On the security of eBPF Linux kernel extensions

Riccardo Sisto
Politecnico di Torino, Italy

HAIC Talks - October 20th, 2025 - Aalto

Outline

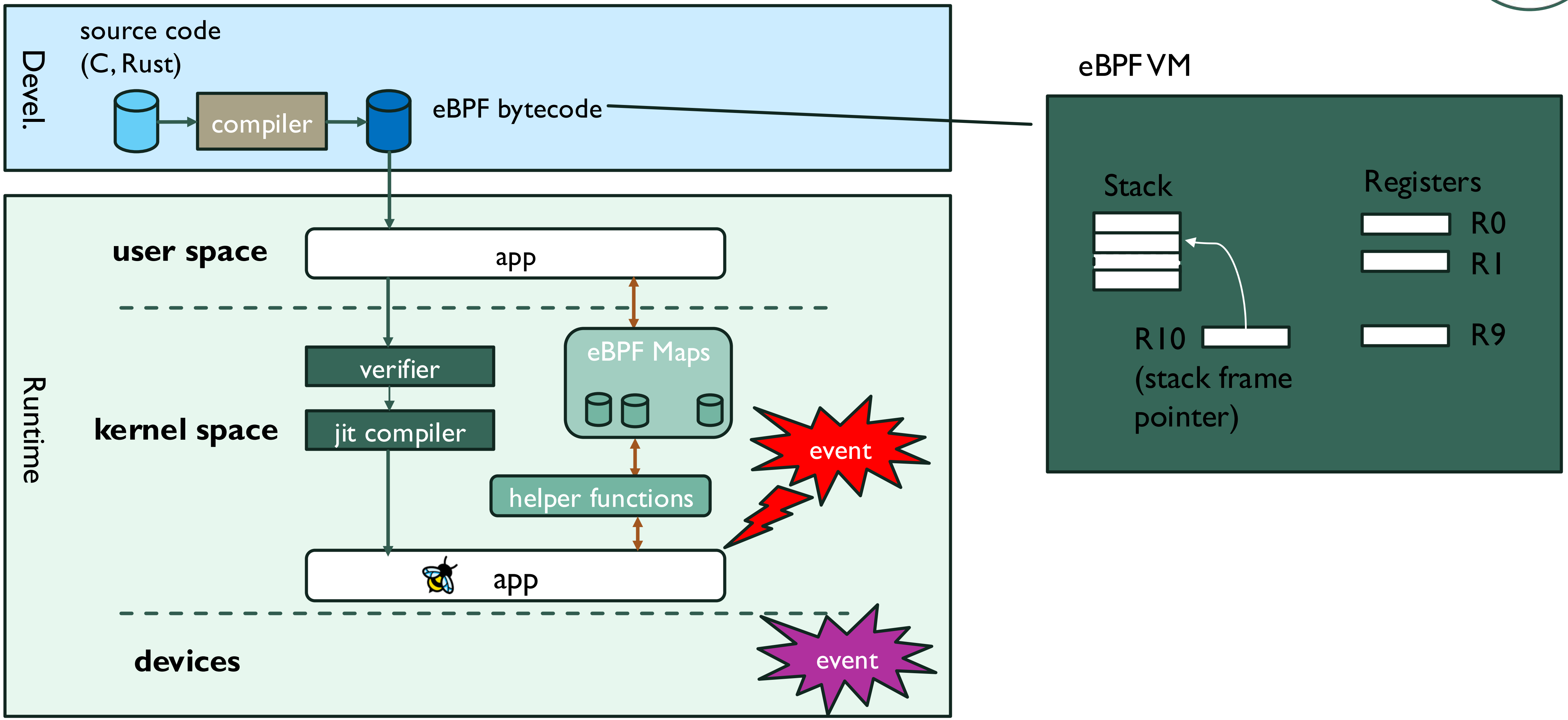
- eBPF Linux Kernel Extensions – How they work and why they are security critical
- eBPF attack surfaces and vulnerabilities
- Possible Mitigations
- Recommendations and Conclusions

The eBPF Revolution

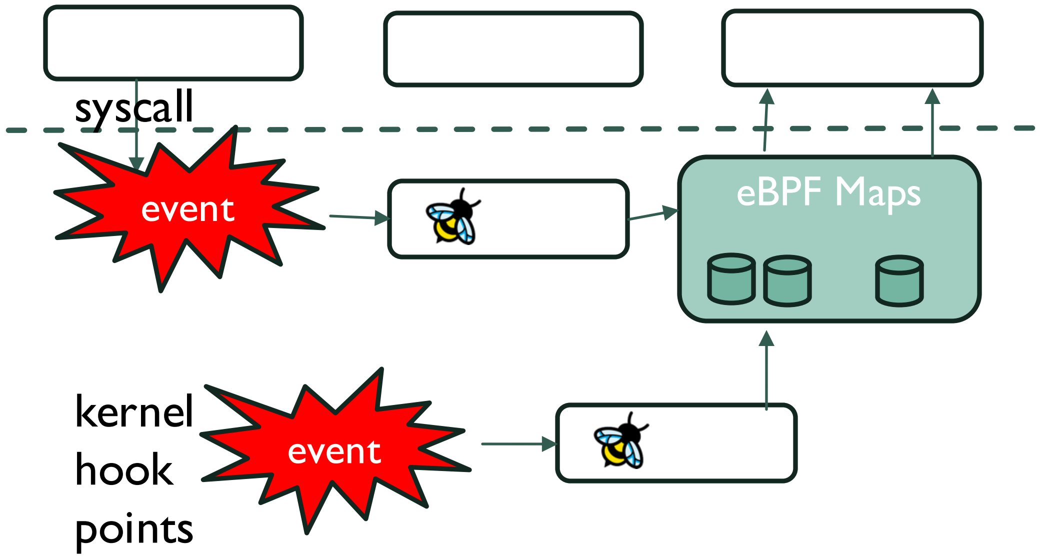
- eBPF: Dynamic Kernel Programmability
  - no need to rely on kernel modules (flexibility)
  - runtime changes without kernel recompilation
  - kernel safety/isolation mechanisms
  - high performance
- Many monitoring/observability and security enforcement applications

