Tracee: High Throughput of eBPF Events for Execution Patterns Detections

Rafael David Tinoco
Nadav Strahilevitz

@rafaeldtinoco
@NDStrahilevitz

Aqua Security

Quick Introduction about the Project

Tracee is a Runtime **Security** and **Forensics** tool for Linux. It uses Linux eBPF technology to trace your system and applications **at runtime** and **analyzes collected events** in order to detect suspicious **behavioral patterns**.

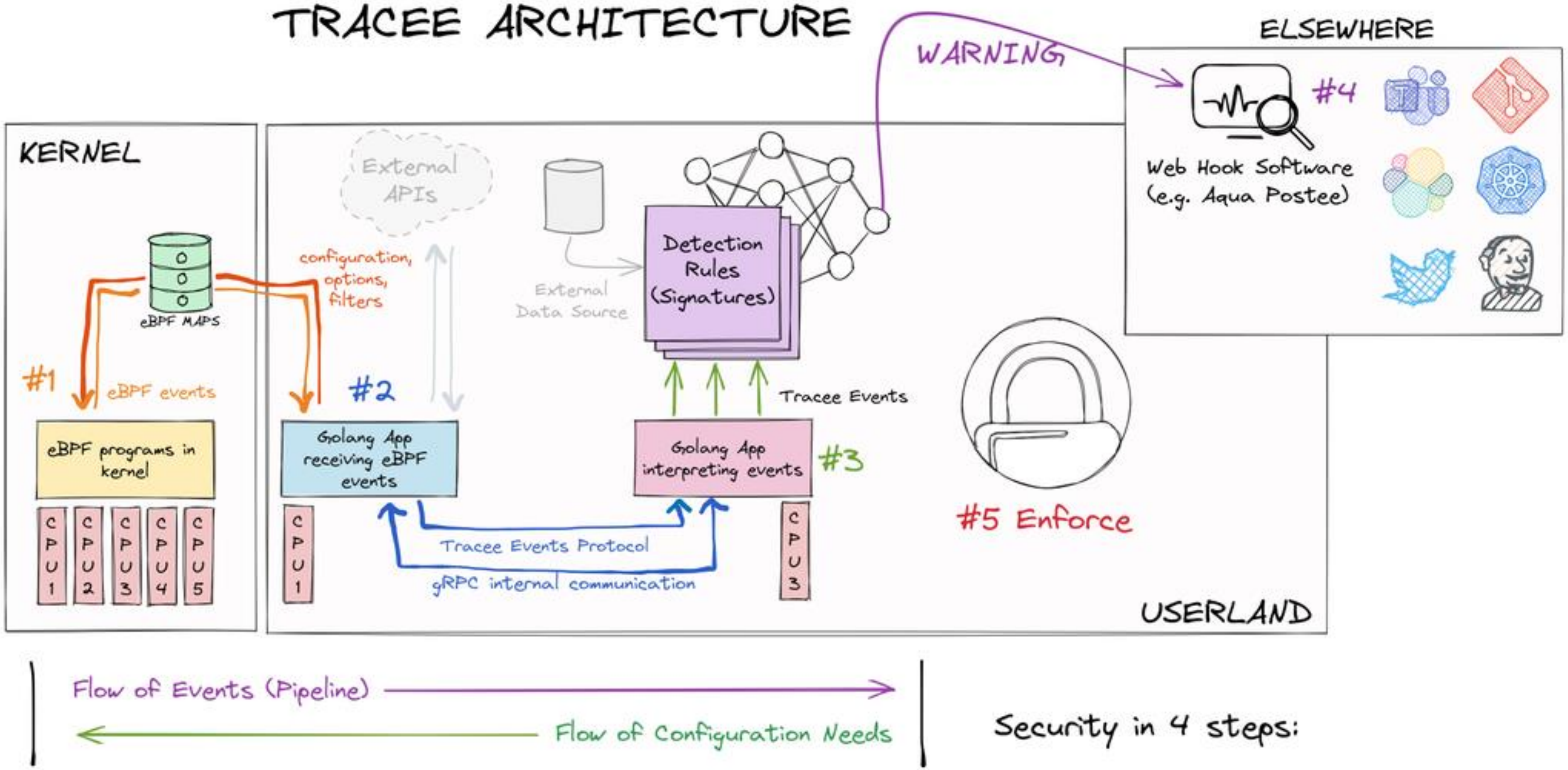
Tracee is composed of the following sub-projects, which are hosted in the **aquasecurity/tracee** repository:

Tracee-eBPF	Linux Tracing and Forensics using eBPF
Tracee-Rules	Runtime Security Detection Engine

https://aquasecurity.github.io/tracee/latest/

Project Author: Yaniv Agman (Tracee Team Leader)

TRACEE ARCHITECTURE



Tracee eBPF Event on OpenAt

Event Type: OpenatEventID
Name: "openat"
Probe: "openat" syscall

Params: {
  Type: "int", Name "dirfd"
  Type: "const char", Name: "pathname"
  Type: "mode_t", Name: "mode"
}

Tracee Rules Signature

Metadata:
  ID: TRC-X

Selected Events:
  "OpenAt"

