Protocol Tracing with eBPF

September 23, 2021

Omid Azizi, Yaxiong Zhao, Ryan Cheng, John P Stevenson, Zain Asgar

A CNCF sandbox project

CLOUD NATIVE COMPUTING FOUNDATION

file:///tmp/PXL_20210923_0447545
11.jpg

About Me

Hi, I'm Omid

Twitter: @oazizi
Principal engineer at New Relic.
Founding engineer at Pixie Labs (@pixie_run)

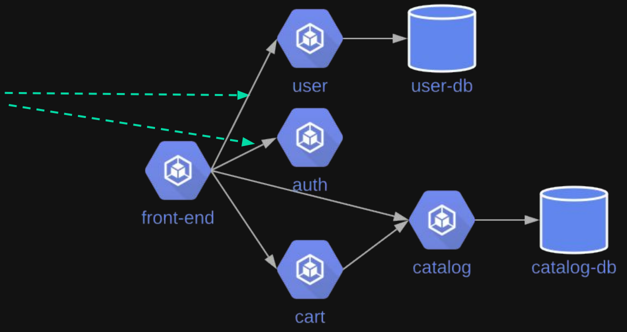
Introduction

In 2019, we set out to build a no-instrumentation observability platform.

- Our Vision: Help developers understand and debug their K8s apps.

First goal: Trace *application* network messages.

- HTTP, then other protocols.



We had two key requirements:

(1) **No instrumentation**: No code modifications, no redeployments.

(2) **Low overhead**: Always active.

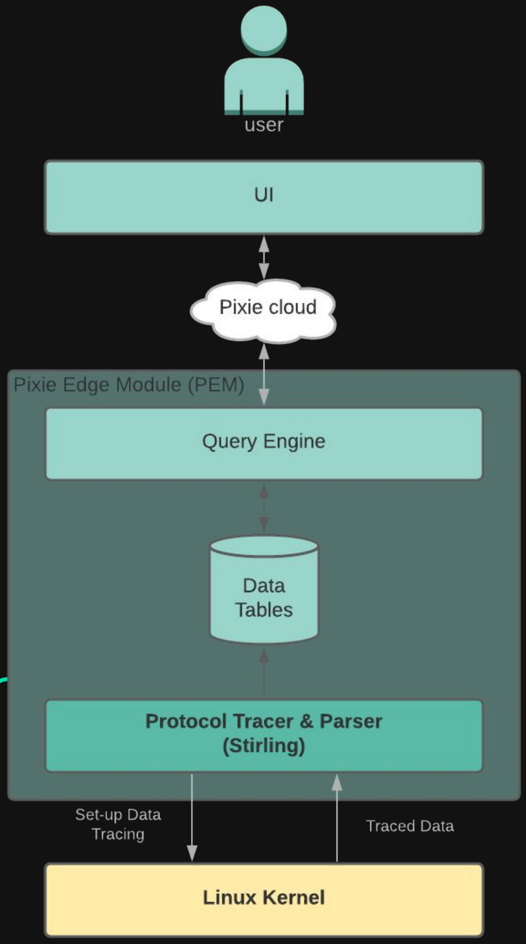
Overview

No instrumentation + low overhead ➪ **eBPF**.

General approach:

- Capture data in kernel-space with eBPF.
- Process data in user-space (protocol parsing).
- Store data into tables for querying by user.

Focus of this talk



https://work.withpixie.ai/live/clusters/gke_pl-pixies_us-west1-a_oazizi?max_num_records=1000&script=Scratch Pad&start_

PIXIE **Cluster:** gke:oazizi RUN

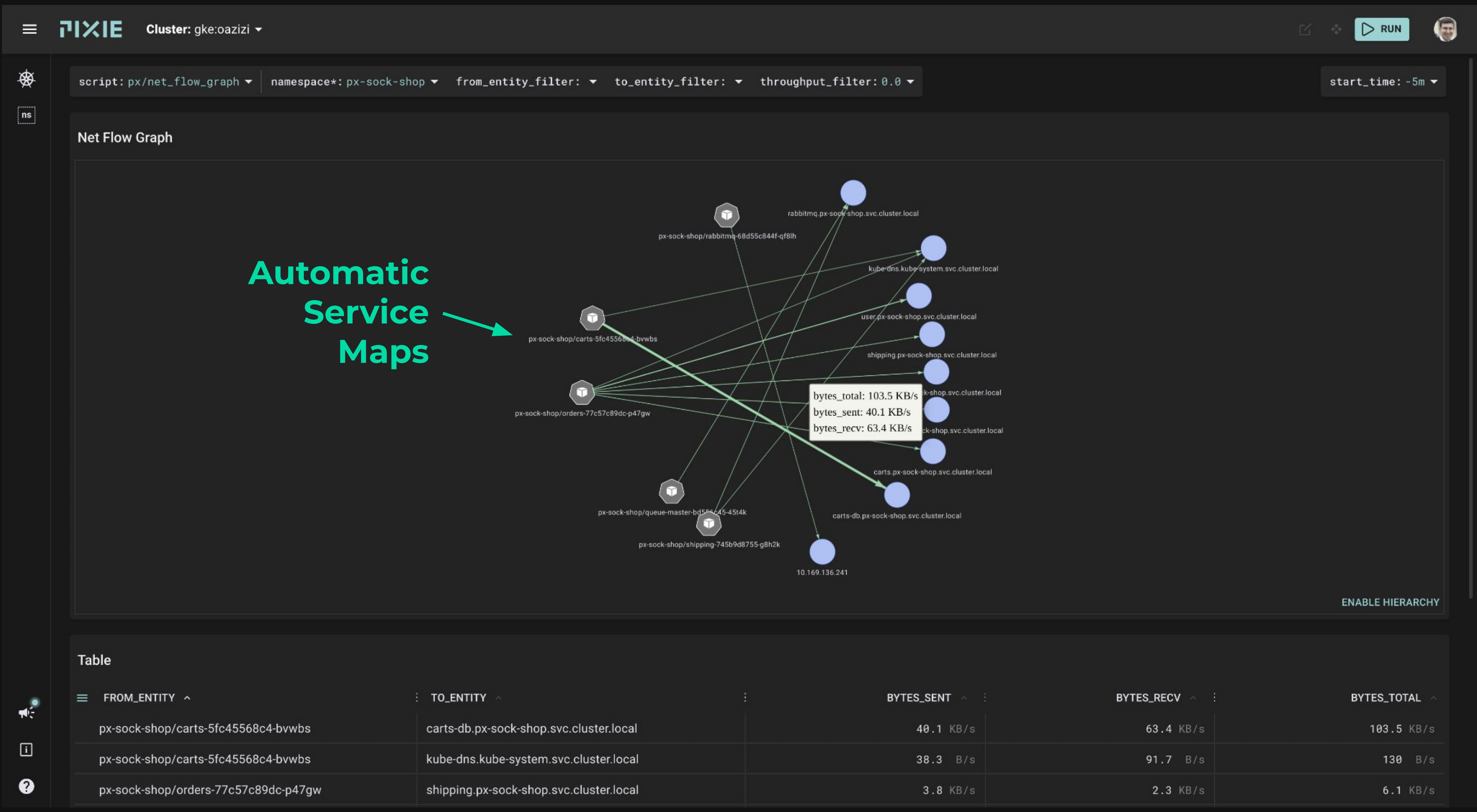
script: Scratch Pad | max_num_records: 1000 | start_time: -5m

