ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

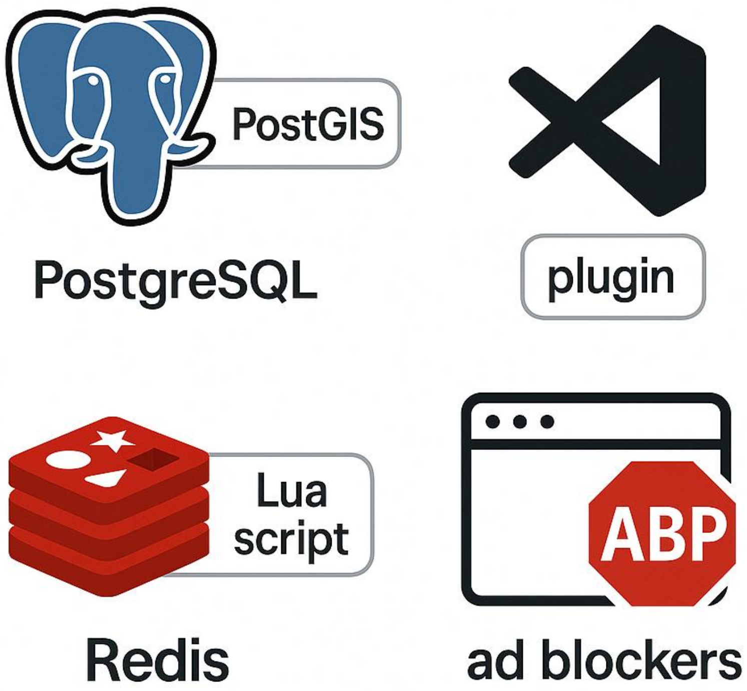
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

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?

- Easier to maintain and update

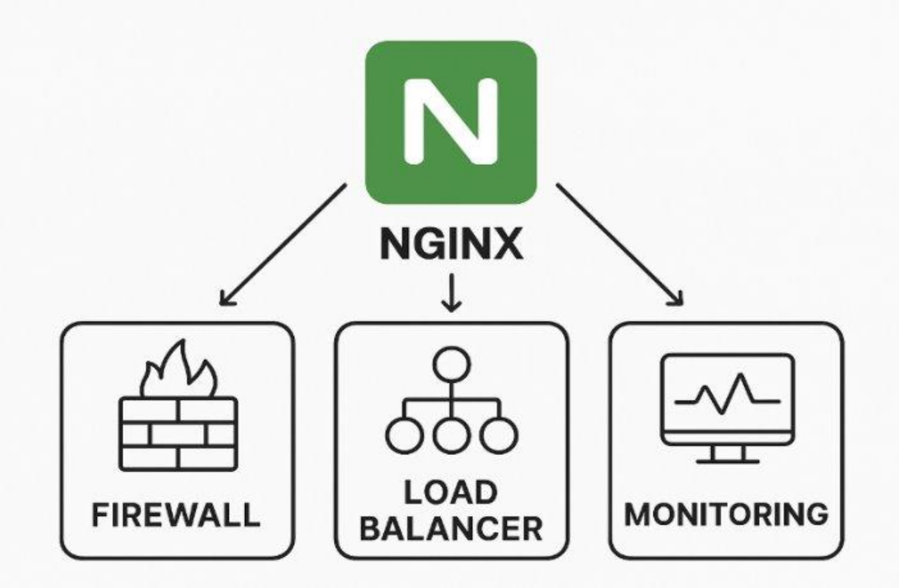
Nginx firewall example

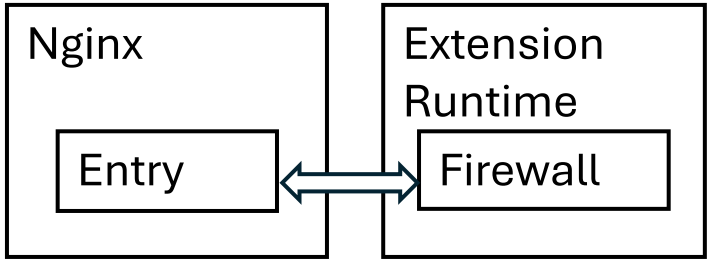
Before deployment, user:

- 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.