cilium tetragon Falco

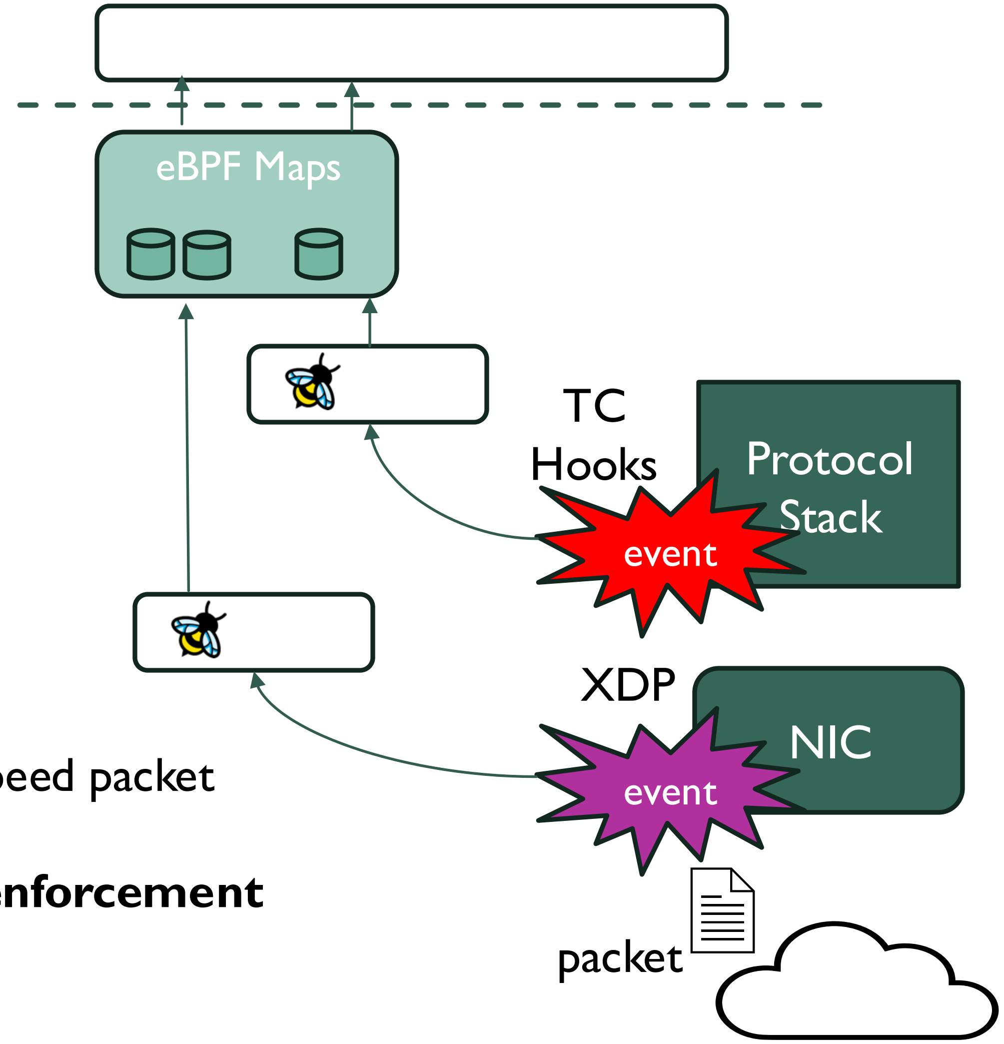
How eBPF works



eBPF Applications

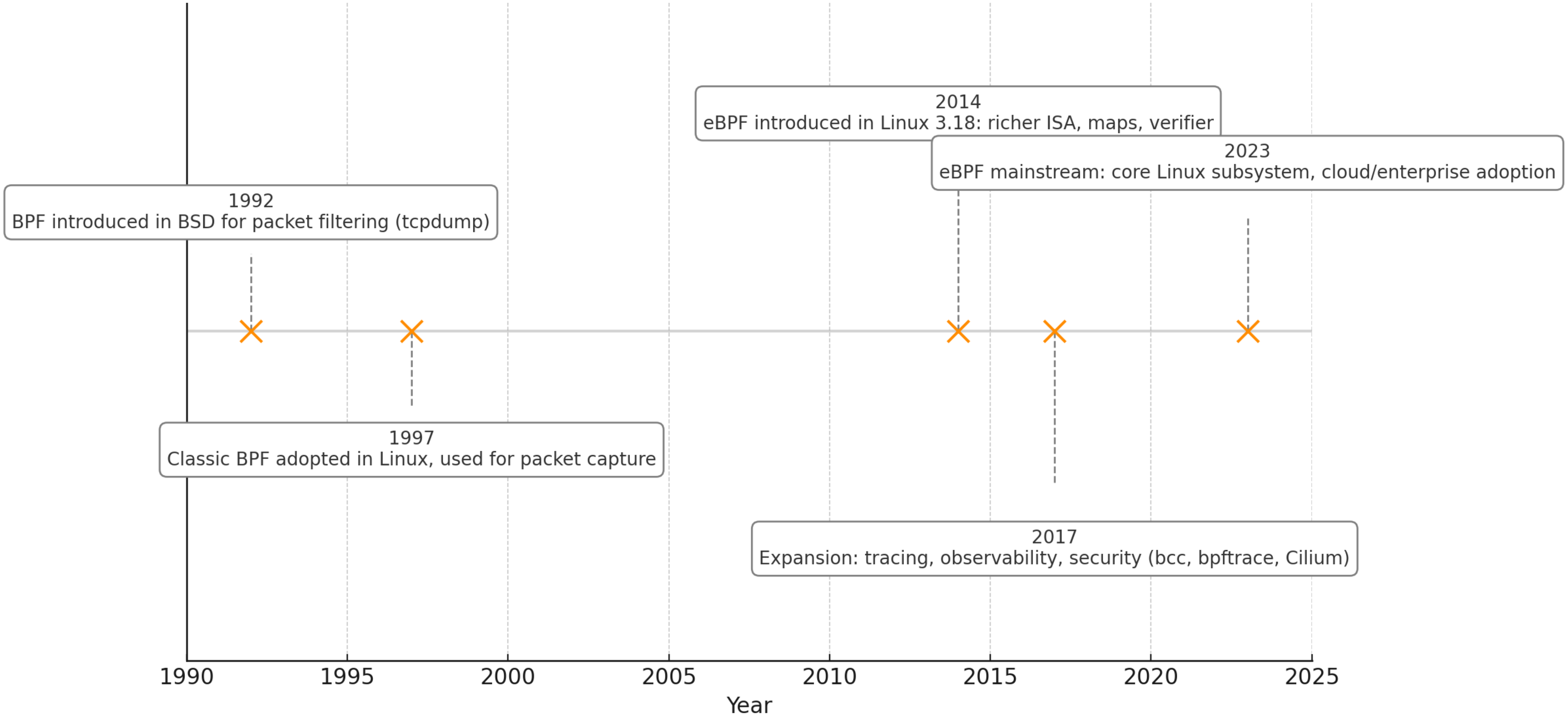


System Monitoring and Tracing (performance, observability)
Security enforcement



Networking (high-speed packet processing)
Network security enforcement

eBPF History



Executing code in kernel space: Possible Hazards and Threats

- eBPF code runs in kernel space
- eBPF code could accidentally or intentionally (malicious code)
  - leak kernel private data (confidentiality)
  - make the kernel crash or become unstable/unavailable (availability)
  - make the kernel diverge from its intended behaviour (integrity)
- Because of eBPF "power", consequences may be quite serious
  - dangerous helpers (read/write any process's use space memory, override return values)
  - networking eBPF can disrupt/manipulate networking functions
- How is all this prevented by the eBPF Ecosystem?

eBPF Safety/Security Measures: Capabilities

- Main goals:
  - allow eBPF program loading and attaching only to trusted processes
  - selectively allow specific operations on specific program types/features
  - enable differentiated verifier checks based on process trust level
- How: by exploiting the Linux Capability system

No capability
- load/read/attach only certain program types (e.g. socket filter on own socket)
- limited program features and helper functions

CAP_BPF
- create maps
- load/read more types of programs
- use advanced program features (e.g. bounded loops)
- …

