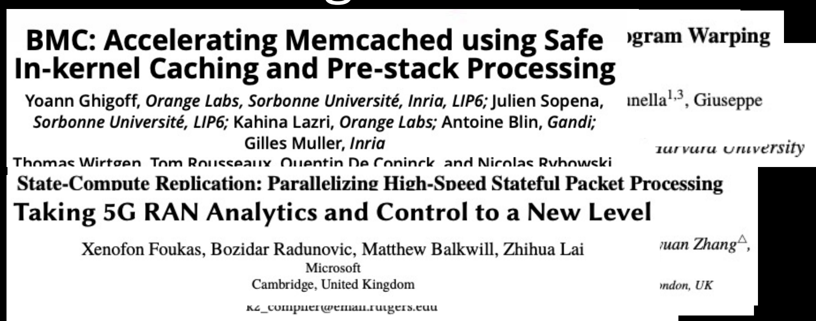
SoK: Challenges and Paths Toward Memory Safety for eBPF

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

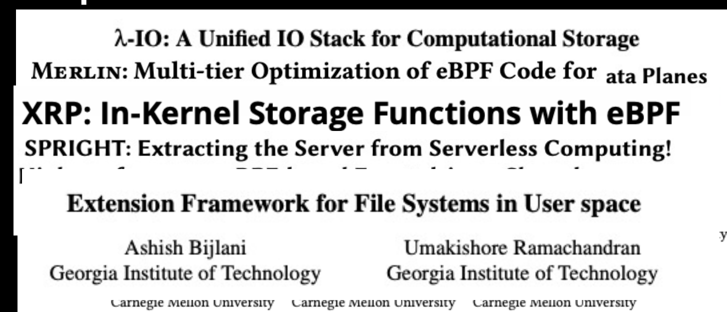
* Penn State † EPFL ‡ UC Riverside

eBPF Has Been Drawing More Attention

- Networking

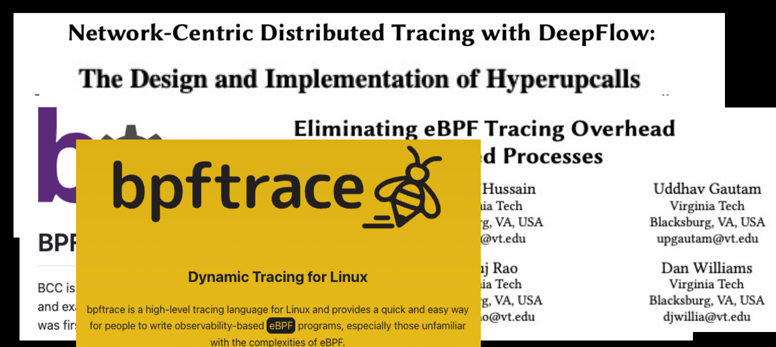


- Optimization

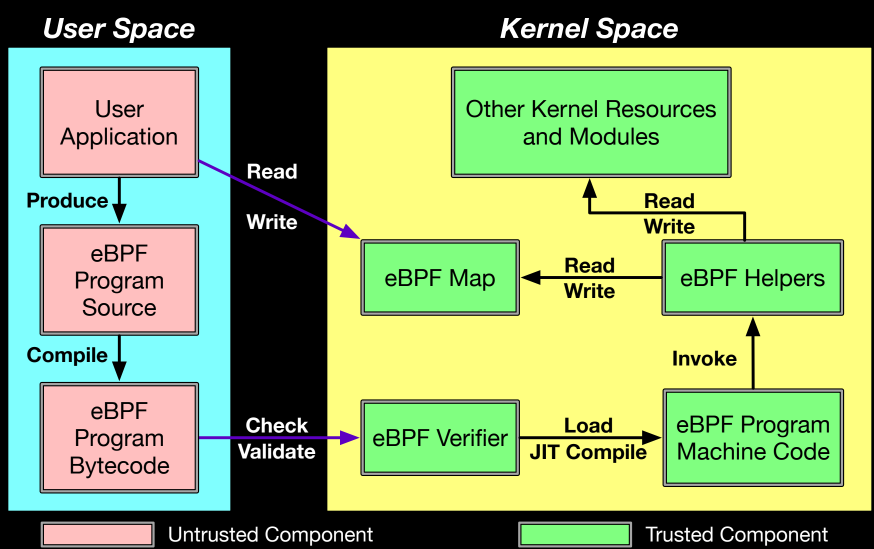


- Security

- Tracing



eBPF Workflow and Its Trust Model



Do they work as expected?

