
# Extending Applications Safely and Efficiently

Yusheng Zheng
*UC Santa Cruz*

Tong Yu
*eunomia-bpf Community*

Yiwei Yang
*UC Santa Cruz*

Yanpeng Hu
*ShanghaiTech University*

Xiaozheng Lai
*South China University of Technology*

Dan Williams
*Virginia Tech*

Andi Quinn
*UC Santa Cruz*

## Abstract

This paper presents the Extension Interface Model (EIM) and bpftime, which together enable safer and more efficient extension of userspace applications than the current state-of-the-art. EIM is a new model that treats each required feature of an extension as a resource, including concrete hardware resources (e.g., memory) and abstract ones (e.g., the ability to invoke a function from the extended application). An extension manager, i.e., the person who manages a deployment, uses EIM to specify only the resources an extension needs to perform its task. bpftime is a new extension framework that enforces an EIM specification. Compared to prior systems, bpftime is efficient because it uses extended Berkeley Packet Filter (eBPF)-style verification, hardware-supported isolation features (e.g., Intel MPK), and dynamic binary rewriting. Moreover, bpftime is easy to adopt into existing workflows since it is compatible with the current eBPF ecosystem. We demonstrate the usefulness of EIM and bpftime across 6 use cases that improve security, monitor and enhance performance, and explore configuration trade-offs.

## 1 Introduction

Developers extend their software to customize it for the needs of a particular deployment. For example, extensions can improve application performance [2, 40], add custom features [48, 50], enhance security [46, 62], and enable application observability for performance monitoring [61, 77] and debugging [39, 45]. Software extensions do not require modifying the original application, which enables customization while ensuring that the deployment can easily integrate maintenance updates from upstream repositories. Many software applications support extensibility, including web browsers [21, 43], HTTP servers [17, 25], text editors [24, 64], and databases [49, 59].

To extend software, a developer defines new logic for the program as a set of extensions, and associates each extension with a specific location in the host application, called an *extension entry*. When a user invokes the application, the system loads the host application and the user's configured extensions. Each time an application thread reaches an extension entry, the thread jumps to the associated extension. It executes the extension in the extension runtime context; once the extension completes, the thread returns to the host at the point immediately after the extension entry.

This paper presents a new approach for specifying the interface between an extension and a host, called the *Extension Interface Model* (EIM), and a new extension runtime system, called bpftime, that together provide safer and more efficient software extensions. EIM and bpftime are motivated by the challenges current extension frameworks face in balancing key extensibility features.

First, current extensibility frameworks struggle to navigate the tradeoff between extension interconnectedness and safety. On the one hand, enabling extensions that perform meaningful work requires *interconnectedness*, i.e., the ability to observe or modify the host application's state and to execute functions defined by the host. On the other hand, application deployments require that their extensions are safe, i.e., that an extension failure cannot harm the health of the host application and the underlying system. Achieving safety requires restricting an extension's behavior to limit both the system resources it consumes (e.g., the memory it uses, the files it opens) and the host interactions it performs (e.g., the host state it reads). Interconnectedness and safety are in tension because interconnectedness often necessitates allowing an extension to modify the host in potentially unsafe ways. We introduce the term *extension manager* to describe the person who is responsible for configuring the extensions on a given deployment and thus must navigate the tradeoff between interconnectedness and safety.

To maximize system safety, an extension manager should follow the principle of least privilege, granting extensions only the minimal set of features their use cases require. For example, a deployment that supports observability extensions for debugging and monitoring may have different interconnectedness/safety needs than one that supports extensions for customizing application behavior. Unfortunately, current

extension frameworks poorly support specifying and enforcing such deployment-specific interconnectedness/safety tradeoffs. Many frameworks (e.g., safe language runtimes [23, 38], NaCl [75]) cannot express an interconnectedness/safety tradeoff. Instead, they rely on applications to enforce their own safety, which is ad hoc and error-prone (see §2). Other frameworks (e.g., lwC [37], RLBox [44], Shreds [11]) do not support fine-grained limits on extensions. They either cannot restrict certain unsafe extension behaviors or cannot do so on a per-extension-entry basis. Finally, some frameworks (e.g., Orbit [31], Wedge [7]) do support fine-grained interconnectedness/safety tradeoffs, but they are not designed for extensibility and require modifying host application source code to impose different tradeoffs.

Our first contribution, EIM, supports fine-grained interconnectedness/safety tradeoffs. EIM's key idea is to represent the extension features needed for interconnectedness or restricted for safety through a single abstraction called a *resource*. For example, a resource can represent an extension's ability to call a host function or read a host variable. EIM represents the ability to use resources with capabilities [55, 60, 71]. An EIM specification is produced by two parties: the original application developer and the extension manager. First, developers define capabilities that represent the resources the host application can provide to extensions, essentially enumerating the extension interconnectedness that the host application supports. Then, during deployment, the extension manager creates extension classes that specify the set of capabilities allowed at a particular extension entry, essentially choosing the interconnectedness/safety tradeoff for each extension entry. In sum, EIM specializes interfaces for fine-grained protection, such as those for access control of OS objects (e.g., SELinux [54]) or browser manifest files [22], to support the fine-grained tradeoffs necessary for software extensions.

EIM specifications are runtime-agnostic, and we could enhance an existing extension framework to enforce them. However, current extension frameworks poorly navigate the tradeoff between three properties: extension safety, as specified in an interconnectedness/safety tradeoff; extension isolation, which prevents a host application from harming an extension and is necessary for security monitoring extensions; and extension efficiency, which requires that extensions execute at near-native speed. Current frameworks are inefficient because they employ heavyweight techniques for isolation and safety. For example, many frameworks (e.g., Orbit [31], lwC [37], Wedge [7]) provide new operating system-level isolation abstractions and require context-switch-like overhead to switch between a host and an extension. Other frameworks (e.g., Wasm [23], NaCl [75]) enforce safety and isolation using software fault isolation (SFI), which is much slower than native execution [29].

Our second contribution is bpftime, a new extension runtime that efficiently supports EIM and extension isolation using two design principles. First, the system uses lightweight approaches to provide extension safety and isolation. It enforces the safety in an EIM specification without any runtime overhead using extended Berkeley Packet Filter (eBPF)-style verification, and it enforces isolation with minimal overhead using ERIM-style intraprocess hardware-supported isolation [66, 72]. Second, bpftime introduces concealed extension entries, which improve efficiency by eliminating runtime overhead from extension entries that are not in use by a running process. Concealed extension entries use binary rewriting [10, 14, 18, 70] to inject an extension entry into a host only when a user loads an associated extension. While prior work uses techniques similar to bpftime's verification, isolation, and rewriting techniques, bpftime is the first system combine them to satisfy EIM's requirements.

Additionally, bpftime is fully compatible with eBPF, streamlining the system's path to adoption. With eBPF compatibility, not only can current users of eBPF extensions (e.g., uprobes) seamlessly adopt bpftime[1], but bpftime extensions can also share state and interact closely with eBPF kernel extensions, thereby supporting extensibility use cases that require extending both the kernel and applications.

We maintain bpftime as an open source project [2]; bpftime has 1,000 stars on GitHub, more than 20 contributors, and several PRs per month. Our users currently rely on bpftime and EIM for many use cases, including observability, fault injection, hot patching, and other application customizations. Inspired by these users, we present 6 use cases that highlight the benefits of bpftime and EIM. These use cases explore design tradeoffs, improve security, monitor performance, and enhance system efficiency. In particular, we use bpftime to monitor a microservice application, create new durability configurations in Redis, cache metadata operations in FUSE, implement an SSL-supporting distributed tracing tool, monitor system calls for performance analysis, and enhance webserver security.

We evaluate bpftime's performance on the aforementioned 6 use cases. We find that bpftime improves the throughput of profiling microservices by a factor of 1.5 compared to eBPF. bpftime enables a Redis durability configuration that loses orders of magnitude less data in a crash while decreasing throughput by only about 10%. bpftime enables FUSE caching that accelerates operations by orders of magnitude. The system adds only 2% overhead when extending Nginx, which is up to 5× lower than state-of-the-art alternatives such as WebAssembly, Lua, ERIM [66], and RLBox [44]. bpftime reduces the overhead of SSL traffic monitoring by a factor of 3.79 compared to native eBPF. Moreover, bpftime offers configurations that prevent monitoring overhead from affecting unmonitored processes, a feature eBPF cannot provide. We also use microbenchmarks to illustrate the key features that enable bpftime's performance advantages and demonstrate bpftime's compatibility with eBPF.