CAP_PERFMON
- load/attach tracing programs

CAP_NET_ADMIN
- load/attach networking programs
- …

CAP_SYS_ADMIN
- no limits

eBPF Safety/Security Measures: The Verifier

- Main goal: reject eBPF programs that are not safe/secure to run
- How:
  - by performing **static code analysis**: CFG validations + a sort of symbolic execution (state exploration)
    - check no instruction is unreachable
    - check bytecode validity (no invalid instructions)
    - track possible register content types/values
    - check only the admitted helper functions are called with arguments satisfying their required constraints
    - check memory safety (bounds checking, pointer checking, stack safety) and no undefined behaviour
    - check program termination (1M instruction limit, bounded loops only,...)
    - check no information is leaked (if unprivileged) + access control
    - check the process holds the required capabilities

no formal soundness proof

helper functions are not checked because trusted

eBPF Safety/Security Measures: Jit and Runtime

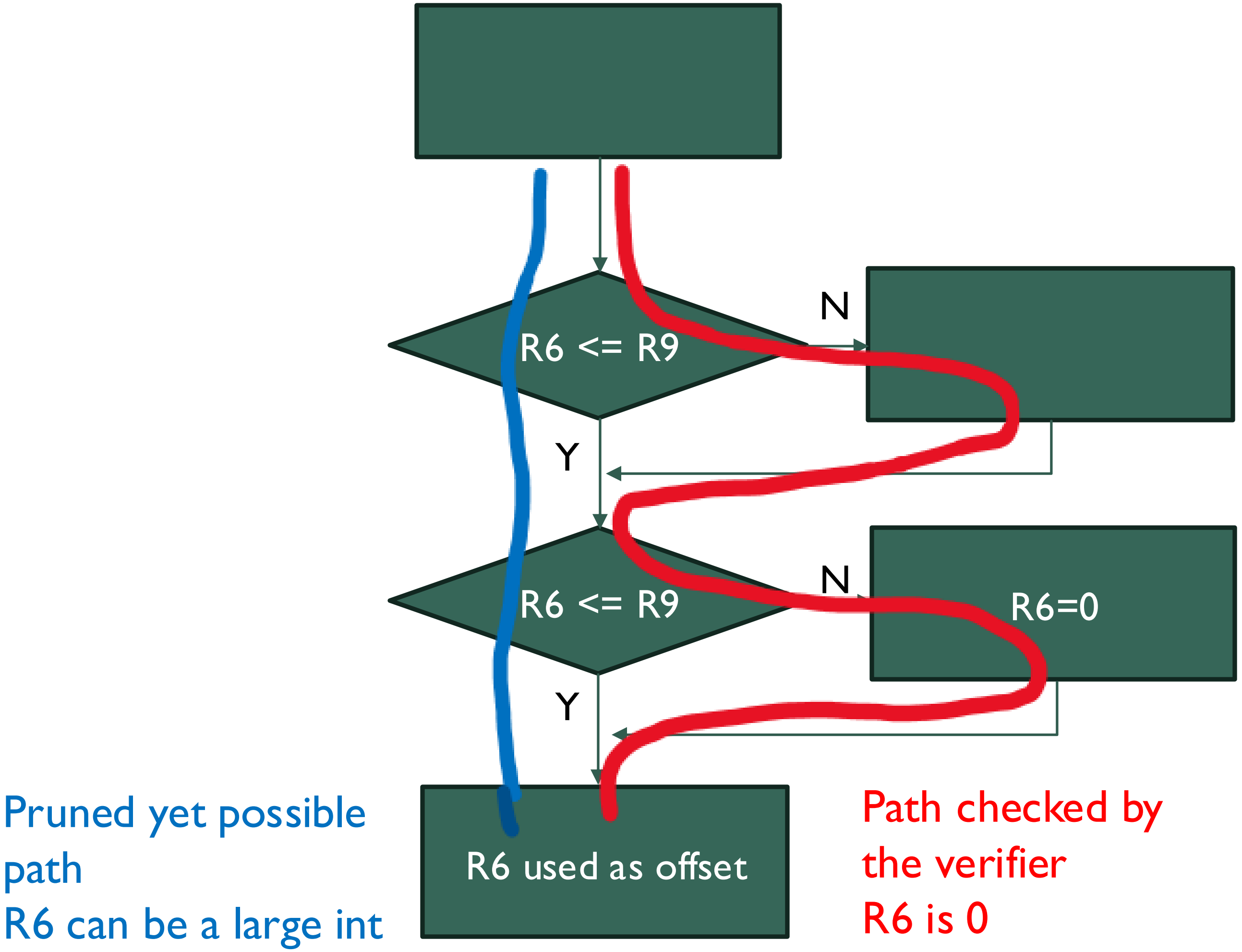
- Main goal: prevent some runtime issues
- How:
  - prevention of certain undefined behaviours (e.g. divide by zero)
  - protection against transient execution attacks (e.g., Spectre)

Limitations of eBPF Security

- Criticism on the eBPF security approach and its maturity have been expressed by several researchers
- Main issues:
  - vulnerabilities
  - verification problem complexity/approach

Vulnerabilities in the Verifier: unsound checks
Example: CVE-2023-2163

- Error in path pruning
- Linux kernels affected
  - 5.3 –> 5.4.242 (excluding)
  - 5.5 -> 5.10.179
  - 5.11 -> 5.15.109
  - 6.2 -> 6.2.13
- Exploitation
  - malicious code can
    - cause the verifier evaluate R6 as 0 while it is not
    - cause out of bounds memory access



Vulnerabilities in the Verifier: missing checks
Example: CVE-2021-4204

- Missing size check for helper functions bpf_ringbuf_submit and bpf_ringbuf_discard
- Such functions can be called with any offset value, leading to memory corruption
- Linux kernels affected
  - before 5.8.0
- Exploitation
  - call vulnerable functions with offset that exceeds reserved space to cause out of bounds memory access

Helper functions do not check arguments

```
void *p = bpf_ringbuf_reserve(&rb, 64, 0);
if (!p)
  return XDP_ABORTED;
...
bpf_ringbuf_submit(p + 128, 0);
```