- **eBPF programs should not perform unsafe memory accesses.**
- eBPF helper functions are trusted kernel APIs but do not have any validations.
- 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 bug 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;
}
```

```
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;
}
```

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

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

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

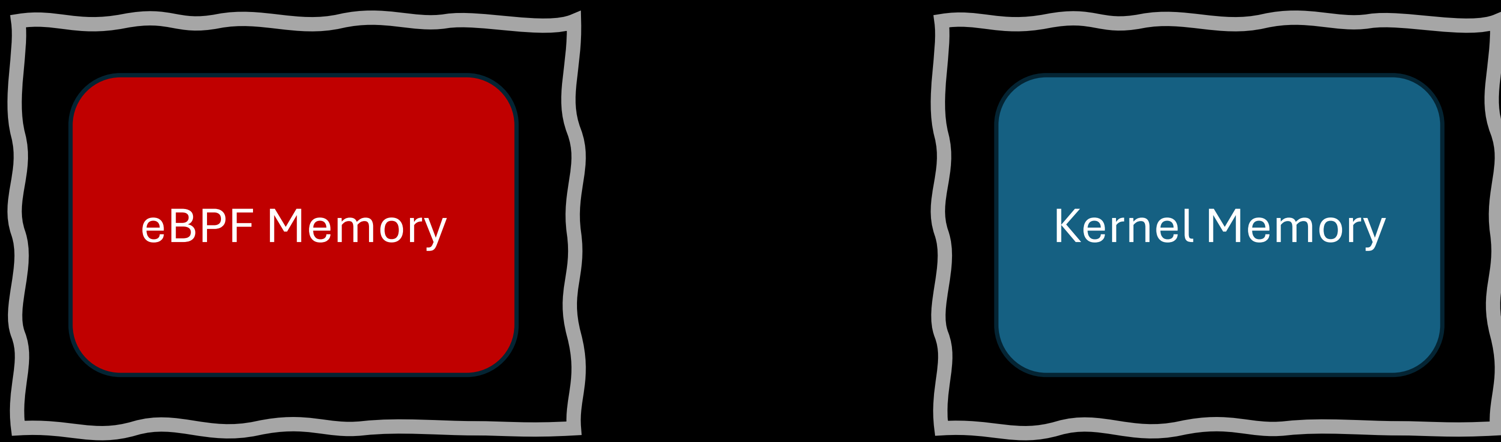
Take Capability CAP_BPF as an Example

- 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 execution** remains common in practice

Research Directions

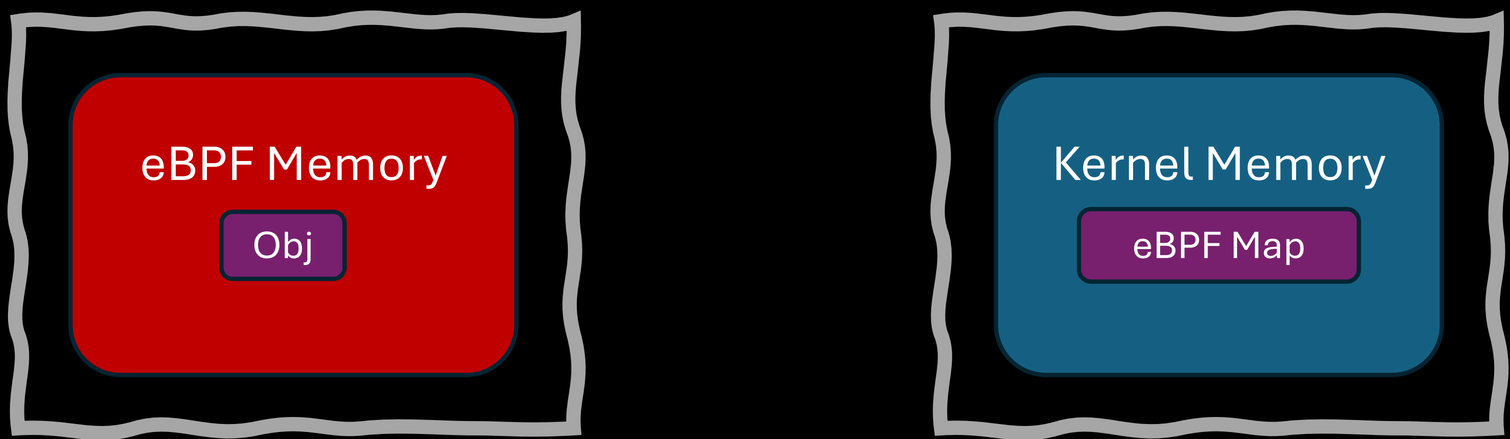
- Fuzzing
  - Inherently incomplete