[1] Note: eBPF uprobes do not support all bpftime features.

[2] available at https://github.com/eunomia-bpf/bpftime.

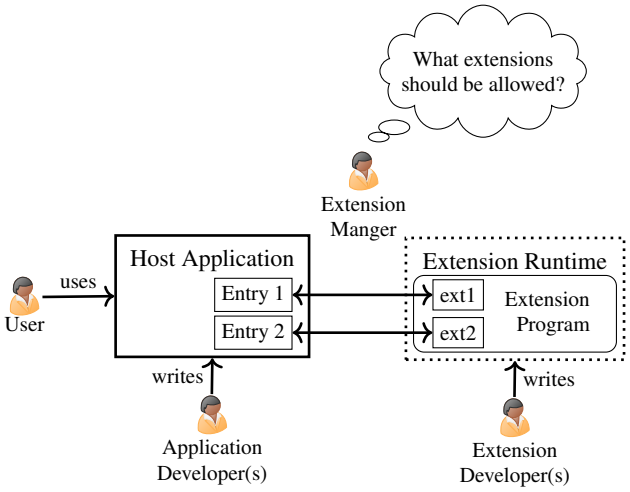



Figure 1: A process extended with two extensions, ext1 and ext2, with associated extension entries, entry1 and entry2, respectively. The application developer(s) and extension developer(s) write the host application and extension program, respectively. The User uses the host application. The Extension Manager decides which extensions to allow and use in the deployment.

In sum, our contributions are:

- EIM, which allows users to specify fine-grained interconnectedness/safety tradeoffs.
- bpftime, an efficient extension runtime system that enforces EIM specifications and isolation through two separate verification techniques and concealed extension entries.
- An evaluation of EIM and bpftime demonstrating their usefulness and efficiency in 6 use cases.

In the rest of the paper, we motivate EIM and bpftime (§2), describe EIM (§3), explain the design and implementation of bpftime (§4), discuss 6 use-cases (§5), evaluate bpftime(§6), describe related work (§7), and conclude (§8).

# 2 Motivation

System extensions augment an application without modifying its source code to customize behavior, enhance security, add custom features, and observe behavior. By supporting application modifications without requiring source code changes, extensions allow a customized deployment to integrate maintenance updates from upstream repositories easily and can provide assurances of security and safety. The rest of this section discusses the principal roles of system extensions (2.1), provides an example web-server use-case (2.2), articulates the key properties of extension frameworks (2.3), discusses limitations of the current state-of-the-art (2.4), and articulates the threat model (2.5).

## 2.1 Roles

The system extension usage model considers four key principals. The *application developers* are a group of trusted developers who write the original application, while the *extension developers* are a group of trusted developers who create the extensions. System extensions assume that both the application developer(s) and extension developer(s) are trusted but fallible, so applications and extensions might be exploitable but are not intentionally malicious. Next, the system extension model includes an *extension manager*, a trusted individual that installs and manages the extensions; the model relies on the manager to be both trusted and infallible. Finally, *users* are untrusted individuals who interact with the extended application; users can be malicious and may try to craft inputs that would trigger vulnerabilities in otherwise benign code.

Figure 1 provides a representation of an extended application and shows the role of each principal. The application developers write the host application. The extension developer creates the extension program, which can read and write application state and execute application-defined functions. The extension manager is responsible for deciding which extensions to use at each extension entry. Finally, users produce input that interacts with the host application and, indirectly, the extension program.

## 2.2 Web-Server Example

Consider an instance of Nginx deployed as a reverse proxy. The application developers write the server, while the extension developers provide a suite of possible extensions to deploy on the system for monitoring, firewalls, and load balancing. The extension manager determines the extensions for the deployment and the privileges to provide each extension. First, the manager uses an extension program that monitors traffic to detect reliability issues [27]. Second, the manager deploys an extension program that implements a firewall that returns a 404 response for URLs that are indicative of SQL injection and cross-site scripting attack. Finally, the manager deploys an extension program to perform load balancing across the possible servers downstream from the proxy by periodically contacting downstream servers to measure system load [26].

## 2.3 Key Extension Framework Features

Extension use-cases require three key features:

**Fine-grained Safety/Interconnectedness tradeoffs.** Extensions must be *interconnected*, i.e., able to interact with the host application. Host interactions include reading/writing host state and executing host-defined functions. The Nginx example extension programs highlights both of these features: extensions in the observability use-case require read access to host state, while the firewall, load balancing, and statistics

| 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. |

Table 1: Example issues caused by extension safety violations.

extensions call Nginx functions to parse user requests and return responses.

At the same time, an extension manager wishes to ensure that extensions are *safe*, i.e., do not harm the health of the deployment. Safety does not aim to thwart malicious extensions, but rather to ensure that a bug in the extension cannot harm the reliability or security of the host. Extension bugs have lead to catastrophic consequences in production, including livelock, system crashes, and remote code execution; Table 1 provides examples.

There is no single definition of safety appropriate for all extension use-cases, since safety and interconnectedness are in tension. Instead, the manager should be able to follow the principle of least privilege and allow each extension to perform only the actions necessary for the interconnectedness required for its task. For example, the Nginx observability extensions only need to read from specific host states, but would be unsafe if allowed to write to them. In contrast, the Nginx firewall extension needs to read/write to a different set of host states, but would be unsafe if it were allowed to read the observability states.

Safety rules that can be applied to each individual extension are *fine-grained safety/interconnectedness tradeoffs*. We note two desirable properties. First, the same extension entry may need to support different safety/interconnectedness tradeoffs since it may be useful for separate extension use-cases. Second, while the universe of possible interconnectedness features depends on the host application and thus requires a knowledgeable application developer to modify the host source code, the manager should be able to modify the safety/interconnectedness tradeoffs for their extensions without changing the original application.

**Isolation.** Extensions must be *isolated*, i.e., not be harmed by the host application. Isolation does not thwart a malicious host application, but rather ensures that attackers cannot circumvent extension-based security by exploiting bugs in the host application. Isolation requires ensuring that host applications cannot modify extension states. The Nginx firewall extension is an example that relies upon isolation.

**Efficiency.** Extensions should be *efficient*, i.e., execute at near-native speed, since they may be deployed on the hot path of production systems. For example, the Nginx load balancer extension is invoked on the hot path for all user requests.

## 2.4 The Limitations of State-of-the-Art

We describe the limitations of existing extension frameworks.

**Native Execution.** A number of extensibility approaches execute the application and extension in the same execution context, essentially treating the extension as a component of the original program. Such approaches include those that use dynamic loading (e.g., `LD_PRELOAD`) and dynamic binary instrumentation [8, 39, 45]). These systems provide efficiency but neither isolation nor support fine-grained safety/interconnectedness tradeoffs.

**SFI-based tools.** Many extension frameworks use SFI [69] to provide isolation, including XFI [16], NaCL [75], RLBox [44], and language-based sandboxes such as WebAssembly [23] and Lua [38]. Some of these tools (e.g., Lua, WebAssembly, and NaCL) do not provide an interface for safety/interconnectedness tradeoffs. Instead, they rely on the host application to check for safety violations, an approach that has proven buggy (see Table 1). Other tools (e.g., RLBox, XFI) lack fine-granularity or the ability to limit certain extension behaviors. Additionally, SFI-based tools are typically inefficient since they validate extension behavior at runtime [29].

**Subprocess Isolation.** Subprocess isolation systems, such as Wedge [7], Shreds [11], lwC [37], and Orbit [31], separate extensions from the host application through operating system isolation abstractions. Such systems ensure isolation. However, some lack fine-grained interconnectedness/safety tradeoffs (Lwc and Shreds), while others (Orbit and Wedge) could provide such tradeoffs, but only after code changes to the host application, because they are not designed for extensibility. Finally, such systems struggle to be efficient for frequently-executed extension use-cases, since they require context-switch-like overheads when switching between the host application and its extensions.

**eBPF uprobes.** While it is usually used for kernel extensions, the extended Berkeley Packet Filter (eBPF) framework provides userspace extensions through the uprobe interface. eBPF uprobes are isolated from the host application, but do not support fine-grained interconnectedness/safety tradeoffs. Moreover, eBPF uprobes are not efficient. eBPF uprobes place a software breakpoint on every extension entry, causing the system to trap into the kernel to execute each extension.