Nginx

Entry

Extension Runtime

Firewall

Extension Problems & requirements

Real-world safety violations:

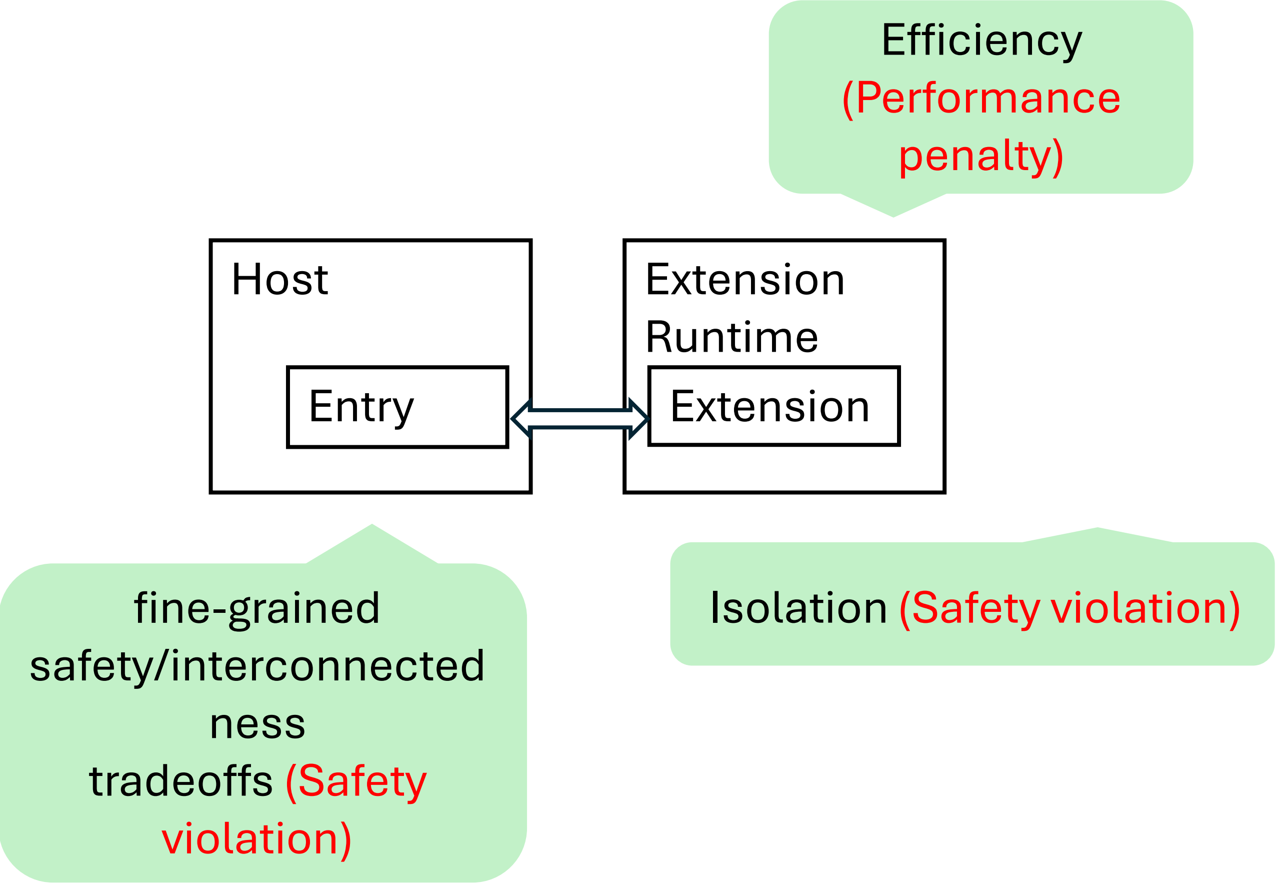
- Bilibili CDN outage, Apache buffer overflow, Redis RCE

Performance penalty:

- WebAssembly/Lua impose 10-15% overhead

Requirements:

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



Efficiency (Performance penalty)

Host

Entry

Extension Runtime

Extension

fine-grained safety/interconnectedness tradeoffs (Safety violation)

Isolation (Safety violation)

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

Efficient support for EIM and isolation through userspace eBPF runtime

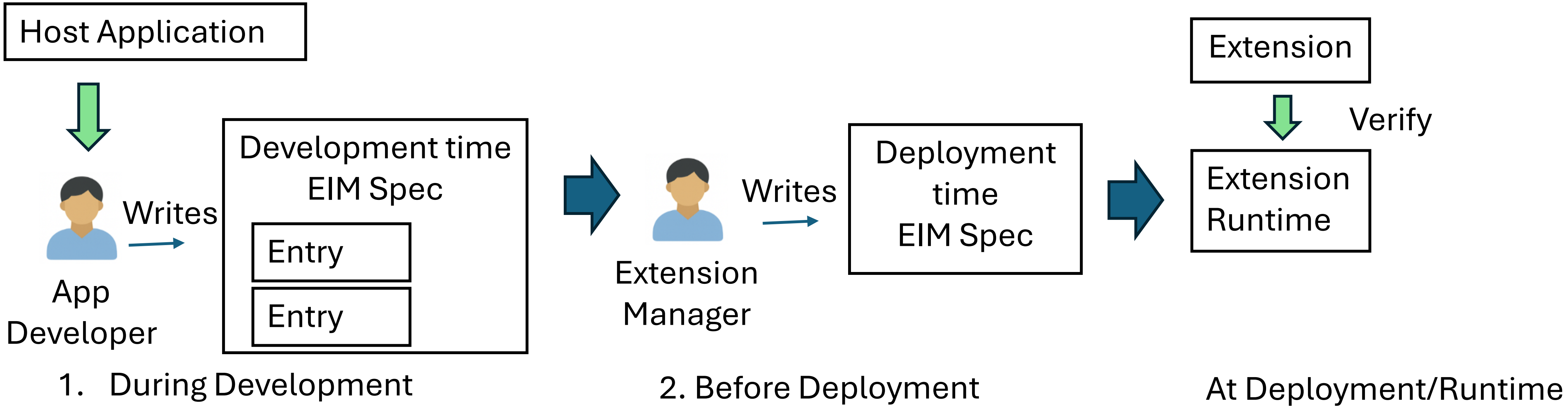
Up to 6x less overhead than current state-of-the-art!

Outline

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

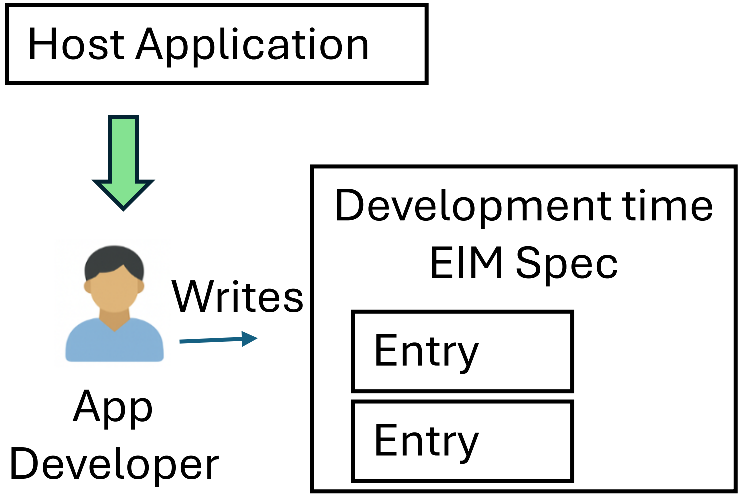
EIM: Extension Interface Model

- Challenge:
  - Enable fine-grained safety/interconnectedness trade-offs
- Solution:
  - Two-phase specification (Development Time and Deployment Time)
  - Model all resources as capabilities