Vulnerabilities in Other Ecosystem Components
Example: CVE-2024-56614

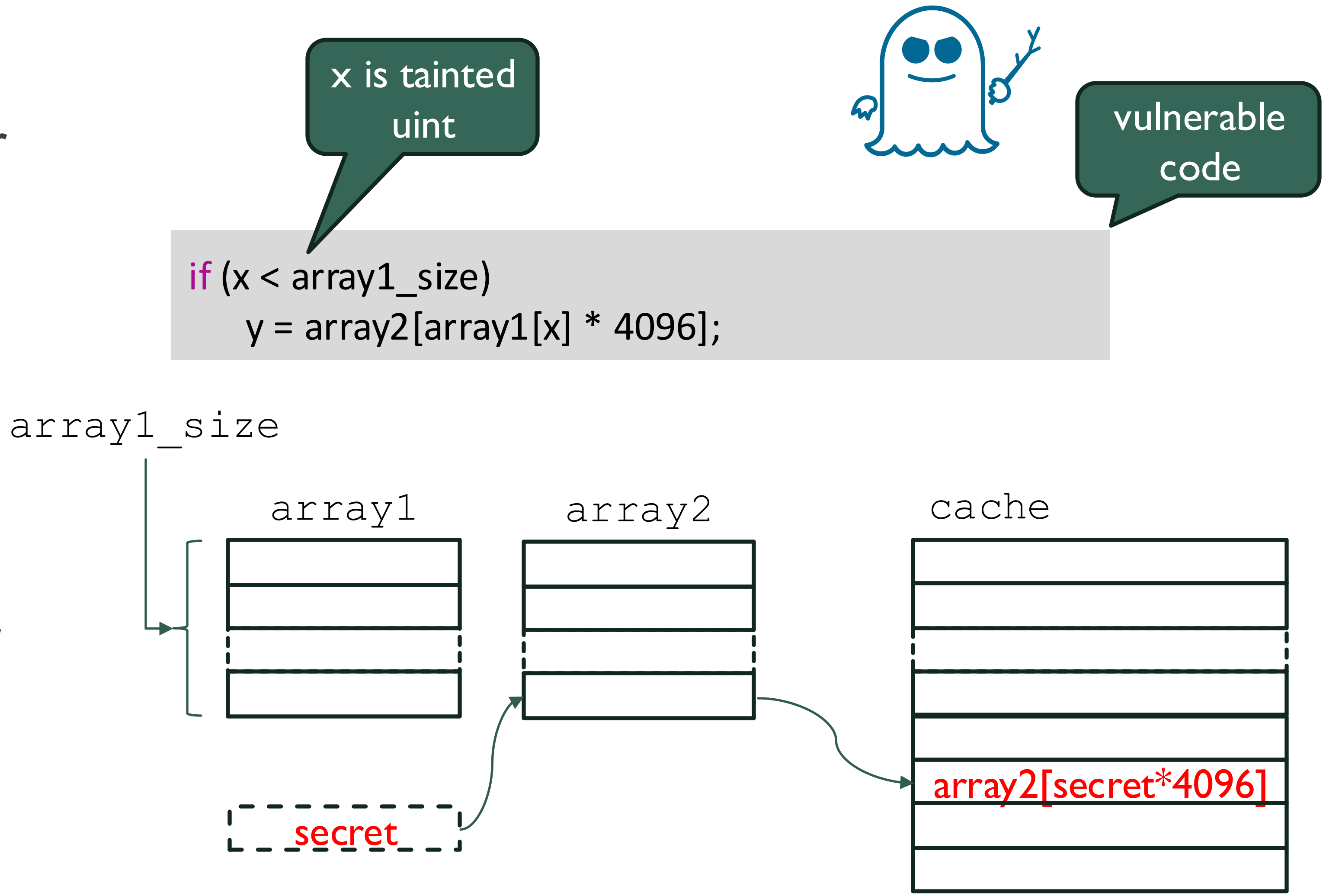
- Vulnerability in the kernel function that implements the syscall command to delete an element in an XSK map
- Linux kernels affected
  - 4.18 –> 5.15.174
  - 5.16 -> 6.1.120
  - 6.2 -> 6.6.66
  - 6.7 -> 6.12.5
- Exploitation
  - create XSK map with maxentries=0x80000000+N
  - add and then delete entry with key 0x80000000+X with 0<X<N => k = - (0x80000000-X)
  - memory outside map is corrupted

```
static long xsk_map_delete_elem(struct bpf_map *map, void *key)
{
  struct xsk_map *m = container_of(map, struct xsk_map, map);
  struct xdp_sock __rcu **map_entry;
  struct xdp_sock *old_xs;
  int k = *(u32 *)key;

  if (k >= map->max_entries)
    return -EINVAL;

  spin_lock_bh(&m->lock);
  map_entry = &m->xsk_map[k];
  old_xs = unrcu_pointer(xchg(map_entry, NULL));
  if (old_xs)
    xsk_map_sock_delete(old_xs, map_entry);
  spin_unlock_bh(&m->lock);

  return 0;
}
```

max_entries is u32
k converted to u32
for comparison

Transient Execution Attacks

- eBPF programs are subject to transient execution attacks (such as Spectre) as any other program
- Example: Spectre v1 (conditional branch misprediction)
  - As speculative execution is essential for performance, HW mitigations are partial, and SW mitigations necessary.
  - Attack causes info leakage



eBPF Spectre Mitigations

- For V1 Spectre
  - Verifier: replace variables with constants if possible
  - Verifier: analyse paths that could be reached by speculation
  - Verifier: insert speculation barriers where necessary

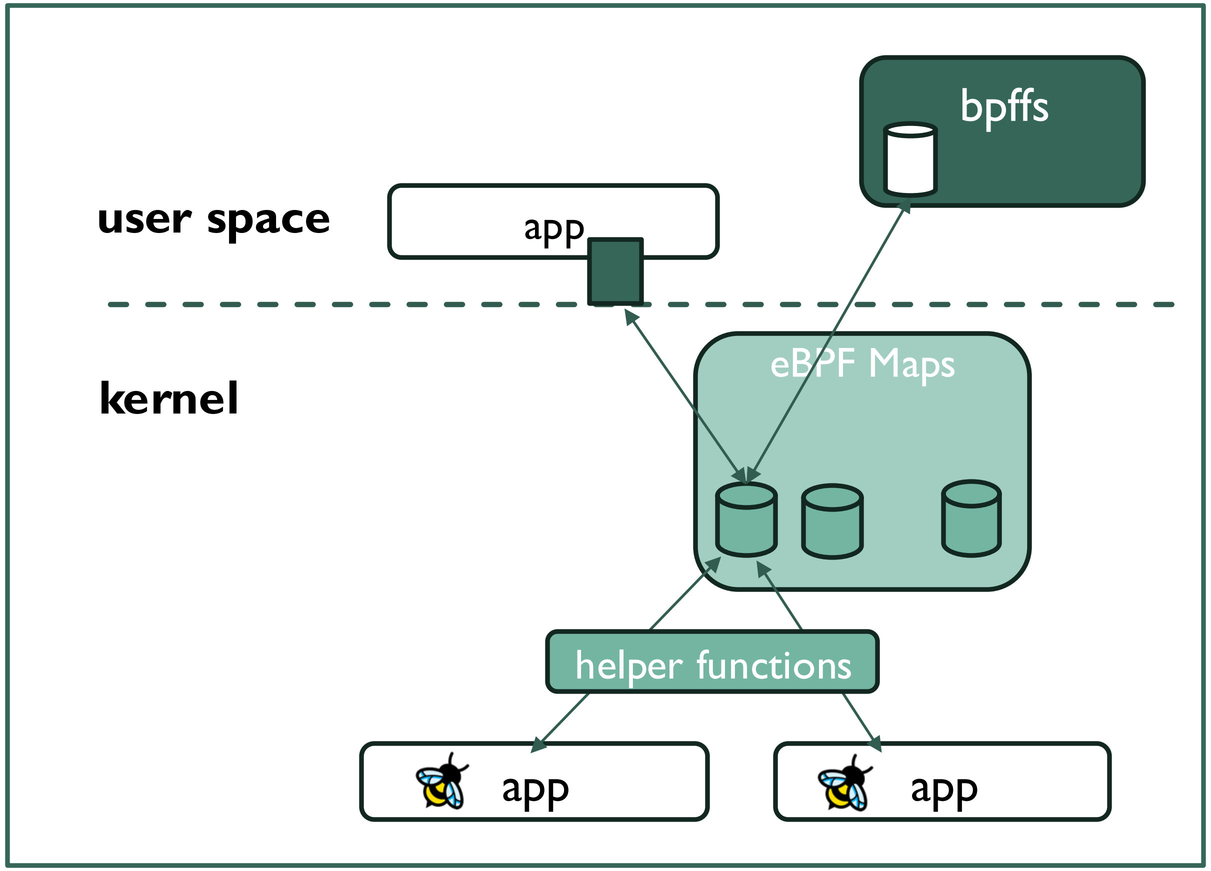
Recent addition

- For V2 Spectre
  - JIT Compiler: retpoline (return trampoline) : speculative execution is trapped into a loop until the real branch address is computed
  - JIT Compiler: remove speculation by using direct calls when possible
- Applied to unprivileged eBPF code only