**Aspect-oriented programming.**. Aspect-oriented programming allows extensions, but existing aspect-oriented languages do not support safety/interconnectedness tradeoffs. For example, if an AspectJ extension were exploited [34], the attacker would have unrestricted ability to observe and modify the original host application.

## 2.5 Threat Model

The system extension threat model is as follows. First, it assumes that the extension manager accurately and completely identifies the correct safety/interconnectedness tradeoff for each extension entry. This means that a buggy extension cannot corrupt, crash, or hang an application through the interface that the extension manager allows. Second, the model assumes that the control-flow of the application cannot be modified or corrupted even if an application is compromised—essentially equivalent to control flow integrity [1]—since an attacker could otherwise circumvent extension execution. Given these limitations, the threat model considers two key threats. First, it considers buggy extensions that accidentally crash or hang the application through errant pointers, infinite loops, or stack corruption that are outside of their allowed safety/interconnectedness tradeoff. Second, it considers compromised applications that modify or corrupt the state used by extensions to alter the extension's behavior.

# 3 Extension Interface Model (EIM)

The Extension Interface Model (EIM) is a new model for specifying fine-grained interconnectedness/safety tradeoffs. Using the model, an extension manager can follow the principle of least privilege and specify a tradeoff that would enable extensions to perform their tasks (i.e., sufficiently interconnected) with minimal potential harm to the system (i.e., are sufficiently safe). The model is sufficient to prevent past extension bugs from harming production; e.g., an extension manager can use EIM to prevent each of the bugs in Table 1. An extension framework (e.g., bpftime) can later ensure that the extensions loaded into an application follow their EIM specification, thereby ensuring their safety.

EIM's key idea is to represent the extension features that might be necessary for interconnectedness or restricted for safety as a *resource*. Such resources include both classical systems resources, such as compute cycles, and host application interactions, such as the ability to call a host function or read/write to a host variable. EIM models the ability to use a resource as a capability [55, 60, 71]. An EIM specification encodes fine-grained interconnectedness/safety tradeoffs by specifying the set of capabilities that a loaded extension is allowed to use when configured for a given extension entry.

An EIM specification consists of two separate components. First, the EIM specification includes a development-time configuration, prepared by an application developer, that specifies the set of possible safety and interconnectedness features (§3.1). Second, an EIM specification includes a deployment-time configuration, prepared by an extension manager, that specifies precise interconnectedness/safety tradeoffs (§3.2).

## 3.1 Development-time EIM Specification

```
1 State_Capability(
2 name = "readPid",
3 operation = read(ngx_pid))
4
5 Function_Capability(
6  name = "nginxTime",
7  prototype = (void) -> time_t,
8  constraints = {rtn > 0})
9
10 Extension_Entry(
11  name="processBegin"
12  extension_entry = "ngx_http_process_request",
13  prototype = (Request *r) -> int))
14 Extension_Entry(
15  name="updateResponseContent"
16  extension_entry = "ngx_http_content_phase",
17  prototype = (Request *r) -> int*)
```

Figure 2: An EIM development-time specification for a simplified version of the Nginx observability use-case.

The development-time EIM specification encodes the possible interconnectedness features and extension entries of the host application. Since these features are tightly coupled with the host application, the development-time EIM is created by an application developer while developing the host application. A development-time EIM specification defines three sets of entries: state capabilities, function capabilities, and extension entries. Figure 2 provides an example of a development-time EIM specification for the Nginx observability use-case.

**State Capability.** A state capability expresses the ability to read or write to a global state located in the host. A state capability includes a name and an operation of the form `read(var)` or `write(var)`, which specify the capability to dereference and read or write to a variable, `var`, respectively. Figure 2 includes a state capability, `readPid`, that enables reading the global variable in Nginx that stores the process id of the current worker process (Lines 1– 3).

**Function Capabilities.** A function capability expresses the ability to execute a host-provided function. An application developer defines a new function capability by providing the capability a name, the function prototype of the function, and a set of constraints that provide pre- and post-conditions for the function. Figure 2 includes a function capability in Lines 5–8. The capability specifies the ability to call Nginx's function for getting a timestamp. The function constraints provide the post-condition that its return value is positive.

**Constraints.** Constraints ensure that extensions safely use host-provided functions and can encode two things. First, constraints can encode binary relationships between arguments and return values, high-level semantic facts, and boolean operators over other constraints. Second, constraints can encode high-level fats about the function that an extension framework could support. Currently, EIM supports allocation facts indicating that a function's return was allocated, IO facts in-

```
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)})
```

Figure 3: A deployment-time EIM specification for the simplified Nginx observability use-case.

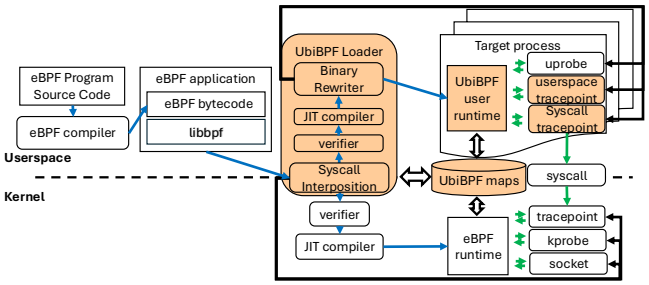



Figure 4: bpftime Design. White components are from eBPF; orange components are new to bpftime. Blue arrows show execution flow when compiling and loading an eBPF application. Green arrows show execution flow when an eBPF extension executes. White arrows with black outline indicate components that interact with eBPF maps

dicating that the function requires the capability to perform IO, annotation facts that indicate a relationship between arguments equivalent to those that linux provides through current eBPF annotations [35], and read/write facts indicating that the caller must hold read/write capabilities for a specified field within a function argument.

**Extension Entry.** An extension entry specifies the points in the host application that an extension can override. EIM supports extension entries that specify a function—i.e., extending an EIM extension entry replaces the definition of the associated function. An extension entry includes a name, the name of the original function from the host application, and the prototype of the function. Conveniently, many applications already include extensibility support through function interposition that an application developer can reuse for the EIM specification. However, developers may need to define and call new functions in the host application to support emerging extension use-cases. Figure 2 includes two extension entries, `processBegin`, for extending Nginx immediately after it is finished parsing a request, and `updateResponseConent`, for modifying the response that Nginx returns to a client.

## 3.2 Deployment-time EIM Specification

Deployment-time EIM specifications identify the interconnectedness/safety tradeoffs for a deployment. In particular, the specification identifies the capabilities that an extension requires for its task. Extension Managers write the deployment-time EIM specifications, since identifying the right tradeoff requires reasoning about a deployment's extension use-cases and available resources.

A deployment-time EIM specification includes a set of *Extension Classes*, each representing an interconnectedness/safety tradeoff at an extension entry. An extension class includes a name, the name of an extension entry from the development-time EIM specification, and a set of capabilities allowed by the class. An extension class' allowed capabilities can include capabilities outlined in the development-time EIM specification, state capabilities over the arguments in the extension entry's prototype, and resource capabilities, which specify an ability to consume a hardware resource. EIM provides two such resources, `instructions` and `memory`.

Figure 3 is a deployment-time EIM for the simplified Nginx observability use-case. The specification includes constraints that prevent extension bugs; e.g., it prevents the extension livelock that caused a production outage to an Nginx deployment [73]. The specification includes two extension classes, one for each of the extension entries defined in Figure 2. The allowed capabilities (Lines 4 and 8) include state capabilities (e.g., `read(r)`), resource capabilities (e.g., `instructions < inf`), and function capabilities (e.g., `nginxTime`).

# 4 bpftime Design

EIM specifications are platform independent, so we could expand existing extension frameworks to enforce them. However, existing frameworks provide safety and isolation through inefficient heavyweight operating-system supported isolation [7, 11, 31, 37] or software-fault isolation [23, 38, 44, 75]. Expanding such systems to enforce EIM's capability model would further degrade their efficiency and yield an unusual system.