  - Hard to generate eBPF programs that both pass verifier and trigger bugs
- Isolation
  - Aims to restrict access to eBPF and shared (eBPF map) memory
  - Does not address risks from **indirect kernel access**
- Runtime Checks
  - Limited by the resource constraints and instruction limits.
- Static Validation
  - Existing approaches are either unsound or incomplete.

Limitation – Isolation as Example



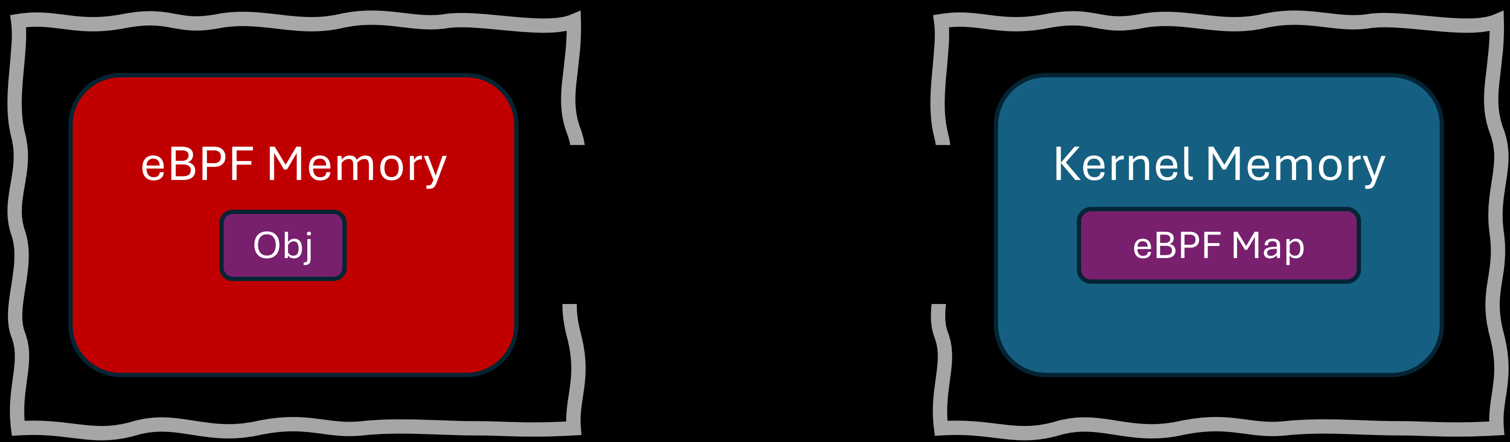
- Isolation separates eBPF memory and Kernel memory
- Unauthorized memory accesses are prevented at isolation boundary

Limitation – Isolation as Example



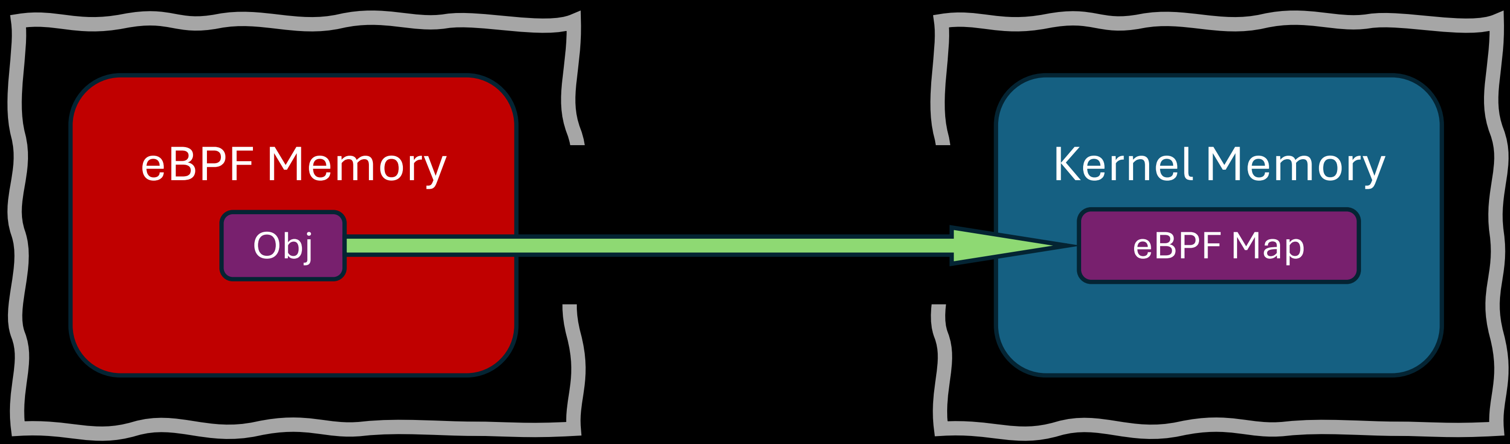
- However, objects in eBPF program can be transmitted to be saved in kernel data (e.g., eBPF map) via eBPF helpers.

Limitation – Isolation as Example



