SoK: Challenges and Paths Toward Memory Safety for eBPF

Kaiming Huang*, Mathias Payer† , Zhiyun Qian‡, Jack Sampson*, Gang Tan*, Trent Jaeger‡

* Penn State † EPFL ‡ UC Riverside

eBPF enables unprivileged code in the kernel

- Networking

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff, *Orange Labs, Sorbonne Université, Inria, LIP6;* Julien Sopena, *Sorbonne Université, LIP6;* Kahina Lazri, *Orange Labs;* Antoine Blin, *Gandi;* Gilles Muller, *Inria*

ogram Warping

nella[1,3], Giuseppe

arvard University

State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing

Taking 5G RAN Analytics and Control to a New Level

Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, Zhihua Lai
Microsoft
Cambridge, United Kingdom

uan Zhang△,

ndon, UK

kz_compiler@email.rutgers.edu

- Optimization

λ-IO: A Unified IO Stack for Computational Storage

MERLIN: Multi-tier Optimization of eBPF Code for ata Planes

XRP: In-Kernel Storage Functions with eBPF

SPRIGHT: Extracting the Server from Serverless Computing!

Extension Framework for File Systems in User space

Ashish Bijlani, Georgia Institute of Technology
Umakishore Ramachandran, Georgia Institute of Technology

Carnegie Mellon University Carnegie Mellon University Carnegie Mellon University

- Security

HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64

Di Jin, Vaggel

Falco
powered by eBPF
CLOUD NATIVE COMPUTING FOUNDATION
GRADUATED PROJECT

An Analys
Vu

Xijia Che
niversity and BNRist

Li

Ofek Kirzner and Adam Morrison, *Tel Aviv University*

George Mason University *Tsinghua University*

- Tracing

Network-Centric Distributed Tracing with DeepFlow:

The Design and Implementation of Hyperupcalls

Eliminating eBPF Tracing Overhead
d Processes

bpftrace

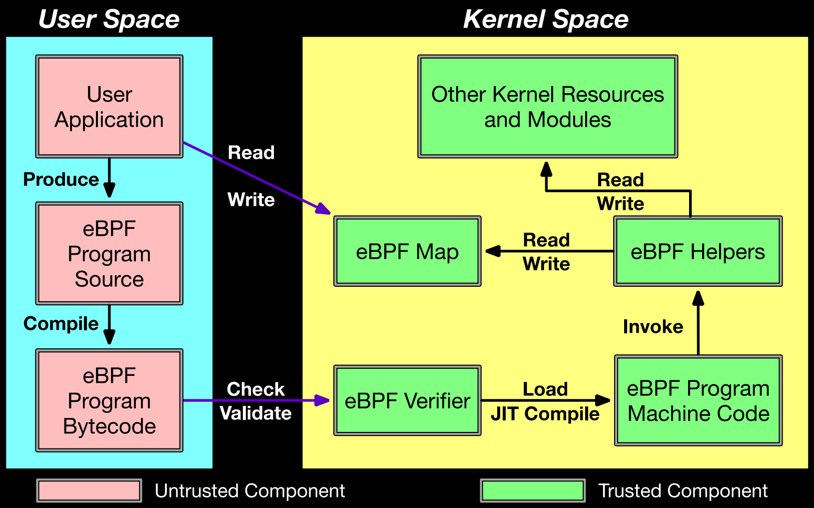
Dynamic Tracing for Linux

bpftrace is a high-level tracing language for Linux and provides a quick and easy way for people to write observability-based eBPF programs, especially those unfamiliar with the complexities of eBPF.

Uddhav Gautam
Virginia Tech
Blacksburg, VA, USA
upgautam@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

eBPF Workflow and Its Trust Model



But are they really safe?

- eBPF programs **must not perform unsafe memory accesses**.
- eBPF helper functions are trusted **but not validated** kernel APIs.
- The verifier must ensure that accesses to the kernel data **do not populate memory errors**.
- The verifier **must be free of implementation bugs**, as any bug can be exploited to load unsafe programs.
- The eBPF trust model relies **critically on the eBPF verifier** to enforce memory safety.