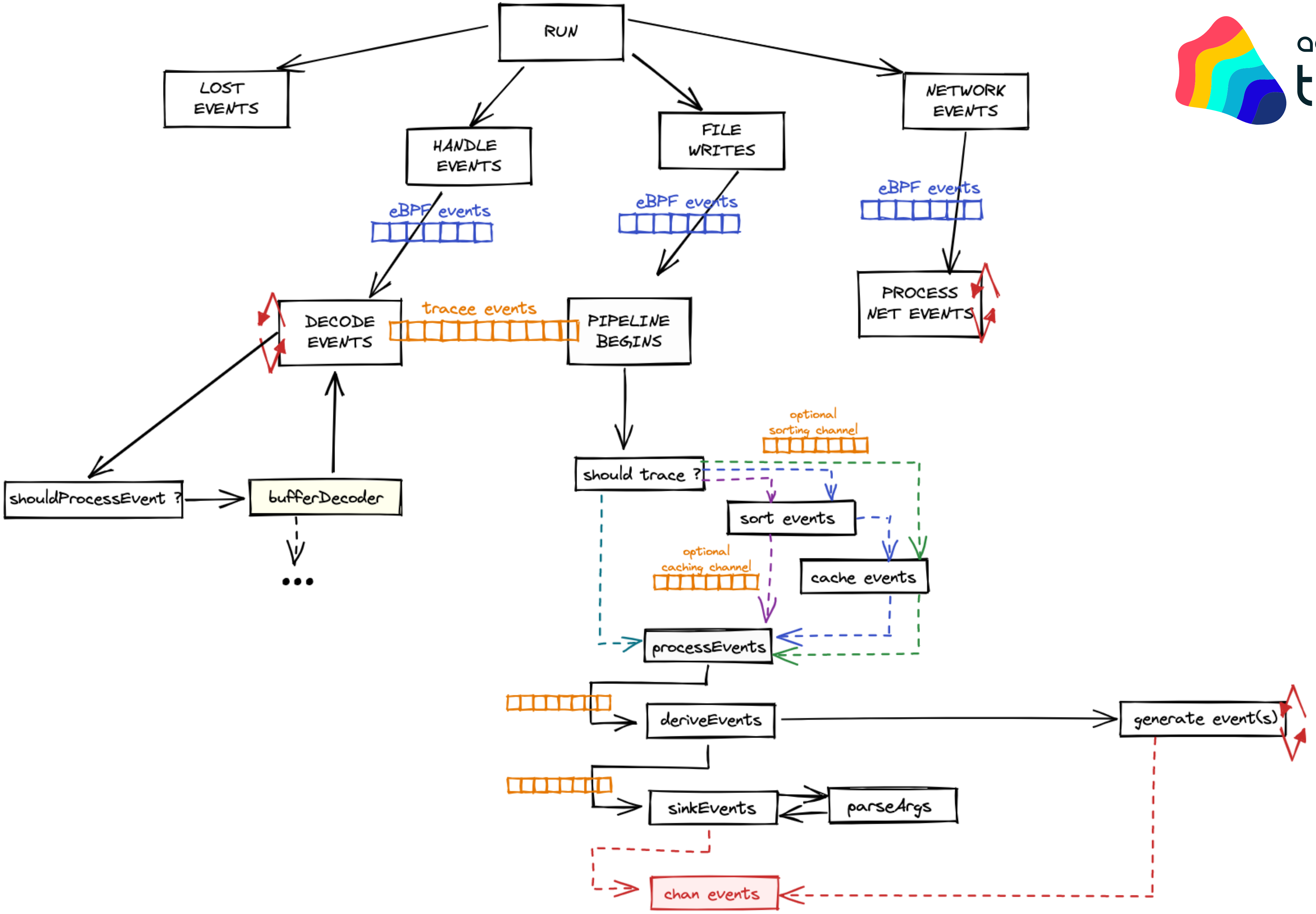
Match:
  "Pathname" == "/dir/filename"

Security in 4 steps:

- #1 Generate
- #2 Collect
- #3 Detect
- #4 Consume
- #5 Enforce

aqua tracee

INTRO
DEMO


RUN
LOST EVENTS
HANDLE EVENTS
FILE WRITES
NETWORK EVENTS
eBPF events
eBPF events
eBPF events
PROCESS NET EVENTS
DECODE EVENTS
tracee events
PIPELINE BEGINS
optional sorting channel
should trace ?
sort events
optional caching channel
cache events
shouldProcessEvent ?
bufferDecoder
...
processEvents
deriveEvents
generate event(s)
sinkEvents
parseArgs
chan events
aqua tracee

FACTS TO CONSIDER

Facts to Consider

1. Old Kernels support (lack of eBPF enhancements)
2. eBPF program types and overhead
3. Doubled data copies over the pipeline
4. Resource consumption thresholds
5. Runtime vs Frequency:
    1. Number of probes
    2. Frequency probes are fired
    3. How complex the probe handler is

Kernel Overhead – Runtime x Frequency

- Hook overhead = logic runtime * hook frequency
- In eBPF: we can measure overhead with bpftool prog show
- Average runtime = run_time_ns/run_cnt
- From experimentation: run_time_ns includes tail calls.

PERF DEMO

Kernel Overhead - Sys Enter ⇔ Sys Exit

- Tracee can track all running syscalls.
- Hooking sys_enter and sys_exit allows a view of all syscalls, but they are very frequent calls.
- Initializing an event was done for every enter/exit even if the syscall it's not needed.
- Other events require information on preceding syscalls.
- Solution - Split the logic up to tail calls by need.
- Result: Significant overhead decrease.
- Caveat: Some events must declare some syscalls as tail call dependencies.

aqua

eBPF support, available helpers and overhead

- TC eBPF programs need interface attaching/detaching
- v5.2 cgroup eBPF programs **LACK OF SUPPORT**:
  - cgroup/sock_release bpf_get_socket_cookie
  - generic eBPF helpers (uid) bpf_sk_storage_get
- v5.3: added bounded loops support
- v5.5: added eBPF trampoline support (calling convention JIT)
  - Fentry & Fexit (less overhead)
- v5.7: LSM
  - Flow control (block access instead of killing PIDs)
  - Avoid TOCTOU
- v5.9: socket lookup hook

Example of eBPF needed tricks to support older kernels

```c
//
// Socket Ingress/Egress eBPF program loader (right before and right after
//

SEC("kprobe/__cgroup_bpf_run_filter_skb")
int BPF_KPROBE(cgroup_bpf_run_filter_skb)
{
    // runs BEFORE the CGROUP/SKB eBPF program

    event_data_t data = {};

    if (!init_event_data(&data, ctx))
        return 0;

    // Both ingress & egress (mostly ingress) might run from a kthread and
    // need processing. Instead of "should_trace", check if current socket
    // a socket being traced or not.

    struct sock *sk = (void *) PT_REGS_PARM1(ctx);
    struct sk_buff *skb = (void *) PT_REGS_PARM2(ctx);
    int type = PT_REGS_PARM3(ctx);

    // obtain socket inode
    u64 inode = BPF_READ(sk, sk_socket, file, f_inode, i_ino);
    if (inode == 0)
        return 0;

    switch (type) {
        case BPF_CGROUP_INET_INGRESS:
        case BPF_CGROUP_INET_EGRESS:
            break;
        default:
            return 0; // wrong attachment type, return fast
    }

    // save args for kretprobe
    struct entry entry = {0};
    entry.args[0] = PT_REGS_PARM1(ctx); // struct sock *sk
    entry.args[1] = PT_REGS_PARM2(ctx); // struct sk_buff *skb

    // prepare for kretprobe using entrymap
    u32 pid = data.context.task.host_pid;
    bpf_map_update_elem(&entrymap, &pid, &entry, BPF_ANY);

    // NOTE: The net task context is the event context's "task context" wh
```

```c
SEC("kretprobe/__cgroup_bpf_run_filter_skb")
int BPF_KRETPROBE(ret_cgroup_bpf_run_filter_skb)
{
    // runs AFTER the CGROUP/SKB eBPF program

    event_data_t data = {};
    if (!init_event_data(&data, ctx))
        return 0;

    // Both ingress & egress (mostly ingress) might run from a kthread and
    // need processing. Instead of "should_trace" here, check if there is a
    // current skb timestamp entry (added by the entry) and delete it.

    // pick from entry from entrymap
    u32 pid = data.context.task.host_pid;
    struct entry *entry = bpf_map_lookup_elem(&entrymap, &pid);
    if (!entry) // no entry == no tracing
        return 0;

    // pick args from entry point's entry
    // struct sock *sk = (void *) entry->args[0];
    struct sk_buff *skb = (void *) entry->args[1];

    // cleanup entrymap
    bpf_map_delete_elem(&entrymap, &pid);

    // use skb timestamp as the key for cgroup/skb
    u64 skbts = BPF_READ(skb, tstamp);

    // only continue if netctx exists
    net_event_context_t *netctx = bpf_map_lookup_elem(&cgrpctxmap, &skbts);
    if (!netctx)
        return 0;

    // delete netctx after cgroup ebpf program runs
    bpf_map_delete_elem(&cgrpctxmap, &skbts);

    return 0;
}
```

