MorphOS: An Extensible Networked Operating System

PETER OKELMANN, Technical University of Munich (TUM), Germany
ILYA MEIGNAN–MASSON, Technical University of Munich (TUM), Germany
MASANORI MISONO, Technical University of Munich (TUM), Germany
PRAMOD BHATOTIA, Technical University of Munich (TUM), Germany

This paper introduces MorphOS, an extensible networked operating system that addresses the runtime inflexibility of unikernels for dynamic, stateful network-intensive applications like Virtual Network Functions (VNFs). While unikernels offer superior performance and minimal resource overheads, their traditional update mechanisms require costly rebuilds and restarts, leading to service disruption and state loss. MorphOS addresses this by integrating eBPF to enable dynamic, verified code execution for seamless updates to packet processing logic. It employs an out-of-band verification service to offload computationally intensive verification tasks and utilizes hardware-assisted memory isolation (Memory Protection Keys) for enhanced execution hardening. Our evaluation of MorphOS with four VNF implementations demonstrates significant benefits: MorphOS drastically reduces reconfiguration time, effectively amortizes verification costs, and achieves up to 3× better performance compared to Linux-based VNF deployments, all while preserving the inherent lightweightness of unikernels. MorphOS thus paves the way for adaptable, efficient, and state-preserving networked applications in cloud environments.

CCS Concepts: • **Networks** → **Programming interfaces**; **Middle boxes / network appliances**; *In-network processing*; • **Computer systems organization** → **Maintainability and maintenance**; • **Software and its engineering** → Input / output; Automated static analysis; Just-in-time compilers; Software safety.

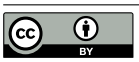
Additional Key Words and Phrases: Unikernels, Reconfigurability, Virtual Network Functions, eBPF

ACM Reference Format:
Peter Okelmann, Ilya Meignan–Masson, Masanori Misono, and Pramod Bhatotia. 2025. MorphOS: An Extensible Networked Operating System. *Proc. ACM Netw.* 3, CoNEXT4, Article 30 (December 2025), 25 pages. https://doi.org/10.1145/3768977

1 Introduction

Operating Systems (OSes) are the cornerstone that provides a performant and reliable platform to fast networked applications [22, 47, 63, 64, 67, 83, 116]. Several studies optimize OSes and network stacks to improve performance for serverless [64, 109, 113], Virtual Network Functions (VNFs) [12, 45, 89], and instance chaining [85, 93, 114]. Recently, unikernels [64] have gained attention as a promising approach among them [57, 67]. Unikernels are specialized operating systems designed to run in cloud environments and target a single application. Unikernels expose more low-level OS primitives to applications than general-purpose OSes, enabling applications to assemble, e.g., network stacks that are optimal for their specific workload. By optimizing its components at compile time, unikernels enable high performance, reduce image size, and offer rapid boot-up times, making them a compelling option for cloud deployments. This holds particularly true for performance-critical VNFs [9, 15, 42, 60, 88, 98, 101]

Authors' Contact Information: Peter Okelmann, okelmann@in.tum.de, Technical University of Munich (TUM), Germany; Ilya Meignan–Masson, ilya.meignan-masson@tum.de, Technical University of Munich (TUM), Germany; Masanori Misono, masanori.misono@in.tum.de, Technical University of Munich (TUM), Germany; Pramod Bhatotia, pramod.bhatotia@tum.de, Technical University of Munich (TUM), Germany.



ACM 2834-5509/2025/12-ART30
https://doi.org/10.1145/3768977

that offer a paradigm shift from traditional, hardware-centric network infrastructure to agile, scalable, and software-defined network functions running on OSes, e.g., in cloud virtual machines (VMs).

However, unikernel-based applications inherently face significant limitations in terms of reconfiguration and extensibility [71]. Changing processing logic requires rebuilding and redeploying not only the application but the entire unikernel. Although unikernels realize faster boot times than traditional VMs, rebooting results in losing all state, which may lead to interruption of connections or introduce a non-negligible performance overhead in the case of VNFs [67, 71]. Applications such as state-of-the-art VNFs attempt to mitigate the lack of OS-level reconfigurability with runtime parameters and configuration. However, such approaches pay for the gained flexibility with increased runtime-complexity and hence performance [33, 43].

To that end, we ask the following question: *Can unikernels provide a generic mechanism for networked applications to enable flexible live-reconfiguration while maintaining performance and safety properties?*

To address this question, we propose MorphOS, a novel extensible networked operating system designed to overcome the runtime inflexibility of unikernels. Our key insight is that a satisfactory reconfiguration mechanism must expose control flow and state management capabilities. MorphOS is first to introduce verified and JIT compiled eBPF programs to the world of unikernels (*MorphOS eBPF runtime*), realizes OS abstractions for applications to leverage eBPF for flexible live-reconfiguration (*MorphOS hookpoints*), and offers a modular high-performance network stack that integrates with hookpoints (*NetStack*). We build upon eBPF programs [68] as a suitable foundation for MorphOS because it has a well-established ecosystem of fast execution environments and safety mechanisms [19, 39].

To provide safe reconfigurability to applications, MorphOS overcomes three key challenges imposed by unikernels. *First*, existing application-level reconfiguration mechanisms are often limited in scope and lack sufficient programmability. The traditional approach of recompiling and redeploying unikernel applications not only disrupts service but also results in the loss of critical runtime state. With MorphOS, applications leverage eBPF hookpoints (§ 5.1) to make algorithms and decisions fully programmable and live-reconfigurable while retaining runtime state (§ 5.3). *Second*, the single-address-space nature of unikernel-based applications risks a surge in outages triggered by the insertion of faulty eBPF programs. Unfortunately, verifiers are not suitable for integration into lightweight unikernels. MorphOS proposes an out-of-band verification service (§ 5.4) to enforce the correctness of the eBPF programs while maintaining fast reconfiguration times and the lightweightness of the unikernel. *Third*, unikernels reduce the amount of safety isolation compared to traditional multi-tenant OSes. However, such isolation is necessary because the complexity of automated verification makes eBPF verifiers error-prone and unfit to enforce safety guarantees. MorphOS, therefore, introduces a lightweight hardening technique for unikernels based on Memory Protection Keys (MPK) available with all x86 CPU vendors (§ 5.5) that is optimized for data-intensive eBPF environments such as VNFs.

We implement MorphOS on top of Unikraft [55] and adapt the Prevail [39] verifier to offer an out-of-band verification service. We integrate the Click [67] modular router with MorphOS APIs to build a set of reconfigurable and eBPF-driven VNF prototypes called MorphClick: a forwarder, a firewall, Deep Packet Inspection (DPI), and Network Address Translation (NAT). In our evaluation, we compare these eBPF implementations with native implementations on Unikraft and Linux. We evaluate MorphOS across three dimensions: lightweight reconfigurability (§ 8.1), correctness and safety (§ 8.2), and performance (§ 8.3). The results show that MorphOS reduces reconfiguration times while maintaining the lightweightness of unikernels. The MorphOS correctness verification cost amortizes over time, and MPK-based safety hardening is effective at the cost of 25-41ns of processing added per packet. MorphOS is 1.6×-3.0× as fast as Linux while eBPF can hurt (↓10%, NAT) but also benefit (↑18%, IDS) performance compared to native implementations. All of our code is available at https://github.com/TUM-DSE/MorphOS.

Contributions. MorphOS makes the following contributions:

- **eBPF for unikernel applications**: MorphOS brings fast eBPF execution environments to unikernels via JIT compilation to tackle their lack of reconfigurability.
- **Verification with unikernels**: MorphOS introduces an out-of-band verification service as an eBPF verification model applicable for lightweight unikernels and suited to replace runtime checks of eBPF interpreters.
- **eBPF hardening for unikernels**: MorphOS proposes a hardware-assisted lightweight isolation mechanism suited to harden the safety of eBPF programs in unikernels against verifier bugs.

2 Background: OS Architectures for Network Stacks

Network stacks handle a wide variety of tasks and consist of three components: device drivers, protocol layers, and application interfaces. Network stack architectures in kernels, userspace, and unikernels differ significantly in terms of design, performance, and flexibility (see Table 1).

Feature	Kernel Stacks	Userspace Stacks	Unikernels
Performance	Low	High	Specialized
Flexibility	Low	High	App-OS codesign
App types	General-purpose	High-performance	Cloud-native

Table 1. Comparison of network/driver stacks.

Kernel and userspace stacks. Kernel-based stacks tightly integrate all three components with each other and expose rich functionality to general-purpose applications, e.g., via standard POSIX interfaces [31, 95]. However, traditional kernel stacks suffer from performance overhead due to expensive event handling (interrupts, sockets), context switches, and page table invalidations that occur for every system call [11, 26, 55]. Userspace stacks such as DPDK [5] bypass kernel limitations such as customizability and replace eventing with polling. They support high-performance workloads, including Software Defined Networking (SDN) [7, 8, 35, 43, 90] and Network Function Virtualization (NFV) [9, 33, 47, 50, 79, 82]. However, userspace stacks still run on traditional kernels, which slows down interrupts [48], limits hardware-software co-design for kernel-managed resources (e.g., IOMMU, page tables) [59, 119], and prevents fine-grained cooperative scheduling [49, 92].

Unikernel stacks. Unikernels, such as MirageOS [64], OSv [53], and Unikraft [55] are specialized OSes designed for cloud virtualized environments. Unikernels embed themselves, including the network stack, directly into the application as a library. By replacing system calls with library calls, unikernels enable additional compile-time optimizations and promote the co-design of network stack functionalities. Consequently, unikernels' performance and lightweightness reduce the attack surface, improve resource efficiency, and shorten boot times, making them well-suited for cloud-native apps such as VNFs and microservices [63, 65]. To support the need for fast and scalable networking, Linux hosts can combine unikernel VMs with fast software switches (e.g. VPP [7] with DPDK).

Unikernel performance. To demonstrate the effectiveness of unikernels, we evaluate the performance of Click [67] VNFs running on Unikraft and Linux. We measure the maximum receive/send rate, throughput, and round-trip latency (detailed in § 8). Fig. 1 shows the results. Unikraft sustains 2-3× higher packet reception rates and significantly improves transmit and latency performance. This improvement stems from unikernel's single-application design, enabling a unified address space and more compiler optimizations by replacing syscalls with function calls [64]. Our unikernel-based VNF, e.g., yields to the scheduler only after packet processing to minimize tail latencies caused by preemption.