May discourage unprivileged eBPF

Other security-related issues: Isolation Limitations Example: Maps

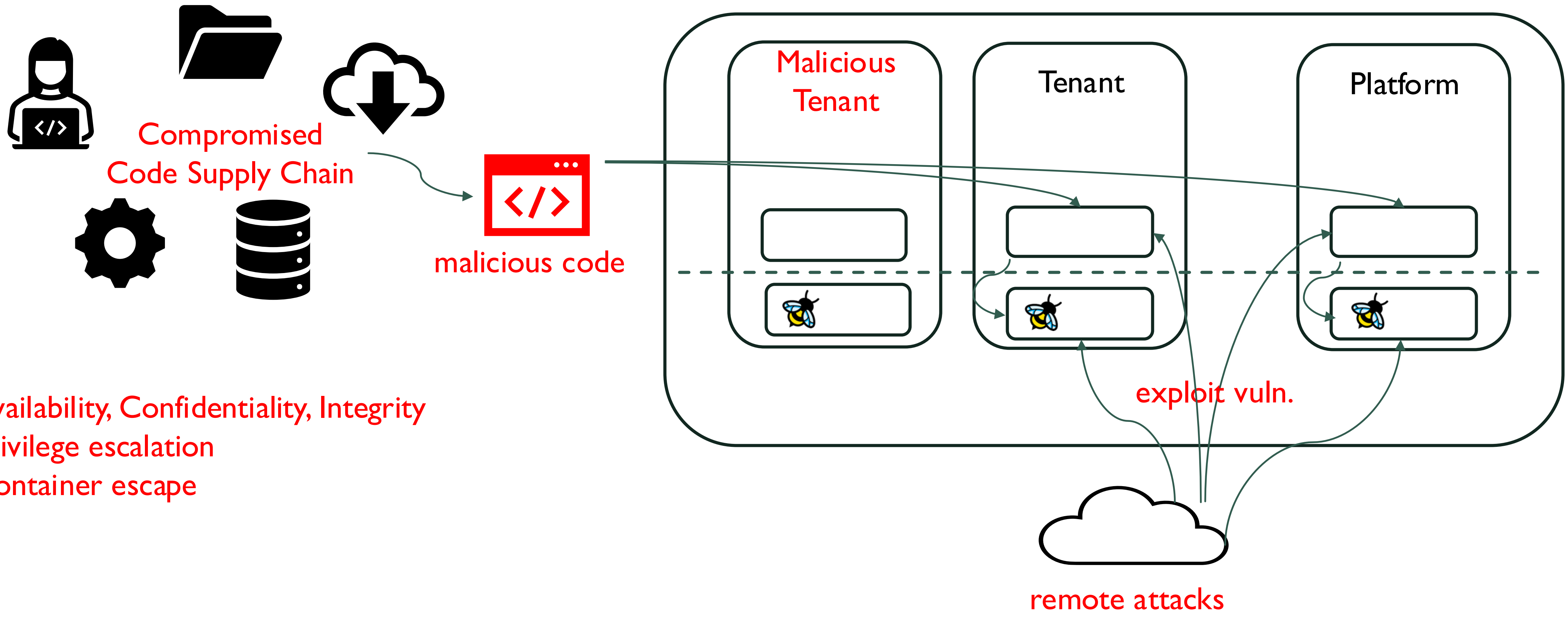
- eBPF map isolation depends on access to map file descriptors or pinned map files in the BPF file system
- Getting access to a map file descriptor or map file enables a process to access the map
  - requires privileges (capabilities and filesystem permissions)
  - risk of improper access to other program's maps



Example: Attacks based on eBPF map FD Hijacking

- Since Linux 5.6 the system call **`pidfd_getfd`** lets a process get a duplicate of another process's FD (requires CAP_SYS_PTRACE)
- By using this feature and iterating over FD values, an attacker process having CAP_SYS_PTRACE can access and even modify an eBPF map used by a target eBPF program
- The attack was demonstrated to work against FALCO, which holds a list of dispatchers in a map to delegate monitoring tasks to specific eBPF apps

eBPF Threat Model



Availability, Confidentiality, Integrity
Privilege escalation
Container escape

Security Vulnerabilities the eBPF system does not try to prevent

- Software vulnerabilities can be introduced in eBPF programs as in any other program
- Not all of them violate memory safety or access control policies

  => the verifier does not detect them
- Example: programming bugs in security controls
  - relevant when eBPF code is used for security control

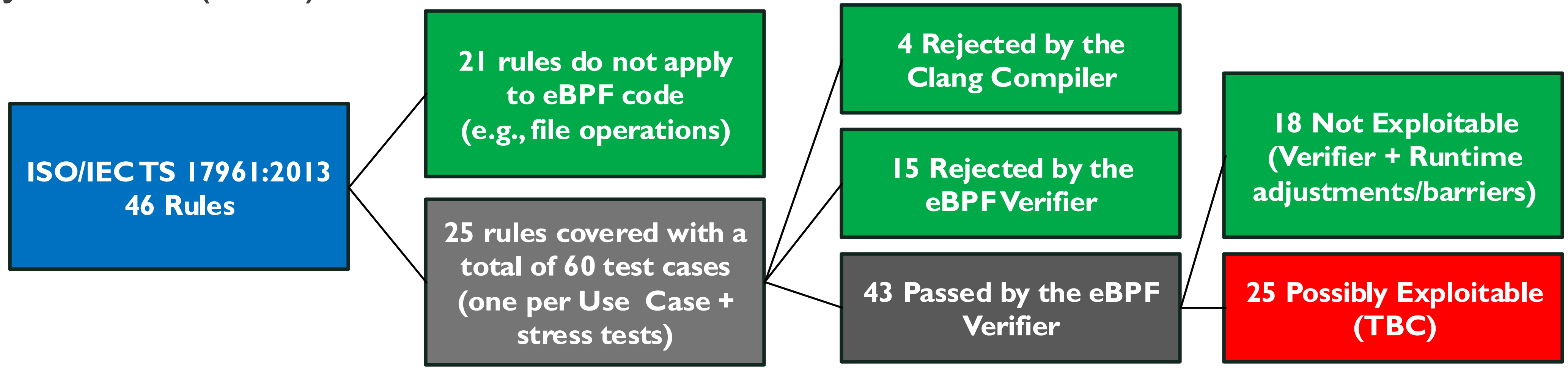
drop packets with payload length field exceeding threshold

```
unsigned int curr;
```

```
{ unsigned int length;
  // extract length from payload
  ...
  if (curr + length > MAX)
     return XDP_DROP;
  else
     curr += length;
}
```