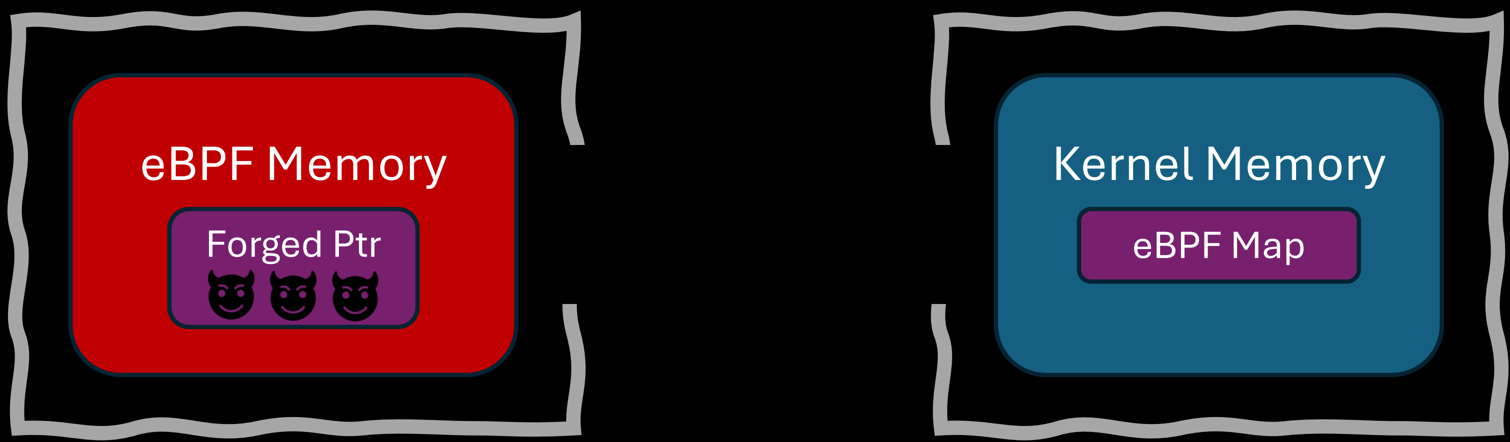
- However, objects in eBPF program can be transmitted to be saved in kernel data (e.g., eBPF map) via eBPF helpers.
- The access to kernel data through such pointers need to be preserved.

Limitation – Isolation as Example



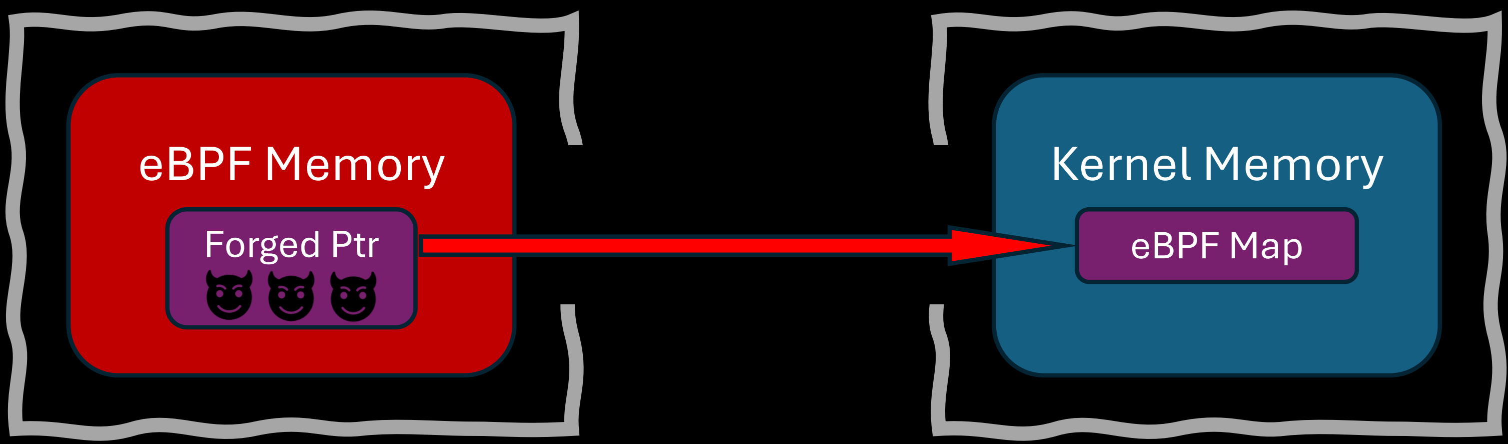
- However, objects in eBPF program can be transmitted to be saved in kernel data (e.g., eBPF map) via eBPF helpers.
- The access to kernel data through such pointers need to be preserved.

Limitation – Isolation as Example



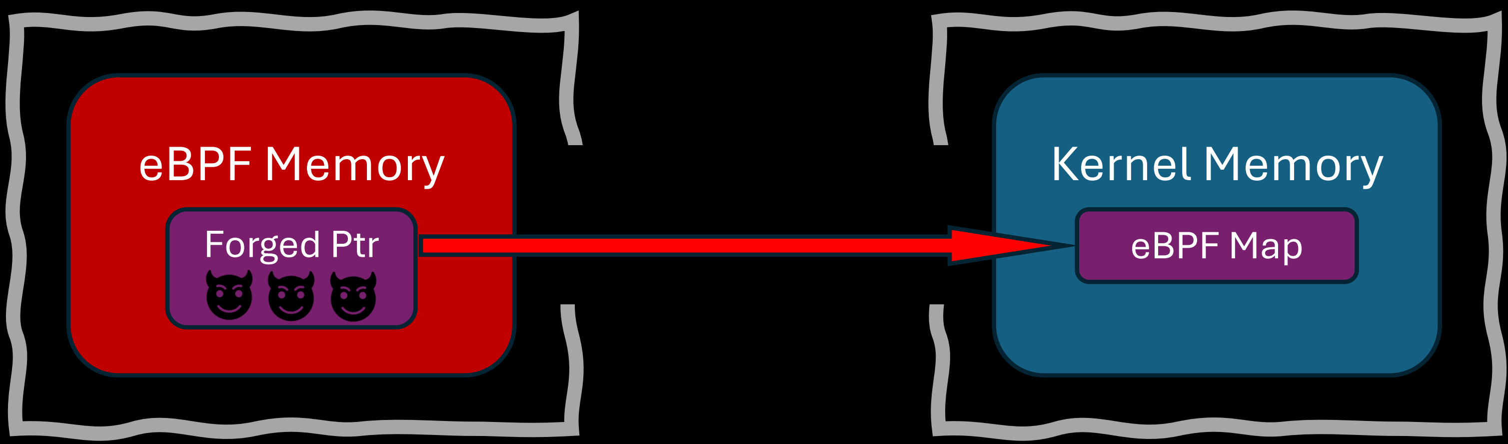
- Such opened-gates allows attacker to
  - exploit the memory errors in eBPF program
  - forge a pointer arbitrarily

Limitation – Isolation as Example



- Such opened-gates 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.