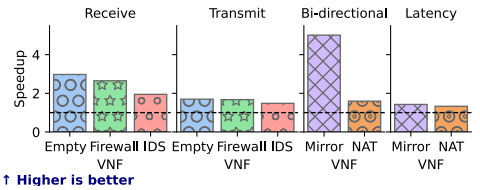


Fig. 1. The speedup of the Unikraft unikernel over the Linux kernel on Click VNFs (64B packets).

Unikernels for cloud infrastructure. Cloud services commonly rely on shared infrastructure provided by fast networked applications [16, 94, 98, 109]. Consolidating the infrastructure of different

	MorphOS	XDP [20]	ClickOS [67] / Metron [50]	BESS [43] / E2 [82]	VPP [7]	Open vSwitch [90]	xOMB [3]
Reconfiguration of the	application	kernel	application				
Implemented on the	OS-level		application-level				
Routing parameters	yes	yes	yes	yes	yes	yes	VNF dependent
Processing graph	yes	yes	yes	yes	recompile	no graph	yes
Processing logic	yes	yes	recompile	recompile	recompile	dyn. libs	dyn. libs

Table 2. Availability of live reconfiguration mechanisms in different VNF systems.

tenants on a single instance improves utilization and the benefit of unikernel specializations. Next, we explain how such an approach can negatively impact flexibility and reconfigurability.

3 Motivation: Limitations of Unikernel-based Network Stacks

While unikernels are a promising approach to realizing high-performance networked applications (§ 2), the underlying deployment models in public clouds face the following reconfigurability limitations. First, although the lightweightness of unikernels enables fast startup, restarting unikernels leads to loss of network state in memory [112]. Second, eBPF improves runtime extensibility [71], yet ensuring safety through verification is costly due to the complexity of verifiers. Third, this complexity also complicates the correctness of verification guarantees. We elaborate on each limitation below.

3.1 Lack of Flexible Reconfigurability of Unikernels

Reconfigurability is crucial for dynamically operating latency-sensitive networked applications. However, unikernel compile-time optimizations require applications to recompile and reboot for reconfiguration, introducing non-negligible overhead and potential service disruption.

Limitation of unikernel reconfiguration. To evaluate reconfiguration cost in unikernel-based networked applications, we measure the reconfiguration times for (1) the Click [54] VNF on Linux, (2) Click on Unikraft, and (3) Linux's XDP [20]. XDP accelerates network functions by running eBPF programs in the kernel, bypassing Linux's slow network stacks [20, 44]. For Click, we measure the time to reconfigure all elements, including application restarts. For Unikraft, this involves booting the unikernel VM, while for XDP, we measure program replacement and verification times.

Fig. 2 shows the result. While Click's configuration times are comparable across platforms, restarting the Unikraft kernel alongside incurs overhead. Restarts lead to network interruptions and result in the loss of memory state [112], further degrading network performance, particularly for stateful functions such as NAT or connection-based load balancers, due to the disruption of connections. Linux's promising XDP approach avoids kernel restarts using eBPF but entails costly verification, making XDP reconfiguration slower than Click on Linux.

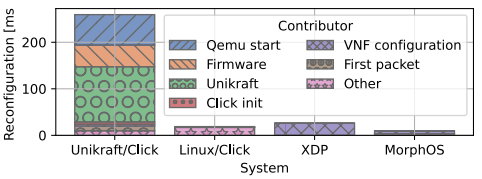


Fig. 2. Live reconfiguration of Click VNFs/XDP.

Meta's highly variable use of eBPF programs exemplifies the need for frequent, state-preserving reconfiguration, which could also unlock new use-cases [13]: On average, Meta combines three eBPF programs for a service, but on 1 out of 20 days, 13 programs are combined, e.g., to account for changing load scenarios, network contention, or VM migrations. Supporting reconfiguration beyond frequencies seen today could enable new use-cases like dynamically reacting to network anomalies by adding in-network retransmissions or blocking suspicious traffic early in the network pipeline.

Insufficient existing reconfiguration mechanisms. Actually, reconfiguration mechanisms in current networked applications also have limitations. Table 2 summarizes the mechanisms of various VNFs such as XDP [20], Click [54], VPP [7], Open vSwitch [90], and xOMB [3]. Many networked applications implement reconfiguration on the *application-level* by exposing selected options defined by the developer, e.g., high-level routing parameters in VPP or processing graph configuration in

Click/BESS [7, 43, 54]. ClickOS [67] offers a control endpoint to change processing-element-specific parameters at runtime. While effective, this approach limits the scope to pre-envisioned use cases and trades flexibility for implementation complexity and performance. Changing the ClickOS processing graph, which is typically required to modify stateful algorithms, still requires a restart. Instead, users desire an application-independent reconfiguration mechanism that facilitates state-preserving algorithm modifications. Therefore, some VNFs employ *OS-level reconfiguration*. For example, XDP allows fast updates by replacing eBPF programs in the Linux kernel (see Fig. 2). However, Linux's eBPF mechanisms only influence the kernel network stacks, but not applications, and introduce a costly verification [52]. Lastly, *orchestration-level reconfiguration* can employ, e.g., live-VM-swapping to mitigate the downtime of unikernel restarts. Nevertheless, state-of-the-art systems risk terminating live connections when the NAT connection state is lost. The burden remains on developers to extend applications with incremental migration of the critical state.

Limitation #1: Insufficient reconfiguration mechanisms. Unikernels face performance issues with reboot-based reconfiguration and a flexibility/performance tradeoff in current live reconfiguration methods, demanding a new, efficient, flexible, and state-preserving reconfiguration mechanism.

3.2 Complexity of Verifying Extensions

In Linux, eBPF enables safe kernel customization, e.g., of the network stack through event-driven execution of extensions. Its verification ecosystem [39, 80] ensures memory isolation, the absence of undefined behavior, and bounded termination, making eBPF promising for extending and reconfiguring unikernel applications.

	Linux verifier	Prevail
Lines of code	16,000	65,000
False positive rate	8 / 192	0 / 192
Supports loops	no	yes
Runtime complexity	$O(n^k)$	$O(n^3)$

Table 3. eBPF verifiers have a large code base that grows with increasing accuracy [39].

Challenges of integrating verifiers with unikernels. However, verifiers are inherently unsuitable for direct integration into unikernels for three reasons. *First, resource overhead.* eBPF verifiers, such as the one in the Linux kernel, require significant memory and processing power to perform exhaustive static analysis. This conflicts with the specialized, single-purpose nature of unikernels, which are designed to minimize resource usage and maximize performance. In the context of VNFs, where low-latency operation is essential, it is unacceptable that control-plane tasks such as verification steal significant resources. *Second, code complexity.* The verifier's implementation is complex and adds substantial code size to the unikernel, contradicting the unikernel's lightweight design. Prevail consists of 65k lines of code and requires a C++ runtime, bloating its binary size to 2.4MB, comparable to the Unikraft kernel. *Third, isolation.* Verifiers themselves are prone to malicious attacks due to their large attack surface, as demonstrated by recent CVEs [77, 78]. Unikernels, as single-application OSes, do not provide sufficient isolation mechanisms to safely co-locate a verifier with an application. Table 3 summarizes the complexity of the eBPF verifiers.

Limitation #2: Impractical On-Unikernel eBPF Verification. The complexity of eBPF verification hinders its adoption in high-performance unikernels, necessitating a lightweight mechanism to ensure correct eBPF extensions for safe unikernel extensibility.

3.3 Verification Correctness Challenges

Limitations of eBPF verifiers. The complexity of eBPF verifiers makes it increasingly difficult to generate proofs as eBPF programs grow more intricate. We survey related work that mentions Common Vulnerabilities and Exposures (CVEs) [72] in existing eBPF verifiers (see § A) and find that 23 out of 32 vulnerabilities are a result of bugs in the verifier [61, 62, 66, 117]. Despite ongoing efforts, verification of the Linux eBPF verifier is incomplete [66, 110, 111]. As an example, Fig. 3 shows a CVE found in the Linux verifier [76] where certain eBPF programs escape the verification. The

program starts a loop at `L1`, loading a value from memory to be used as the loop break condition. The verifier, unfortunately, uses an erroneous offset calculation for `L1`, starting the loop verification in line 2 instead of 1. This mistake causes the verifier to wrongly prune subsequent execution paths as unreachable. eBPF can use the subsequent iterations to generate an unverified program state that accesses private kernel data. While memory isolation is a key source for security vulnerabilities [103], 76% of CVEs in the Linux verifier affect memory isolation [61].

Limitation #3: Reliability & safety gaps in eBPF verifiers. Due to the complexity of eBPF verifiers, their reliability and ability to guarantee memory isolation and safety are limited. Additional memory isolation hardening is necessary for network infrastructure applications shared between tenants.

```
1 L1: r11 = *(u64 *)(r10 -8) // start loop
2    if r11 == 0x0 goto L2  // exit loop
3    r11 -= 1
4    *(u64 *)(r10 -8) = r11
5    goto L1              // repeat loop
6 L2: w0 = 0
7    exit
```

Fig. 3. CVE-2024-42072: The verifier misinterprets the jump label to `L1` as line 2, missing verification of the loop.

4 Overview

To overcome the limitations of specialized unikernel applications, we design MorphOS along three design goals: (1) keeping unikernels lightweight and reconfigurable for networked applications, (2) safe multi-tenancy through verification, and (3) hardening safety against verifier bugs.

4.1 MorphOS's System Architecture

MorphOS is a unikernel that provides abstractions and infrastructure for applications to leverage reconfigurable programs. Due to the multi-tenancy of cloud networked applications, traditional dynamically loaded libraries provide insufficient isolation guarantees. To this end, MorphOS introduces eBPF programs (abbreviated as "programs") as an additional layer on top of the unikernel-based application. Fig. 4 shows its architecture, which consists of three building blocks: First, the *control plane* exposes an external interface for tenants to reconfigure the unikernel by replacing eBPF programs in the eBPF runtime. Second, the *eBPF runtime* maintains the active eBPF programs along with their state. The runtime guards transitions into and out of the eBPF context to isolate tenant-provided eBPF programs. Third, the *data plane* runs the core application logic in the unikernel. MorphOS provides the application with the means to define custom hookpoints, optimized network access, and isolated memory pools to efficiently ensure data security at the boundary of eBPF programs.