Traced HTTP Traffic

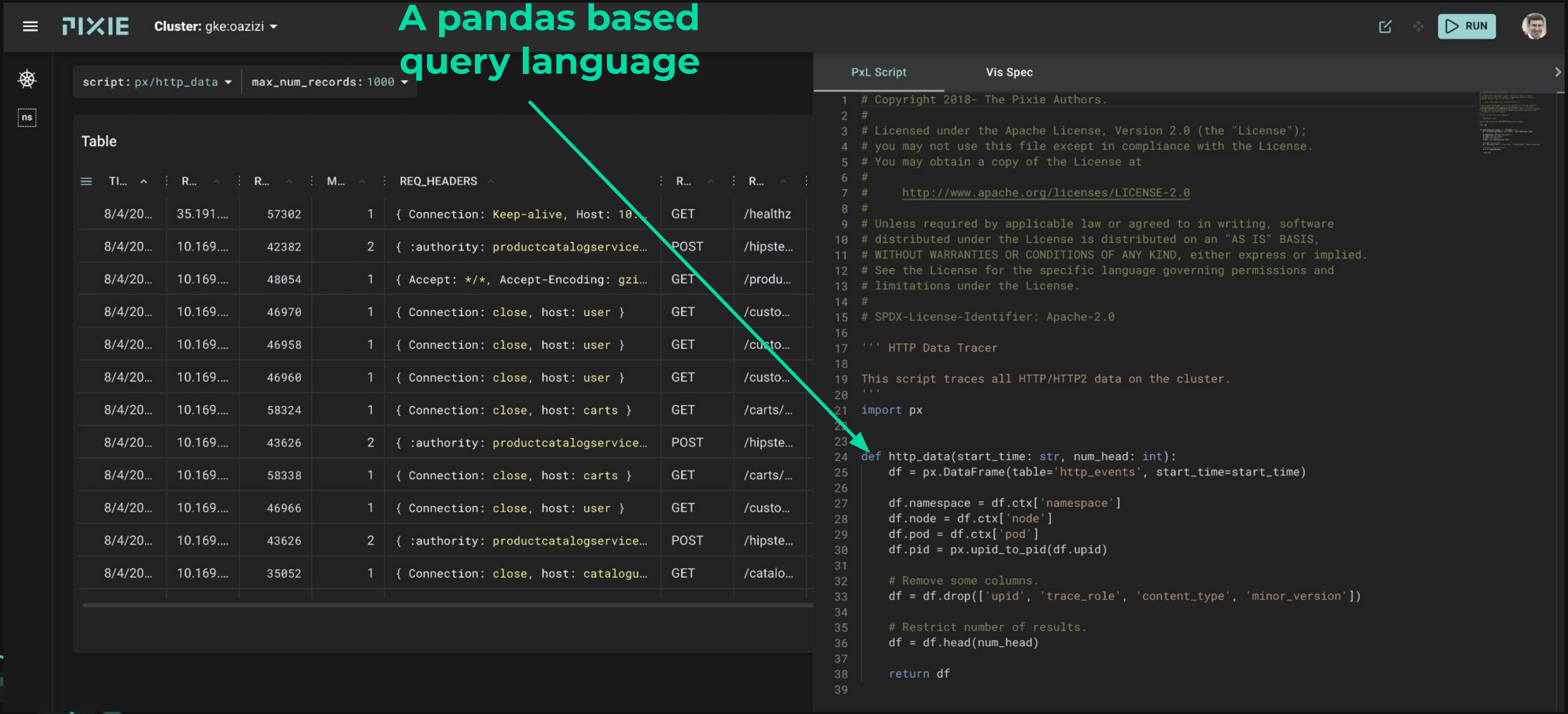
Table

TIME_	POD	REMOTE_ADDR	REQ_ME...	REQ_PATH	REQ_BODY	RESP_HEADERS	RESP...	RESP_BODY
9/22/2021, 3:58:18 PM	px-sock-shop/front-end-...	10.169.137.128	GET	/login		{ Connection: keep-...	200	Cookie is s
9/22/2021, 3:58:18 PM	px-sock-shop/carts-85bf...	10.169.137.125	GET	/carts/57a98d98e4b006...		{ Connection: close	202	<removed:
9/22/2021, 3:58:18 PM	px-sock-shop/payment-...	10.169.137.130	POST	/paymentAuth	{ address: { id: ...	{ Content-Length: 51	200	{ authori
9/22/2021, 3:58:18 PM	px-sock-shop/shipping-7...	10.169.137.130	POST	/shipping	{ id: d1fd39fa-87...	{ Content-Type: app...	201	{ id: d1f
9/22/2021, 3:58:18 PM	px-sock-shop/carts-85bf...	10.169.137.125	DELETE	/carts/57a98d98e4b006...		{ Connection: close	202	<removed:
9/22/2021, 3:58:18 PM	px-sock-shop/carts-85bf...	10.169.137.125	POST	/carts/57a98d98e4b006...	{ itemId: 808a2de...	{ Connection: close	201	{ id: 614
9/22/2021, 3:58:18 PM	px-sock-shop/orders-7c...	10.169.137.125	POST	/orders	{ customer: http:...	{ Connection: close	201	{"id":"614
9/22/2021, 3:58:18 PM	px-sock-shop/shipping-7...	10.169.137.130	POST	/shipping	{ id: 2bbfa8d4-97...	{ Content-Type: app...	201	{ id: 2bb
9/22/2021, 3:58:18 PM	px-sock-shop/orders-7c...	10.169.137.125	POST	/orders	{ customer: http:...	{ Connection: close	201	{"id":"614
9/22/2021, 3:58:18 PM	px-sock-shop/carts-85bf...	10.169.137.125	POST	/carts/57a98d98e4b006...	{ itemId: 03fef6a...	{ Connection: close	201	{ id: 614
9/22/2021, 3:58:18 PM	px-sock-shop/carts-85bf...	10.169.137.125	POST	/carts/57a98d98e4b006...	{ itemId: 808a2de...	{ Connection: close	201	{ id: 614
9/22/2021, 3:58:18 PM		35.191.10.207	GET	/healthz		{ Content-Length: 1...	200	{ lastUpd