Example of eBPF needed tricks to support older kernels

```
// SKB eBPF programs
//

static __always_inline u32 cgroup_skb_generic(struct __sk_buff *ctx)
{
    // IMPORTANT: runs for EVERY packet of tasks belonging to root cgroup

    u64 skbts = ctx->tstamp; // use skb timestamp as key for cgroup/skb program

    net_event_context_t *neteventctx = bpf_map_lookup_elem(&cgrpctxmap, &skbts);
    if (!neteventctx)
        return 1;

    struct bpf_sock *sk = ctx->sk;
    if (!sk)
        return 1;

    sk = bpf_sk_fullsock(sk);
    if (!sk)
        return 1;

    nethdrs hdrs = {0}, *nethdrs = &hdrs;

    return CGROUP_SKB_HANDLE(family);
}
```

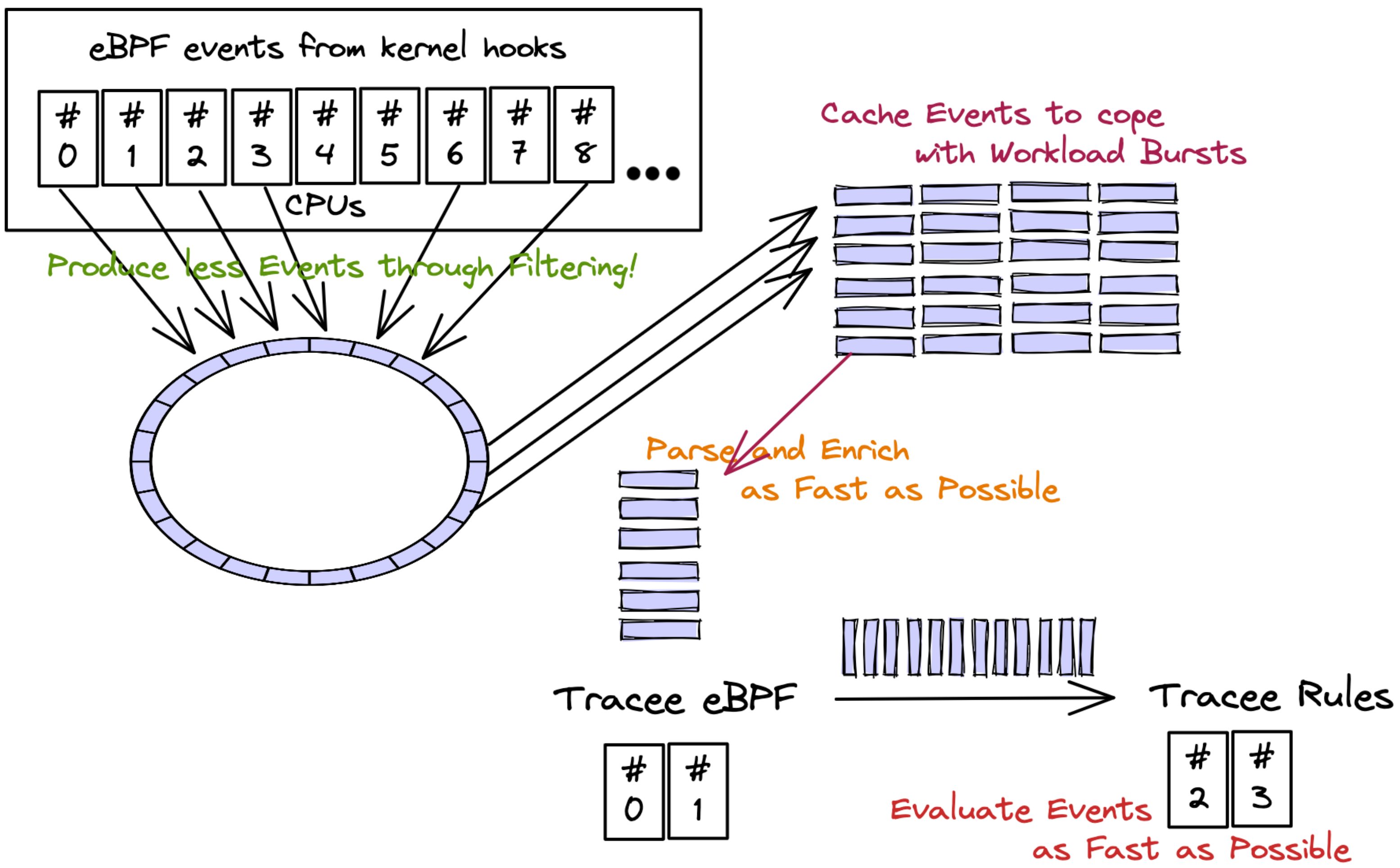
THE PIPELINE

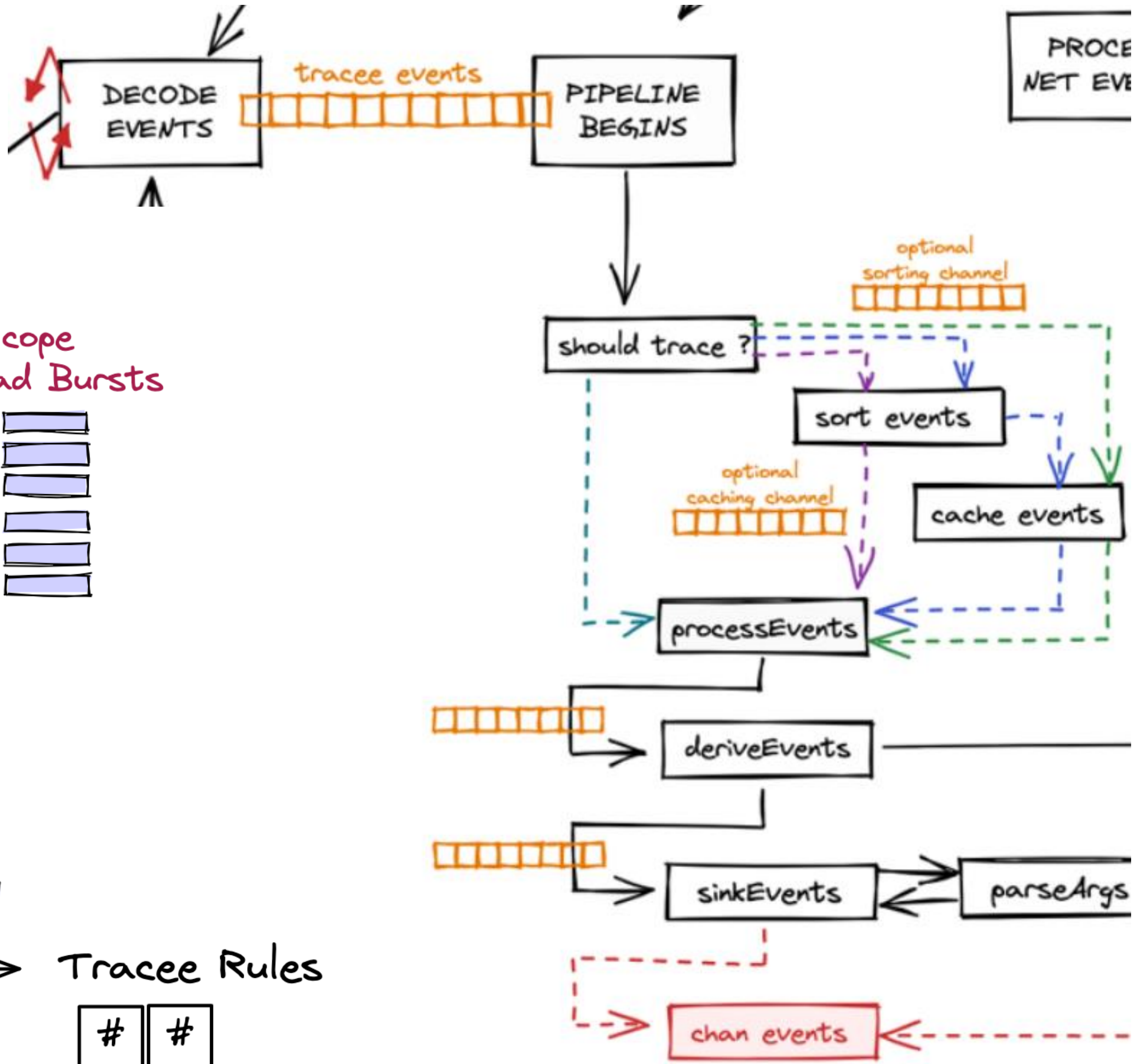
Concept: Producer:Consumer ratio

- Definition: the ratio of maximal sustained theoretical throughput between a producer and a consumer.
- If the ratio is < 1, expect lost events due to a bottleneck.
- Note: Filtering makes measuring tricky because the producer might not reach the consumer's maximum throughput.