Deployment model: MorphOS as a service. We envision cloud providers to offer MorphOS-based services as optimized applications, following existing deployment models (§ 2) and relieving cloud tenants from developing their own. MorphOS minimizes infrastructure overhead by co-locating cloud-tenant applications on a single, high-throughput unikernel instance that multiplexes tenant connections between the tenant's eBPF programs. Like the existing configuration, cloud providers expose MorphOS's eBPF updating mechanisms via APIs to provide flexibility to tenants.

Safely isolating eBPF. MorphOS safely isolates eBPF programs from the unikernel and its application, which is critical in the envisioned deployment model where performance-optimized networked applications are shared across tenants. Providers must balance tenant programmability with maintaining the unikernel/application security and functionality. Therefore, MorphOS employs an out-of-band *Verification service* that statically verifies the safety of a given eBPF program. The eBPF JIT runtime in the unikernel restricts the eBPF program to only call safe, predefined functions. MorphOS only accepts verified eBPF code to maintain its integrity and trust.

4.2 MorphOS Workflow

The general workflow of MorphOS applications is as follows. First (Fig. 4 ①), cloud service providers build highly optimized networked applications such as the flexible Click [54] VNF with MorphOS and offer them as a hosted service that is shared between tenants for efficiency. Second ②, tenants using

the service extend its functionality with eBPF programs and submit them to the provider's eBPF registry. One tenant may deploy packet filters for his endpoints to protect them against DDoS attacks, while another may deploy deep packet inspection to secure internal traffic between his services. The verification service certifies the safety of submitted programs and yields a cryptographic certificate on success. Third ③, tenants insert verified eBPF programs from their collection into MorphOS via a dedicated network link managed by the *Control endpoint*. Fourth ④, the control plane validates the certificate, JIT-compiles the eBPF bytecodes into native instructions, and attaches the verified program to the corresponding hookpoint in the application. Next, we describe how providers make applications more reconfigurable using MorphOS hookpoints.

4.3 Programming and Threat Model

Provider: Application programming. MorphOS enables providers to build networked applications that process network input and return responses. It offers a POSIX-compatible development environment with libc to simplify app creation. For high-performance processing of received packets ⑤, applications can also directly access network drivers via the lightweight *NetStack*. MorphOS provides isolated buffer pools for eBPF I/O to enable efficient data sharing with the MPK isolation domains that run eBPF programs. To allow for reconfigurability and extensibility, providers can embed eBPF hookpoints into their applications that are called ⑥ as the application processes network traffic. Additionally, MorphOS includes predefined eBPF function types and helper functions that cloud providers can extend to suit specific application requirements.

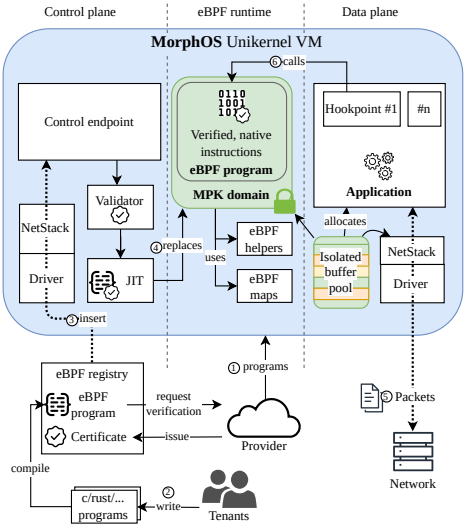


Fig. 4. MorphOS makes specialized unikernels configurable and safely extensible with eBPF.

Tenant: Configurability through eBPF programming. Tenants write eBPF programs to customize provider applications at defined hookpoints. These programs are compiled from languages like C and Rust and use provider-defined helper functions for safe OS access. To ensure safety, tenants acquire a certificate from the provider's verification service when pushing to the eBPF registry. Once certified, tenants update eBPF programs in MorphOS at runtime to modify application functionality.

Threat model. MorphOS faces a primary threat from cloud tenants deploying potentially malicious or buggy eBPF programs. These untrusted programs could access private application data, cause denial of service by hogging CPU or memory resources, or breach the sandbox and VM isolation to compromise system integrity. The model assumes a trusted cloud provider manages the eBPF verification service and applications while tenants develop eBPF programs to customize functionality. Although the verifier is the initial defense, MorphOS does not consider it infallible. Thus, runtime checks for resource consumption and hardware-assisted isolation (e.g., MPK) are crucial for an additional protection layer to detect successful attacks. Aside from the MorphOS instance, the verifier exposes a significant attack surface to tenants. MorphOS decouples the verifier into a stand-alone service to be deployed on hardened clusters, e.g., in isolated and ephemeral VMs. Orthogonal work hardens against VM escape attacks and hardware side-channels, and improves cryptographic ciphers.

4.4 Key Ideas

We present the following key ideas of MorphOS to make cloud apps extensible and reconfigurable.

#1 OS-level support for application reconfigurability. Due to their single-application nature, unikernels have drawbacks in terms of flexibility and reconfigurability. Unfortunately, application-level configuration mechanisms are often incomplete and lack programmability (§ 3.1).

MorphOS overcomes these limitations with a generic live-reconfiguration mechanism built into the OS that makes VM/application restarts obsolete and avoids losing state. *MorphOS exposes access to eBPF hookpoints*, which unikernel applications use to make algorithms and decisions fully programmable and reconfigurable in a live manner (see § 5.1, § 5.3).

#2 Out-of-band eBPF verification. Powerful eBPF verifiers have large code bases and exhibit high runtime complexity, impeding their integration into kernels. Prevail [39] approaches this problem by moving verification into a dedicated process [70]. Unfortunately, this approach does not generalize to unikernels, which do not support multi-processing (§ 3.2). Furthermore, this approach requires integrating many libraries that the verifier depends on into the unikernel as well.

MorphOS decouples resource-intensive verifiers from the unikernel through cryptographic certificates and instead runs them out-of-band on different hosts (§ 5.4). Ahead-of-time verification accelerates updating eBPF programs significantly while adhering to the design goals of unikernels.

#3 eBPF hardening with MPK. Current real-world eBPF verifiers are still improving their safety guarantees (§ 3.3) because the verification process is complex. eBPF verification is hard to implement correctly, as demonstrated by related work that has repeatedly uncovered flaws in verifiers [28, 61, 62, 117]. Unverified JIT compilers add another potential point for safety violations.

Today's eBPF systems are therefore not fit to enforce isolation guarantees between cloud providers and tenants. Deployment scenarios envisioned for MorphOS require safety hardening that goes beyond correctness guarantees. *MorphOS hardens the unikernel against eBPF escaping verification* with MPK-based memory isolation (§ 5.5).

5 Design

5.1 The OS Abstraction

MorphOS presents a dual-layer abstraction model designed to enable dynamic system reconfiguration through eBPF-based extensibility. Existing systems lack the necessary runtime definition necessary to integrate eBPF, the verifier, and MPK hardening. The first layer, *MorphOS hookpoints*, concerns the application developer. The MorphOS hookpoint abstraction provides the developer with conveniently accessible infrastructure to seamlessly enrich applications with reconfigurable functionality implemented with eBPF. The second layer, *MorphOS eBPF runtime*, concerns the cloud tenant who uses MorphOS applications. In this layer, MorphOS provides a base runtime environment with essential but generic functionality for tenants to extend applications. Application developers can extend the eBPF runtime layer with their application-specific helper functions. MorphOS exposes each layer via a dedicated interface, which we describe next.

Type	Signature

Load eBPF	`morphos_set_cert_authority(ca);`
	`morphos_register_helper(id, helper);`
	`morphos_share_map(id, map_name, id);`
	`morphos_load(id, file, signature, use_jit);`

Runtime	`morphos_alloc_input_buf();` `morphos_call(program, inputbuf, params ...)`

Table 4. *MorphOS hookpoints* allow apps to safely extend their functionality with eBPF programs.

MorphOS system APIs. Table 4 presents the system API for MorphOS applications. The API includes functions for loading, managing, and executing eBPF programs while ensuring security through certificate validation and controlled resource sharing. `morphos_set_cert_authority()` sets a trusted certificate authority for eBPF program signatures. Next, the application customizes the eBPF runtime by registering native helper functions that eBPF programs can invoke (`morphos_register_helper()`) and setting up eBPF state to be shared between invocations and programs (`morphos_share_map()`). The `morphos_load()` function handles program loading consisting of just-in-time (JIT) compilation, relocation of executable code, and linking available helper functions. At runtime, `morphos_alloc_input_buf()` ensures secure memory allocation for input data, and `morphos_call()` executes eBPF

programs with specified parameters. Together, these functions provide a robust and secure mechanism for extending applications in MorphOS, balancing flexibility with strict security guarantees.

MorphOS eBPF interface. From the eBPF context, MorphOS and other external functionality are only available through safe wrappers called *helper functions*. The *MorphOS eBPF runtime* interface defines these helper functions for the eBPF program (Table 5). The interface is divided into four types: the *(a)* eBPF entry point, *(b)* OS functionality, *(c)* network buffer management, and *(d)* eBPF maps. (a) The entry point defines the eBPF program's entry function signature, which includes pointers to input buffers, as well as packet metadata such as port numbers. (b) The OS-type API offers several default helper functions to get time, randomness, or do logging. (c) The network buffer API offers helpers such as `bpf_grow_buffer()` to dynamically adjust packet buffer sizes (e.g., for encapsulating packets in tunnel protocol headers). (d) Map helpers enable accessing inter-invocation state through lists or key-value maps. Functions like `bpf_map_lookup/update_elem()` allow eBPF programs to access and modify data in maps, supporting tasks like flow tracking and stateful processing.

5.2 Network Stack

MorphOS introduces *NetStack*, a robust and modular network architecture designed to support dynamic reconfigurability and high-performance packet processing. The *NetStack* abstraction encompasses drivers, network protocol stacks, application buffer management, and event callbacks. Applications use this abstraction to either choose a minimal set of stack components to minimize overhead or use the full stack through the POSIX-like API. Central to the design of *NetStack* are the buffer management, the stack component bypass, and the event model.

