# Extending Applications Safely and Efficiently

Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, XiaoZheng Lai, Dan Williams, Andrew Quinn

OSDI 25

## Intro & Background

**Motivation for Extension Frameworks**

- Many modern applications (e.g., web servers, databases, browsers) support plugin-style extensions for:
  - **Performance optimization** (e.g., custom query optimizers [1, 38])
  - **New functionality** (e.g., domain-specific logic [46, 48])
  - **Security hardening** (e.g., dynamic taint analysis [44, 59])
  - **Observability and debugging** (e.g., performance tracing, failure analysis [37, 43, 58, 74])
- A typical extension model provides:
  1. **Extension Entrypoints**—predetermined "hooks" in the host application where the extension can run.
  2. **Runtime Loading**—when a thread reaches a hook, execution jumps into the extension and returns afterwards.

**Key Challenges**

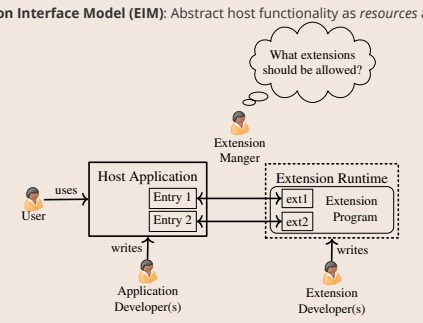
1. **Interconnectedness vs. Safety**
   - Extensions need access to application state (interconnectedness), but that must be limited to avoid buggy or malicious behavior that could crash or compromise the app (safety).
   - Existing frameworks rarely give fine-grained control over exactly which parts of the application an extension may read or write.
2. **Isolation vs. Efficiency**
   - Process- or container-level isolation is heavy: frequent context switches impose large performance penalties.
   - Software fault isolation (SFI) can be lighter weight but still incurs significant overhead.
   - The ideal is **in-process**, low-overhead isolation with minimal runtime cost.

**Shortcomings of Existing Approaches**

- **Native execution** (e.g., LD_PRELOAD, dynamic binary rewriting) is fast but offers no real isolation or safety.
- **Software Fault Isolation (SFI)** (e.g., Native Client, WebAssembly, Lua, RLBox, XFI) enforces isolation but either lacks fine-grained permission controls or still incurs nontrivial overhead.
- **Subprocess or micro-VM** (e.g., lwC, Shreds, Orbit, Wedge) can isolate, but often require application modifications and incur high IPC/context-switch costs.
- **eBPF uprobes** isolate extensions reasonably well but cannot enforce per-hookpoint permissions in user space, and each user-side hook requires a kernel trap, hurting performance.

**Our Solution at a Glance**

- **Extension Interface Model (EIM)**: Abstract host functionality as *resources* and *capabilities*.



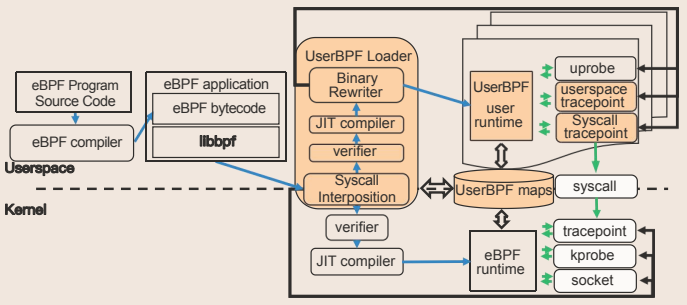

- **Extension Entry Declarations**
  - Annotate the host functions that become hookpoints for extensions, including name and signature.
  - Example:

```
Extension_Entry(
  name           = "processBegin",
  extension_hook = "ngx_http_process_request",
  prototype      = (Request *r) -> int
);
Extension_Entry(
  name           = "updateResponseContent",
  extension_hook = "ngx_http_content_phase",
  prototype      = (Request *r) -> int*
);
```

  - These annotations are extracted at compile time to build the *development-time EIM specification*.
2. **Deployment-Time EIM Configuration**
   - For each declared Extension Entry, an **Extension Class** specifies:
     1. A capability set (subset of the declared state/function capabilities).
     2. Optional resource limits (e.g., "instructions < ∞," "memory < 8 MB").
   - Example YAML for two Nginx hooks:

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

   - An extension manager (person or script) writes this deployment-time policy to enforce "least privilege" at runtime.