So, we designed and implemented bpftime, a new extension framework for extending compiled applications (e.g., those written with C,C++,Go, etc.). bpftime efficiently enforces EIM specifications and supports extension isolation by employing two design constraints. First, the system uses separate lightweight approaches for EIM enforcement and for isolation, similar to how KFlex [15] uses two separate verification techniques for kernel extensions. Namely, bpftime enforces EIM safety/interconnectedness tradeoffs without imposing any runtime overhead by employing eBPF-style verification and provides efficient extension isolation using ERIM-style [66] intra-process hardware-supported isolation. Second, bpftime introduces concealed extension entries, which use binary rewriting [10, 14, 18, 70] to inject extensions into the

host application so that extension entries are zero-cost when not in use by an extension.

bpftime is compatible with eBPF: it provides nearly identical development model, execution models, and even reuses some architecture from eBPF. Thus, existing eBPF uprobe use-cases can seamlessly employ bpftime, and users can easy deploy bpftime extensions alongside kernel eBPF extensions. Alas, ensuring eBPF compatibility is challenging. Linux's eBPF ecosystem consists of tightly coupled components—compilers, runtime libraries, and the kernel—that are nearly impossible to disentangle. Instead of decoupling them, prior user-level eBPF systems [28, 53] re-implementing the entire eBPF technology stack and ultimately failed to provide reasonable performance and compatibility. Instead, bpftime identifies a narrow waist in the current eBPF ecosystem and interposes at this point. In particular, the system interposes on eBPF-related system calls and on the shared map mechanism for sharing data across extensions.

We implemented bpftime in 13,000 lines of code; Figure 4 provides the high-level design. bpftime's trusted computing base includes the kernel eBPF verifier, the binary rewriter, the operating system, and hardware-supported intra-process memory protections. Our current implementation only supports isolation on Intel x86, but different hardware vendors have similar technology that bpftime could employ to add support (e.g., ARM memory domains).

The rest of this section describes bpftime's usage model (§4.1), the bpftime loader (§4.2), and the bpftime runtime (§4.3).

## 4.1 bpftime Usage

bpftime follows a usage model similar to eBPF. Application developers first create development-time specifications for their application. Extension developers then write programs using eBPF. Extension managers later create deployment-time EIM, and load the extension into the host application with eBPF system calls. The bpftime loader intercepts these calls, validates extensions against their class-defined capabilities, initializes bpftime maps, sets up the bpftime runtimes, and attaches extensions to their extension entries. During execution, the runtime enforces extension isolation.

We elaborate on bpftime's support for EIM and bpftime extension programs.

### 4.1.1 bpftime support for EIM

To reduce the challenge of maintaining consistency between an application and its EIM specification, bpftime supports development-time EIM specifications through the use of annotations. Namely, application developers annotate their code using a set of annotations that are based upon eBPF's kfunc annotations [35] to specify state capabilities, function capabilities, and extension entries. During compilation, bpftime uses static analysis to extracts the development-time EIM specification from the annotations as well as symbol and type information from the application's debug symbols. The system encodes the resulting specification at the binary level that could, in principle, support any compiled language; the tool currently supports C/C++.

In addition to the application-specific development-time EIM specification, bpftime adds generic extension entries and function capabilities. The system adds `uprobe` and `uretprobe` execution entries for every host-defined function; the host logically executes these entries at the beginning and return of their affiliated function, respectively. Similarly, bpftime adds `sysenter` and `sysexit` execution entries for system calls. The prototype for the `uprobe` and `sysenter` entries include the arguments for the associated function or systemcall, respectively, while the prototype for `uretprobe` and `sysenter` includes the return value for the associated function or systemcall return, respectively. Additionally, bpftime defines a number of runtime-defined helper functions for common tasks and encodes them as function capabilities in the development-time EIM specification. For example, it adds a function capability to modify the return value for `uretprobe` and `sysexit` probes and separate capabilities to interact with bpftime-provided shared memory for process-local memory, inter-process memory, or process-kernel memory.

bpftime supports concealed extension entries when it generates the development-time EIM specification. The system removes the calls to all defined extension entries from the compiled program, both the developer-specified and automatically generated ones, thereby ensuring that any entries that are not used by an extension do not impose a runtime cost.

bpftime supports deployment-time EIM specifications through a YAML configuration language. In addition to extension classes that an extension manager defines, bpftime supports two built-in extension class types that auto-generate useful extension entries. First, bpftime supports observability, which adds an observability extension class for all auto-generated entries (i.e., `uprobe`s, `uretprobe`s, `sysenter`s, and `sysexit`s). Each observability extension class allows capabilities to read the entry's arguments, use a bounded number of instructions, and use a subset of the bpftime-provided function capabilities. Second, bpftime supports customizability, which adds customizability for all auto-generated entries. Each customizability extension class allows capabilities to read/write the entry's arguments using a bounded number of instructions and to use many of the bpftime-provided functions.

### 4.1.2 bpftime support for extension programs

bpftime supports extension programs that are similar to those supported by traditional eBPF. A developer writes an event-based program consisting of two parts: (1) a set of bpftime extensions and (2) a userspace control application that loads

and communicates with the extensions. Each user-defined extension specifies an associated extension class from EIM. Meanwhile, the eBPF application uses standard APIs (e.g., libbpf) to share data and interact with extensions through bpftime maps.

## 4.2 bpftime Loader

The bpftime loader prepares the extensions for execution by loading them into the host application and initializing the bpftime maps required for the eBPF application. To cope with the complexity of ensuring compatibility with eBPF, the loader is a thin interposition layer between eBPF applications and the kernel. The loader exposes the eBPF-related system calls interface from the eBPF ecosystem to userspace, but implements them using standard POSIX abstractions (files, unix sockets, etc.). Thus, bpftime supports control applications that use standard eBPF runtimes (e.g. bcc and libbpf) and are compiled using a standard eBPF compiler without requiring kernel modifications. The loader consists of two key components, the verifier, which implements bpftime's eBPF-style verification of extension safety, and the binary rewriter, which supports extensions that use concealed extension entries.

**The Verifier.** The verifier ensures that extensions are allowed by their associated extension class in the EIM specification. It receives each extension as an eBPF bytecode program. The verifier first performs a few basic verification checks: it ensures that an extension's function signature matches its extension class and that the extension only calls functions allowed by the extension class' function capabilities.

Next, the verifier converts the EIM specification into constraints that it adds to the eBPF bytecode program and uses the eBPF verifier to verify the extension. First, bpftime parses DWARF debugging information from the host application to produce BTF information to add types to the eBPF bytecode. The verifier supports function capabilities by replacing each call to a function capability in the extension with a mock new eBPF kfunc that it generates; using mock kfuncs allows bpftime to reuse the eBPF verifier's support for external functions. Additionally, the verifier converts all function capability constraints into assertion statements that it inserts into the eBPF bytecode before the generated kfunc. Next, the verifier uses the extension class' capability to add additional constraints. It supports state capabilities by modifying the function prototype (e.g., adding `const` for read only arguments) and encodes resource capabilities using verifier-supported clauses [3]. Finally, the verifier uses the verifier to ensures that the extension upholds the EIM specification, is memory and type safe, and will not execute a hardware exception.

Once verified, extensions pass through a userspace JIT compiler, which compiles them into native code. The JIT compiler passes the compiled extensions to the rewriter.

[3]The Linux kernel provides minimal support for custom resource constraints, so our implementation falls back to using PREVAIL [19] for these.

**The Rewriter.** The rewriter uses `ptrace` to pause the host application and load the bpftime user runtime (§4.3) into it, which allows the system to support use-cases that extend live applications; the system could use existing live-update solutions, such as MCR [20] and KUP [33], to improve performance. The rewriter instruments the program's instructions using Frida [18] and libcapstone [36] to ensure that extensions are invoked when the host reaches their associated extension entry. For extension entries that relate to a single instruction (e.g., `uprobes`, `uretprobes`), the rewriter uses a standard instruction trampoline: it replaces the instructions that were originally at the hook point with a `call` into a preamble for the point's associated eBPF function and places the overwritten instructions at the end of its preamble. Supporting extension entries that relate to system calls is more complex since an application can execute a system call instructions (e.g., `sysenter`) at any point throughout their application. Thus, the runtime iterates through all of the instructions in the application and places a trampoline on all system call instructions. Since `sysenter` is smaller than a trampoline, the system uses zpoline [74], which uses the zero page to accommodate two-byte call instructions. The current implementation only supports a single extension per entry point; users requiring multiple extensions can use a dispatcher pattern to individually call each extension, similar to the design used by libxdp.