Showing 68 - 81 / 1000 records


PIXIE
Cluster: gke:oazizi
RUN
ns
script: px/net_flow_graph
namespace*: px-sock-shop
from_entity_filter:
to_entity_filter:
throughput_filter: 0.0
start_time: -5m
Net Flow Graph
Automatic Service Maps
rabbitmq.px-sock-shop.svc.cluster.local
px-sock-shop/rabbitmq-68d55c844f-qf8lh
kube-dns.kube-system.svc.cluster.local
user.px-sock-shop.svc.cluster.local
shipping.px-sock-shop.svc.cluster.local
px-sock-shop/orders-77c57c89dc-p47gw
bytes_total: 103.5 KB/s
bytes_sent: 40.1 KB/s
bytes_recv: 63.4 KB/s
carts.px-sock-shop.svc.cluster.local
carts-db.px-sock-shop.svc.cluster.local
px-sock-shop/shipping-745b9d8755-g8h2k
10.169.136.241
ENABLE HIERARCHY
Table
FROM_ENTITY
TO_ENTITY
BYTES_SENT
BYTES_RECV
BYTES_TOTAL
px-sock-shop/carts-5fc45568c4-bvwbs
carts-db.px-sock-shop.svc.cluster.local
40.1 KB/s
63.4 KB/s
103.5 KB/s
px-sock-shop/carts-5fc45568c4-bvwbs
kube-dns.kube-system.svc.cluster.local
38.3 B/s
91.7 B/s
130 B/s
px-sock-shop/orders-77c57c89dc-p47gw
shipping.px-sock-shop.svc.cluster.local
3.8 KB/s
2.3 KB/s
6.1 KB/s

PxL scripts:
A pandas based query language

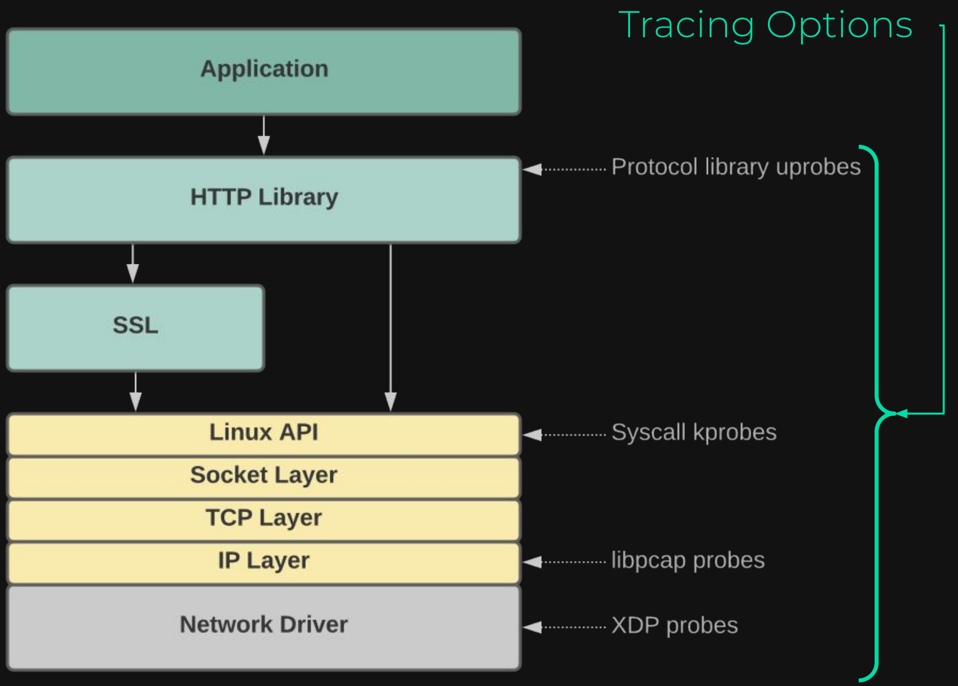


Building a Protocol Tracer

Where to Trace the Data?

Many options in the software stack:

We preferred tracing as close to the application layer as possible.



Approaches Compared

	protocol library uprobes	syscall kprobes	libpcap/XDP
Tracing overhead	Low	Low	Low
Scalability & Stability	Uprobes per library, Probe targets may change	High	High
Parsing effort	None	Protocol parsing	Packet processing & protocol parsing
SSL tracing	Cleartext available	Data encrypted	Data encrypted

We chose to use syscall kprobes on functions such as send() & recv().

- Rationale: close to the application layer, but stable API.

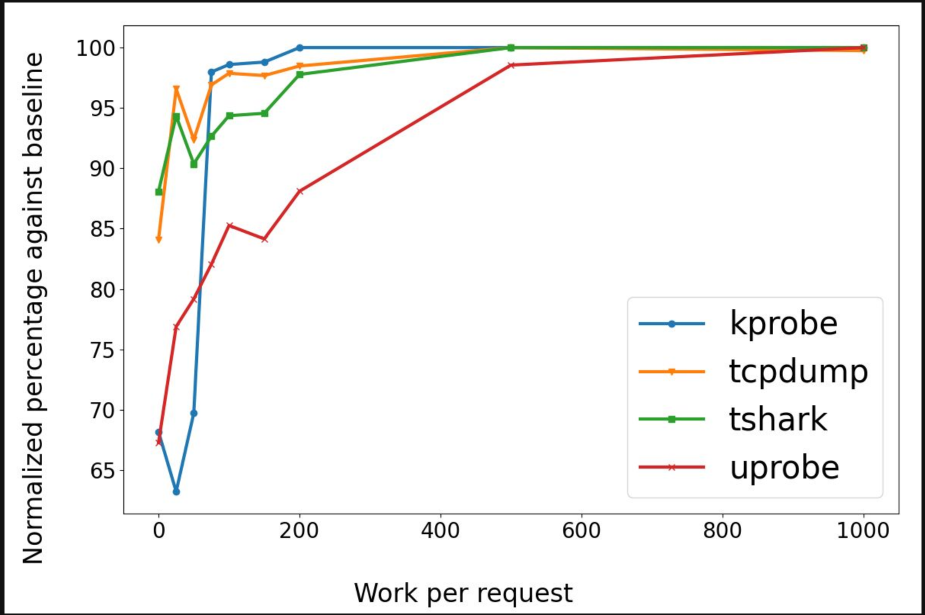
Performance Overhead

Production servers are typically in this range, since they do real work.