### 2.1 EIM Model

1. **Development-Time EIM Specification**
   - **State Capabilities**
     - Written as `read(var)` or `write(var)` to specify allowed global variables or struct-field accesses.
     - Example:

```
State_Capability(
  name      = "readPid",
  operation = read(ngx_pid)
);
```

   - **Function Capabilities**
     - Specify which host functions an extension may call, including their prototypes and optional pre/post-conditions.
     - Example:

```
Function_Capability(
  name        = "nginxTime",
  prototype   = (void) -> time_t,
  constraints = { return_value > 0 }
);
```

   - At development time, the application author declares which global variables or functions an extension may access (e.g., `read(pid)`, `write(r->headers)`, `call get_time()`).
   - At deployment time, the extension manager chooses a minimal set of capabilities for each hook (adhering to "least privilege").
- **bpftime Extension Runtime**: A lightweight, in-process framework that:
  - Uses a **verifier** similar to eBPF's to guarantee extension bytecode matches the declared EIM constraints—zero runtime cost in user space.
  - Leverages Intel Memory Protection Keys (MPK) via an ERIM-style approach to enforce in-process isolation without context switches.
  - Applies **binary rewriting** to "conceal" hookpoints so that, if no extension is loaded, there is literally no added cost at that hook.
  - Maintains compatibility with existing eBPF tools (bcc, libbpf, bpftrace) by intercepting and rewriting eBPF loading and Map operations (bpftime Maps).

### 2.2 bpftime Loader

1. **eBPF Syscall Interception**
   - bpftime intercepts standard eBPF loading syscalls (`bpf()`, map-related calls) in user space.
   - It implements a user-space shim that translates those requests into calls to the real kernel eBPF API via UNIX-domain sockets or file descriptors.
   - This maintains full compatibility with bcc, libbpf, bpftrace, etc., requiring no kernel code changes.
2. **Verifier**
   - Accepts raw eBPF bytecode and the selected Extension Class's capability constraints.
   - Translates each declared capability (e.g., "can call get_time," "can read pointer r->headers") into a set of clauses or predicates that the eBPF verifier understands.
   - Ensures the loaded bytecode obeys these constraints—no out-of-bounds pointer use, no disallowed calls—purely at load time, so there is zero runtime-execution overhead.
3. **Binary Rewriter**
   - Uses `ptrace` to pause the target process and injects the bpftime runtime library into its address space.
   - Relies on Frida + Capstone to rewrite machine instructions at each declared hookpoint:
     1. At every `uprobe/uretprobe` hook, replace the original instruction(s) with a jump (trampoline) into the extension's entry stub, then execute the original instructions upon return from the extension.
     2. For arbitrary syscall hooks (e.g., `sysenter`), use a "z-poline" instructionation technique to insert jumps with minimal instruction-overhead.
   - **Concealed Hookpoints**: If no extension is currently loaded for a given hook, the code at that hookpoint remains exactly as before—no added branches or jumps—so the cost is literally zero when unused.

### 2.3 bpftime Runtime

1. **In-Process Isolation via MPK**
   - Follows an ERIM-style approach using Intel Memory Protection Keys (MPK):
     1. Each extension is assigned a distinct MPK indices; extension code/data pages are tagged with that key.
     2. On entry into the extension, `WRPKRU` flips the PKRU so that pages tagged for that extension become writable/exec-readable.
     3. On exit, `WRPKRU` flips PKRU back so that extension pages become non-writable at user level.
     4. The host application cannot tamper with extension code or data, ensuring strong isolation, all without expensive context switches or syscalls.
   - Fully implemented on x86-64; ARM support can be added via similar ARM PK mechanisms (AMPK).
2. **bpftime Maps (Efficient Data Structures)**
   - Unlike native eBPF Maps (which require syscalls), bpftime Maps operate entirely in user space, providing zero-syscall access:
     - **Local (Non-shared) Mode**: Single-process, per-extension local hash tables.
     - **Inter-process Shared Mode**: Shared memory segments for multiple processes to share state.
     - **Kernel-Backed Mode**: Hybrid mode where the map is kernel-visible for use by both kernel probes and user-space extensions.
   - Supported data structures: Hash Map, Array, LPM Trie, Ring Buffer, Perf Event Array, per-CPU variants, etc.
   - Internally optimized with lock-free or per-CPU data structures so that typical operations (lookup, update, delete) are an order of magnitude faster than native eBPF Maps.