## 4.3 bpftime Runtime

The bpftime runtime executes extensions in the same process as the original application to ensure efficiency. The runtime implements bpftime's intra-process isolation approach to provide extension isolation, which is less expensive than the isolation techniques of existing extension frameworks (e.g., software-fault isolation). Finally, the runtime implements bpftime maps, which are compatible with eBPF maps.

**Intraprocess Isolation.** The bpftime runtime uses ERIM-style intraprocess isolation to ensure extension isolation [66]. The runtime ensures that the application cannot modify extension logic by setting the memory pages that contain extensions and extension trampolines to be non-writable. The runtime ensures the integrity of extension memory using intra-process memory protections. During loading, the bpftime loader allocates a memory protection key for the extension's memory. The loader adds instructions that use the allocated key (i.e., `WRPKRU`) immediately before all userspace extensions, and adds instructions to reset the key immediately before returning to the original application. To protect the key itself, bpftime loads the key's value directly into the extension and sets the extension's memory permissions to be non-readable. The system is currently susceptible to the syscall-based attacks on ERIM [13], but could adopt Jenny's syscall filtering defenses [56] to resolve the issue.

**bpftime Maps.** Conventional eBPF provides maps to store

state across extension invocations. Unfortunately, eBPF maps support access to userspace programs through expensive system calls. So, bpftime provides a new map implementation to support system call free access across extension invocations, including bpftime extensions, eBPF kernel extensions, and bpftime control applications. Additionally, bpftime maintains compatibility with the existing eBPF ecosystem by interposing on eBPF map system calls and adding logic that uses bpftime maps instead of traditional eBPF ones. bpftime maps support three sharing modes: local process (nonshared), shared across multiple processes, and shared across processes and the kernel. bpftime maps provide a wide-range of data structures, including hash maps, arrays, LPM tries, ring buffers, perf event arrays, per-CPU variants. Additionally, bpftime maps are highly efficient: they offer per-cpu variants to remove contention and use lock-free synchronization when necessary.

## 5 Use Cases

We maintain bpftime as an open source project and have cultivated a small use-base. EIM and bpftime have many use-cases, including observability, hot patching, and security enforcement. Inspired in part by the experiences of our users, we craft 6 use-cases of EIM and bpftime that explore design tradeoffs, improve security, monitor performance, and improve performance.

**Nginx Plugin.** An extension manager manages an Nginx deployment that frequently receives suspicious traffic and wishes to deploy an extension to ensure that the system remains secure. While Nginx currently supports extensions through Lua and WebAssembly, the extensions framework offer poor safety/interconnectedness trade-off and impose high overhead since the extensions copy data to-and-for Nginx instead of operating over the same objects. Since Nginx is already designed for extensibility, creating the development time EIM specification only requires an application developer add annotations to current Nginx extension functions. The extension manager specifies the deployment time EIM by adding a single extension class to an extension entry that corresponds to processing request. The extension class supports reading from an argument representing the current request and writing to the return value. The manager then deploys an extension that implements a firewall to return a 404 response when a request's URL is indicative of SQL injection and cross-site scripting attacks.

**sslsniff.** Distributed tracing tools struggle to provide useful results when used on encrypted traffic. `sslsniff`, a tool from the bcc [52] project, allows an extension manager to intercepts all SSL/TLS data within userspace to enable better distributed tracing for encrypted data. `sslsniff` uses eBPF `uprobes` on SSL/TLS encryption and decryption functions in OpenSSL. It suffers from high overhead due to eBPF `uprobe` inefficiencies. bpftime's automated `uprobes` and observability extension classes enable `sslsniff` to work on bpftime without requiring changes.

**Syscount.** Syscount [3], another bcc tool, allows aggregating system call activity for a process on a running system by using system-wide extensions on `sysenters` and `sysexits`. To limit monitoring to a single process, the tool must observe the system calls of every process on the system and perform manual filtering in the extension. As a result, Syscount imposes overhead on every process, even those that are not monitored. An extension manager can run Syscount on bpftime without any modifications by using bpftime's automated `sysenter` extension entries and its automated observability extension classes. Since bpftime adds Syscount extensions into only the monitored process, the Syscount overhead is localized to the monitored processes.

**DeepFlow.** DeepFlow is an open-source observability platform for observing a microservice's behavior across the kernel and userspace with eBPF. Unfortunately, Deepflow can lead to as much as a 50% drop in application throughput due to the overheads of eBPF `uprobes`. We ported DeepFlow to bpftime, modifying 10 out of the 5,000 lines of eBPF code, so that the tool could benefit from bpftime's efficiency. DeepFlow extensions use bpftime's automated uprobe and system call extension entries. DeepFlow uses the system's automated observability extension classes with one caveat: it adds support for using process-kernel shared maps.

**FUSE Caching.** The Filesystem in Userspace (FUSE) framework offers reliability and security advantages compared to in-kernel alternatives, but imposes considerable runtime overhead because it requires numerous additional context switches for every I/O systemcall [67]. ExtFuse [6] eliminates much of the overhead from using FUSE by enabling a FUSE filesystem to push its logic into the kernel. However, ExtFuse is invasive: it requires a custom kernel module that is difficult to maintain. In this use-case, an extension developer uses bpftime to implement and support the same benefits of ExtFuse without requiring a custom and difficult to maintain kernel model. They configure bpftime's automated customizability extension classes on the system's automated `syscall` tracepoints and add function capabilities for bpftime helpers that interact with file paths, such as `realpath`, and for using process-kernel shared maps. Then, they configure the deployment to use two extensions, written by an extension developer, that accelerate applications that use FUSE. The first is a metadata cache that accelerates repeated lookups to the same file system entries by extending `open`, `close`, `getdents`, and `stat`. It exploit's bpftime's compatibility with eBPF to maintain cache consistency using a kprobe extension on `unlink` in the kernel [4]. The second extension implements a blacklist to accelerate permission checking for functions that access filesystem entries (e.g., `open`).

**Redis Durability Tuning.** Redis, a key-value store, offers durability through a write-ahead log (called an Append-Only

[4]The second extension must operate in the kernel since non-extended processes may `unlink` a cached file.

```c
int writeback_fin_cnt;

SEC("user_define_ops/fsync_ext")
int BPF_UPROBE(start_fsync, int __fd) {
 if (writeback_fin_cnt == 0) {
   io_uring_wait_and_seen();
 } // else fsync is complete so no wait
 successful_writeback_count = 0;
 io_uring_prep_fsync(__fd);
 io_uring_submit();
 return 0;
}

SEC("kretprobe/file_write_and_wait_range")
int BPF_KPROBE(file_write_and_wait_range,
            struct file *file, loff_t start,
            loff_t end) {
 ... // Check if fsync was from redis/AOF
 __sync_fetch_and_add(&writeback_fin_cnt, 1);
 return 0;
}
```

Figure 5: The Redis Delayed fsync extension.

File (AOF)). By default, Redis offers three durability configurations that tradeoff durability and performance overhead: no AOF, which does not provide durability; everysec, which ensures that writes are durable every second; and alwayson, which ensures that every write is immediately durable. The durability gap between alwayson and everysec is substantial: a crash under everysec can lead to the loss of tens of thousands of updates. Unfortunately, alwayson also reduces throughput by a factor of six compared to everysec.

Redis can use bpftime to provide customizable durability through extensions. Since Redis AOF is not currently designed for extensibility, the application developer needs to add new functions to the source code to support extensibility. The developer defines three new functions and call them at the top of Redis's functions for `write`, `fsync`, and `fdatasync`. Then, they use bpftime annotations to identify the new functions as extension entries. Altogether, this change required about 20 lines of code.

With the updated Redis, the extension manager can explore a custom durability policy for their deployment. They first configure an extension that converts synchronous I/O operations into batched ones by using Linux's `io_uring`. The extension batches *b* I/O operations (calls to `write` and `fsync`) before waiting on them to complete, which ensures that the system loses at most *b* updates in an untimely crash. To use the extension, the extension manager creates two extension classes for `write` and `fsync` that have bounded computation and function capabilities to execute `io_uring` and bpftime map helpers for process-local memory.