Study of C code Security Vulnerabilities not Detected by the Verifier

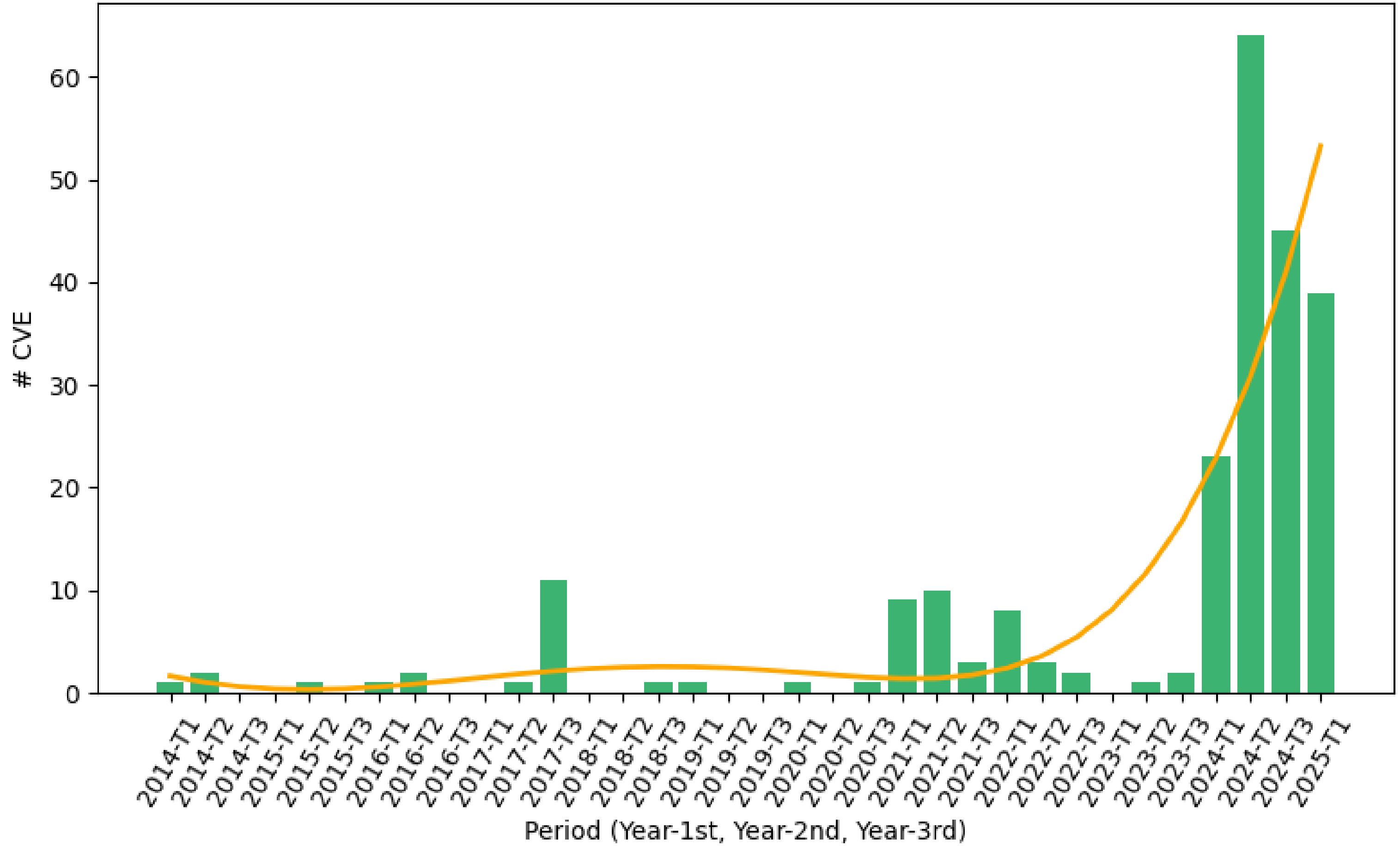
- Study based on ISO/IEC TS 17961:2013 (C secure coding rules)
  - C programming rules whose violation can cause security vulnerabilities
  - Examples:
    - escaping of the address of an automatic object, overflowing signed integers
- Methodology: inject vulnerabilities into real eBPF C code and see what happens
- Preliminary results (WIP)



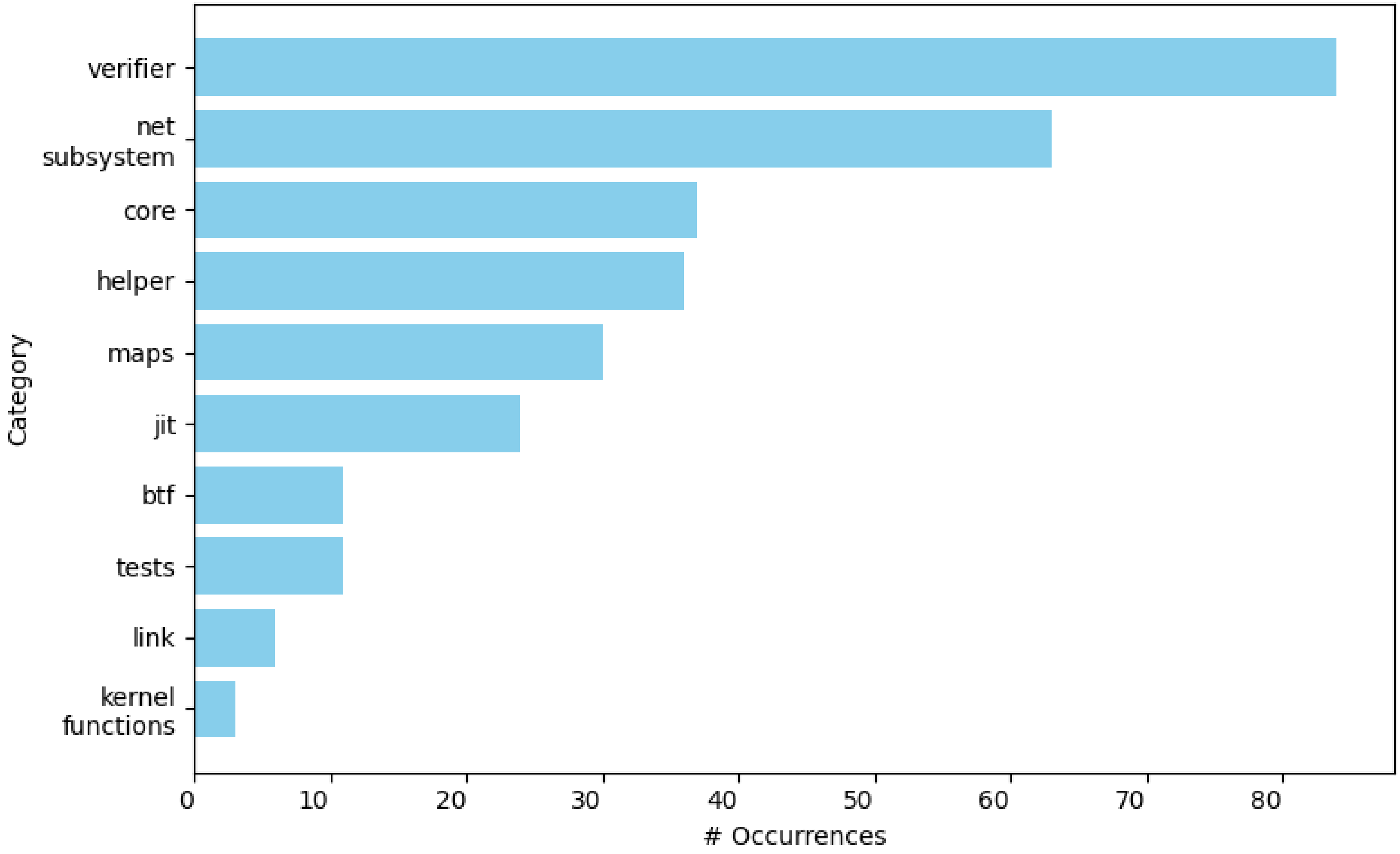
Yes, but how much should we be worried of eBPF-related Vulnerabilities?