Study: Deploy probes on an HTTP server.

- X-axis: the amount of work performed by the per request.

Take-away: kprobe overhead < 2% overhead as long as server is not trivial.



Framework and Requirements

The Pixie data collector (Stirling) is written in C++

- Uses both BCC and BPFTrace for eBPF
- The protocol tracer uses BCC for the greater degree of control.

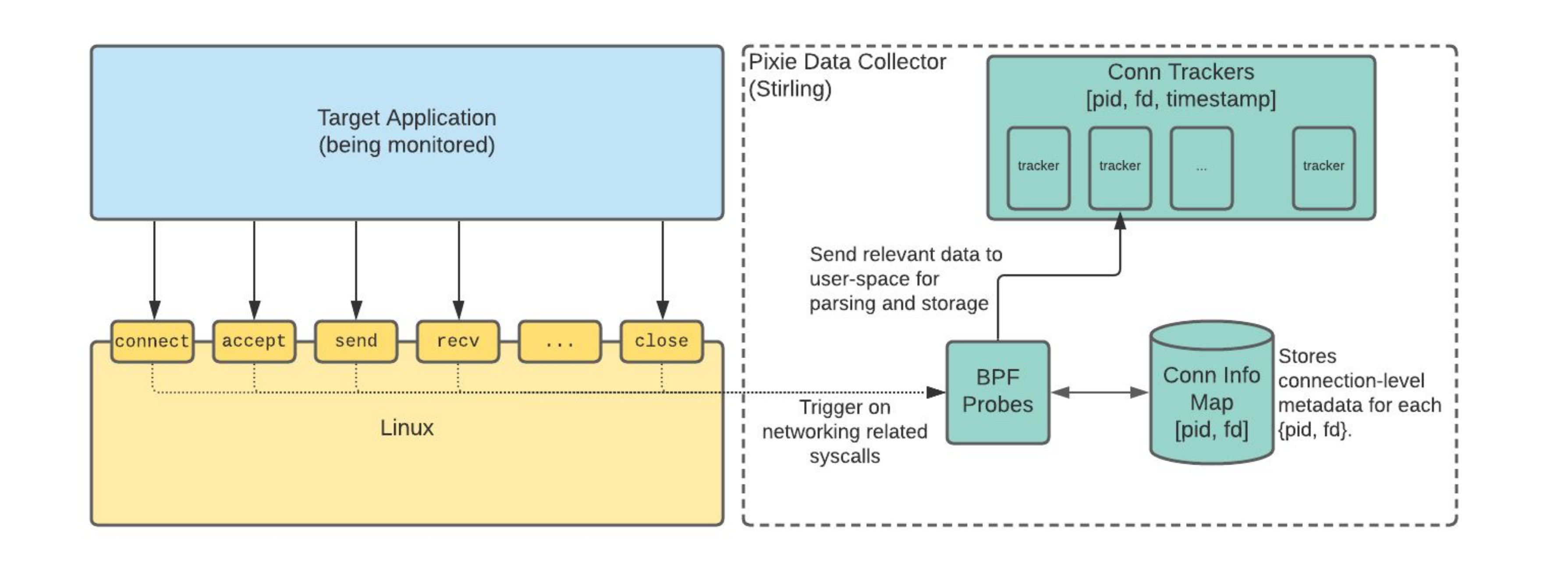
Requirements

- Need to support older kernels: we don't control the target ecosystem.
- Minimum kernel version supported: 4.14

Restrictions

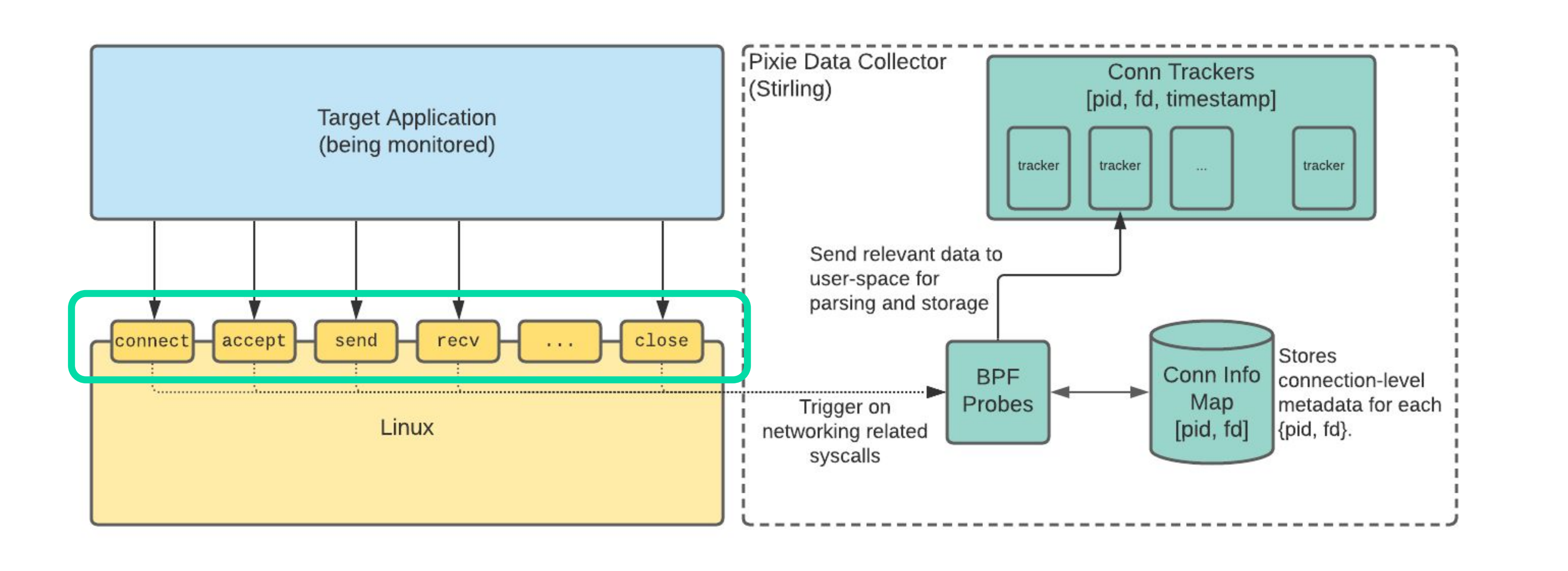
- 4096 instruction limit :(
- No ringbuf :(
- Really want to use libbpf + CO-RE..but we can't :(