- Note: Internally we use pprof flame graphs correlated with event loss ratio and event throughput rates to gauge improvements. As such this ratio is more of an intuitive tool.

aqua

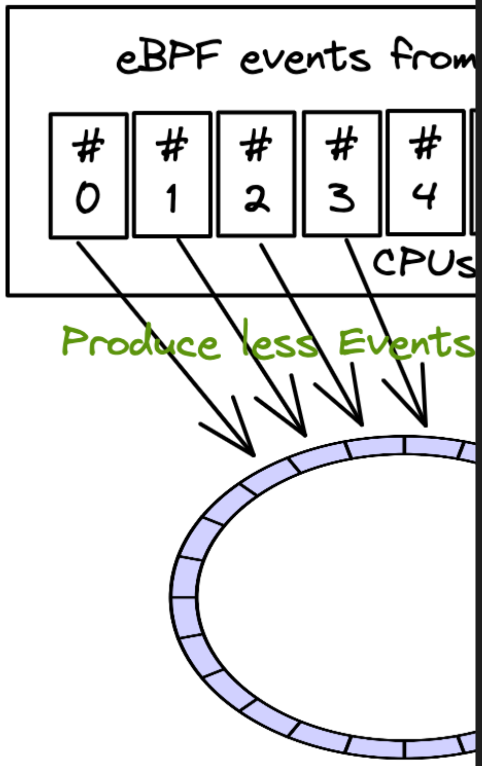
Decoding, Sorting and Caching Events






```
Select different cache types for the event pipeline queueing.
Possible options:
cache-type={none,mem}                                 pick the appropriate cache type.
mem-cache-size=256                                    set memory cache size in MB. only works for cache-type=mem.
Example:
  --cache cache-type=mem                              | will cache events in memory using default values.
  --cache cache-type=mem --cache mem-cache-size=1024  | will cache events in memory. will set memory cache size to 1024 MB.
  --cache none                                        | no event caching in the pipeline (default).
Use this flag multiple times to choose multiple output options
```