Type	Signature
A) Entrypoint	`struct bpf_context { void* packet, void* packet_end, u64 input_port }` `bpf_program(input: bpf_context) -> u64`
B) OS	`bpf_ktime_get_ns() -> u64` `bpf_get_prandom_u32() -> u32`
C) Net bufs	`bpf_grow_buffer(front: usize, back: usize) -> usize`

D) Maps	`#[map(NAME)]`
	`static NAME: {HashMap<K, V>|Array<A>}`
	`bpf_map_lookup_elem(NAME, key: K) -> V`
	`bpf_map_update_elem(NAME, key: K, value: V, flags: u64)`
	`bpf_map_delete_elem(NAME, key: K)`

Table 5. MorphOS eBPF runtime interface.

Central buffer management. The eBPF context, the application, and the operating system share a central memory pool (isolated buffer pool in Fig. 4). This design eliminates the need for redundant memory copies between the application, MorphOS, and eBPF runtimes, enabling zero-copy data transfers. Consequently, applications minimize overhead while maintaining buffer isolation with hardware-assisted memory hardening (§ 5.5).

Network stack bypass. By allowing applications to bypass protocol stacks, MorphOS achieves functionality akin to common kernel-bypass architectures such as DPDK [5]. MorphOS ensures efficient resource and power usage outside of maximum-throughput scenarios by maintaining interrupt-driven event processing for packet handling. Although 32 of 70 DPDK drivers support interrupt mode [4], many DPDK applications do not use them, significantly degrading application packing density. Unlike DPDK, MorphOS optimizes its event scheduling and buffer management for the single-CPU architecture of unikernels, enabling cloud providers to substantially increase VM-to-host ratios [55]. The MorphOS approach contrasts with architectures of general-purpose OSes like Linux, where the network stack introduces significant overhead due to its size and reliance on system calls. These factors increase latency and additional memory copies, hindering performance.

Event model optimized for performance. MorphOS minimizes event handling by directly invoking application callbacks in a run-to-completion model, reducing overhead and eliminating memory copies. It is optimized for single-core execution, maximizing lock-free parallelism when scaling out. This design aligns with the cloud computing paradigm of scaling by allocating more instead of larger VMs. *NetStack* is particularly advantageous for applications where traffic flows are often partitioned in hardware (e.g., using RSS [69] or flow steering VNFs [9]), ensuring that each

CPU core requires only local state. We envision MorphOS deployments to follow these principles, leveraging its lock-free network stack to maximize performance and scalability.

5.3 eBPF Extensibility

The MorphOS abstraction (§ 5.1) increases the reconfigurability of unikernel-based applications through three key components: (1) the endpoint for reconfiguration, (2) the JIT compiler for updating eBPF programs, and (3) the state management for preserving information across reconfigurations.

Reconfiguration mechanism. MorphOS enables tenants to reconfigure applications over the network via the *control endpoint* by replacing eBPF programs. This process works as follows. Tenants first send a new eBPF program to the verification service. Once verified, this eBPF program is stored in the provider's registry. Subsequently, tenants send a reconfiguration command to the MorphOS management network interface to request to set the new program for a certain eBPF hookpoint. Upon the request, MorphOS loads the eBPF program from the registry, verifies its certificate, relocates its code in memory, instantiates eBPF maps, and JIT-compiles the eBPF instructions into native code. Lastly, MorphOS installs the JITted eBPF program into the corresponding eBPF hookpoint.

This registry-based design helps reconfigurations to scale out and remain predictable. First, MorphOS separates the program upload from the reconfiguration command to enable timely insertion of the same program into many MorphOS instances without resubmitting the entire program each time. Second, the design decouples the tenant's eBPF program upload speed from the time-critical reconfiguration path and relieves the tenant from managing a catalogue of verified programs.

JIT compiler. The JIT compiler translates eBPF bytecodes into native instructions to enable high-performance eBPF processing. The design of eBPF bytecode inherently accounts for JIT compilation [51], allowing the compiler to map most eBPF registers directly to native registers in a one-to-one manner. For stack management, the compiler allocates per-program memory regions at page granularity to support MPK isolation (as discussed in § 5.5). To handle the helper functions calls from the eBPF program, the JIT compiler inserts trampoline code. This trampoline saves eBPF register states and converts them to match x86 calling conventions, addressing differences in stack alignment, register preservation, and argument passing. It restores the original state after the function call completes. During MorphOS context switches between eBPF and unikernel environments, the system updates MPK permissions to harden isolation between their distinct stacks and memory domains.

State migration. MorphOS maintains state during eBPF program updates by sharing eBPF maps. Each map gets a unique ID that stays constant across updates. When a new program loads, maps with existing IDs are reused to preserve the state. For data structure changes, new maps can be defined, and developers handle lazy state migration to them in the new eBPF program.

5.4 Decoupled eBPF Verification

Verification of eBPF for unikernels is challenging because automated verification is time and resource-intensive (§ 3.2). MorphOS decouples verification from eBPF injection and enables out-of-band verification to generate a catalog of verified programs to be inserted on-demand.

Verification goals. The MorphOS verifier guarantees that three protection goals hold: The verifier maintains the *operational integrity* of the application and Unikernel, even when programmable by cloud tenants via MorphOS's eBPF elements. For *confidentiality*, the verifier ensures eBPF programs never access data from other MorphOS hookpoints or neighboring tenants. Finally, the verifier ensures *availability* by verifying that eBPF programs will not cause system crashes or block execution, preventing malicious tenants from denying service in shared instances.

Verification process. With MorphOS, the cloud provider offers a verification process that operates outside of unikernels and yields a cryptographic certificate attesting to the program's safety.

We base our verification process on the Prevail [39] verifier to ensure these goals. The verifier guarantees integrity, confidentiality, and bounded execution through a precise analysis of all possible execution flows, and it proves that no illegal memory access occurs. Prevail achieves that by keeping a detailed track of memory values and their possible ranges while avoiding path explosion by merging states across loop iterations. We extend Prevail to understand the guarantees of memory regions used by output values of MorphOS helper functions and to capture the semantics of eBPF function parameters, such as buffer pointers, used in MorphOS hookpoints. § B details how MorphOS supports eBPF that contains loops, receives packet contexts as input, and accesses shared maps.

Finally, users submit the eBPF executable along with the certificate to the MorphOS application. The application checks the certificate using asymmetric cryptography and attaches the eBPF program to the target hookpoint. Thanks to the out-of-band design of the verifier, cloud providers can flexibly dedicate resources to the verification process.

5.5 Hardware-Assisted Memory Safety

The complexity of verification raises concerns about the correctness of its process (§ 3.3). To mitigate this issue, we employ hardware-assisted memory isolation with MPK [2, 46, 81], which is available on x86 and ARM. Table 6 summarizes the isolation properties we provide.

Isolation properties	Verifier	MPK
Helper bugs	×	(✓ maps)
JIT bugs	×	✓
Isolate unikernel from eBPF	✓	✓
Isolate between eBPF	✓	✓
Temporal safety (packet buffers)	✓	✓
Read uninitialized memory	✓	×
Termination	✓	×

Table 6. Verification properties of Prevail [39] and hardening with MPK [2, 46].

MPK. MPK uses the upper bits of the page table entry to assign one of 16 domains to each page (PKey). The CPU's `PKRU` register configures the permission for each domain (read/write bits) for the executing thread. When programs access pages that violate the permissions set in that register, the CPU triggers an exception. Programs use the `WRPKRU` instruction to update the `PKRU` register. This operation is fast (~20 cycles) because it does not require TLB flushes [84].

MPK isolation workflow. When desirable, service providers enable MPK hardening. MorphOS then assigns distinct PKeys to separate memory of the unikernel, applications, eBPF programs, and individual packet buffers. Fig. 5 illustrates the PKey assignments for two tenants sharing a MorphOS instance: K_0 for the unikernel memory, K_1 for an eBPF program's memory, and K_2 for another cloud tenant's eBPF program. Packet buffers are individually protected using the remaining keys K_{3-15}.

By default, MorphOS sets permissions in `PKRU` to P_{uk}, granting access to all memory regions. As the application processes packets, some of them trigger eBPF hookpoints that invoke eBPF programs. When executing eBPF programs, MorphOS updates `PKRU` to operate under P_{eBPF} permissions, which can only access its own memory K_1 and its current packet buffer K_m. When calling helper functions, MorphOS changes the MPK domain to P_{uk}, allowing the helper to access OS and application services, and switches back to P_{eBPF} on return.

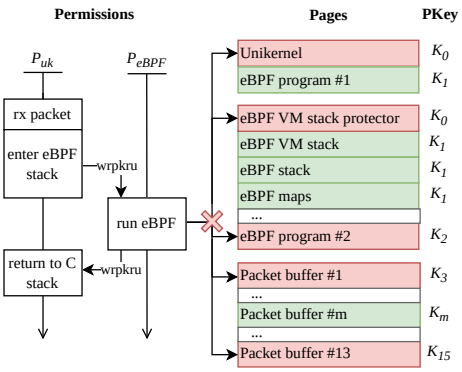


Fig. 5. Permissions and PKeys used for MPK.

To mitigate the limited number of hardware PKeys, MorphOS proposes probabilistic eBPF isolation by assigning K_{1-15} pseudo-randomly to eBPF programs and packet buffers, accepting permission overlaps. § C explains how unlikely the success of attacks becomes with probabilistic isolation. Alternatively, we can also resort to libMPK [84] to achieve deterministic *and* scalable eBPF isolation, overcoming hardware PKey limits through key virtualization.

To support packet batching with probabilistic isolation, MorphOS needs to keep all buffers of a batch in the same MPK domain. When the application splits up batches, e.g., to submit a portion of packets to a different branch of the processing graph, MorphOS needs to change the domain of the diverging buffers. Due to a lack of batching support in the Click version ported to Unikraft, we omit a further exploration of batching designs.

6 MorphOS for VNFs

We envision MorphOS to be used for a variety of cloud applications such as Virtual Network Functions (VNFs) [9, 15, 42, 60, 88, 98, 101] and network-optimized microservices [64, 102]. Performance-critical VNFs particularly profit from MorphOS because VNFs offer a paradigm shift from traditional, hardware-centric network infrastructure to agile, scalable, and software-defined network functions running in cloud VMs. This section explains how we extend the Click [54] VNF with eBPF hookpoints and implement firewalls, Network Address Translation (NAT) [29], and Deep Packet Inspection (DPI) [27, 56, 120] that are fully reconfigurable while retaining state.