Architecture



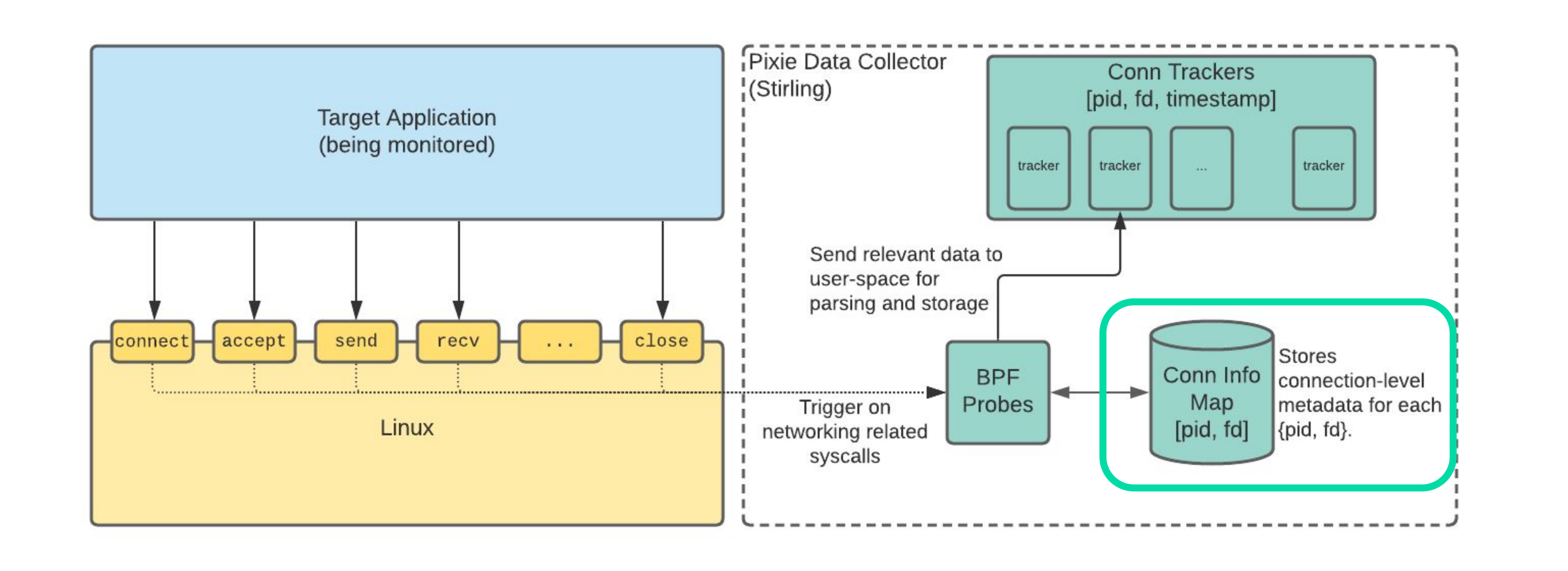
Architecture

1 - Setup probes on network related syscalls.



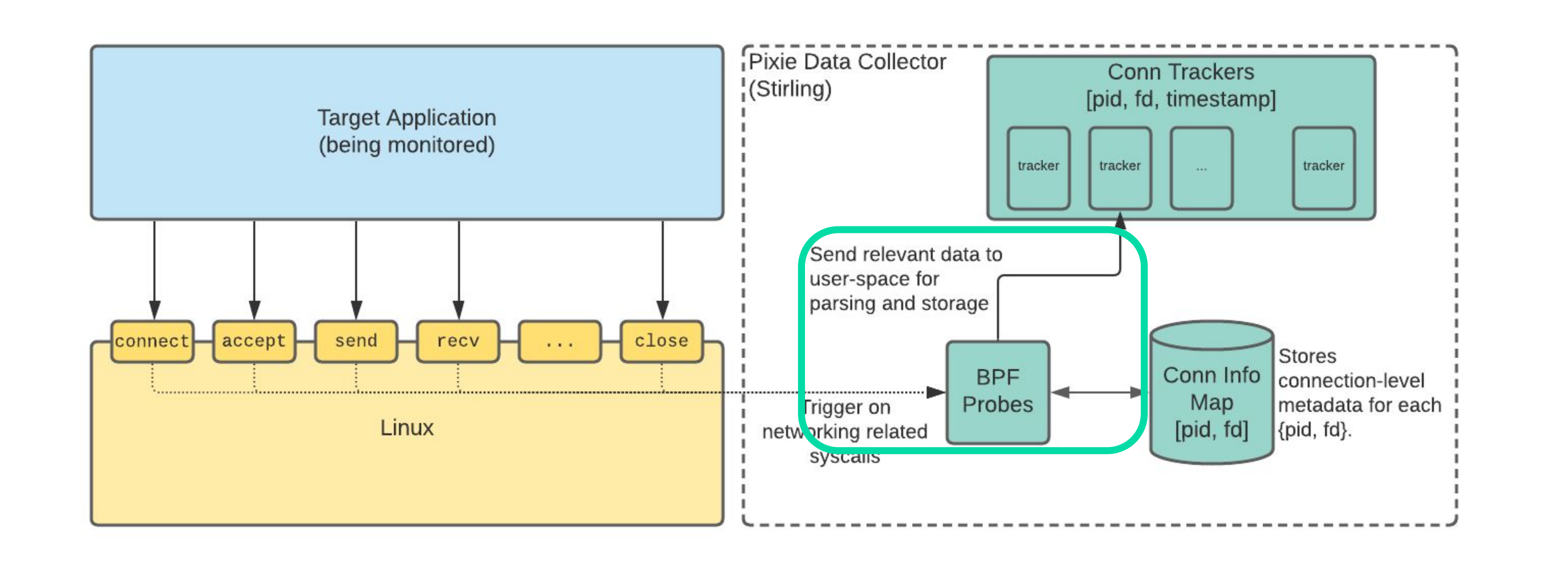
Architecture

2 - Record connection metadata in BPF maps.