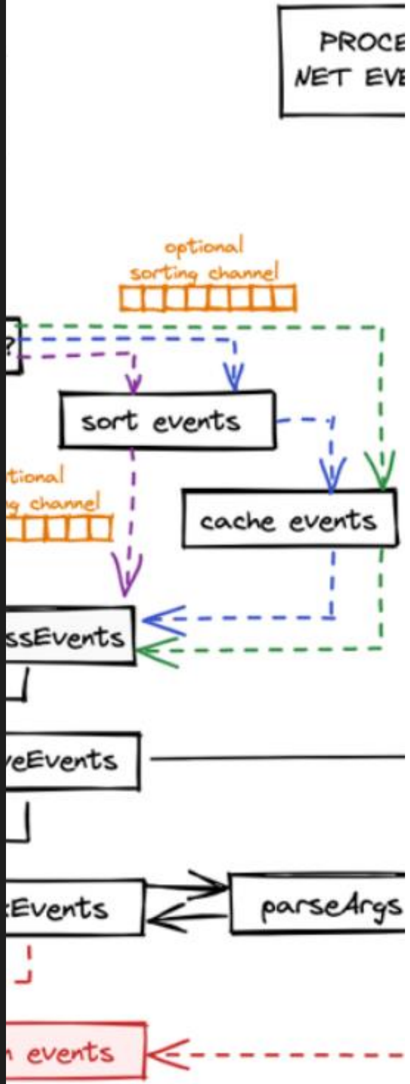
Decodi
and Cac




```go
// Under some circumstances, tracee-rules might be slower to consume events
// than tracee-ebpf is capable of generating them. This requires
// tracee-ebpf to deal with this possible lag, but, at the same,
// perf-buffer consumption can't be left behind (or important events coming
// from the kernel might be loss, causing detection misses).
//
// There are 3 variables connected to this issue:
//
// 1) perf buffer could be increased to hold very big amount of memory
//    pages: The problem with this approach is that the requested space,
//    to perf-buffer, through libbpf, has to be contiguous and it is almost
//    impossible to get very big contiguous allocations through mmap after
//    a node is running for some time.
//
// 2) raising the events channel buffer to hold a very big amount of
//    events: The problem with this approach is that the overhead of
//    dealing with that amount of buffers, in a golang channel, causes
//    event losses as well. It means this is not enough to relief the
//    pressure from kernel events into perf-buffer.
//
// 3) create an internal, to tracee-ebpf, buffer based on the node size.

// queueEvents implements an internal FIFO queue for caching events
func (t *Tracee) queueEvents(ctx context.Context, in <-chan *trace.Event) (chan
    out := make(chan *trace.Event, 10000)
    errc := make(chan error, 1)
    done := make(chan struct{}, 1)

    // receive and cache events (release pressure in the pipeline)
    go func() {
        for {
            select {
            case <-ctx.Done():
                done <- struct{}{}
                return
            case event := <-in:
                if event != nil {
                    t.config.Cache.Enqueue(event) // may block if queue is full
                }
            }
        }
    }()
```



Filtering – Reducing Throughput

- Critical step – reduces further processing down the pipeline and rules engine
- Ideally done as early as possible
- Currently done in two steps:

Kernel (eBPF)

- Less events submitted
- Less kernel time overhead
- More complex to implement
- Used for global context

Userspace

- Saves time for event polling
- Easier to implement new complex logic
- Used for local event context (args and struct)

Filtering – Userland Implementation

- Userland filtering reduces the producer:consumer ratio between the event pipeline and the perf buffer.
- Currently supporting per event context filtering, return value filters and argument filtering.

```go
type Event struct {
	Timestamp           int          `json:"timestamp"`
	ThreadStartTime     int          `json:"threadStartTime"`
	ProcessorID         int          `json:"processorId"`
	ProcessID           int          `json:"processId"`
	CgroupID            uint         `json:"cgroupId"`
	ThreadID            int          `json:"threadId"`
	ParentProcessID     int          `json:"parentProcessId"`
	HostProcessID       int          `json:"hostProcessId"`
	HostThreadID        int          `json:"hostThreadId"`
	HostParentProcessID int          `json:"hostParentProcessId"`
	UserID              int          `json:"userId"`
	MountNS             int          `json:"mountNamespace"`
	PIDNS               int          `json:"pidNamespace"`
	ProcessName         string       `json:"processName"`
	HostName            string       `json:"hostName"`
	ContainerID         string       `json:"containerId"`
	ContainerImage      string       `json:"containerImage"`
	ContainerName       string       `json:"containerName"`
	PodName             string       `json:"podName"`
	PodNamespace        string       `json:"podNamespace"`
	PodUID              string       `json:"podUID"`
	EventID             int          `json:"eventId,string"`
	EventName           string       `json:"eventName"`
	ArgsNum             int          `json:"argsNum"`
	ReturnValue         int          `json:"returnValue"`
	StackAddresses      []uint64     `json:"stackAddresses"`
	ContextFlags        ContextFlags `json:"contextFlags"`
	Args                []Argument   `json:"args"` //Arguments are ordered according their appearance in the original event
}
```

Filtering – Userspace Implementation

Key requirements (argument filtering):

1. Performance – Filtering cost < Filtering benefit.
2. Support all argument types – bypassed through string conversion. Potential Solution: Hardcoding argument types and using Go 1.18 generics.
3. For strings – support equality, prefixes, suffixes and contain inputs (possibly regex in the future).

Filtering – Userspace optimization

```go
func matchFilter(filters []string, argValStr string) bool {
    for _, f := range filters {
        prefixCheck := f[len(f)-1] == '*'
        if prefixCheck {
            f = f[0 : len(f)-1]
        }
        suffixCheck := f[0] == '*'
        if suffixCheck {
            f = f[1:]
        }
        if argValStr == f ||
            (prefixCheck && !suffixCheck && strings.HasPrefix(argValStr, f)) ||
            (suffixCheck && !prefixCheck && strings.HasSuffix(argValStr, f)) ||
            (prefixCheck && suffixCheck && strings.Contains(argValStr, f)) {
            return true
        }
    }
    return false
}
```

```go
valLen := len(val)
if set.minLen == math.MaxInt || valLen < set.minLen {
    return false
}

for _, prefixLen := range set.lengths {
    if valLen < prefixLen {
        return false
    }

    check := val[0:prefixLen]
    if set.Set[check] {
        return true
    }
}
return false
```

```go
if f.equal[val] {
    return true
}
if suffixes.Filter(val) {
    return true
}
if prefixes.Filter(val) {
    return true
}
for contain := range contains {
    if strings.Contains(val, contain) {
        return true
    }
}
```

Filtering – Current limitations and Potential Improvements

Only one global filtering scope

- Can't create parallel conflicting filter scopes.

Limited argument type support

- Non ideal method for maps and struct filtering.
- However, userland implementation allow an easier extension.

Filtering happens before derivation

- This means derived event cannot be filtered easily.