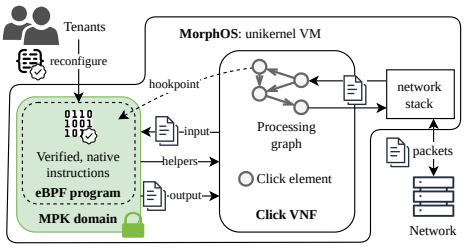


Fig. 6. MorphOS for optimized VNFs as a service.

Integrating Click. We integrate the Click [54] modular software router with MorphOS as a suitable base for highly flexible and reconfigurable VNFs (see Fig. 6). Click users define a packet processing graph by (re-)configuring directional edges between processing modules called *elements* to assemble or change higher-level VNFs. For MorphOS, we add eBPF-based elements to be integrated into processing graphs with different capabilities ranging from filtering (`BPFFilter`), over packet sorting (`BPFClassifier`), to modifying packets (`BPFRewriter`).

Fast, JITed firewalls. Click already offers firewall functionality using elements that iterate over Access Control Lists (ACLs). However, configurable systems such as ACL interpreters prioritize runtime flexibility over compile-time optimizations, resulting in sub-optimal performance [33]. MorphClick instead uses a `BPFClassifier` element where firewall rules are implemented as code that is JIT compiled, eliminating the ACL interpreter and reducing processing overhead.

Frequent reconfiguration of DPI. DPI describes network functions that extensively analyze packet contents, e.g., to enhance firewalls, detect DDoS traffic, or search patterns for Intrusion Detection Systems (IDSs). DPI is hard to generalize for configurability via parameters or data tables, as inspection logic depends on inspected protocols and the application data, leading to either incomplete or inefficient implementations. We instead design a DPI prototype for MorphClick using the MorphOS `BPFFilter` element with a string matching algorithm implemented in eBPF. Fast reconfiguration of MorphClick control flow empowers VNF users to quickly update processing logic to react to changing traffic patterns or apply new defense mechanisms, e.g., against DDoS attacks.

Retaining NAT state. NAT, rate-limiting, and traffic shaping VNFs track per-connection state that is crucial for reliable network connectivity. Restarting such VNFs drops that state, thereby terminating all current connections (NAT) or degrading network conditions (limiting, shaping). Using MorphClick, we build an eBPF-based VNF that implements the NAT core using the MorphOS `BPFRewriter` element. This design replaces VNF restarts with state-preserving eBPF updates while maintaining flexible reconfigurability, therefore eliminating problems induced by state loss.

7 Implementation

We implement a prototype of MorphOS based on the Unikraft [55] unikernel (v0.16.3). MorphOS implements a verification service based on the Prevail [39] verifier (`9f25cee`). We port Click [54]

(a538483) as VNF and uBPF [6] (2c2a682) as a eBPF JIT engine to Unikraft. We extend Unikraft's libclick [97] and integrate it with MorphOS's eBPF hookpoints (§ 6). MorphOS optimizes transmissions from Click to Unikraft by introducing batching and reducing calls to the cooperative scheduler between transmissions. Furthermore, we extend scheduling regarding the ToDevice element to support pull and not only push operations, a critical capability necessary for many single-threaded packet processing graphs. Finally, MorphOS integrates the *isolated buffer pool* and leverages Unikraft's cooperative scheduler to optimize network performance.

The eBPF runtime. MorphOS supports both eBPF interpretation and JIT compilation via uBPF [6]. We extend uBPF to support read-only ELF sections, such as `.data`, that occur in our eBPF binaries compiled from Rust. MorphOS loads the new program by parsing the ELF file to locate function and data sections, copies them into unikernel memory, and rewrites eBPF instructions to access or call the relocated items. eBPF programs specify the used map types at compile time in the `.maps` ELF segment and can read/insert at runtime with helper functions. The eBPF map definitions from the `.maps` section are not relocated but used to allocate and instantiate maps.

The verification service. MorphOS provides an executable that verifies eBPF programs and yields a cryptographic signature if safety can be proven. To conduct verification with Prevail, MorphOS creates an eBPF platform consisting of program input layout, eBPF maps, and helper functions: First, we define the *context* struct that acts as input to each function and points to the bounds of the accessible packet buffer. Second, we define the in-memory layouts of available eBPF *map* types. Third, we define helper function signatures and their parameter types and associated semantics so that Prevail can verify assumptions made by the helper implementations. If program verification succeeds, the service uses OpenSSL to sign a SHA256 hash of the verified program with the ECDSA key of the unikernel administrator (i.e., the cloud provider).

Hardening with MPK. To use MPK, we set the user flag in the page table entries since MPK only works for user pages on our CPUs. MorphOS modifies the allocator to place eBPF-related memory on its own set of pages and tag them with MPKeys. MorphOS splits the stack of eBPF programs from the unikernels stack and switches between them when entering P_{eBPF} contexts so that the unikernels and eBPF stack are isolated from each other with different MPK permissions. In addition, MorphOS modifies the unikernel and JIT compiler to switch `PKRU` between the respective eBPF context permissions P_{eBPF} necessary for a given input buffer and unikernel context permissions P_{uk}.

8 Evaluation

We evaluate MorphOS across the following dimensions: lightweight reconfigurability (§ 8.1), correctness and safety (§ 8.2), and performance (§ 8.3).

Experimental setup. We conduct measurements on two hosts, where one acts as a network load generator (Linux pktgen [108]) and the other one hosts the VM under test. The hosts are connected with 10G links via Intel X520 NICs. For comparability, we run both Linux and Unikraft baselines in single-core VMs because Unikraft is designed for single-core cloud VMs. The hosts use 256GB of RAM and an Intel Xeon 5317 CPU. We use Linux 6.6.1 and Qemu 8.2.6. We provide networking to the VM via VPP 24.06 using DPDK drivers and vhost-user, which we find is faster than Linux network virtualization with vhost.

Experiment variants. We evaluate three VNF systems: as baselines we use Click on Linux (*Linux*) and Click on Unikraft (*Unikraft*) where all Click elements are implemented natively. We compare them to MorphClick (*MorphOS*), which implements core processing in eBPF without hardening. Each system supports our implemented network function use cases (§ 6): a firewall that alternates between permitting and dropping traffic to UDP/TCP ports, an Intrusion Detection System (IDS) that applies

AhoCorasick [1] string matching to identify malicious packet contents, a simple ethernet packet reflector, and a hairpin Network Address Translation (NAT) that accesses state for every packet.

8.1 Lightweight Reconfigurability

RQ1. *Does MorphOS improve live-reconfigurability of unikernel-based applications while maintaining their lightweightness?* We evaluate VNF reconfiguration time and VM image lightweightness.

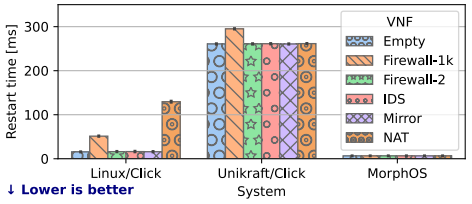


Fig. 7. Live reconfiguration times of VNFs.

Reconfiguration time. We evaluate the reconfigurability of VNFs by measuring the time required to update packet processing logic across three different reconfiguration mechanisms. We use the baselines from § 3.1 and compare them with the time that MorphOS takes to replace eBPF programs. Fig. 7 shows the result. We observe that reconfiguring Click on Unikraft takes 7× longer than Linux because it takes 230ms to restart the unikernel. On Linux, Click reconfiguration times vary depending on the complexity of the VNF configuration. The NAT configuration takes 100ms to configure many protocols and services known by Linux, while Unikraft avoids this overhead by limiting the set of services. MorphOS achieves the fastest reconfiguration, requiring only 18% of Linux's reconfiguration time, because it only updates eBPF programs without updating whole components. MorphOS spends 45% of the time on network communication with the *control endpoint* to trigger the update and only 3.6ms for loading and updating the new eBPF program. The example of firewalls with 1000 rules (firewall-1k) demonstrates the advantages of efficient eBPF reconfiguration as the baselines spend an additional 20ms on initializing the firewall rules.

VM image size. We investigate the impact of MorphOS on the VM size by measuring disk image sizes. Images consist of the unikernel or Linux distribution image containing Click, as well as the configuration files and eBPF programs. We compare MorphOS with the Unikraft unikernel and Linux-based OSes (Alpine Linux (3.19) and Ubuntu (22.04 LTS)).

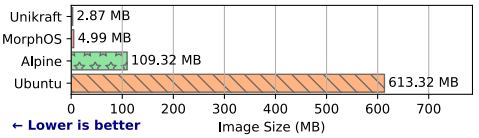


Fig. 8. VM image size of Click in Unikraft and different Linux distributions.

Fig. 8 shows that MorphOS adds around 2MB to the Unikraft image size, with OpenSSL as the biggest contributor. MorphOS is orders of magnitude smaller than even lightweight Linux images such as Alpine, but also than Ubuntu, which bundles many more tools, features, and a larger kernel.

RQ1 takeaway: MorphOS significantly reduces reconfiguration time by making even VNF logic live-reconfigurable with eBPF, without sacrificing the lightweightness of unikernels.

8.2 Correctness and Safety

RQ2. *Does MorphOS effectively enforce correctness and harden safety, and what overhead does the hardening induce?* We measure verification-related contributors to reconfiguration time and analyze how they may amortize over time. We then evaluate the effectiveness of our MPK hardening.

Verification-induced reconfiguration time. We conduct white-box measurements to analyze where MorphOS spends time while replacing eBPF programs. First, we identify and measure tasks related to enforcing safety. Second, we calculate how the reconfiguration time can be amortized over time.

There are two sources for reconfiguration time: the eBPF reconfiguration, where the program is replaced, and MorphOS's out-of-band verification. We subdivide both sources into various sub-tasks and measure their execution time for the NAT VNF.

First, Fig. 9 (a) shows that the verification time is significantly longer than the reconfiguration time. During out-of-band preparation of eBPF programs, 25% of time is spent on verification, and 69% of time is spent on the compilation. On the other hand, during reconfiguration, the validation of the verification certificate accounts for only 5% of the time.