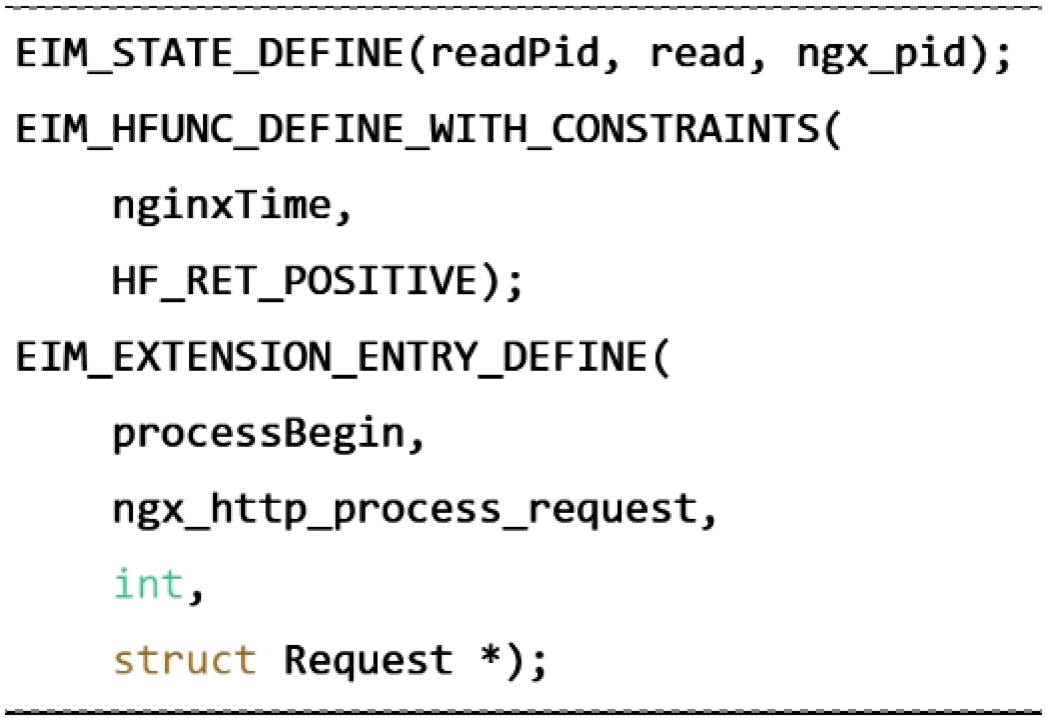


EIM: Development Time Specification

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



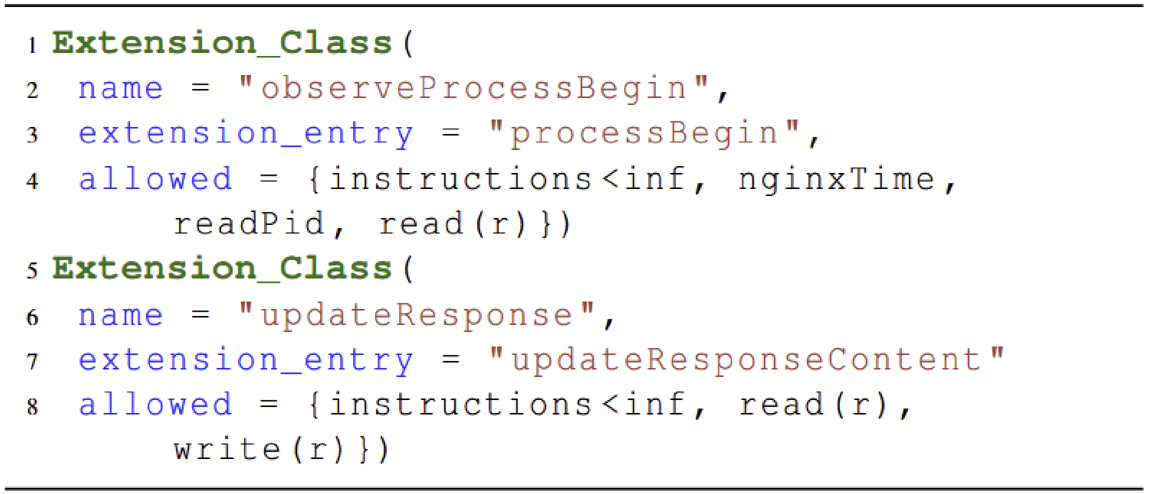
1. During Development



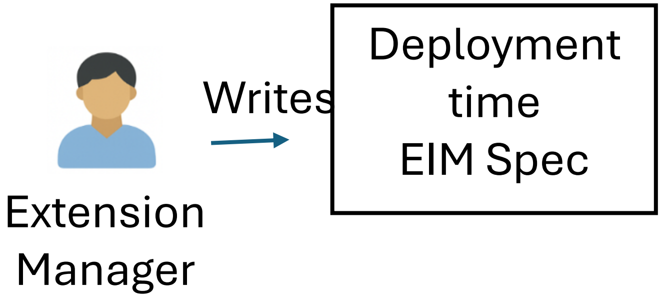
```
EIM_STATE_DEFINE(readPid, read, ngx_pid);
EIM_HFUNC_DEFINE_WITH_CONSTRAINTS(
    nginxTime,
    HF_RET_POSITIVE);
EIM_EXTENSION_ENTRY_DEFINE(
    processBegin,
    ngx_http_process_request,
    int,
    struct Request *);
```

EIM: Deployment Time Specification

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



```
Extension_Class(