However, the performance of highly durable configurations (i.e., small values of *b*) remains poor. So, the extension manager tries another extension. The new extension, *delayed-fsync* (Figure 5), extends the behavior of `fdatasync` so that it waits for the *previous* call to `fdatasync`. This design ensures that the system loses at most 2 updates. The extension also implements a fast-path optimization that extends the kernel to expose a shared variable to userspace that tracks the number of completed `fdatasync` operations on each of a process' open files. The `fdatasync` extension reads from the shared variable and only execute system calls in the event that the previous `fdatasync` has not completed. To use delayed-fsync, the extension manager creates an extension class for `fdatasyc` that has bounded computation and function capabilities to execute `io_uring` and bpftime map helpers for process-kernel shared memory.

# 6 Evaluation

We implement the 6 case studies on bpftime and answer the following questions:

1. How does bpftime's performance overhead compare to state-of-the-art extension tools, including eBPF, Lua, WebAssembly, ERIM [66], and RLBox [44]?
2. Why does bpftime impose lower runtime overhead than existing tools,?
3. How compatible is bpftime with existing extension use-cases and kernel eBPF runtime?

**Experimental Setup.** We evaluate bpftime running on two servers. Server A is a dual-socket Intel Xeon Gold 5418Y Processor (24 cores, 2.00 GHz, 45 MB LLC) with 256 GB DDR5 memory. Server B is a dual-socket Intel Xeon E5-2697-v2 processor (48 cores, 2.7 Ghz, 30 MB LLC) with 256 GB DDR3 memory. Unless otherwise mentioned, each metric is the average of 10 trials. We report averages using geometric mean when that is appropriate (e.g., when calculating an average speedup). We compare bpftime to two baselines: the native execution on each system (native) and the performance running the extension on the linux eBPF ecosystem.

## 6.1 bpftime Performance

This section evaluates bpftime's performance across the 6 case studies. In summary, we find that bpftime offers a significant improvement compared to the current state-of-the-art extension frameworks such as eBPF, Lua, WebAssembly, RLBox [44], and ERIM [66].

### 6.1.1 Nginx Plugin

We evaluated bpftime performance improvements for extending Nginx against the current plugins approaches, including Nginx's builtin support for Lua and WebAssembly, and prior state-of-the-art systems including ERIM [66] and RLBox [44]. We deployed security and tracing modules for each framework in Nginx on System A and benchmark the system using `wrk` (8 threads, 64 connections, 30s tests, averaged over 20 runs). Figure 6 shows throughput results: bpftime introduces 2% overhead compared to native Nginx. In contrast, Lua and WebAssembly incur 11% and 12%, so bpftime achieves 5.5×

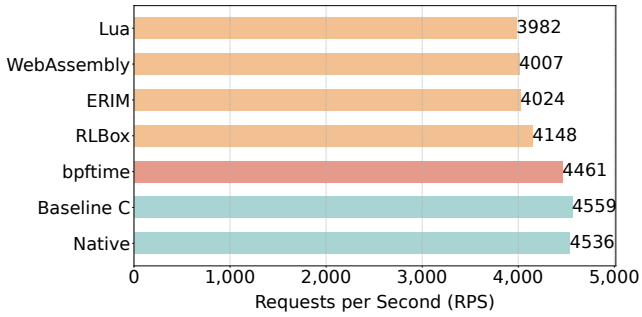
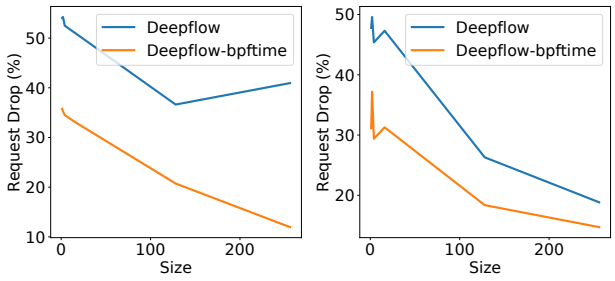



Figure 6: Comparison of extension approaches for Nginx module




(a) HTTP (b) HTTPS

Figure 7: Deepflow throughput overhead

and 6× lower overhead, respectively. ERIM and RLbox impose 11% and 9%, so bpftime achieves 5.5× and 4.5× lower overhead, respectively. In summary, this experiment illuminates a tradeoff between the current extension approaches and bpftime: bpftime and EIM will require more development time to write the specification and pass the verifier when compared to prior work, but significantly improve efficiency and safety.

### 6.1.2 Deepflow

We deploy Deepflow on System B and use it to monitor a microservice that returns a random string for each request, implemented on a Golang server, and configured to use either HTTP and HTTPS. We route traffic to the service using `wrk`, configured to use 10 threads with 500 concurrent connections for a 10-second test duration. We execute the tests with three configurations: native microservice execution without Deepflow, Deepflow using eBPF, and Deepflow-bpftime using bpftime.

We measure the server's throughput as the response size increases from 1KB to 256KB (Figure 7). Without DeepFlow, the microservice achieves 250,000 requests/s for small responses and 47,000 requests/s for large responses. With DeepFlow using eBPF, throughput drops by up to 54%. bpftime improves DeepFlow throughput by at least a factor of

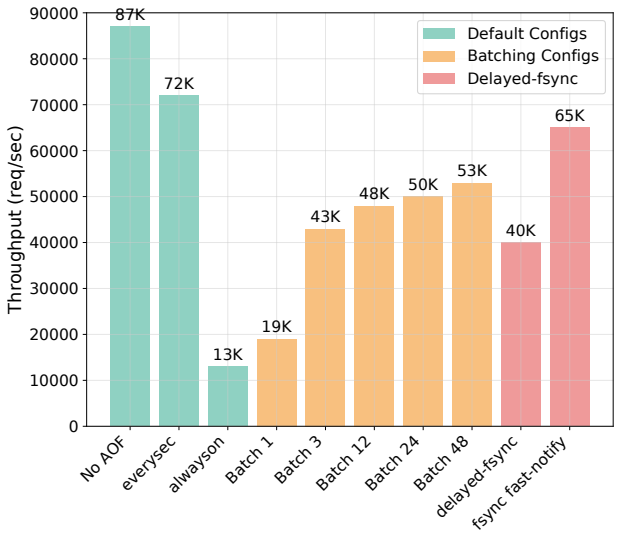



Figure 8: Redis throughput with bpftime Batch-I/O and Delayed-fsync extensions

1.5. DeepFlow's remaining overhead comes from the system's large set of probes that it uses throughout both the kernel and in userspace.

### 6.1.3 Durability Tuning

We evaluate the Durability Tuning use-case on System A and use redis-bench [65] to send 1M set requests with 5 parallel clients and 3 byte payloads. We execute Redis with the built-in configurations (always on, everysec, and No AOF), varied batch sizes (1, 3, 12, 24, and 48), delayed-fsync, and delayed-fsync with the fast-notify optimization. We configure Redis with always on when configured to use a batching and delayed-fsync. Figure 8 shows the throughput of each of the configurations.

The batched-I/O exposes interesting durability/performance tradeoffs. Using a batch size of 48 improves Redis always on throughput by a factor of 4.17, and incurs a modest durability cost of losing 24 updates in the event of a crash (roughly half of the function in a batch are `writes`). For comparison, the everysec configuration can lose 72,000 updates in the event of an untimely crash. Thus, a batch size of 48 reduces data loss relative to everysec by three orders of magnitude. Additionally, we observe that the batched-I/O implementation improves Redis performance even for small batches: e.g., a batch size of 1 improves Redis always on throughput by a factor of 1.51.

Delayed-fsync presents a powerful configuration. On its own, delayed-fsync achieves a throughput of 40k requests/second, which is 4.15 times larger than the Redis always on's throughput. Adding the fast notify user-kernel interaction further accelerates Redis to 65k request/second, more than 5

| Test | Native (s) | bpftime (s) |
|---|---|---|
| Passthrough, fstat | 3.65 | .176 |
| LoggedFS, fstat | 7.40 | .184 |
| LoggedFS, openat | 17.0 | 0.074 |
| Passthrough, find | 5.1 | 1.6 |

Table 2: FUSE operation latency.