Memory Safety in eBPF Context

- Goal: Memory safe accesses in eBPF programs and kernel use of eBPF-generated pointers
- Spatial Safety
  - All accesses of a memory object must only access memory within the object's allocated region.
- Type Safety
  - All accesses of a memory object must only access the same data types for each offset and each field.
- Temporal Safety
  - All accesses of a memory object must not access the object's allocated region before allocation nor after the object's deallocation.

Identify Unsafe Memory Operation in eBPF

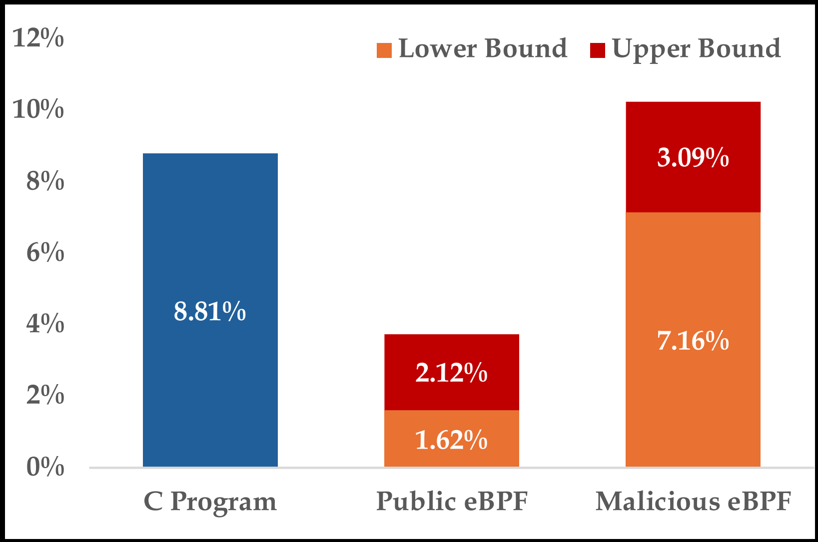
- Hypothesis: eBPF should be close to memory safe in terms of low fraction of unsafe operations, but how to identify them?

- Approach – DataGuard (NDSS 2022) and Uriah (CCS 2024)

- 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 absolute 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 use.
- **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.
  - Good start for full memory safety validation and enforcement
- Insight 2: Must ensure that all eBPF operations cannot exploit victim objects to prevent illicit modification of shared data/pointers
  - Ideally, make eBPF programs run in a memory safe manner
  - E.g., Easier to validate temporal safety statically
- Insight 3: Must ensure that all pointer values shared with the kernel are memory safe

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
- Advancing finer-grained isolation
  - Pointer forging for indirect corruption
  - Cross-boundary interface vulnerabilities
- Migrating to memory-safe languages