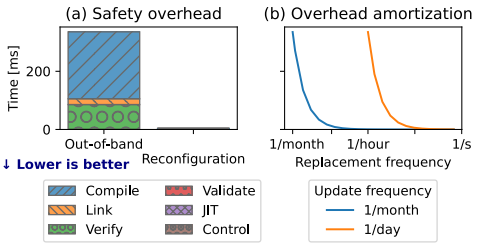


Fig. 9. (a) Time spent on ensuring safety during out-of-band verification and live-reconfiguration. (b) Out-of-band overhead amortizes over replacements of eBPF programs.

Second, we analyze the amortization of out-of-band tasks. Out-of-band verification is only necessary when developers update and recompile the eBPF code. A case study about tens of eBPF programs at Meta reveals that reconfigurations happen more frequently than code updates: Meta employs NetEdit [13] to manage frequent and dynamic eBPF reconfigurations, which surpass manual review capabilities. Despite this, Meta changes eBPF code only with a frequency f_u up to once a month [13]. On the other hand, the reconfiguration overhead occurs with a frequency f_r. The out-of-band verification time t_v amortizes over multiple reconfigurations to $t_a = t_v * f_u / f_r$. Fig. 9 (b) shows the amortized overhead t_a for different reconfiguration frequencies. For daily reconfigurations ($f_r = 1/\text{day}$), the original 335ms out-of-band cost reduces to 0.5ms. In such cases, MorphOS achieves reconfiguration in 3.4ms, rendering compilation and verification costs negligible through amortization.

Hardening with MPK. Next, we evaluate the effectiveness of MorphOS's use of MPK to harden security. We survey literature that reports bugs in existing eBPF systems [61, 62, 66, 117]. We then classify the bugs by attack vector and test if MorphOS's MPK isolation protects against such attacks.

Fig. 10 shows our identified bug classes. The majority (69%) originate in verifier bugs that mispredict how register values change or how the execution flow branches. Further, 25% of bugs stem from faulty helper functions that do not adhere to the model guaranteed to the verifier.

CVEs	MorphOS hardening	Category
12	yes	Verifier: register value tracking: *e.g. verifier predicts eBPF null check to pass when it must not*
8	only map get helpers	Helper functions: *return value must point to eBPF memory*
6	yes	Verifier: branch pruning: *the verifier's model of jumps matches the JIT compiler's model (see Fig. 3)*
4	yes	Verifier: context value tracking: *context pointer must not be null*
2	no	Verifier: crash

Fig. 10. MorphOS's hardening with MPK is effective against common vulnerabilities (CVEs § A).

We then assess against which bug classes the MorphOS hardening is effective. For bugs in register value tracking, branch pruning, or context value tracking, MPK effectively protects MorphOS by triggering an emergency stop once the malicious program proceeds to access private memory. MorphOS also detects programs that exploit eBPF map get helpers to return private data as they run in the isolated eBPF MPK domain. Against bugs in other helpers, our hardening is ineffective because these helpers must run in the un-isolated domain to access the unikernel or application memory. To summarize, the hardware-assisted safety hardening of MorphOS protects against a big class of attacks, effectively improving the safety of applications.

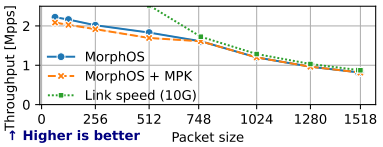


Fig. 11. MorphOS (firewall) with and without MPK-based hardening.

MPK performance overhead. We evaluate the overhead of the MPK hardening using the firewall VNF. We compare MorphOS maximum receive throughput with and without MPK hardening enabled. Fig. 11 presents the results. For small-sized packets, MPK introduces up to 5-7.5% overhead (25-41ns

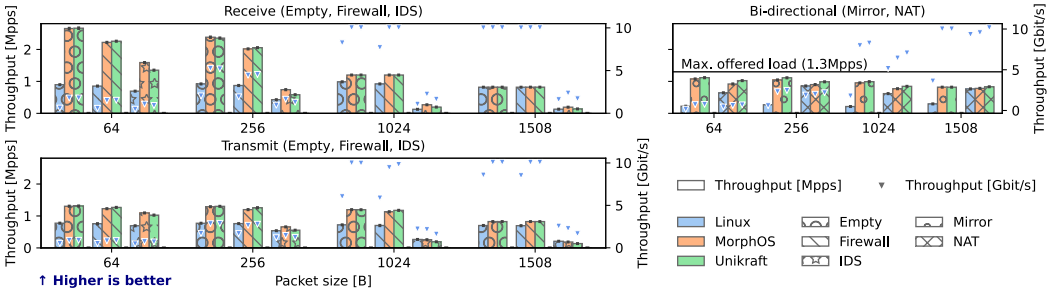


Fig. 12. VNF throughput with MorphOS, Linux, and Unikraft. MorphOS has 1.5×-2.9× lower per-packet overhead than Linux. For larger packet sizes, throughput in Gbit/s ▼ approaches the 10Gbit link speed.

per packet) on the `WRPKRU` instruction inserted by the JIT compiler to change PKey permissions around the eBPF program. As the packet size and processing time increase towards the 10Gbit link speed, the hardening overhead diminishes and becomes negligible.

RQ2 takeaway: MorphOS out-of-band verification amortizes over time, making verification costs negligible for scenarios found in the industry. MorphOS hardens effectively against several safety violation categories at the cost of up to 41ns additional processing time per packet.

8.3 Performance

RQ3. *Does MorphOS's use of eBPF introduce performance overhead?* First, we evaluate the throughput and latency of our use cases. Second, we compare parametrized firewalls with our eBPF-based one.

Throughput. We evaluate the throughput of five VNFs. We measure receive and transmit performance by generating and counting packets with Click on the system under test. For the two inherently bidirectional VNFs, we generate and measure traffic on the load generator.

Fig. 12 shows the results. Full-sized packets saturate the 10G links in most cases, but smaller packets reveal per-packet overhead differences. Linux is 1.6×-3.0× slower than MorphOS for receive and transmit. We observe the biggest speedup for the bidirectional mirror VNF. Even though we limit the offered load to 1.3Mpps, the Linux kernel stack steals the application's CPU time to drop packets due to the lack of cooperative scheduling. MorphOS use of eBPF incurs at most 10% overhead (NAT) compared to native Unikraft. The JIT compiler minimizes overhead and, in some cases, improves performance. For the IDS VNF, compiler optimizations enable MorphOS to outperform Unikraft by 18%, as unmodified Click algorithms lack awareness of runtime configuration during compile time. While our 10G link does impose a limitation in some tests with big packets, we expect our comparison to unmodified Unikraft to remain representative: Our results for different packet sizes support, that MorphOS does not add overhead that depends on packet size, such as packet copies.

Latency. We evaluate latency by measuring the round-trip delay of packets with the bi-directional VNFs. We use MoonGen [30] to sample latency at 1000Hz under 100kpps of load and present the cumulative distribution function (CDF) of the latency histogram. Fig. 13 shows two tight groupings of distributions: the slower Linux VNFs, and the faster MorphOS and Unikraft ones. The distributions within each grouping match closely because the latencies are dominated by the OS and network stack instead of the VNF logic. Overall, Unikraft and MorphOS reduce median latencies by 28% over Linux.

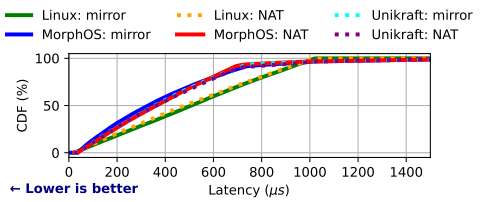


Fig. 13. End-to-end latency of bi-directional VNFs.

Large firewalls. Finally, we evaluate the effectiveness of MorphOS's JIT compilation by comparing large firewalls with native Click. The MorphOS firewall is defined in eBPF code and JIT compiled, whereas the native one is configured via Access Control List (ACL) parameters. We add port filter rules that alternate between dropping and passing packets and send traffic to these ports in a round-robin fashion. Fig. 14 shows that while Click's native firewall performance decreases with more rules, MorphOS consistently performs best. The native Click throughput approaches zero beyond 10k rules, whereas MorphOS with JIT-compiled eBPF maintains 1.2Mpps. This is due to eBPF compilers optimizing firewall logic into match-case constructs with constant lookup time, avoiding iteration over large rule tables. Running the firewall with an eBPF interpreter confirms the scalability of eBPF, albeit with lower base performance.

RQ3 takeaway: For throughput, Click VNFs with MorphOS are 1.6×-3.0× faster than with Linux and on par with Unikraft. While eBPF can have up to 10% overhead, our IDS and firewall studies show that JIT-compiled eBPF can also improve performance over highly parametrized traditional VNFs.

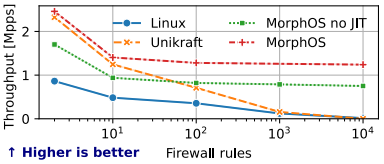


Fig. 14. Receive throughput depending on the number of installed firewall rules.

9 Related work

OS designs for network-intensive applications. Several operating systems present optimizations to accelerate networked applications, e.g., through more efficient intra-host communication [63, 67], data processing [12, 32, 47], and scheduling [49, 92]. Specifically, ClickOS [67] focuses on optimizing the operating system's and Xen's [10] network stacks for Click, while MorphOS primarily addresses the runtime inflexibility of unikernels in supporting dynamic, stateful, network-intensive applications. Notably, MorphOS uses Click solely as a representative application, and its reconfiguration mechanism is fundamentally decoupled from the unikernel application itself. MOS [47] is a library OS optimized for data-plane performance that provides the high-level infrastructure necessary to develop high-level middleboxes. Shinjuku [49] improves tail latencies by fine-grained preemption of long-lasting requests. MorphOS is orthogonal to this work because its APIs facilitate extending the unikernel with similar optimizations. However, MorphOS adds the goal of application reconfigurability and optimizes the network stack to safely support dynamic eBPF quickly.