Memory Safety Issue in eBPF Verifier

- eBPF verifier has been becoming **a significant source of bugs**.
  - **46 CVEs** in eBPF verifier in 2024.
  - **325 Syzbot-reported bugs** related to eBPF submodule in Linux Kernel.
- Checks are **unsound** and **incomplete**.
  - Bugs left unchecked amid removal of safety checks by optimizations.
  - Checks are incomplete for ensuring full memory safety.
- Checks are **limited in scope** in terms of complete workflow.
  - Checks of the verifier are limited to the eBPF bytecode.

Memory Safety Issue in eBPF Verifier

- Checks are **limited in scope** in terms of complete workflow.
  - Checks of the verifier are limited to the eBPF bytecode.

```
static void *__dev_map_lookup_elem(
            struct bpf_map *map, u32 key){
             struct bpf_dtab *dtab = container_of(map, struct
                                                  bpf_dtab, map);
             struct bpf_dtab_netdev *obj;
             if (key >= map->max_entries)
                          return NULL;
             obj = rcu_dereference_check(dtab->netdev_map[key],
                                         rcu_read_lock_bh_held());
             return obj;
}
```

No checks in `map_lookup_elem` to ensure the initialization of `obj`.

```
SEC("classifier")
int example_prog(struct __sk_buff *skb) {
   int index = 0; // Key for accessing dev_map
   int *dev_ifindex;
   // Use dev_map_lookup_elem to retrieve the interface
   dev_ifindex = dev_map_lookup_elem(&dev_map, &index);
   if (!dev_ifindex) {
      return TC_ACT_SHOT; // Drop packet if fails
   }
   // Uninitialized memory access
   *dev_ifindex += 1; // KMSAN uninit warning
   // Final decision to accept or drop the packet
   return TC_ACT_OK;
}
```

Attacker can easily forge a malicious eBPF program to exploit UBI.

Kernel Defenses

Category	Kernel Defensive Features	Description
Required Defense	eBPF Verifier	Validates security of eBPF programs.
Optional Defense	Capability CAP_BPF	Permits only privileged users to attach eBPF programs.
	BPF LSM (Linux Security Modules)	Enforces access control over eBPF programs.
	BPF Type Format (BTF) and CO-RE	Validates data type and version compatibility.
General Defense	CFI and Execute-Only Memory (XOM)	Prevents control flow hijacking and code reuse attacks.
	Memory Tagging	Prevents pointers from being tampered and forged.
	Shadow Stacks	Protects return addresses.
	kASAN	Detects memory errors at runtime.
	kASLR	Randomizes memory layout.
	SMAP and SMEP	Prevents unauthorized user-space memory access in kernel mode.

- eBPF-specific defenses are limited by **optional settings** and left room for attacks with limited privilege.
- General defenses **fail to fully block** eBPF-based attacks.