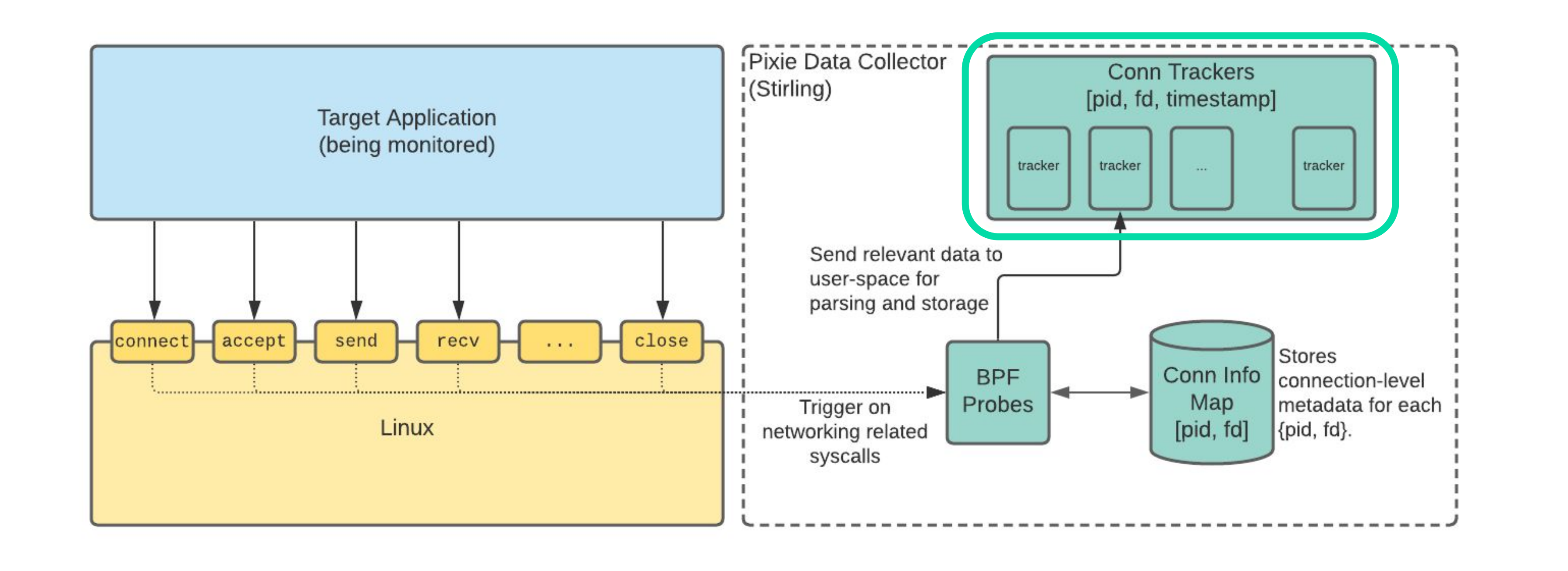
Architecture

**3 - Infer protocol with basic rule-based classification as a simple filter.
Transfer connection information and data through two perf buffers.**



Architecture

4 - Track connections in user-space with ConnTrackers.
Parse ConnTracker data into structured messages.



So, it all just works...right?

The general approach of tracing syscalls has some benefits

- Avoided the complexity of the network layer.
- Easy correlation of events to PID

But the approach is not without its challenges, including:

- Dealing with the variety of syscalls.
- Finding the remote endpoint address.
- Implementing protocol inference in eBPF.
- Dealing with stateful protocols (HTTP/2) and encrypted traffic (TLS).

Challenges of Tracing Syscalls

Tracing syscalls is a double-edged sword.

- Benefit: The stable API makes our probes portable across kernel versions.
- Con: Over the years, many ways of doing the same thing have evolved.
    - We have to account for all of them.

The protocol tracer probes a total of 17 Linux syscalls.

List of Syscalls

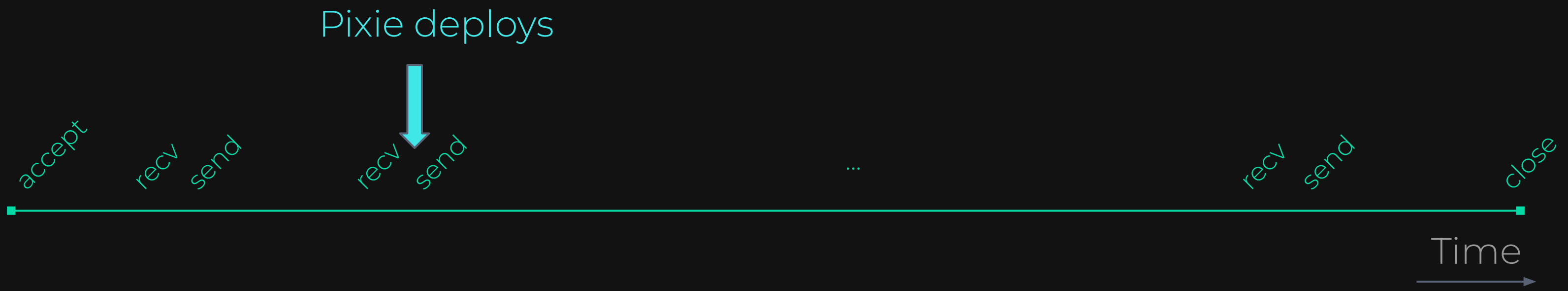
Connection management	Recv variants	Write variants	Special purpose
connect accept accept4 close	read readv recv recvfrom recvmsg recvmmsg	write writev send sendto sendmsg sendmmsg sendfile	sock_alloc sock_sendmsg sock_recvmsg

Challenges of Tracing Syscalls: Examples

Example	Problem	Our Solution
read & **write** syscalls are used for both file I/O and sockets.	When we trace these syscalls, we end up with more than network traffic.	Trace **sock_sendmsg** & **sock_recvmsg** to select only the socket traffic.
accept may be called with a NULL addr argument.	When NULL, the remote endpoint address is not directly accessible.	Trace internal **sock_alloc** calls to figure out missing address.
Variants like **sendmsg** & **recvmsg** have multiple data chunks.	BPF doesn't support loops.	Unrolled loop over a bounded number of chunks (45). Lose data beyond that.

Challenges of Tracing Mid-Stream

As an observability tool, we may not see the entire connection stream.



Problem for long-lived streams: we won’t know the remote endpoint.

- So we resolve endpoints from user-space.