- eBPF-related CVEs: Time Distribution

2014-2025: **249** eBPF-related CVEs published



eBPF-related CVEs : Affected Modules



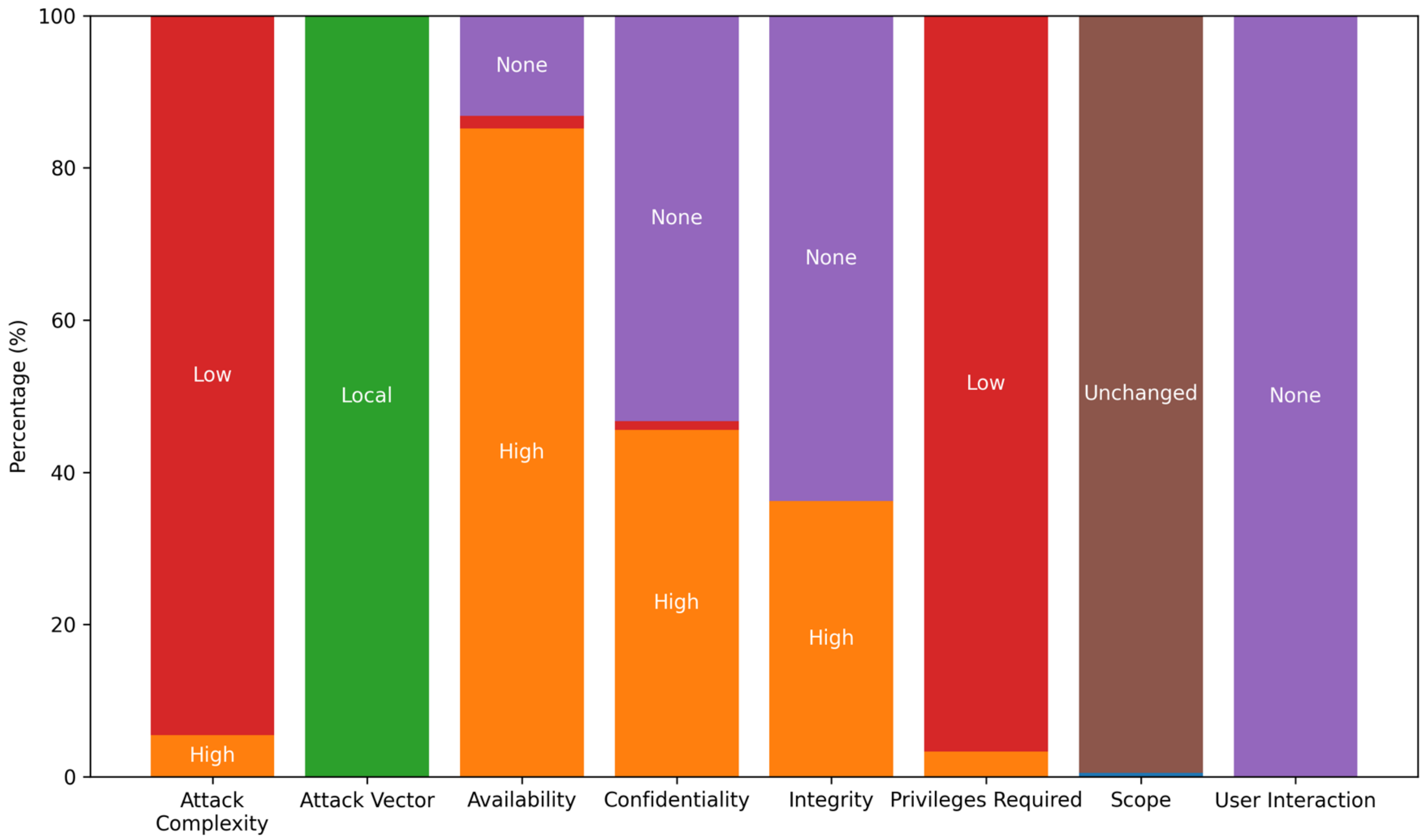
eBPF-related CVEs : Severity

eBPF-related vulnerabilities – CVSS Score

Average: 6.3

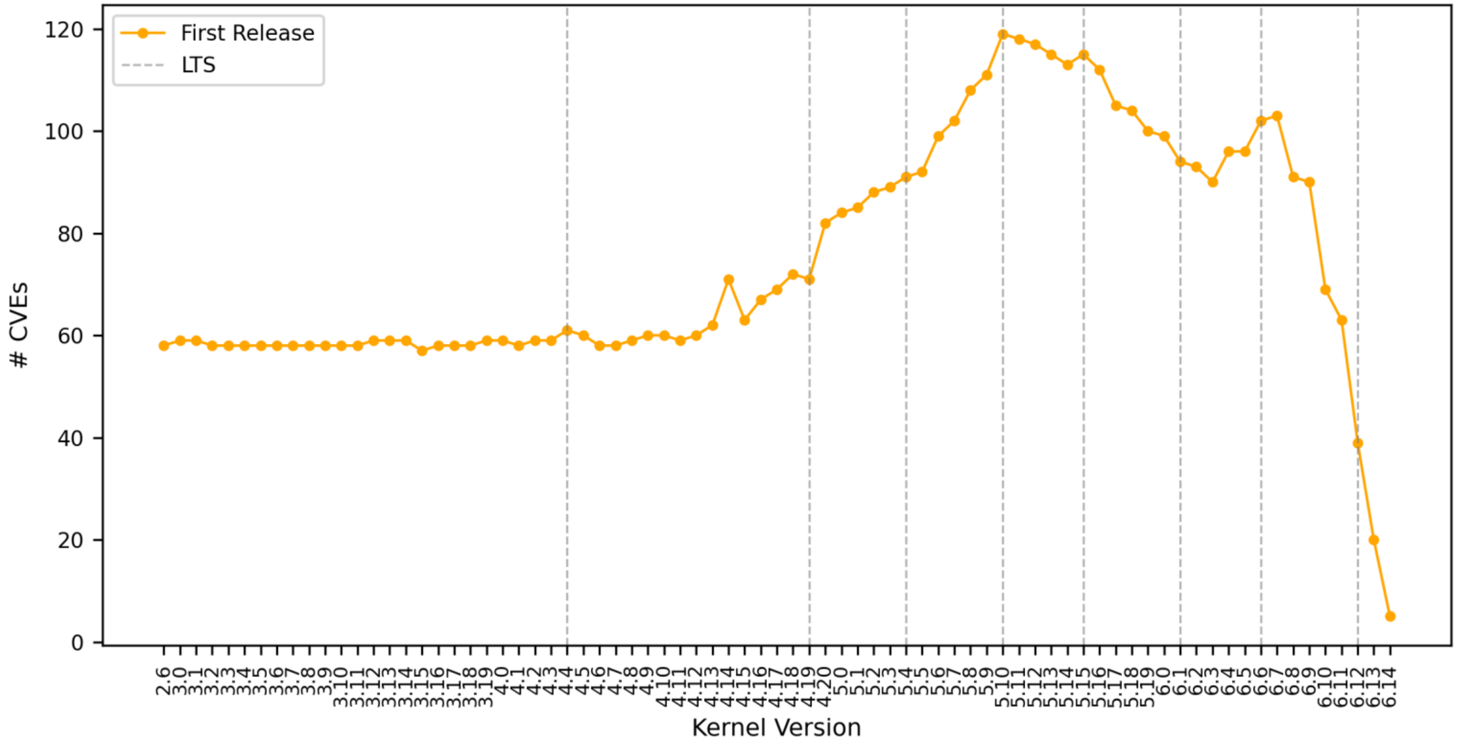
Median: 5.5

Maximum: 8.8



eBPF-related CVEs : Exposure to CVEs of Linux Initial Releases and Patch Latency

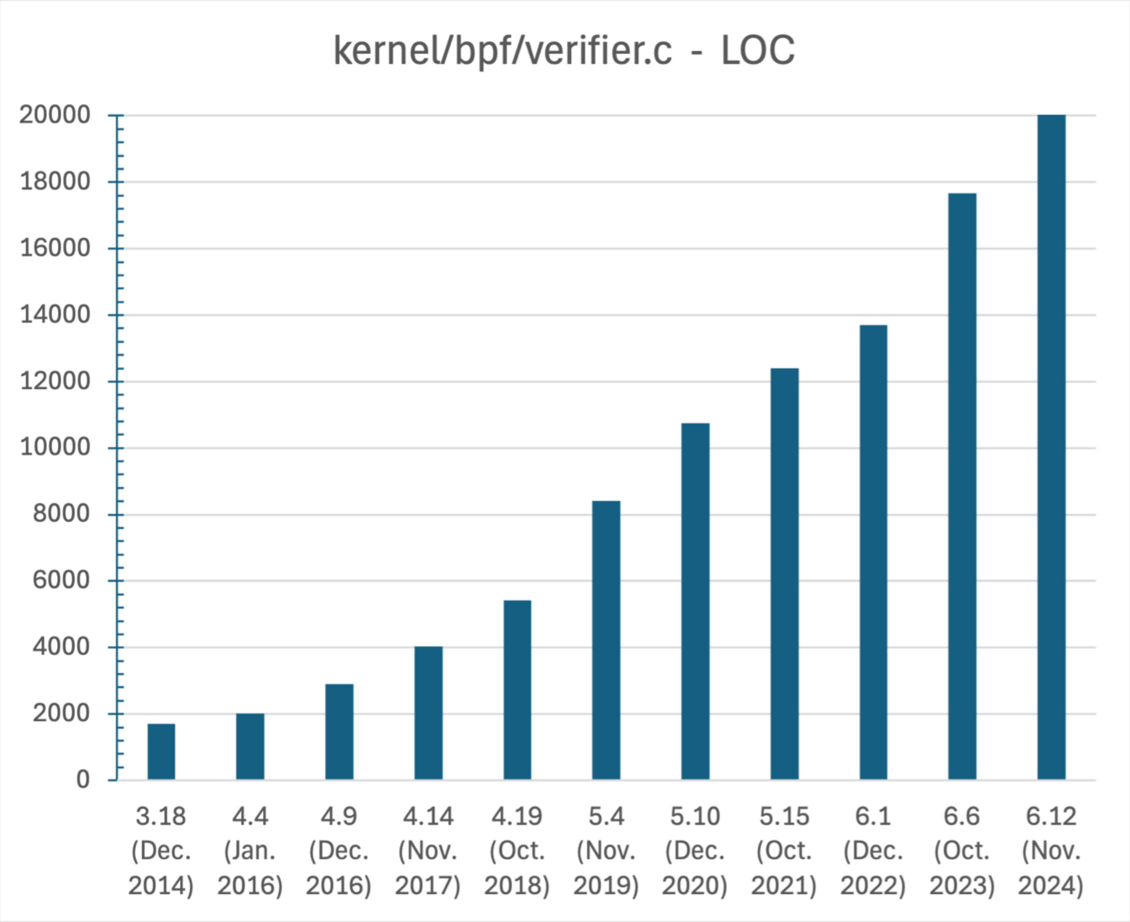
- High Patch Latency:
  - average: 1038 days
  - median: 737 days
  - no correlation with severity



R. Rizza, R. Sisto, F. Valenza, “Analysis of the eBPF Vulnerabilities in the Linux Kernel”, in **CRISIS 2025**,Gatineau, CA, October 10-12, 2025, to appear.

The Concern of the eBPF Verifier Complexity Increase

- Many vulnerabilities have been fixed, but with an increasing complexity.
- Will the process ever converge?
- Complexity increased not only in the verifier (20KLOC)
  - more than 200 helper functions



Mitigation Techniques: Improving Verification

- Several proposals in literature to improve the verifier
  - PREVAIL[1]
  - Verifying the Verifier[2]
  - Hyperbee[3]: idea of combining several different verifiers

[1]G. Jin, J. Li, and G. Briskin, "Research report: Enhanced ebpf verification and ebpf-based runtime safety protection," IEEE S&P 2024
[2]H. Vishwanathan, et al. "Verifying the Verifier: eBPF Range Analysis Verification", CAV 2023
[3]Y. Wang, D. Li, and L. Chen, "Seeing the invisible: Auditing ebpf programs in hypervisor with hyperbee," in 1st Workshop on EBPF and Kernel Extensions (eBPF '23)

Mitigation Techniques: Isolation and Memory Safety Enforcement

- Software-based Fault Isolation solutions
  - SafeBPF[1], SandBPF[2]
- Hardware-based solutions
  - MOAT[3], HIVE[4]

[1]S.Y. Lim et al., "Safebpf: Hardware-assisted defense-in-depth for ebpf kernel extensions," in CCSW 2024

[2]S.Y. Lim, et al., "Unleashing unprivileged ebpf potential with dynamic sandboxing," in eBPF '23

[3]H. Lu, et al, "MOAT: Towards safe BPF kernel extension," in 33rd USENIX Security Symposium 2024

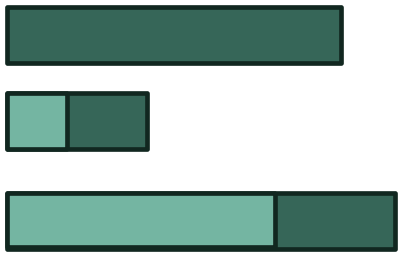