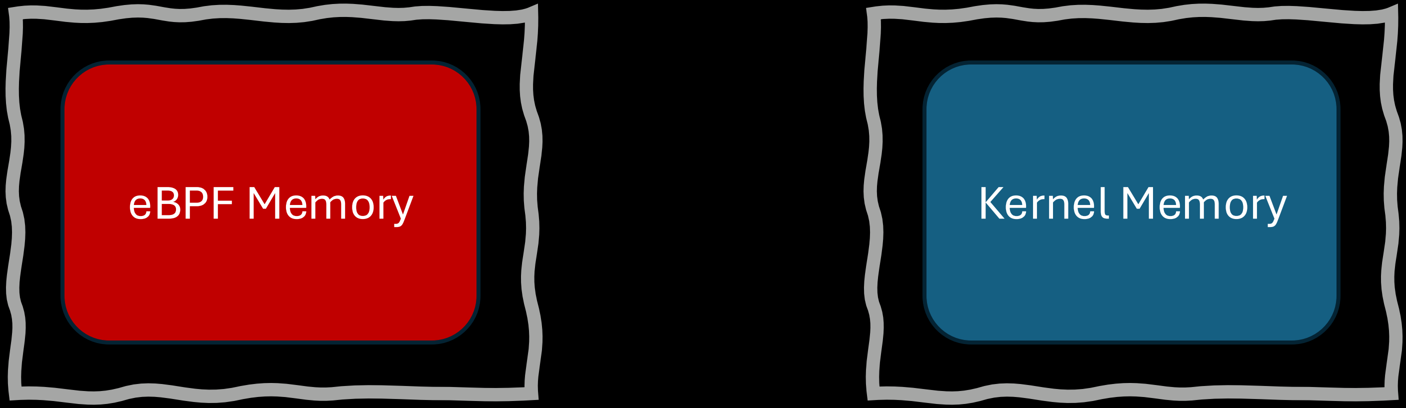
Take Capability CAP_BPF as an Example

- Introduced in Linux 5.8 (Aug 2020).
  - Designed to **restrict unprivileged users** from attaching **eBPF** programs.
- CAP_BPF is not a hard restriction.
  - Users can **opt out** and still attach eBPF programs.
- Privileged enforcement **reduces flexibility**.
  - Vendors such as Cilium rely on **unprivileged eBPF**.
- CAP_BPF illustrates the **tension between security and usability**.
  - Unprivileged eBPF execution **remains common** in practice.

Research Directions

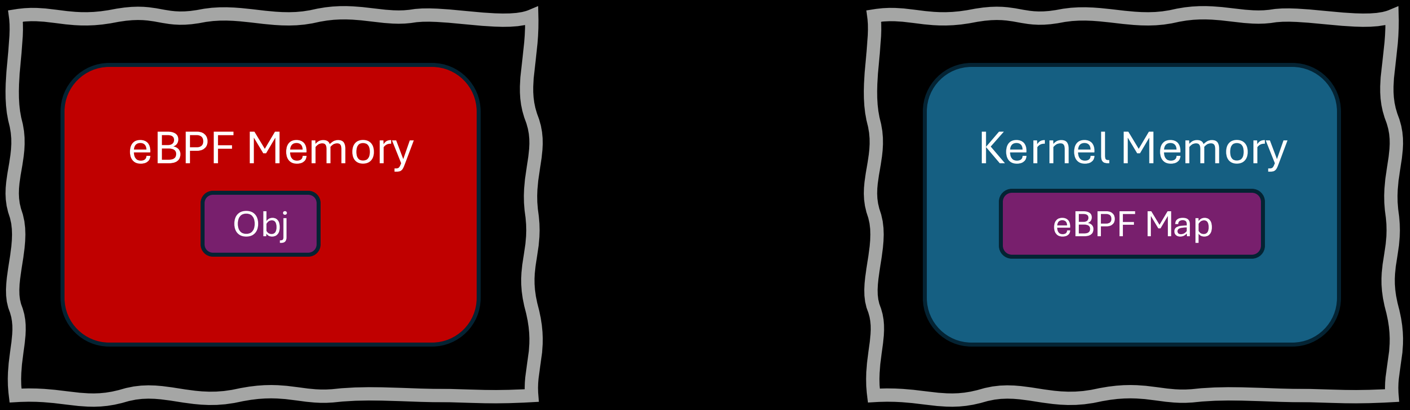
- **Fuzzing**
  - Inherently **incomplete**.
  - **Hard** to generate eBPF program that both **pass verifier** and **trigger bugs**.
- **Isolation**
  - Relies on specific hardware features or support.
  - Does not address risks from **indirect kernel access**.
- **Runtime Checks**
  - Limited by the **resource constraints** and **instruction limits**.
- **Static Validation**
  - Existing approaches are either **unsound** or **incomplete**.

Limitation – Isolation as Example



- Isolation **separates** eBPF memory and Kernel memory.
- Unauthorized memory accesses are prevented at isolation boundary.