Userland < Kernel Performance

- Filtering in kernel could massively reduce initial throughput.
- Newer kernels might enable full implementation.
- Need to explore in older kernels.

aqua

Event Processing

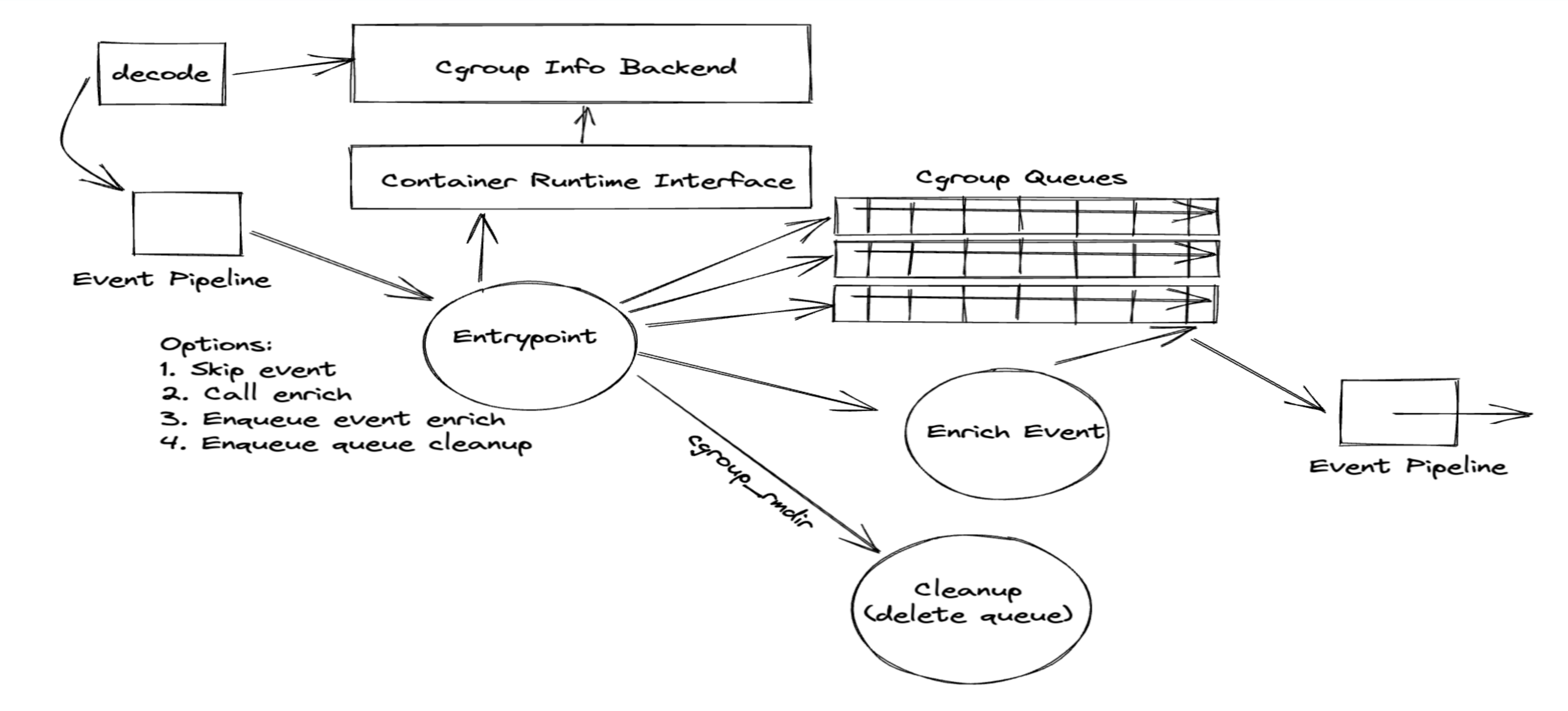
Event Processing Categories			
Logic Hooks • Early in pipeline • Performance impact is workload dependent • Example: cgroup parsing	**Event Enrichment** • Complexity of enrichment varies • Simple: Process Info for network events • Complex: Querying Container Data	**Event Derivation** • Create new events in the pipeline • Suspect to a lost event • Finish up kernel logic in userspace • Example: Container Created event	**Argument Parsing** • Make event arguments user readable • Technically optional but practically required

aqua

Container Enrichment - Challenges

1. Container detection is based on cgroup paths – parsing is pattern based.
2. Correlating container id with image and name is practically impossible in the kernel.
3. Detecting runtimes on a system, multiple runtimes and nested runtimes may exist.
4. Each container runtime has it's own quirks.
5. Container runtime interface requests are synchronous, our pipeline is not. (note: event interfaces may not exist, and do not include container annotations which we use for kubernetes awareness).

Container Enrichment - Non Blocking Architecture



Container Enrichment – Possible Alternatives

eBPF uprobes

- Possibly the most efficient way
- Challenges:
  1. Daemons are written in go – plan9 calling conventions
  2. Parsing golang structs is less trivial than C
  3. Golang doesn't play nice with uprobes sometimes*

Merging with event APIs

- May reduce overhead of runtime requests
- Not every runtime has an event API (cri-o)

*For more info on golang and eBPF uprobes see this article:
https://medium.com/bumble-tech/bpf-and-go-modern-forms-of-introspection-in-linux-6b9802682223

Derivation (Possible Delays): Symbols Loading

```
./dist/tracee-ebpf \
    --trace=symbols_loaded \
    --trace=symbols_loaded.symbols=fopen \
    --trace=symbols_loaded.library_path!=libc
```

Shared Object Loaded (Security Mmap File)

eBPF event

Symbols Loaded

Userland Derived Event

Library Name

Host Symbols Loader

Container Symbols Loader

Path Resolver (Cont. to OS)

Load SO Symbols (ELF)