 name = "observeProcessBegin",
 extension_entry = "processBegin",
 allowed = {instructions<inf, nginxTime,
      readPid, read(r)})
Extension_Class(
 name = "updateResponse",
 extension_entry = "updateResponseContent"
 allowed = {instructions<inf, read(r),
      write(r)})
```



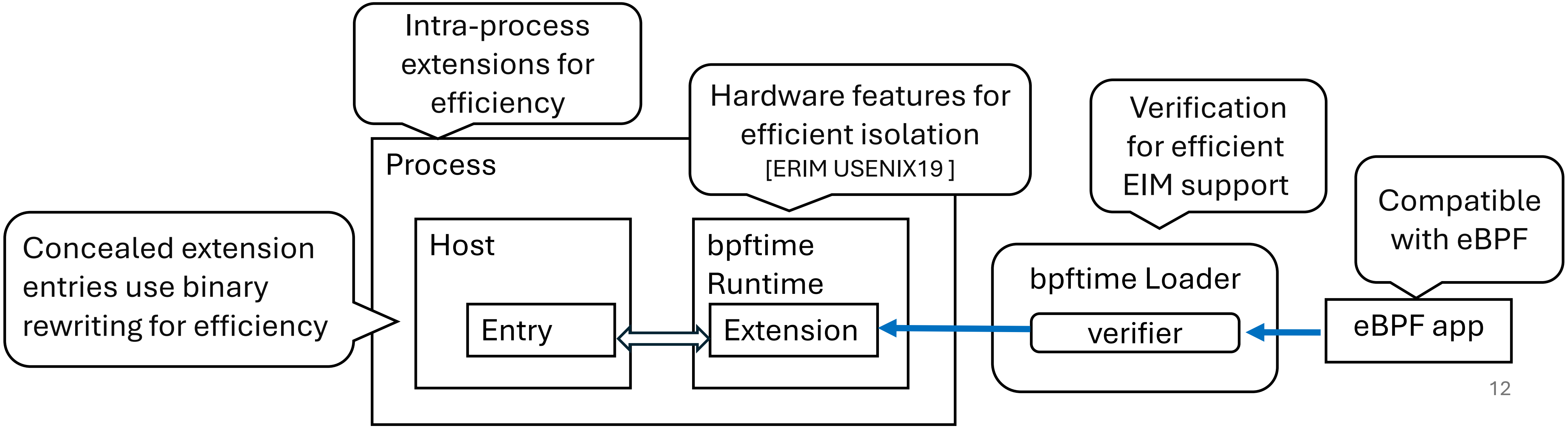
2. Before Deployment

Outline

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

bpftime: userspace eBPF extension framework

- Challenge:
  - efficiently support EIM and isolation
- Solution:
  - exploit eBPF-style verification, binary rewriting, and hardware features

