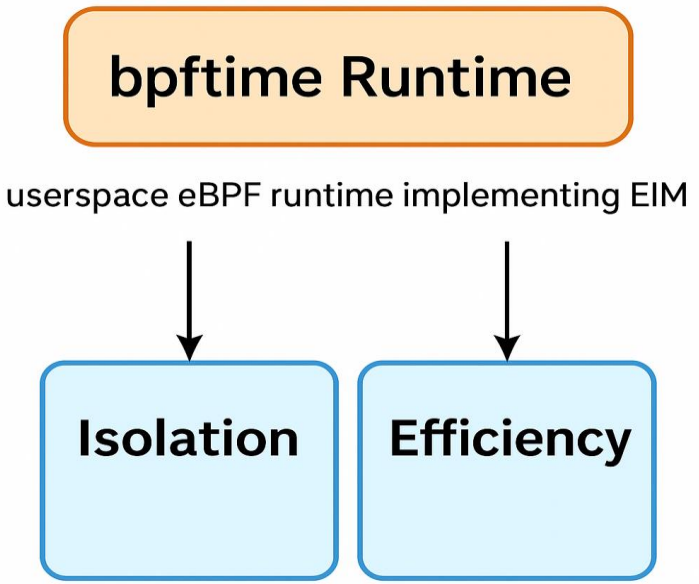


Contributions

- **Extension Interface Model (EIM)**



- **bpftime**



- Up to 6x less overhead than current state-of-the-art!

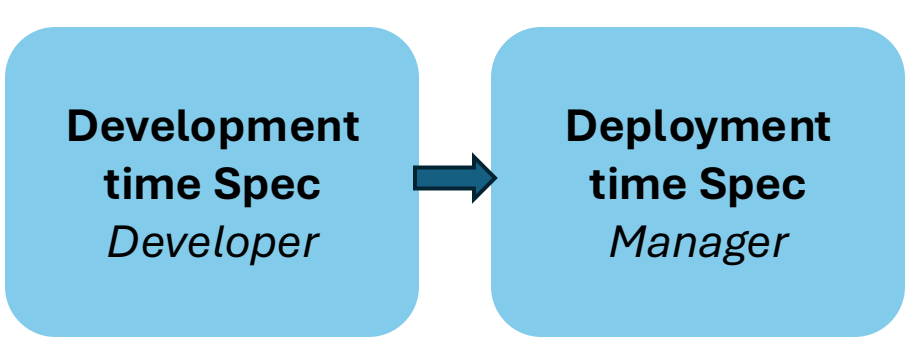
Contributions

Extension Interface model (EIM)

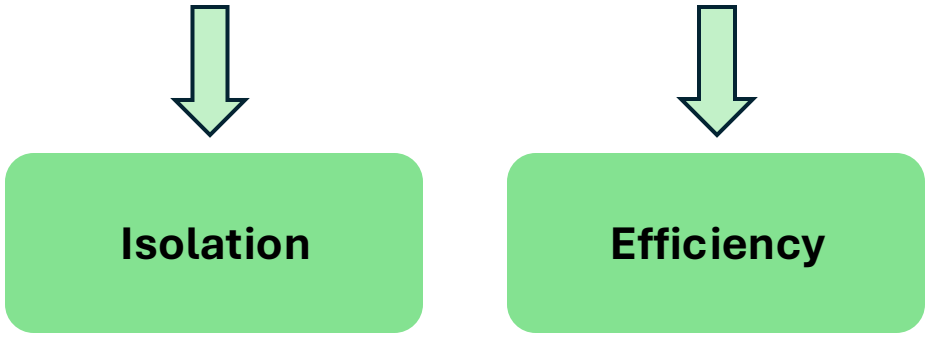
Navigate fine-grained safety/interconectedness trade-offs for extensions