[4]P. Zhang, et al., "Hive: a hardware-assisted isolated execution environment for ebpf on aarch64," in 33rd USENIX Security Symposium 2024

Mitigation Techniques: Security Monitoring

- LSM (Linux Security Modules)
  - SELinux
  - AppArmor
- eBPF-LSM
  - eBPF programs attached to LSM hooks
  - custom checks (e.g. CVE-specific mitigations)
- Tetragon

Memory-Unsafe Operations in eBPF Programs and (non)-Effectiveness of Defenses

- Huang et al.[1] recently studied the occurrence of memory-unsafe operations in commonly used eBPF programs and malicious programs
  - Fraction of unsafe operations evaluated by means of static analysis techniques
    - C programs: 8.81%
    - public eBPF programs: 1.62%-3.74%
    - malicious eBPF programs: 7.16%-10.25%



  - effectiveness of several defenses evaluated by means of experimental evidence and analysis
    - no defense fully or soundly covers any category of memory-unsafe operations

[1] *K. Huang et al., "SoK: Challenges and Paths Toward Memory Safety for eBPF", IEEE S&P, 2025.*

Some Recommendations for better eBPF security

- Avoid unsupported kernels and update kernel regularly
  - if an unsupported kernel is really necessary, consider
    - porting patches
    - adopting other hardening/mitigation techniques for the relevant CVEs (e.g. eBPF-LSM)
- Apply supply chain security good practices
- Give eBPF programs the least privilege
- Do not rely on the verifier only
  - mitigation techniques can help (even if they are not definitive solutions)
  - use security-specific static code analysis tools when developing eBPF code

J. Kelly, J. Callaghan & A. Martin, "eBPF Security Threat Model", https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20%E2%80%94%20eBPF%20Security%20Threat%20Model.pdf .

Conclusions

- eBPF: very powerful, but very dangerous
- eBPF security made much progress
  - but much remains to do
  - and the complexity increase is a big concern
- Diverse approaches to security are necessary to use eBPF securely

Riccardo Sisto

riccardo.sisto@polito.it

Politecnico di Torino - Netgroup:

https://netgroup.polito.it

ELASTIC EU project:

https://elasticproject.eu