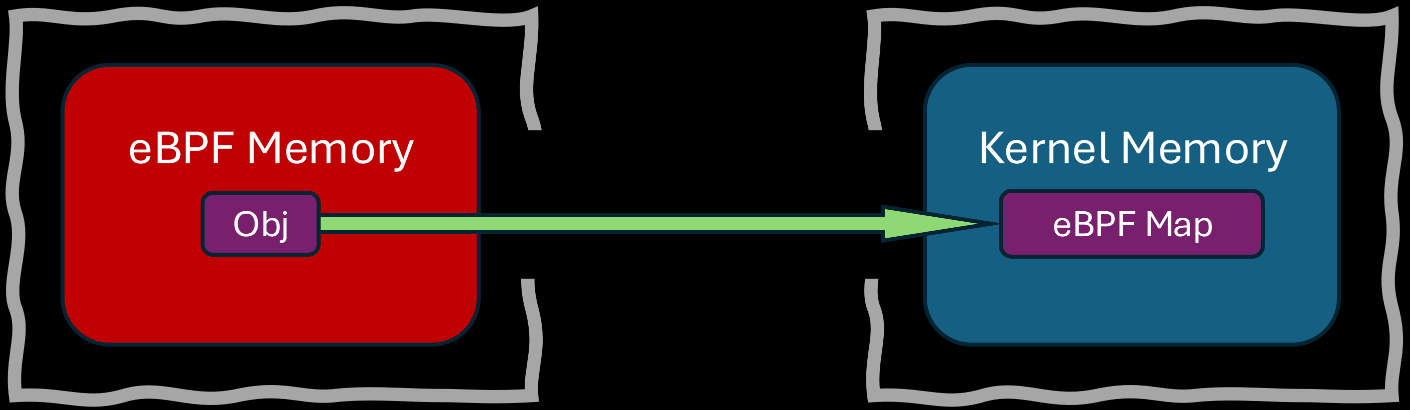
Limitation – Isolation as Example



- However, objects accessed in eBPF programs (e.g., pointers to valid kernel memory) **can be stored in kernel** (e.g., eBPF map) via eBPF helpers.

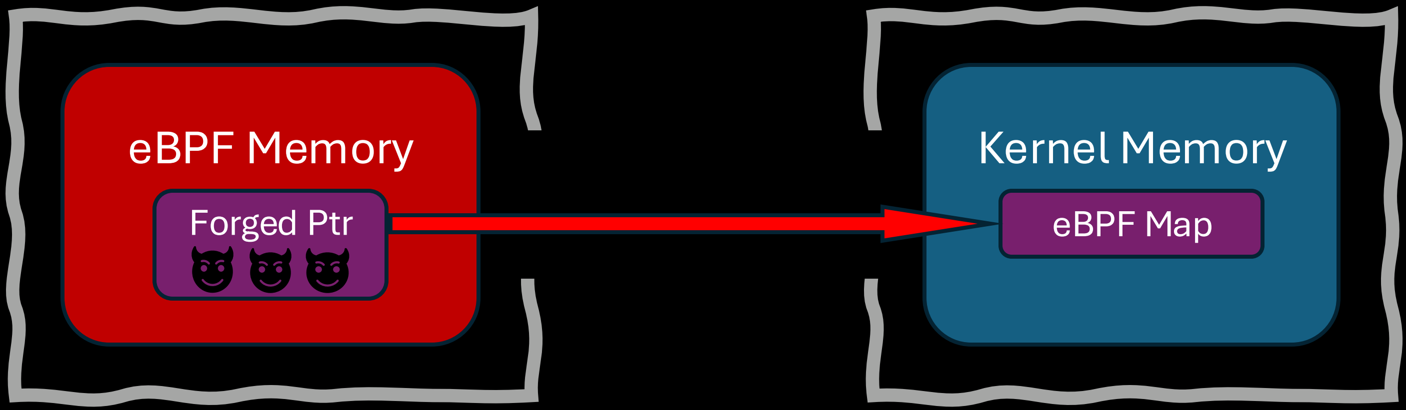
```
struct val_t val = { ... };  // Defined on eBPF stack
bpf_map_update_elem(&my_map, &key, &val, BPF_ANY); // store object to kernel data

void *ptr = bpf_get_current_task();  // get pointer to current task_struct
bpf_map_update_elem(&my_map, &key, &ptr, BPF_ANY); //store pointer to kernel data
```