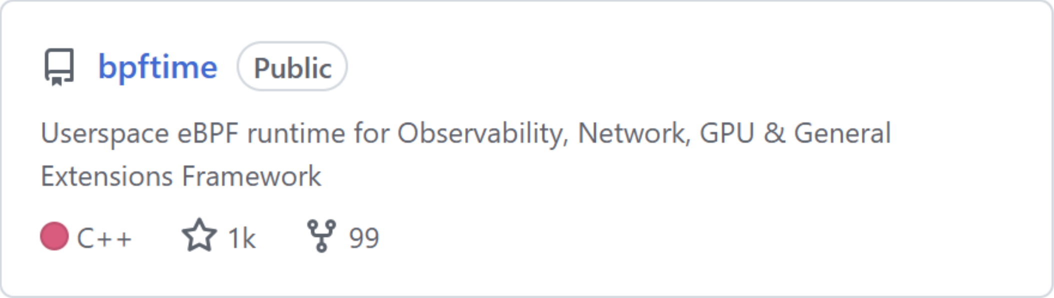


Intra-process extensions for efficiency
Hardware features for efficient isolation [ERIM USENIX19]
Verification for efficient EIM support
Compatible with eBPF
Process
Concealed extension entries use binary rewriting for efficiency
Host
Entry
bpftime Runtime
Extension
bpftime Loader
verifier
eBPF app

Outline

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

Six Real-World Use Cases



bpftime Public

Userspace eBPF runtime for Observability, Network, GPU & General Extensions Framework