LRU map of all OS libraries loaded, and imported/exported symbols

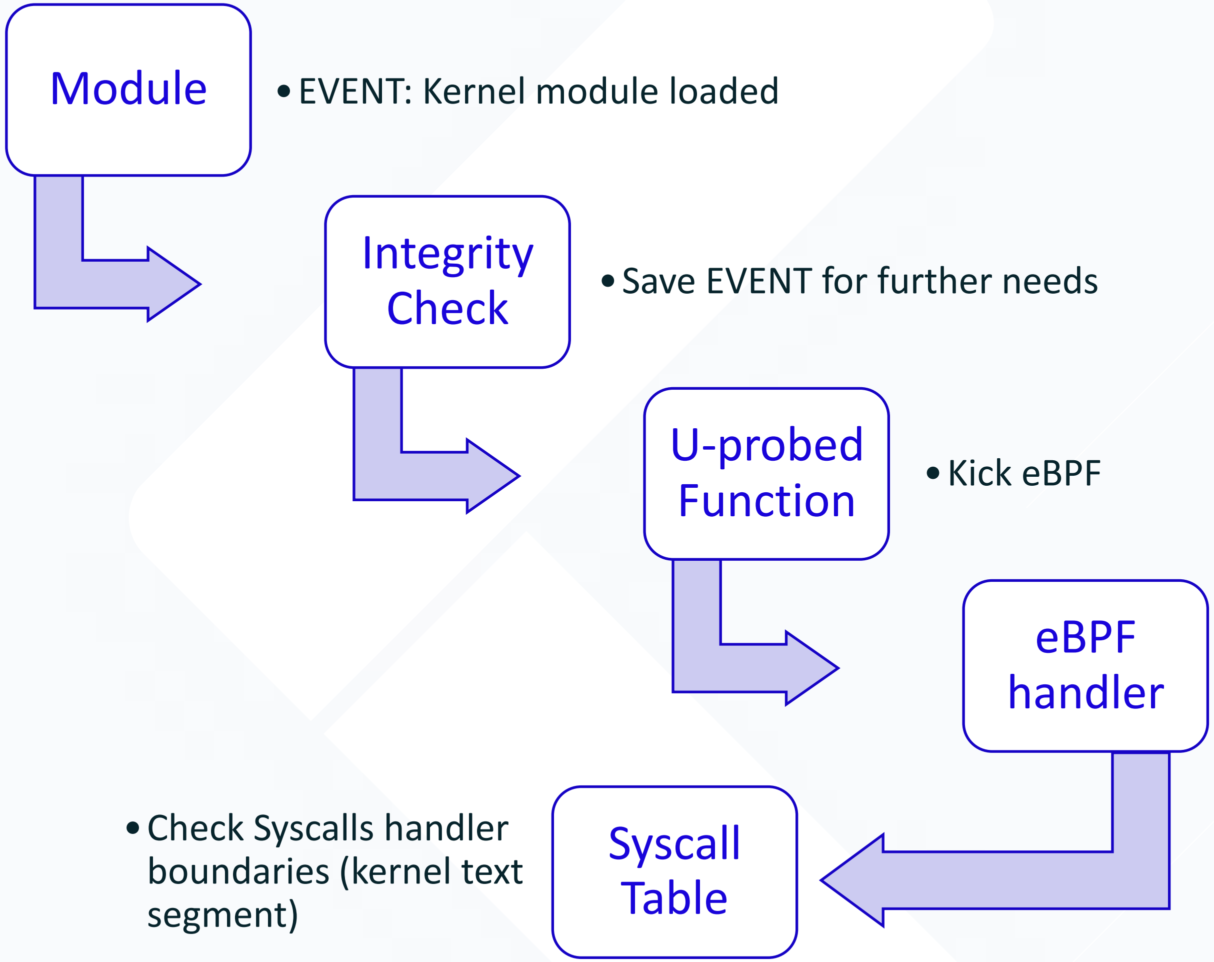
```
// try to access the root fs via another process in the same mount namespace
pids := cPathRes.mountNSPIDsCache.GetBucket(uint32(mountNS))
for _, pid := range pids {
    procRootPath := fmt.Sprintf("/proc/%d/root", int(pid))
    // fs.FS interface requires relative paths, so the '/' prefix should be
    _, err := fs.Stat(cPathRes.fs, strings.TrimPrefix(procRootPath, "/"))
    if err == nil {
```

aqua

Feature Credits: Alon Zivoni (Aqua Research)

Derivation (Context Savings): Kernel Hook Detections

- **Syscalls Table**




```go
// in case FinitModule and InitModule occurs it means that a
// and we will want to check if it hooked the syscall table
case events.DoInitModule:
	_, ok1 := t.events[events.HookedSyscalls]
	_, ok2 := t.events[events.HookedSeqOps]
	_, ok3 := t.events[events.HookedProcFops]
	if ok1 || ok2 || ok3 {
		err := t.updateKallsyms()
		if err != nil {
			return err
		}
		t.triggerSyscallsIntegrityCheck(*event)
		t.triggerSeqOpsIntegrityCheck(*event)
	}
```

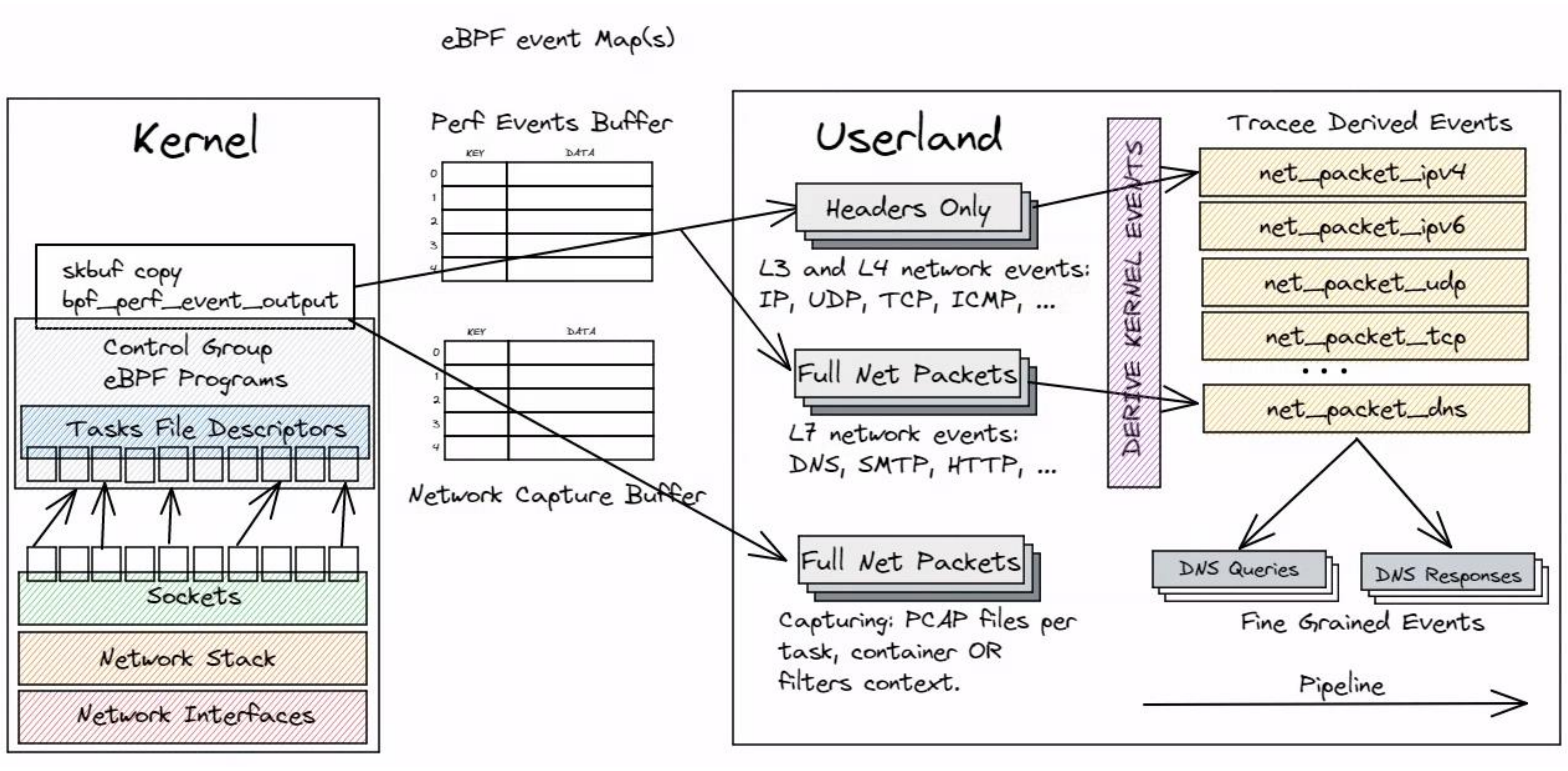
```go
// triggerSyscallsIntegrityCheck is used by a Uprobe to trigger an eBPF program
// that prints the syscall table
func (t *Tracee) triggerSyscallsIntegrityCheck(event trace.Event) {
	_, ok := t.events[events.HookedSyscalls]
	if !ok {
		return
	}
	eventHandle := t.triggerContexts.Store(event)
	t.triggerSyscallsIntegrityCheckCall(
		uProbeMagicNumber,
		uint64(eventHandle),
	)
}

//go:noinline
func (t *Tracee) triggerSyscallsIntegrityCheckCall(
	magicNumber uint64, // 1st arg: allow handler to detect calling convention
	eventHandle uint64) {
}
```