Unikernels. Unikernels, while appealing, face practical challenges in production systems due to limited reconfigurability, unlike traditional servers with tools like AWS Systems Manager or GCP Guest Agent [17, 40, 87, 96, 99, 104, 105]. Existing efforts like uIO [71] extend unikernel functionality using eBPF interpretation [68], though primarily for auxiliary tasks like inspection and debugging rather than core application logic. While uIO enables additional task execution, invoking these functionalities from the application side requires further integration. VampOS [112] enables individually restarting kernel components. MorphOS distinguishes itself by leveraging JIT-compiled eBPF for flexible, high-performance packet processing. Unlike uIO, which uses MPK to help programmers voluntarily limit programming error impact, MorphOS transparently applies MPK to harden eBPF program isolation between inserted programs and the core application – a critical requirement for multi-tenant networked applications.

eBPF for high-performance networking. eBPF is widely used for high-performance networking in Linux environments, as seen with Cilium's cloud-native solutions [21] and Chaining-Box's disaggregated service chaining [18]. Janus [36] further demonstrates its use in real-time systems and SPRIGHT [93] for high-performance serverless functions. Similar to these efforts, MorphOS utilizes eBPF for fast packet processing, but within the context of networked unikernel applications.

Following its success on Linux, eBPF has been adopted in non-standard Linux environments, notably Windows [70], which combines standalone eBPF interpreters (uBPF [6]) and verifiers (Prevail [39]). Additionally, Craun et al. [25] propose decoupled eBPF verification for Linux-based embedded systems. MorphOS also integrates eBPF [6] and Prevail [39] for its eBPF ecosystem. However, MorphOS distinguishes itself by presenting a concrete eBPF verification integration specifically for a unikernel-based system.

Safe execution of networked applications. Several prior studies investigate techniques for the safe execution of networked applications, including the use of memory-safe programming languages [22, 64, 83], compilers [14, 34, 100], and OS-level sandboxing mechanisms [38, 79, 106, 118]. SURE [85] isolates a secure portion of a unikernel from the user application with MPK for serverless computing. ShieldBox [107] and Trusted Click [24] protect the integrity of the packet processing elements themselves by running Click elements in secure SGX [23, 58] enclaves. Compared to these studies, MorphOS leverages the eBPF runtime and its verifier to ensure safe execution.

Other studies explore the verification of the functional correctness of networked applications [91, 115]. Vigor [115] expands automatic verification to VNFs written in C. TinyNF [91] formally verifies the correctness of the VNF's underlying driver. These works are orthogonal to our work.

Driven by the several vulnerabilities found in Linux's eBPF verifier (e.g., [73–75]), several recent works propose sandboxing JIT-compiled eBPF code to add additional isolation guarantees by utilizing MPK on x86-64 (Moat [62]), or ARM PAC (Hive [117]) and ARM MTE (SafeBPF [61]), or employing SFI (KFlex [28]) for Linux eBPF. Our work also uses MPK hardening, but closes design gaps specific to data-intensive VNFs by integrating it with the MorphOS *NetStack*. Additionally, we fundamentally design MorphOS to scale to many program instances by combining eBPF with probabilistic MPK hardening, which contrasts with the poor scalability of eBPF-free MPK networking, such as Pegasus [86].

10 Conclusion

In conclusion, MorphOS proposes a new OS design that enables runtime reconfiguration and extensibility for applications such as Virtual Network Functions (VNFs) while maintaining the advantages of unikernels. By leveraging eBPF for dynamically updatable code execution, out-of-band verification to enforce correctness, and hardware-assisted safety isolation, MorphOS ensures that applications remain dependable across reconfigurations. The evaluation results highlight the ability to reduce reconfiguration time, amortize verification cost, and output Linux in terms of performance and lightweightness. MorphOS establishes a foundation for live-reconfigurable application logic while maintaining the lightweight characteristics of unikernels, paving the way for more adaptive and efficient networked operating systems.

Artifact and supplementary material. MorphOS is available at https://github.com/TUM-DSE/MorphOS. The appendix covers additional details for the eBPF vulnerabilities CVEs and associated eBPF verification in MorphOS.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful comments. This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), and Google Research Grants. The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6), 1975.

[2] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming Revision 3.40 – Section 5.6.7. https://docs.amd.com/v/u/en-US/24593_3.42, 2023. Accessed: 2025-05-28.

[3] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2012.

[4] DPDK authors. Overview of networking drivers. https://doc.dpdk.org/guides/nics/overview.html, 2025. Accessed: 2025-05-28.

[5] DPDK authors. Dpdk: Accelerating network performance. https://www.dpdk.org/, [n.d.]. Accessed: 2025-05-28.

[6] UBPF authors. Iovisor/ubpf: Userspace eBPF VM. https://github.com/iovisor/ubpf, [n. d.]. Accessed: 2025-05-28.

[7] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine*, 56(12), 2018.

[8] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. Combined Stateful Classification and Session Splicing for High-Speed NFV Service Chaining. *IEEE/ACM Transactions on Networking*, 29(6), 2021.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, 2003.

[11] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M. Runge, Florian Wohlfart, Bernd E. Wolfinger, and Georg Carle. A study of networking software induced latency. In *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX, 2014.

[13] 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, 2024.

[14] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. SafePM: a sanitizer for persistent memory. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022.

[15] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016.

[16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX, 2023.

[17] Bryan Cantrill. Unikernels are Unfit for Production. https://www.tritondatacenter.com/blog/unikernels-are-unfit-for-production, 2016. Accessed: 2025-05-28.

[18] Matheus S. Castanho, Cristina K. Dominicini, Magnos Martinello, and Marcos A. M. Vieira. Chaining-Box: A Transparent Service Function Chaining Architecture Leveraging BPF. *IEEE Transactions on Network and Service Management*, 19(1), 2022.

[19] Yang Chen and Xinfeng Shu. Formal verification of eBPF program security based on PTL. In *Proceedings of the 2023 6th International Conference on Artificial Intelligence and Pattern Recognition*. ACM, 2024.

[20] Diptanu Gon Choudhury. XDP-Programmable Data Path in the Linux Kernel. *login Usenix Mag.*, 43(1), 2018.

[21] Cilium. Cilium – eBPF-based Networking, Observability, Security. https://cilium.io/, [n. d.]. Accessed: 2025-05-28.

[22] Snabb Contributors. Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb, 2024. Accessed: 2025-05-28.

[23] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086, 2016. Accessed: 2025-05-28.

[24] Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted Click: Overcoming Security issues of NFV in the Cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2017.

[25] Milo Craun, Adam Oswald, and Dan Williams. Enabling eBPF on Embedded Systems Through Decoupled Verification. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. ACM, 2023.

[26] Willem de Bruijn. Zero-copy networking. *LWN.net*, 726917, 2017.

[27] Gonzalo De La Torre Parra, Paul Rad, and Kim-Kwang Raymond Choo. Implementation of deep packet inspection in smart grids and industrial Internet of Things: Challenges and opportunities. *Journal of Network and Computer Applications*, 135, 2019.
[28] 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*. ACM, 2024.
[29] Kjeld Borch Egevang and Pyda Srisuresh. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, 2001. Accessed: 2025-05-28.
[30] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015.
[31] Paul Emmerich, Daniel Raumer, Alexander Beifuß, Lukas Erlacher, Florian Wohlfart, Torsten M. Runge, Sebastian Gallenmüller, and Georg Carle. Optimizing latency and CPU load in packet processing systems. In *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. IEEE, 2015.
[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, 1995.
[33] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2021.
[34] Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia. Cage: Hardware-Accelerated Safe WebAssembly. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, 2025.
[35] Xenofon Foukas, Navid Nikaein, Mohamed M. Kassem, Mahesh K. Marina, and Kimon Kontovasilis. FlexRAN: A Flexible and Programmable Platform for Software-Defined Radio Access Networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. ACM, 2016.
[36] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5G RAN Analytics and Control to a New Level. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. ACM, 2023.
[37] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Exploiting Sparsity in Difference-Bound Matrices. In Xavier Rival, editor, *Static Analysis*. Springer Berlin Heidelberg, 2016.
[38] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. ACM, 2014.
[39] 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*. ACM, 2019.
[40] Google. Guest environment | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/images/guest-environment, [n. d.]. Accessed: 2025-05-28.
[41] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*. Springer International Publishing, 2015.
[42] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network Function Virtualization: Challenges and Opportunities for Innovations. *IEEE Communications Magazine*, 53(2), 2015.
[43] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkley, 2015. https://courses.grainger.illinois.edu/ece598hpn/fa2022/papers/softnic.pdf.
[44] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. Demystifying the Performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019.
[45] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management*, 12(1), 2015.
[46] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Order Number: 253668-078US – Chapter 4.6.2 Protection Keys. https://cdrdv2.intel.com/v1/dl/getContent/671190, 2022. Accessed: 2025-05-28.
[47] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX, 2017.
[48] Yunhong Jiang and Wei Wang. Towards Low Latency Interrupt Mode DPDK. In *Proceedings of the DPDK Summit 2017*, 2017.
[49] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for usecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX, 2019.

[50] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX, 2018.
[51] Linux kernel developers. BPF Instruction Set Architecture (ISA). https://docs.kernel.org/bpf/standardization/instruction-set.html, [n. d.]. Accessed: 2025-05-28.
[52] Linux kernel developers. eBPF verifier. https://docs.kernel.org/bpf/verifier.html, [n. d.]. Accessed: 2025-05-28.
[53] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX, 2014.
[54] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3), 2000.
[55] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the 16th European Conference on Computer Systems*. ACM, 2021.
[56] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ACM, 2006.
[57] Tytus Kurek. Unikernel Network Functions: A Journey Beyond the Containers. *IEEE Communications Magazine*, 57(12), 2019.
[58] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017.
[59] Viktor Leis and Christian Dietrich. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.*, 17(8), 2024.
[60] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX, 2023.
[61] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In *Proceedings of the 2024 on Cloud Computing Security Workshop*. ACM, 2024.
[62] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. MOAT: Towards Safe BPF Kernel Extension. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX, 2024.
[63] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2015.
[64] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems*. ACM, 2013.
[65] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.
[66] Tina Marjanov and Alice Hutchings. SoK: Digging into the Digital Underworld of Stolen Data Markets . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025.
[67] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2014.
[68] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX, 1993.
[69] Microsoft. Introduction to receive side scaling (rss). https://learn.microsoft.com/en-gb/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2022. Accessed: 2025-05-28.
[70] Microsoft. eBPF for Windows. https://microsoft.github.io/ebpf-for-windows/, [n. d.]. Accessed: 2025-05-28.
[71] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. uIO: Lightweight and Extensible Unikernels. In *Proceedings of the 15th ACM Symposium on Cloud Computing*. ACM, 2024.
[72] MITRE. Common vulnerabilities and disclosures. https://cve.mitre.org/, 2025. Accessed: 2025-05-28.
[73] MITRE. CVE-2020-27194. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27194, [n. d.]. Accessed: 2025-05-28.