Limitation – Isolation as Example



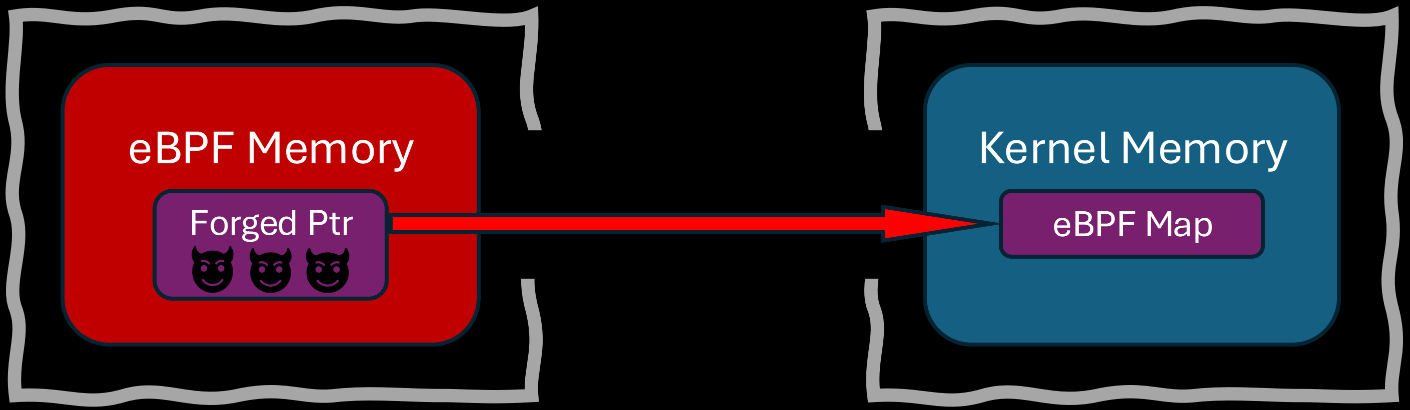
- However, objects accessed in eBPF programs (e.g., pointers to valid kernel memory) **can be stored in kernel** (e.g., eBPF map) via eBPF helpers.
- The accesses to kernel data through such pointers , or through pointers in such objects, **need to be preserved** for functionality.

Limitation – Isolation as Example



- Such data sharing **allows attacker** to
  - exploit the memory errors in eBPF program;
  - forge a pointer arbitrarily;
  - Pass the pointer to the kernel (through helpers) for unauthorized accesses.
  - Known as **Cross-boundary Interface Vulnerabilities** (CIVs).

Limitation – Isolation as Example



- Attacker can use the forged pointer to **escalate exploitability**.
  - Examined by EPF and Interp-flow Hijacking attacks.

- *Linux eBPF - new privilege escalation techniques* – Pentera Labs.

Protection Scope of Existing Defenses

		eBPF-Only			Shared Objs		
		Spatial	Type	Temp	Spatial	Type	Temp
eBPF Verifier [30]	V	◐	◔	◔	◐	◔	○
HyperBee [47]	V	◐	◔	○	○	○	○
KFuse [49]	V	◐	◔	○	◐	◔	○
PREVAIL [112]	V	◐	◔	◔	◐	◔	○
SandBPF [36]	II	◕	○	○	◕	○	○
SafeBPF [37]	II	◕	○	○	◕	○	○
HIVE [39]	II	◕	○	○	◕	○	○
MOAT [38]	II	◕	○	○	◕	○	○
Prevail2Radius [107]	T	◔	◔	○	◔	◔	○
Seccomp-eBPF [131]	T	◔	○	○	◔	○	○
TnumArith [43]	T	◔	○	○	○	○	○
RangeAnalysis [44]	T	◔	○	○	○	○	○