- Same for “**SeqNetOps**” and “**Proc File Operations**” hooks

Feature Credits: Asaf Eitani & Itamar Maouda (Aqua Research)

Derivation (Raw Data Translation): Networking



Derivation (Raw Data Translation): Networking

```
//
// Network Protocol Event Types
//
NetPacketBase: {
    ID32Bit:  sys32undefined,
    Name:     "net_packet_base",
    Internal: true,
    Dependencies: dependencies{
        Capabilities: []cap.Value{cap.NET_ADMIN},
    },
    Probes: []probeDependency{
        {Handle: probes.CgroupSKBIngress, Required: true},
        {Handle: probes.CgroupSKBEgress, Required: true},
        {Handle: probes.SockAllocFile, Required: true},
        {Handle: probes.SockAllocFileRet, Required: true},
        {Handle: probes.CgroupBPFRunFilterSKB, Required: true},
        {Handle: probes.CgroupBPFRunFilterSKBRet, Required: true},
    },
    Sets:   []string{"network_events"},
    Params: []trace.ArgMeta{},
},
NetPacketIPBase: {
    ID32Bit:  sys32undefined,
    Name:     "net_packet_ip_base",
    Internal: true,
    Dependencies: dependencies{
        Events: []eventDependency{
            {EventID: NetPacketBase},
        },
    },
    Sets: []string{"network_events"},
    Params: []trace.ArgMeta{
        {Type: "bytes", Name: "payload"},
    },
},
NetPacketIPv4: {
    ID32Bit: sys32undefined,
    Name:    "net_packet_ipv4",
    Dependencies: dependencies{
        Events: []eventDependency{
            {EventID: NetPacketIPBase},
        },
    },
    Sets: []string{"network_events"},
    Params: []trace.ArgMeta{
        {Type: "const char*", Name: "src"},
        {Type: "const char*", Name: "dst"},
        {Type: "trace.ProtoIPv4", Name: "proto_ipv4"},
    },
},
NetPacketIPv6: {
    ID32Bit: sys32undefined,
```

```
func deriveNetPacketIPv4Args() deriveArgsFunction {
    return func(event trace.Event) ([]interface{}, error) {
        var ok bool
        var payload []byte

        // initial header type

        if event.ReturnValue != 2 { // AF_INET
            return nil, nil
        }

        // sanity checks

        payloadArg := events.GetArg(&event, "payload")
        if payloadArg == nil {
            return nil, noPayloadError()
        }
        if payload, ok = payloadArg.Value.([]byte); !ok {
            return nil, nonByteArgError()
        }
        payloadSize := len(payload)
        if payloadSize < 1 {
            return nil, emptyPayloadError()
        }

        // parse packet

        packet := gopacket.NewPacket(
            payload[4:payloadSize], // base event argument is: |sizeof|[]byte|
            layers.LayerTypeIPv4,
            gopacket.Default,
        )

        if packet == nil {
            return []interface{}{}, parsePacketError()
        }

        layer3 := packet.NetworkLayer()

        switch l3 := layer3.(type) {
        case (*layers.IPv4):
            var ipv4 trace.ProtoIPv4
            copyIPv4ToProtoIPv4(l3, &ipv4)

            return []interface{}{
                l3.SrcIP.String(),
                l3.DstIP.String(),
                ipv4,
            }, nil
        }

        return nil, notProtoPacketError("IPv4")
    }
}
```

Encoding/Decoding

- Tracee uses a named pipe to deliver events to the rule's engine.
- Byte encoding is required for this operation.
- Encoding and Decoding is the MOST significant bottleneck in tracee as it affects the producer:consumer ratio of all critical parts.
- Current encoding: GOB.

Encoding/Decoding - Alternatives

- Practically tracee has only one requirement for its encoding method: Type Safety across boundaries.
- Gob guarantees type safety but is not optimal enough.
- OOTB - most encodings don't keep the golang type info across boundaries without modifications.
- Current alternatives seem to be:
  1. Protocol Buffers – Code generation guarantees type info out of the box. Caveat: Argument types are all over the place and must be hardcoded.
  2. MessagePack – Almost works out of the box but loses type info (int32 -> int8). Might work with modifications to code generation.
  3. Flat Buffers – However map types do not have native support.
  4. Roll your own.
- Our best option currently is to move our event definition to the protocol buffer format, and hardcode our argument types, 1.18 generics seem to be a good option.

aqua

CONCLUSIONS

How to Solve or Address the Problem + Q/A

1. To use **different eBPF programs** (or set of programs) for older and newer kernels, by designed feature.
2. To select carefully **what to probe** and implement **fast-paths** on each hook in order to return as early as possible.
3. Multiple **parallel in-kernel filter scopes** are needed. Narrow each scope to max amount of event arguments to produce less events.
4. **Userland** pre-process, context-saving and filtering will **still have its place**.
5. Events consumer **can't block the pipeline**, no matter what: enqueue and defer (async) work.
6. **Keep types** over the pipeline (and in all event consumers). Protocol Buffers unmarshalling, keeping specific kernel types (int -> uint8).

Thank you! Questions ?
Reach us at:
https://slack.aquasec.com
#tracee , #libbpfgo, #btfhub, #ebpf

Rafael David Tinoco
Nadav Strahilevitz

@rafaeldtinoco
@NDStrahilevitz

Aqua Security