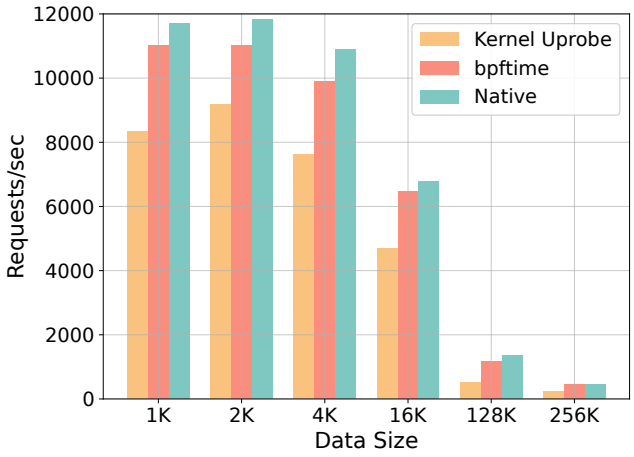



Figure 9: SSlsniff test with nginx

times larger than Redis alwayson. In sum: delayed-fsync with fast notify is only 10% slower than Redis everysec, but it reduces the number of updates that could be lost in a crash by 5 orders of magnitude.

#### 6.1.4 FUSE

We evaluate the performance improvement from using bpftime to implement caching in FUSE when running on System A. We deploy two FUSE file systems: Passthrough, which passes filesystem operations to the underlying file system directly, and LoggedFS [41], which logs all file system operations to a file before passing them to the underlying file system. We measure the end-to-end latency of three workloads: one that issues 100000 `fstatat` calls to a file in the FUSE directory, one that issues 100000 `openat` calls to a file in the FUSE directory, and one that uses the linux utility `find` to travel through the linux 6.7 source code directory. Table 2 shows the results on FUSE and with bpftime caching. We observe that the caching extension implemented with bpftime accelerates the latency of the workloads by a factor of up to 2.4 orders of magnitude.

#### 6.1.5 Sslsniff

We evaluate `sslsniff` on System B while monitoring Nginx. Using the `wrk` benchmark, configured with 4 threads, 512 concurrent connections, and a 10-second test duration. The evaluation includes three configurations: native, `sslsniff` with eBPF, and `sslsniff` with bpftime. We measure nginx throughput when varying response size from 1 KB to 256 KB; figure 9 shows the results. eBPF `sslsniff` significantly impacts Nginx's performance, reducing throughput by up to 28.06% compared to native execution. In contrast, bpftime `sslsniff` has a much smaller effect, with a worst-case throughput reduction of only 7.41

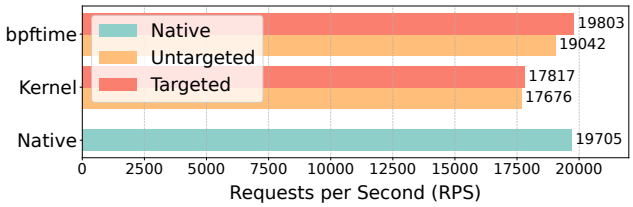



Figure 10: syscount test with nginx

| Bench Type | eBPF(ns) | bpftime (ns) |
|---|---|---|
| Uprobe | 2561.57 | 190.02 |
| Uretprobe | 3019.45 | 187.10 |
| Syscall Tracepoint | 151 | 232 |
| User memory read | 23.3 | 1.5 |
| User memory write | 23.9 | 1.4 |
| hash_map_update | 50.8 | 23.8 |
| hash_map_delete | 19.5 | 10.1 |
| hash_map_lookup | 9.8 | 22.0 |

Table 3: Microbenchmark comparison of bpftime and eBPF

#### 6.1.6 Syscount

We evaluate syscount on System B using the same Nginx server config as in the sslsniff evaluation. Figure 10 compares the measured throughput across five configurations. eBPF Syscount reduce throughput on monitored and unmonitored processes by 10.3% and 9.6%, respectively. In contrast, bpftime syscount reduces throughput on the monitored process by 3.36% and does not affect the unmonitored process.

### 6.2 Microbenchmark Performance

We use microbenchmarks to analyze bpftime's performance.

**bpftime vs. eBPF.** Table 3 compares bpftime performance to eBPF on microbenchmark operations. First, we create microbenchmarks that measure the overhead of each type of extension entry (`uprobe`, `uretprobe`, or `syscall tracepoint`) on System A. bpftime is more than an order of magnitude faster than eBPF for `uprobe` and `uretprobe`, but about 1.5x slower for `syscall tracepoint`. Then, we measure the latency of using helper functions to access userspace memory and eBPF maps and to update and delete hash entries on System A. We observe that bpftime operations are up to an order of magnitude faster than comparable eBPF operations, since bpftime's extension compiler can inline calls and does not require the address space of kernel eBPF

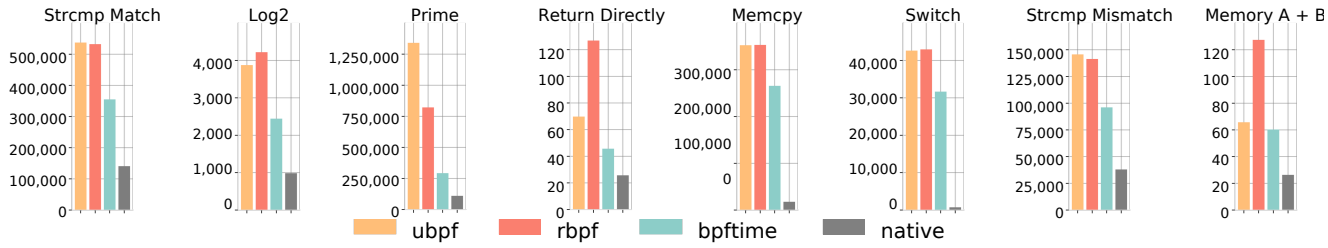



Figure 11: Performance comparison of bpftime, ubpf, rbpf, and native on microbenchmarks.

**bpftime execution engine efficiency:.** We evaluate the efficiency of bpftime's runtime compared to ubpf [28] and rbpf [28] on 8 micro-benchmarks, including integer heavy computation ('log2', 'prime'), memory heavy workloads ('memory a+b', 'memcpy'), cstring benchmarks ('strcmp match', 'strcmp mismatch'), and control-flow benchmarks ('return directly', 'switch'). Figure 11 shows the latency of each microbenchmark and indicates that bpftime is significantly faster than existing userspace eBPF runtimes. Namely, bpftime accelerates average benchmark latency by a factor of 1.53 and 1.72 compared to ubpf and rbpf, respectively.

**bpftime Load latency.** We evaluate the latency of loading an extension with bpftime on system A. We write an extension that monitors `malloc` in `libc`, load it into a running process, and observe a 48ms load latency. For comparison, we load the same extension using LD_PRELOAD, and observe a 30ms load latency.

**bpftime Cost Breakdown.** We isolate the performance improvements of bpftime's two key optimizations: MPK isolation and concealed extension entries. We create a microbenchmark that repeatedly executes a null extension function. The microbenchmark calls the extension function with three settings: first, through its trampoline; second, without the trampoline but with the bpftime execution engine; and third, without the trampoline or execution engine. We observe that the average latency of calling the extension function with the trampoline is 190ns, without the trampoline but with the extension entry is 106ns, and without both is 1.35ns. This means that concealed extension entries save 1.35ns for every unused extension entry that the application executes dynamically; since extension managers often use bpftime's automated extension entries (see §5), these small per entry savings yield large end-to-end performance improvement. The tradeoff is that concealed extension entries require a trampoline that adds an additional 84ns for every executed extension. We reran the experiments with MPK isolation on and off, and did not observe a measurable difference in average extension execution.

## 6.3 bpftime Compatibility

We evaluate the compatibility of bpftime with existing eBPF extensions. We test 17 BCC [52] and bpftrace [68] tools on bpftime without code changes. Also, bpftime fails only one test in the the bpf-conformance test suite [32], whereas ubpf and rbpf fail 22 and 23, respectively.

# 7 Related Work

EIM and bpftime improve the safety and efficiency of software extensions in userspace. EIM allows extension manager to specify fine-grained interconnectedness/safety tradeoffs, while bpftime enforces the tradeoffs efficiently. We outline the related work on userspace extensions, kernel extensions, Intra-process Isolation, and Userspace eBPF.