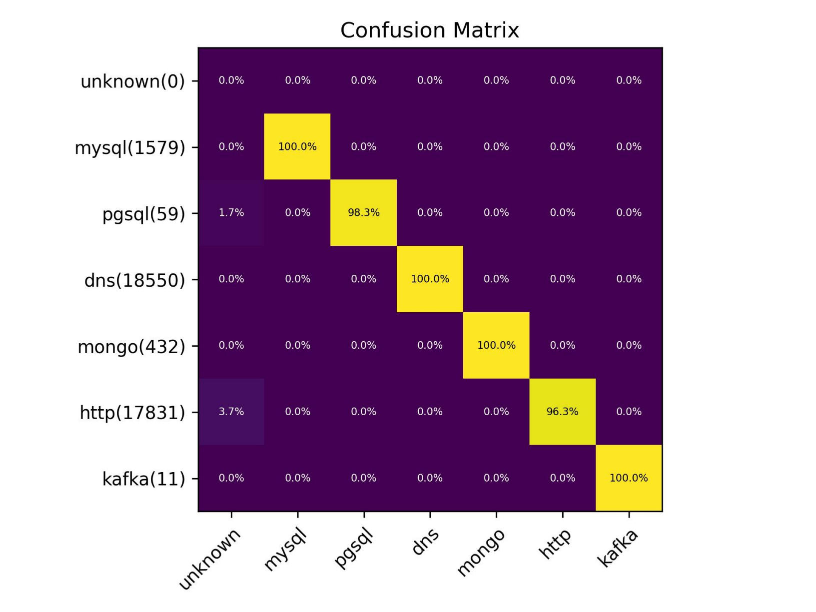
eBPF-Side Protocol Inference

To filter data transfers to user-space, we apply protocol inference in BPF.

- Just a filter: False positives are okay.
- Example for HTTP:

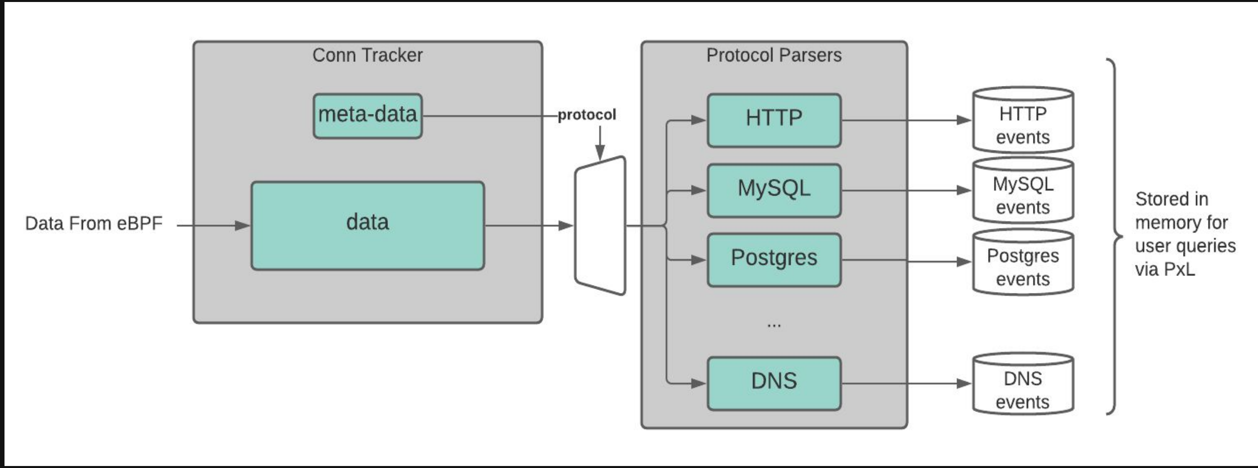
```
static __inline enum MessageType infer_http_message(const char*
buf, size_t count) {
  ...

  if (buf[0] == 'H' && buf[1] == 'T' && buf[2] == 'T' && buf[3] == 'P') {
        return kResponse;
  }
  ...
```

Likelihood that our inference eventually identifies the right protocol



Pluggable Protocol Parsers

Architecture consists of pluggable protocol parsers



Supported Protocols List
HTTP MySQL Postgres Redis Cassandra Kafka NATS DNS gRPC*

*gRPC is traced with dedicated uprobes

We are working on making it easier to contribute protocols

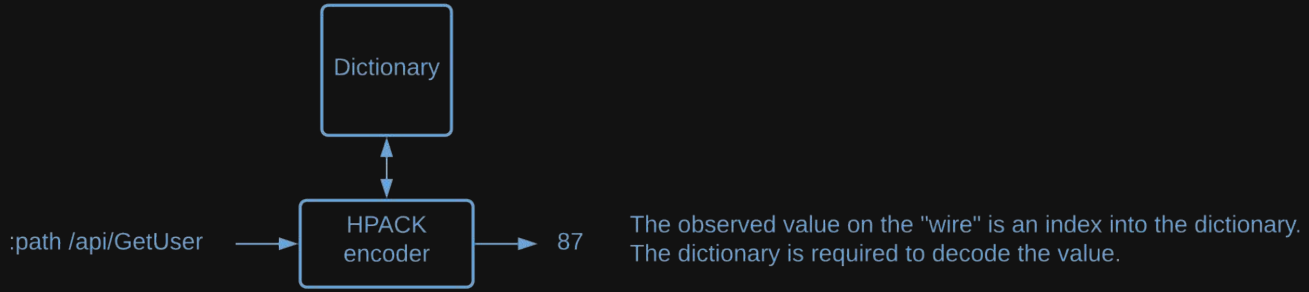
- Including a contribution guide

When kprobes are not enough: Tracing gRPC and TLS

Tracing HTTP/2 and gRPC: The problem

The kprobe-based approach has been mostly effective, but...

- HTTP/2 includes a *stateful* compression scheme called HPACK.
- HPACK uses a dynamic dictionary of common header values.
- We can’t decode the headers if we don’t have the dictionary.



Tracing HTTP/2 and gRPC: What to do?

Unfortunately, we can’t count on knowing the dictionary.

- We may deploy after the HTTP/2 connection was made
- We may lose data through the perf buffer.