Bpftime runtime

Efficient support for EIM and isolation through userspace eBPF runtime



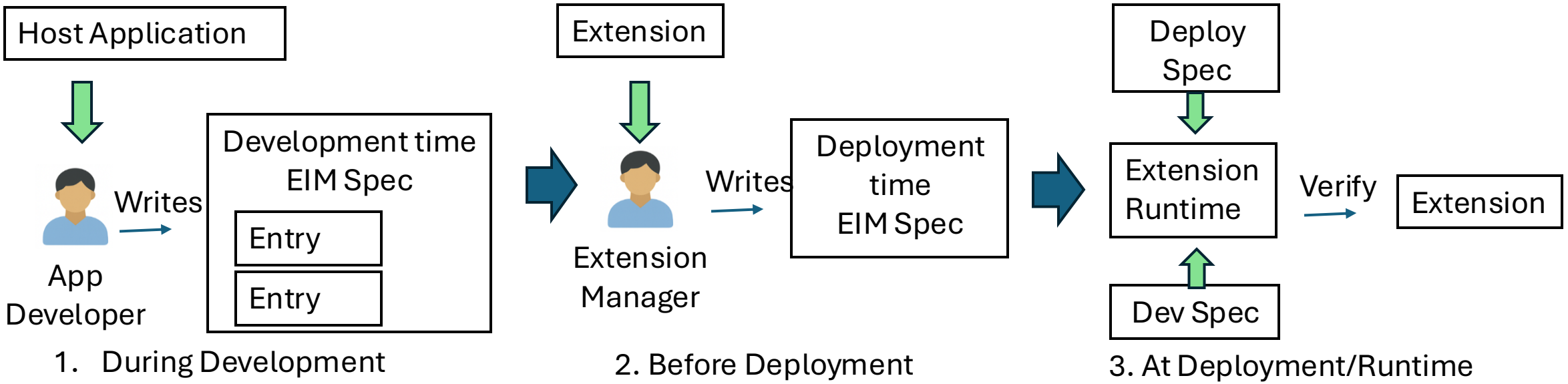
Two phase Specification



- Up to 6x less overhead than current state-of-the-art!