## Implementation

## Use Cases

We implemented and evaluated six representative scenarios to demonstrate bpftime's security, customizability, observability, and performance advantages.

1. **Nginx Plugin (Web Server Security Extension)**
   - **Goal**: Deploy a firewall extension in Nginx's forward-proxy mode to block SQL-injection and XSS attacks.
   - **Implementation**:
     - At the `processBegin` hook, define an Extension Class that only allows reading/writing the current `Request` object (`read(r)`, `write(r)`).
     - The extension inspects the URL; if malicious, return a 404 immediately.
   - **Result**: When the extension is loaded, bpftime adds only ~2% overhead—much lower than Lua, WebAssembly, ERIM, or RLBox.
2. **sslsniff (Distributed HTTPS Tracing)**
   - **Goal**: Observe encrypted TLS/SSL traffic end-to-end for distributed tracing/observability.
   - **Conventional eBPF Approach**: Uprobes on OpenSSL's `SSL_read`/`SSL_write` cause up to ~30% throughput drop.
   - **bpftime Implementation**:
     - Automatically place hookpoints at each uprobe site.
     - Create an "Observability" Extension Class that permits reading pointers needed to record metadata.
     - Write trace records into a shared bpftime Map only if constraints are met.
   - **Measured Results (Figure 8)**:
     - eBPF uprobes: up to 28.06% throughput reduction.
     - bpftime: up to 7.41% throughput reduction.
3. **Syscount (Per-Process Syscall Counting)**
   - **Goal**: Count system calls *only* from a target process rather than all processes.
   - **Conventional Approach (bcc)**: Place kprobes on every process; filter in user space—incurs system-wide overhead.
   - **bpftime Implementation**:
     - Hook only `sysenter`/`sysexit` for the target process using an "Observability" Extension Class.
   - **Measured Results (Figure 9)**:

| Scenario | Throughput (RPS) |
|---|---|
| Native (no hooking) | 19,705 |
| Kernel uprobe (unfiltered) | 17,676 (−10.24%) |
| bpftime for target process (filtered) | 19,042 (−3.36%) |
| Kernel uprobe (with user-side filter) | 17,817 (−9.57%) |
| bpftime (unmonitored processes) | ~19,800 (≈ native) |

   - bpftime introduces ~3.36% overhead to the monitored process and zero overhead to others; native eBPF imposes ~10% everywhere.
4. **DeepFlow (Microservices Observability Platform)**
   - **Goal**: Provide end-to-end tracing across kernel and user space for Go-based microservices.
   - **Conventional eBPF Implementation**: Uprobes on many Go runtime functions cause up to 50% drop in throughput.
   - **bpftime Implementation**:
     - Modify ~10 lines of extension code to reuse automatic hook injection.
     - Use an Observability Class to limit capabilities to safe reads of call stacks/IDs.
   - **Measured Results (Figure 6)**:

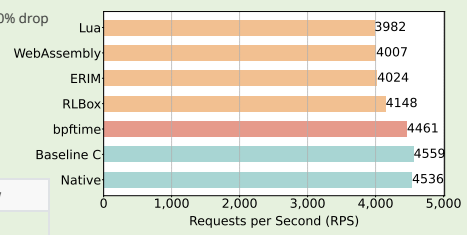
| Workload | Native Throughput | eBPF DeepFlow | bpftime DeepFlow |
|---|---|---|---|
| Small Replies | 250 k RPS | 115 k RPS (−54%) | 170 k RPS (−32%) |
| Large Replies | 47 k RPS | 21 k RPS (−55%) | 31 k RPS (−34%) |