● C++ ☆ 1k 99

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

Customization

- Nginx Firewall
- Redis Durability
- FUSE Metadata Cache

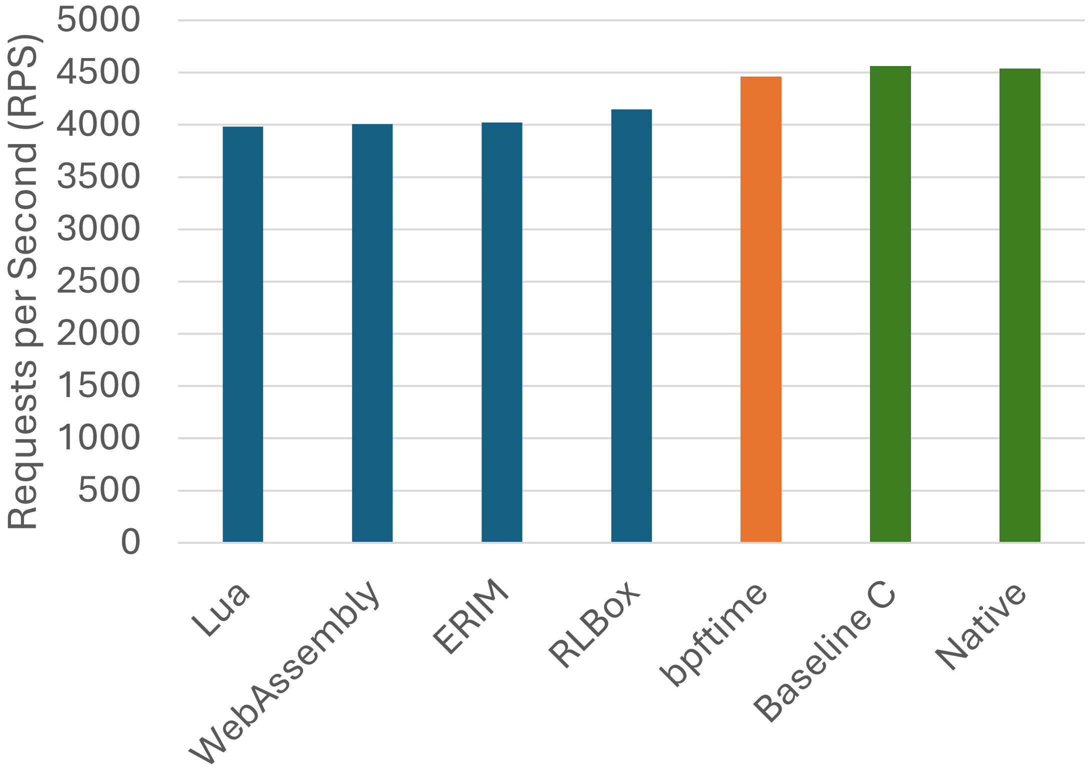
Observability

- DeepFlow
- Syscount
- Sslsniff

Customization: Nginx firewall

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

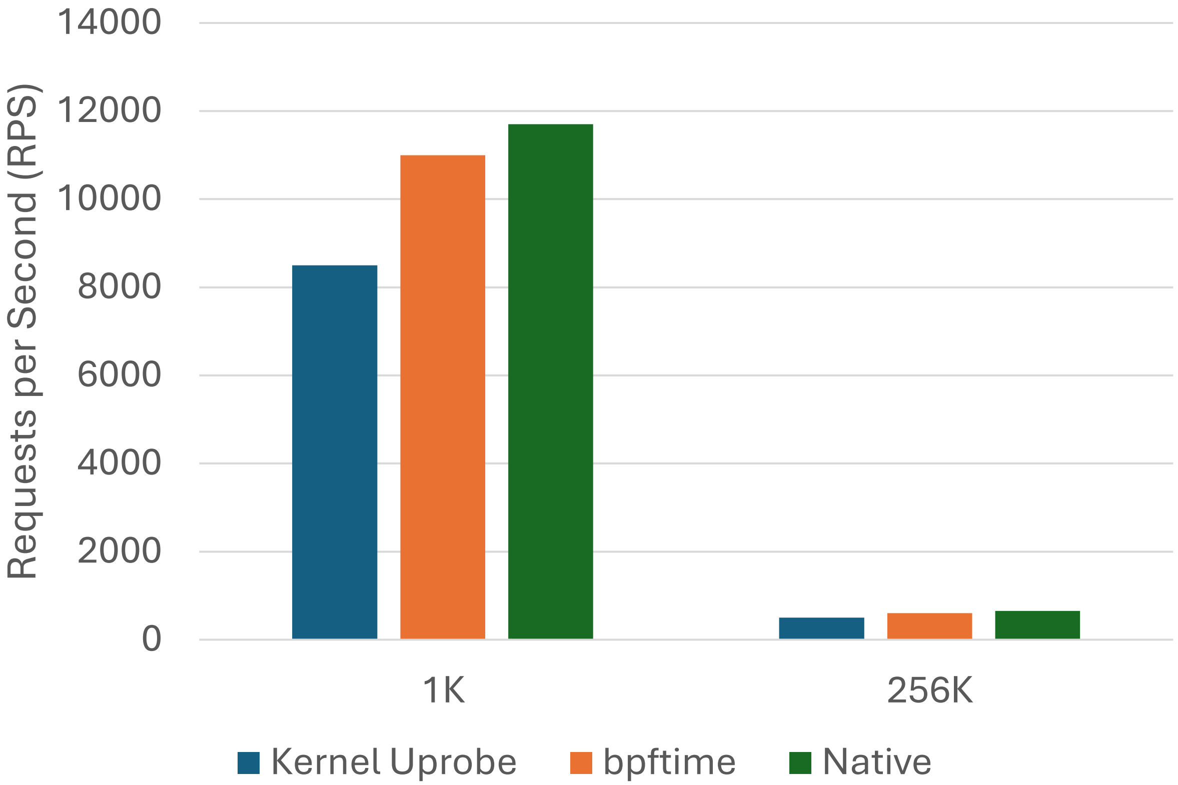
better



Observability: sslsniff

- Maximum 21% less overhead than kernel eBPF

better



Contributions

Questions?