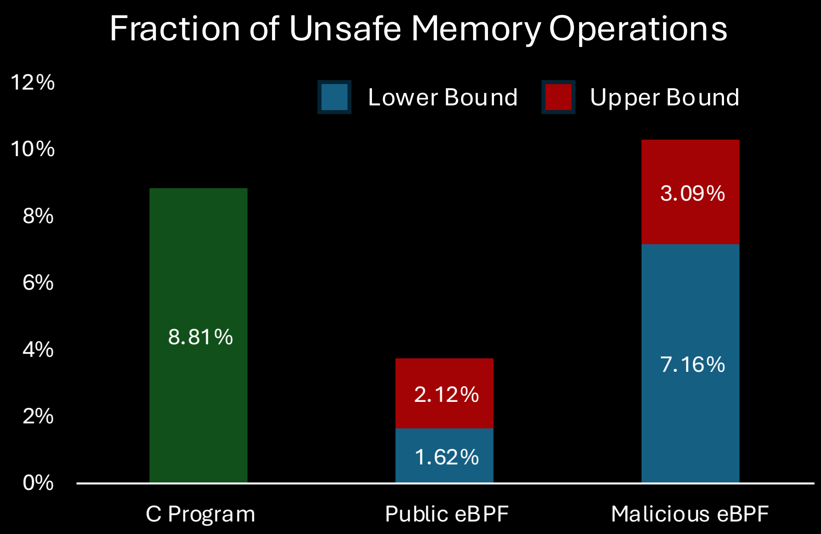
None of the defenses, whether currently deployed or proposed in research, fully or soundly cover any category of unsafe ops.

Identify Unsafe Memory Operation in eBPF

- **How far are we from memory safe eBPF?**
- Achieving memory safety demands protecting unsafe operations.
- **Hypothesis:** eBPF should be close to memory safe in terms of low fraction of unsafe operations, but how to identify them?
- Approach – DataGuard (NDSS 2022) and Uriah (CCS 2024)
- https://github.com/Lightninghkm/Unified-Memory-Safety-Validation
- Dataset
  - Public eBPF programs – Linux Kernel and BCC.
  - Malicious eBPF programs – CVE PoCs and Syzbot reproducers.
  - General C programs – evaluated by DataGuard and Uriah.

Fraction of Unsafe Memory Operations

- **General C Program**
  - Fraction **similar to Malicious eBPF programs** but **far more in number** of unsafe ops.
- **Malicious – higher fractions of unsafe ops**
  - **7.16% (lower bound)** to **10.25% (upper bound).**
  - Despite being crafted to exploit bugs, the **verifier still limits unsafe memory accesses**.
- **Public - significantly lower** fractions
  - **1.62% (lower bound)** to **3.74% (upper bound)**
  - This gap is due to **missing kernel-specific constraint information** in static analysis.
  - Upper bound reduced to 1.74% with updated static analyses for **kernel constraints extraction**.



How Far are We toward Memory-safe eBPF?

- Insight 1: eBPF’s **linear design** makes the fraction of unsafe ops low.
  - ideal for full memory safety validation and enforcement.
- Insight 2: challenge lies not in the complexity of code, but in **precisely identifying and protecting all critical unsafe operations**.
  - Memory accesses by eBPF programs must be completely ensured for all classes of memory safety.
- Insight 3: Specialized defenses should **target unsafe ops/objs**.
- Insight 4: The impact of eBPF operations on **shared data** (with kernel) must be vetted.

Future Directions

- Enhancing **static memory safety validation**.
  - Extract and apply kernel-specific constraints.
  - Adopt compiler-informed techniques, e.g., Rust, WASM.
  - Incorporate syntactic annotations, e.g., Checked-C, EC, CRT-C.
- Advancing **finer-grained isolation**.
  - Pointer forging for indirect corruption.
  - Cross-boundary interface vulnerabilities.
- Migrating to **memory-safe languages**.
- **Formal verification** or **bounded model checking** for JIT Compiler.
  - Like what have been done for WASM.