   - bpftime delivers ≥1.5× throughput of eBPF DeepFlow in all cases.
5. **FUSE Caching (User-Space File System Cache)**
   - **Goal**: Speed up FUSE-based user-space file systems (Passthrough, LoggedFS).
   - **Conventional FUSE**: Each I/O involves a user↔kernel↔FS transition, incurring high latency.
   - **bpftime Implementation**:
     - Hook `open`, `close`, `getdents`, `stat` syscalls to maintain an in-user-space cache (bpftime Map).
     - Kernel kprobes watch `unlink` to keep cache coherent.
   - **Measured Latency (Table 2)**:

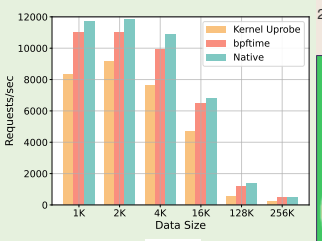
| Scenario | Native FUSE Latency (ms) | bpftime Cache Latency (ms) | Speedup |
|---|---|---|---|
| Passthrough, `fstat` | 3.65 | 0.176 | ≈ 20.8× |
| LoggedFS, `fstat` | 7.40 | 0.184 | ≈ 40.2× |
| LoggedFS, `openat` | 17.0 | 0.074 | ≈ 229.7× |
| Passthrough, `find` | 5.1 | 1.6 | ≈ 3.19× |

6. **Redis Durability Tuning (Custom Persistence Strategies)**
   - **Background**: Redis default persistence policies:
     1. **no AOF**: No durability—crashes lose all data.
     2. **everysec**: fsync every second—can lose ≈ $10^4$ updates on crash.
     3. **alwayson**: fsync on every write—≈ 6× performance penalty.
   - **Goal**: Provide user-space extensions that batch I/O and delay fsyncs for a better performance-durability tradeoff.
   - **Implementation Steps**:
     1. Add three new functions in Redis (for batching writes and deferred fsync).
     2. Annotate relevant write/fsync callsites with bpftime hooks.
     3. Define three Extension Classes:
        - **Batch I/O**: Buffer up to *b* writes before invoking `fsync`, losing at most *b* updates on crash.
        - **Delayed-fsync**: Only invoke `fsync` if a previous `fsync` is still in flight—losing at most 2 updates.
          - **Fast-notify Optimization**: Use a shared counter between user and kernel to skip redundant syscalls when no new fsync is required.
   - **Measured Performance (Figure 7)**:

| Configuration | Throughput (req/s) | Notes |
|---|---|---|
| no AOF | 87 k | (no durability) |
| everysec | 72 k | Might lose ≈ $7.2\times10^4$ updates on crash |
| alwayson | 13 k | (fsync on every write) |
| Batch 1 | ≈ 19 k | Loss ≤ 1 update |
| Batch 3 | ≈ 43 k | +1.7× vs. everysec, loss ≤ 3 updates |
| Batch 12 | ≈ 48 k | +1.9× vs. everysec, loss ≤ 12 updates |
| Batch 24 | ≈ 50 k | +2.1× vs. everysec, loss ≤ 24 updates |
| Batch 48 | ≈ 53 k | +2.3× vs. everysec, loss ≤ 48 updates |
| Delayed-fsync | 40 k | ≤ 2 updates lost; ≈ 4.15× faster than alwayson |
| Delayed-fsync + Fast-notify | 65 k | ≤ 2 updates lost; ≈ 10% slower than everysec, but 5 orders of magnitude fewer losses (2 vs. $7.2\times10^4$) |




bpftime-nginx-module




ssl-nginx




jit_execution_times

## Citation

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg.
"ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)."
In Proceedings of the 28th USENIX Security Symposium (USENIX Security '19), pp. 1221–1238, 2019.

Yuzhuo Jing and Peng Huang.
"Operating System Support for Safe and Efficient Auxiliary Execution (Orbit)."
In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22), 2022.

## Affiliation & Supports

eunomia

UNIVERSITY OF CALIFORNIA · LET THERE BE LIGHT · 1868

SHANGHAITECH UNIVERSITY · 2013

SOUTH CHINA UNIVERSITY OF TECHNOLOGY