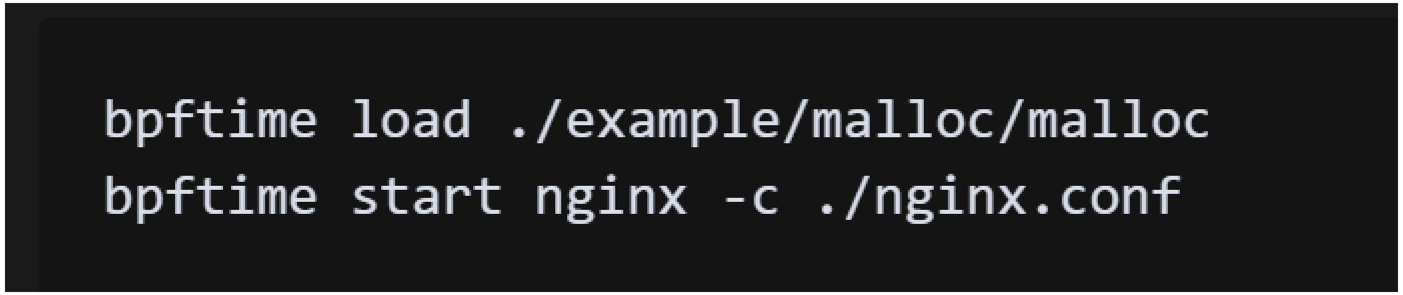
Extension Interface model (EIM)

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

Bpftime

Efficient support for EIM and isolation through userspace eBPF runtime

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

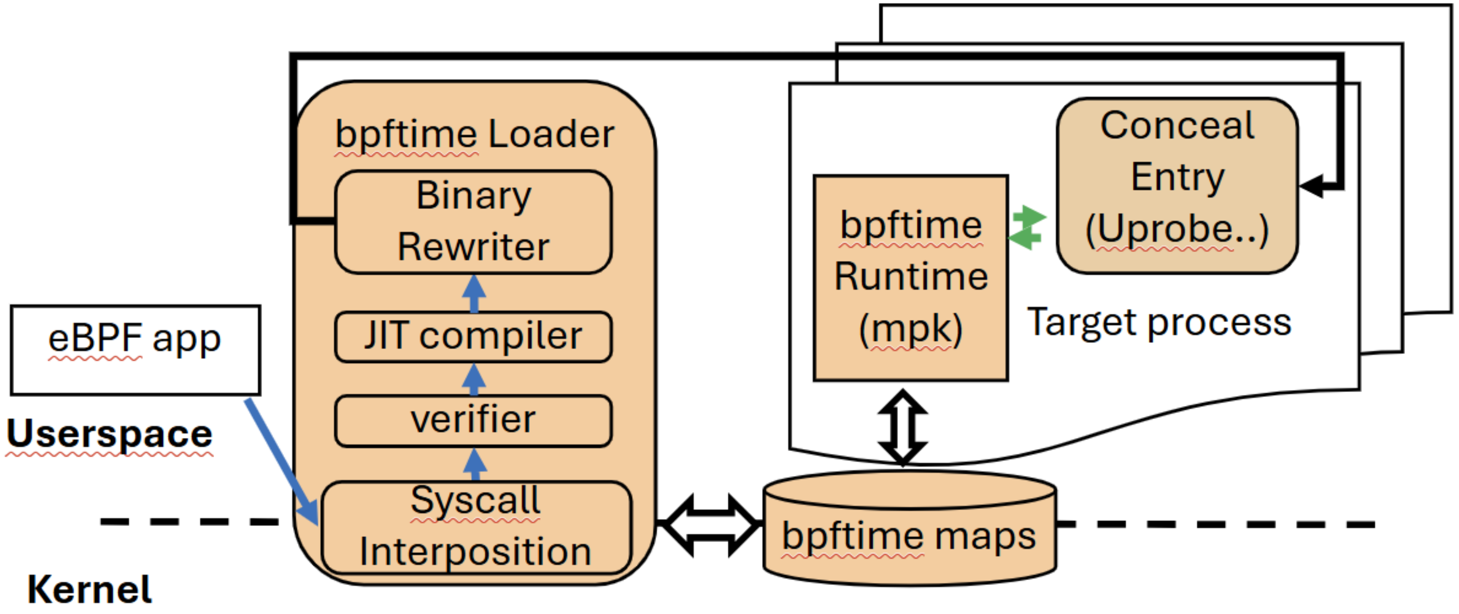


```
bpftime load ./example/malloc/malloc
bpftime start nginx -c ./nginx.conf
```

Backup

bpftime: userspace eBPF extension framework

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


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