**Userspace Software Extensions.** Userspace software extensions fall into three categories. Some systems execute extensions in the same execution context as the host application and thus provide efficiency but not safety or isolation, such as dynamic binary instrumentation tools [8, 39, 45] and `LD_PRELOAD`-based tools. Other systems use SFI [69] to provide isolation, but either do not support safety/interconnectedness tradeoffs [23, 38, 75] or lack the ability to specify such tradeoffs at a fine granularity [16, 44]. Moreover, such systems are typically inefficient due to the overhead of SFI's runtime checks. Finally, some systems [7, 11, 31, 37] use subprocess abstractions for isolation. These systems were not designed for extensions and would be inefficient for extension use-cases since they require context-switch-like overheads to switch between contexts. Moreover, some [11, 37] lack support for specifying fine-grained interconnectedness/safety tradeoffs. Others [7, 31] could support fine-grained interconnectedness/safety tradeoffs, but their usage models do not support extensions: a user of these tools would have to modify their application to extend it.

**Kernel Extensions.** Many systems support kernel extensions [5, 9, 15, 58, 63]; none of these systems support EIM's rich set of fine-grained interconnectedness/safety trradeoffs. Moreover, most [5, 9, 58, 63] use SFI to enforce safety and isolation, which imposes runtime overhead. eBPF has emerged as the de facto extension framework in linux and is increasingly

used in academia [15, 76] and industry [4, 12]. eBPF provides an interface for specifying per hook interconnectedness/safety tradeoffs, but (1) does not support EIM's fine granularity since eBPF bundles large sets of features into sets called program types, (2) does not support all of EIM's features (e.g., eBPF lacks constraints on host-provided functions), and (3) has poor userspace extension support. bpftime's concealed extension entries approach is inspired by recent efforts to support kernel eBPF extensions without traps [30].

**Intra-process Isolation.** bpftime's use of intra-process memory protection is adopted from the approaches taken by Donky [57], ERIM [66] and libmpk [51]. bpftime's key difference between these systems is its target domain—bpftime applies these ideas to extend an existing application without requiring source-code changes, whereas prior work uses intra-process memory protection to provide lightweight protection domains that a rewritten application can use.

**Userspace eBPF.** ubpf [28] and rbpf [53] implement virtual machines for userspace eBPF. These systems have high runtime overhead and incomplete eBPF support.

## 8 Conclusion

In this work, we presented EIM and bpftime, which together provide software extensions that are safer and more efficient. EIM is a new model that allows an administrator to specify fine-grained interconnectedness/safety tradeoffs, effectively reducing the potential for a buggy extension to harm the extended application or underlying system. bpftime is a new extension runtime that enforces EIM specifications. bpftime is more efficient than existing extension frameworks because it uses eBPF-style verification, hardware supported intra-process isolation features, and binary rewriting techniques. Plus, bpftime is easy to adopt since it is compatible with the current eBPF ecosystem. We released bpftime as an open-source project and have cultivated a small user-base. We evaluated EIM and bpftime using 6 use cases to show that they are useful and efficient.

## 9 Acknowledgements

We would like to thank our shepherd, Ryan Huang, and the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Science Foundation under grant CNS-2236966.

## References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), nov 2009.

[2] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. Autosteer: Learned query optimization for any sql database. *Proc. VLDB Endow.*, 16(12):3515–3527, August 2023.

[3] BCC Authors. syscount. https://github.com/iovisor/bcc/blob/master/libbpf-tools/syscount.c.

[4] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. Netedit: An orchestration platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 721–734, New York, NY, USA, 2024. Association for Computing Machinery.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[6] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into Reduced-Privilege compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.

[8] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, page 133–144, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[12] cilium. ebpf-based networking, observability, security solution. https://cilium.io.

[13] Emma Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020.

[14] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[15] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 249–264, 2024.

[16] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[17] The Apache Software Foundation. Developing modules for the apache http server 2.4. https://httpd.apache.org/docs/2.4/developer/modguide.html.

[18] frida. Cross-platform instrumentation and introspection library written in c. https://github.com/frida/frida-gum.

[19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Cristiano Giuffrida, Călin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, page 133–144, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Google. Chrome extensions. https://developer.chrome.com/docs/extensions.

[22] Google Chrome Developers. Manifest - chrome extensions documentation. https://developer.chrome.com/docs/extensions/reference/manifest, 2024. Accessed: 2025-06-02.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[24] David Hu, Sophie Alpert, and Emily Eisenberg. Vim awesome. https://vimawesome.com/.

[25] F5 Inc. Dynamic modules | nginx documentaiton. https://docs.nginx.com/nginx/admin-guide/dynamic-modules/dynamic-modules/.

[26] Kong Inc. Load balancing - kong. https://docs.konghq.com/gateway/latest/get-started/load-balancing/.

[27] Kong Inc. Prometheus - plugin - kong docs. https://docs.konghq.com/hub/kong-inc/prometheus/.

[28] iovisor. Userspace ebpf vm. https://github.com/iovisor/ubpf.

[29] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[30] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, Hani Jamjoom, and Tianyin Xu. Fast (trapless) kernel probes everywhere. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 379–386, Santa Clara, CA, July 2024. USENIX Association.

[31] Yuzhuo Jing and Peng Huang. Operating system support for safe and efficient auxiliary execution. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 633–648, Carlsbad, CA, July 2022. USENIX Association.

[32] Alan Jowett. Cross-platform instrumentation and introspection library written in c. https://github.com/Alan-Jowett/bpf_conformance/.

[33] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant {OS} updates via userspace {Checkpoint-and-Restart}. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 605–619, 2016.

[34] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232, 2007.

[35] Linux kernel maintainers. Bpf kernel functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html.

[36] libcapstone Authors. libcapstone is a disassembly library. https://github.com/libcapstone/libcapstone.

[37] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, Savannah, GA, November 2016. USENIX Association.

[38] LuaLab. Lua the programming language. https://www.lua.org/home.html.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI05*, pages 190–200, Chicago, IL, June 2005.

[40] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Biswajit Mazumder and Jason O. Hallstrom. A fast, lightweight, and reliable file system for wireless sensor networks. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[42] MITRE. Cve-2024-31449: Redis lua script stack overflow. Common Vulnerabilities and Exposures, 2024.

[43] Mozilla. Customize firefox. https://addons.mozilla.org/en-US/developers/.

[44] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[45] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI07*, San Diego, CA, June 2007.

[46] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *12th Annual Network and Distributed System Security Symposium*, San Diego, California, 2005.

[47] NVD. Cve-2021-44790. https://nvd.nist.gov/vuln/detail/CVE-2021-44790.

[48] Inc. OpenResty. Openresty. https://openresty.org/en/.

[49] Oracle. Mysql. https://www.mysql.com/.

[50] PostGIS PSC & OSGeo. About postgis. https://postgis.net/.

[51] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[52] IO Visor Project. Bpf compiler collection (bcc), 2023. Available: https://github.com/iovisor/bcc.

[53] qmonnet. Rust virtual machine and jit compiler for ebpf programs. https://github.com/qmonnet/rbpf.

[54] Red Hat. What is selinux? https://www.redhat.com/en/topics/linux/what-is-selinux, 2024. Accessed: 2025-06-02.

[55] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[56] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.

[57] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694, 2020.

[58] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[59] Avthar Sewrathan and Bryan Clark. Top 8 postgresql extensions. https://www.timescale.com/blog/top-8-postgresql-extensions/.

[60] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In David Kotz and John Wilkes, editors, *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*, pages 170–185. ACM, 1999.

[61] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023.

[62] Asia Slowinski, Traian Stancescu, and Herbert Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 125–137, Boston, MA, June 2012. USENIX Association.

[63] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 207–222, New York, NY, USA, 2003. Association for Computing Machinery.

[64] The Emacs Community Taiwan. awesome-emacs: A community driven list of useful emacs packages, libraries and other items. https://github.com/emacs-tw/awesome-emacs.

[65] Redis Team. Redis benchmark. https://redis.io/docs/management/optimization/benchmarks/.

[66] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[67] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX Association.

[68] IO Visor. bpftrace: High-level tracing language for linux ebpf. GitHub repository, 2023. https://github.com/iovisor/bpftrace.

[69] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[70] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[71] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[72] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2989–3002, 2022.

[73] XRay. Resolving bilibili's major site incident with openresty xray. https://tinyurl.com/4ten75h2.

[74] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, 2023.

[75] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[76] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[77] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, New York, NY, USA, 2023. Association for Computing Machinery.