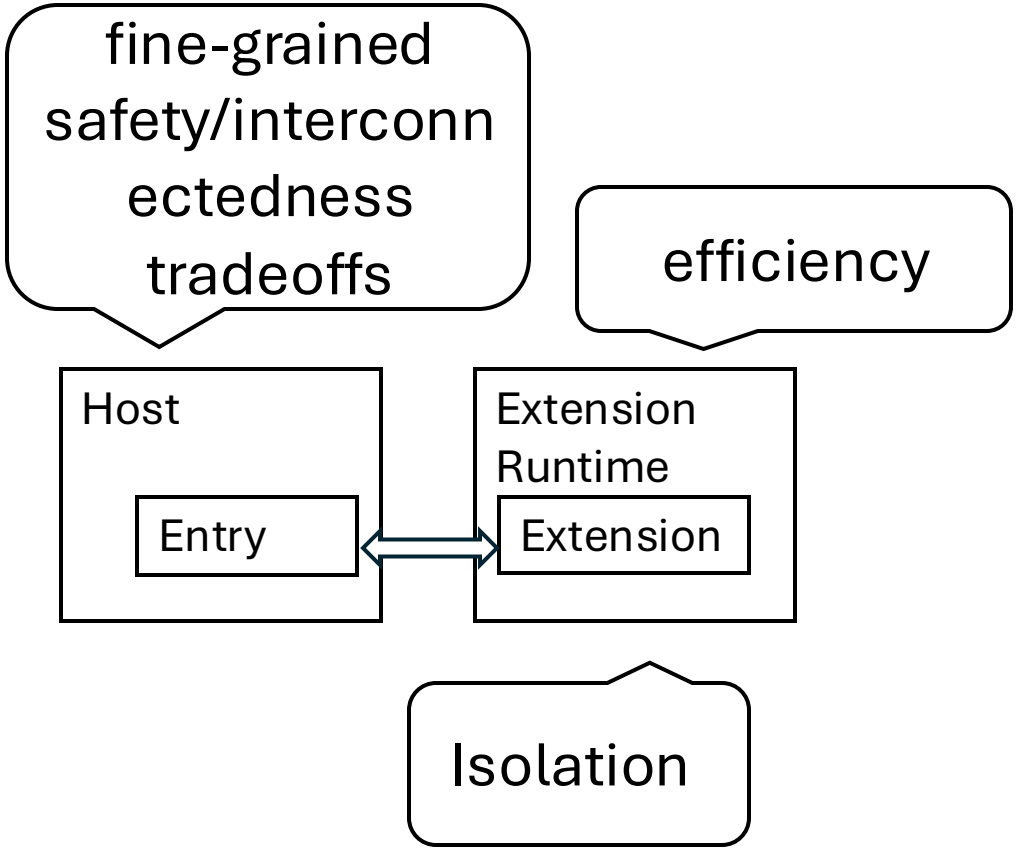
EIM: Extension Interface Model

- Goal: enable fine-grained safety/interconnectedness trade-offs
- Challenge: supporting per deployment tradeoffs
- Solution:
  - Two-Phase Specification (Development-Time and deployment-Time)
  - Model all resources as capabilities



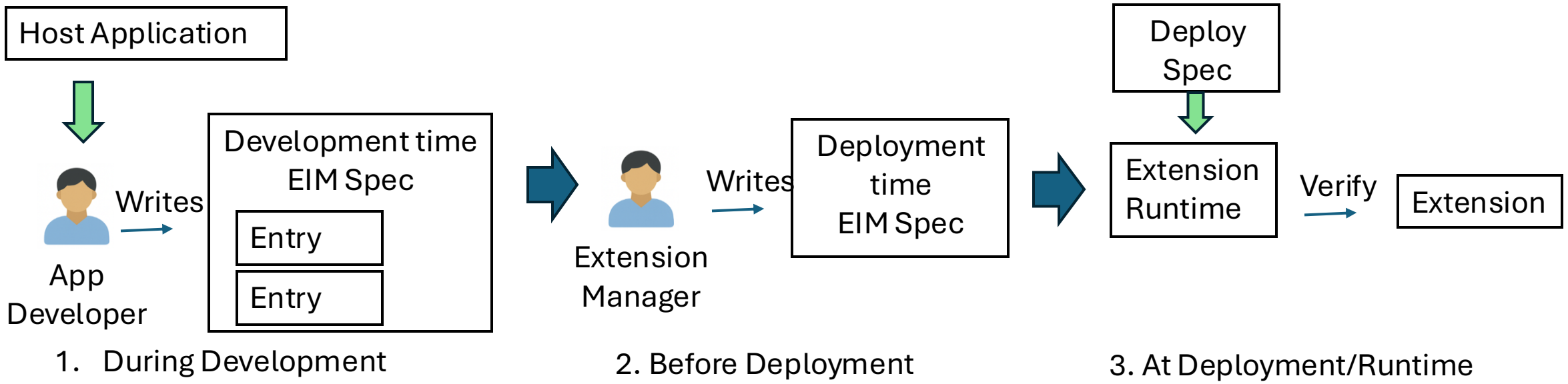
Extension Requirements

- Fine-grained safety and interconnected ness trade-offs
- Isolation:
- Efficiency:



EIM: Extension Interface Model

- Goal: enable fine-grained safety/interconnectedness trade-offs
- Challenge: supporting per deployment tradeoffs
- Solution:
  - Two-Phase Specification (Development-Time and deployment-Time)
  - Model all resources as capabilities



Extension Problems & requirements

Real-world safety violations:

- Bilibili CDN outage, Apache buffer overflow, Redis RCE

Performance penalty:

- WebAssembly/Lua impose 10-15% overhead

Requirements:

- Fine-grained safety and interconnectedness trade-offs
- Isolation
- Efficiency