[74] MITRE. CVE-2020-8835. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835, [n. d.]. Accessed: 2025-05-28.
[75] MITRE. CVE-2021-31440. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31440, [n. d.]. Accessed: 2025-05-28.
[76] MITRE. CVE-2024-42072. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-42072, [n. d.]. Accessed: 2025-05-28.
[77] MITRE. CVE-2024-43837. https://www.cve.org/CVERecord?id=CVE-2024-43837, [n. d.]. Accessed: 2025-05-28.
[78] MITRE. CVE-2024-45020. https://www.cve.org/CVERecord?id=CVE-2024-45020, [n. d.]. Accessed: 2025-05-28.
[79] Priyanka Naik, Akash Kanase, Trishal Patel, and Mythili Vutukuru. libVNF: Building Virtual Network Functions Made Easy. In *Proceedings of the 2018 ACM Symposium on Cloud Computing*. ACM, 2018.
[80] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI 96)*. USENIX, 1996.
[81] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), 2018.
[82] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015.
[83] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX, 2016.
[84] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX, 2019.
[85] Federico Parola, Shixiong Qi, Anvaya B. Narappa, K. K. Ramakrishnan, and Fulvio Risso. SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. ACM, 2024.
[86] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication. In *Proceedings of the Twentieth European Conference on Computer Systems*. ACM, 2025.
[87] Per Buer. Unikernels Aren't Dead, They're Just Not Containers. https://www.infoq.com/presentations/unikernels-includeos/, 2019. Accessed: 2025-05-28.
[88] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. Automatic Parallelization of Software Network Functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX, 2024.
[89] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4), 2015.
[90] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martìn Casado. The Design and Implementation of Open vSwitch. *login Usenix Mag.*, 40(2), 2015.
[91] Solal Pirelli and George Candea. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX, 2020.
[92] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.
[93] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. ACM, 2022.
[94] Tiago Rosado and Jorge Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*. ACM, 2014.
[95] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *5th Annual Linux Showcase & Conference (ALS 01)*. USENIX, 2001.
[96] Jerod Santo. The Big Idea Around Unikernels. https://changelog.com/posts/the-big-idea-around-unikernels, 2021. Accessed: 2025-05-28.
[97] Florian Schmidt, Stefan Jumarea, and Felipe Huici. Click for Unikraft. https://github.com/unikraft/lib-click, 2019. Accessed: 2025-05-28.
[98] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, 2012.
[99] Amazon Web Service. AWS Systems Manager Documentation. https://docs.aws.amazon.com/systems-manager/index.html, [n. d.]. Accessed: 2025-05-28.
[100] Dimitrios Stavrakakis, Alexandrina Panfil, MJin Nam, and Pramod Bhatotia. SPP: Safe Persistent Pointers for Memory Safety. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024.

[101] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[102] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. ACM, 2024.

[103] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.

[104] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. VMSH: Hypervisor-Agnostic Guest Overlays for VMs. In *Proceedings of the 17th European Conference on Computer Systems*. ACM, 2022.

[105] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX, 2018.

[106] Kashyap Thimmaraju, Saad Hermak, Gabor Retvari, and Stefan Schmid. MTS: Bringing Multi-Tenancy to Virtual Networking. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, 2019.

[107] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research*. ACM, 2018.

[108] Daniel Turull, Peter Sjödin, and Robert Olsson. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 82, 2016.

[109] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2021.

[110] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022.

[111] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*. Springer Nature Switzerland, 2023.

[112] Takeru Wada and Hiroshi Yamada. Reboot-based Recovery of Unikernels at the Component Level. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2024.

[113] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022.

[114] Jianing You, Kang Chen, Laiping Zhao, Yiming Li, Yichi Chen, Yuxuan Du, Yanjie Wang, Luhang Wen, Keyang Hu, and Keqiu Li. AlloyStack: A Library Operating System for Serverless Workflow Applications. In *Proceedings of the Twentieth European Conference on Computer Systems*. ACM, 2025.

[115] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying software network functions with no verification expertise. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019.

[116] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 2021.

[117] Peihua Zhang, Chenggang Wu, Xiangyu Meng, Yinqian Zhang, Mingfan Peng, Shiyang Zhang, Bing Hu, Mengyao Xie, Yuanming Lai, Yan Kang, and Zhe Wang. HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX, 2024.

[118] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. ACM, 2016.

[119] Zongpu Zhang, Jiangtao Chen, Banghao Ying, Yahui Cao, Lingyu Liu, Jian Li, Xin Zeng, Junyuan Wang, Weigang Li, and Haibing Guan. HD-IOV: SW-HW Co-designed I/O Virtualization with Scalability and Flexibility for Hyper-Density Cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*. ACM, 2024.

[120] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX, 2020.

Appendix

A eBPF CVE List

We provide a list of Common Vulnerabilities and Exposures (CVE) numbers for each of the vulnerability categories discussed in this paper in Table 7.

Vulnerability/Component: CVEs
Verifier: register value tracking: CVE-2020-27171, CVE-2020-27194, CVE-2020-8835, CVE-2021-31440, CVE-2021-33200, CVE-2021-3444, CVE-2021-3490, CVE-2021-45402, CVE-2022-23222, CVE-2022-2785, CVE-2024-41003, CVE-2024-43910
Helper: CVE-2021-34866, CVE-2021-4001, CVE-2021-4204, CVE-2024-26885, CVE-2024-36937, CVE-2024-42063, CVE-2024-49861, CVE-2024-50164
Verifier: branch pruning: CVE-2021-29155, CVE-2021-33624, CVE-2023-2163, CVE-2023-52920, CVE-2024-42072, CVE-2024-43838
Verifier: context value tracking: CVE-2024-26611, CVE-2024-38566, CVE-2024-42151, CVE-2024-50063
Verifier: crash: CVE-2024-43837, CVE-2024-45020

Table 7. Related work discusses these 32 eBPF vulnerabilities [61, 62, 66, 117].

B eBPF Verification

eBPF verifiers statically verify that eBPF programs are safe by checking that they are self-contained in their designated memory regions. We give an intuition on how eBPF memory safety is upheld through careful runtime and verifier co-design. Afterward, we explain how the design of Prevail [39] ensures effective verification.

eBPF memory safety. The verifier ensures memory safety: eBPF must only access its allocated memory and not read uninitialized memory to prevent modifying or leaking internal information. Therefore, eBPF is restricted to its stack and static allocations defined in the programs `.data` section. Helper functions that safely store data in eBPF maps are the only way to use heap memory because eBPF does not offer instructions or calls to allocate heap memory. Helper functions copy input arguments into their own memory, while the verifier enforces their return values to remain read-only by the eBPF program.

The only time the verifier allows dereferencing a point to external memory is to access the input packet buffer. MorphOS's eBPF functions use a fixed function signature that the verifier understands. It contains a pointer to the start and end of the packet buffer. The verifier ensures that the eBPF code sufficiently checks buffer bounds before accessing its memory.

Statically generating proof. Prevail [39] statically analyzes eBPF programs to verify their memory safety by constructing control-flow graphs to be analyzed using the Crab [41] abstract interpreter.

Prevail knows what proof it needs to generate to verify memory safety because it not only understands eBPF bytecode but is also aware of the surrounding ecosystem: Prevail knows the semantic meaning of the *context* parameters passed as input to the eBPF function, as well as the available helper functions and their memory access semantics.

The key to verification is to prove that all memory accesses a program may do are valid. Prevail tags memory values with invariants and data types to differentiate between numerical values and pointers. Whenever a pointer is modified using numerical values, a proof must be produced that the numerical value is within a range that does not result in out-of-bounds memory accesses. Prevail considers the bounds of three types of memory: First, Prevail tracks the state of individual bytes on the eBPF program's *stack*, which is limited to 512 bytes. Second, the eBPF *context* points to the start and end of a packet buffer for which prevail only tracks accesses as generic values confined by the packet bounds. Third, *shared regions* represent, e.g., eBPF maps that are confined by size and may change their contents at any time because they can be shared with other programs.

Using this memory abstraction, Prevail models the stack of all possible call graphs. It tracks relationships between variables using Zone abstract domains [37]. By tracking all stack cells, Prevail precisely resolves addresses for bounded memory regions like the stack while abstracting packet accesses. Prevail avoids path explosion by merging states across loop iterations with widening or joining operators, optimizing the Zone domain for efficiency.

Prevail's approach to verification supports eBPF programs that contain loops, packet contexts, shared maps, and scales to large programs by avoiding path explosion.

C Probabilistic eBPF isolation

The approach described in § 5.5 to isolating eBPF with MPK is limited by the assumption that there are fewer than 15 packet buffers and eBPF programs so that the MPK isolation works with only 16 PKeys available in hardware.

To cope with this limited number of PKeys, we propose probabilistic isolation with MPK. MorphOS only statically assigns PKey K_0 to the unikernel but accepts permission overlaps by assigning the remaining K_{1-15} pseudo-randomly to eBPF programs and packet buffers. With many in-flight packets, a single key gets inevitably assigned to multiple packet buffer pages as well as K_{eBPF}, causing permission collisions. Consequently, probabilistic MorphOS hardening does not detect all illegal memory accesses.

However, the blast radius of an eBPF program evading MPK isolation by chance, e.g., by randomly guessing the address of a page that shares the K_{1-15} key of its input packet buffer with probability r, is limited to only these pages and only that invocation. Subsequent invocations likely use a different key for the input buffer and hence require a new guess to find another buffer with the same key, reducing the chances of a successful exploit that takes i invocations to $(\frac{1}{15} * r)^i$.

Limitations of MPK hardening. While MorphOS hardens helper functions for eBPF maps by allocating its data with K_{eBPF}, other helpers may pass their inputs to kernel functions, making bugs in unikernel implementations exploitable. In addition, our approach imposes memory overhead as only one packet buffer can be allocated per page because that is the granularity of MPK permissions. For example, we find that Click allocates statically sized buffers of 1.5k bytes regardless of packet size, resulting in a memory overhead of 2.6×. An alternative design can be considered, where packet buffers are copied between the protected and unprotected pages. For large packets, updating the PKey and consequently flushing the TLB can be faster than copying [85].

Received June 2025; accepted September 2025