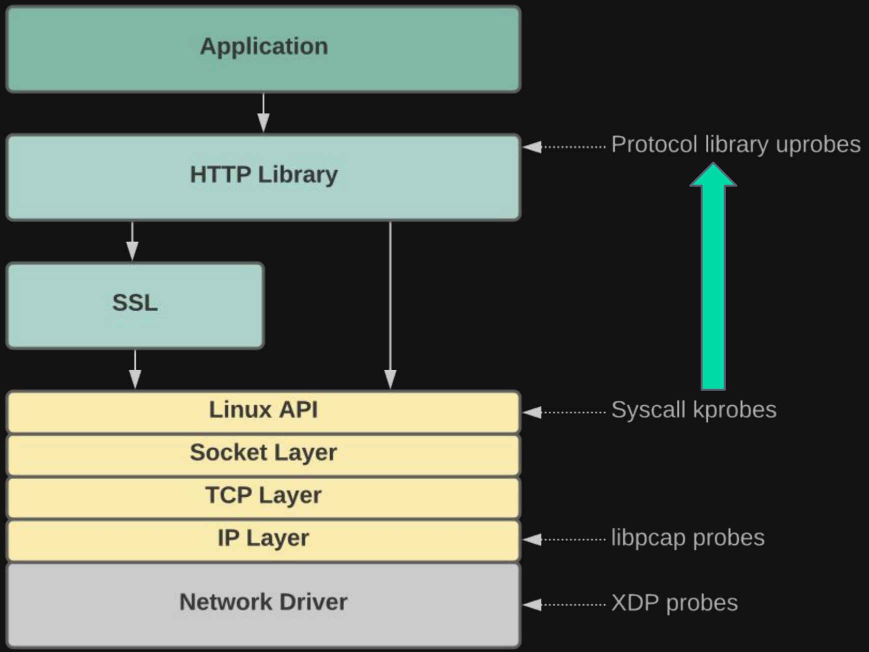
Options we considered:

1) Try to learn the dictionary.
    - Tried it. Too complex..
2) Recover the dictionary state via uprobes.
    - No easy place to probe.
3) Trace the gRPC library directly via uprobes. ⟵ Final solution
    - Not easy, but our only viable option.

Tracing gRPC: Our Approach

Use uprobes to capture data before it's compressed.

We have implemented uprobes for Golang's gRPC library; other libraries are planned.



https://github.com/pixie-io/pixie/blob/main/src/stirling/source_connectors/socket_tracer/bcc_bpf/go_http2_trace.c

Our gRPC Experience: Takeaways

Any protocol that is stateful is hard to decode.

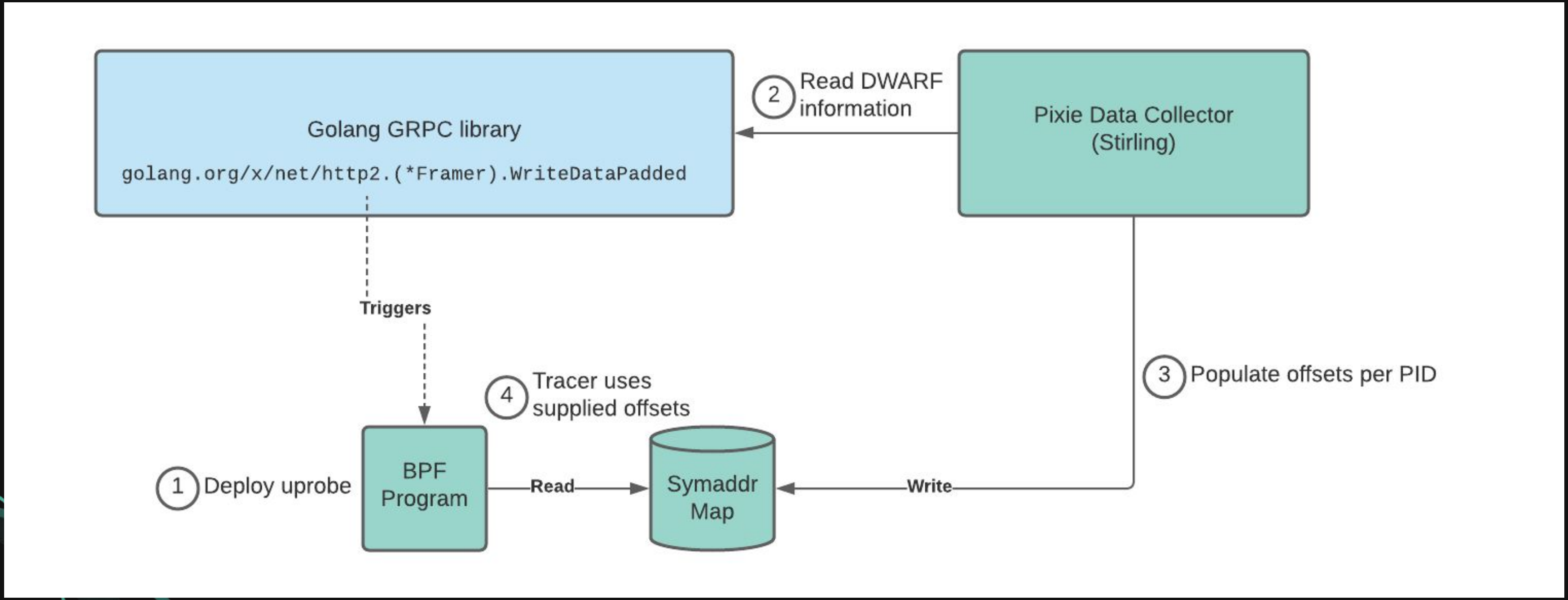
- Compression on individual messages is okay; problem is with dependent state.
- Tools like tWireshark face the same issue: can't decode headers without the state.

The uprobe based approach is hindered by the scalability problem.

- We need uprobes for each gRPC library for full tracing.
- Must take care to place uprobes on functions that appear stable across versions.
- Need debug symbols to make it more robust

Making Uprobes Robust

Read DWARF information to find offsets; pass them to the BPF program.



```c
// Probe for the golang.org/x/net/http2 library's frame writer.
//
// Function signature:
//   func (f *Framer) WriteDataPadded(streamID uint32, endStream bool, data, pad []byte) error
//
// Symbol:
//   golang.org/x/net/http2.(*Framer).WriteDataPadded
int probe_http2_framer_write_data(struct pt_regs* ctx) {
 uint32_t tgid = bpf_get_current_pid_tgid() >>32;
 struct go_http2_symaddrs_t* symaddrs = http2_symaddrs_map.lookup(&tgid);
 if (symaddrs == NULL) {
   return 0;
 }

 // Required argument offsets.
 REQUIRE_SYMADDR(symaddrs->http2_WriteDataPadded_f_offset, 0);
 REQUIRE_SYMADDR(symaddrs->http2_WriteDataPadded_streamID_offset, 0);
 REQUIRE_SYMADDR(symaddrs->http2_WriteDataPadded_endStream_offset, 0);
 REQUIRE_SYMADDR(symaddrs->http2_WriteDataPadded_data_offset, 0);

 // ---------------------------------------------
 // Extract arguments (on stack)
 // ---------------------------------------------

 const char* sp = (const char*)ctx->sp;

 void* framer_ptr;
 bpf_probe_read(&framer_ptr, sizeof(void*), sp + symaddrs->http2_WriteDataPadded_f_offset);
```

SSL Tracing

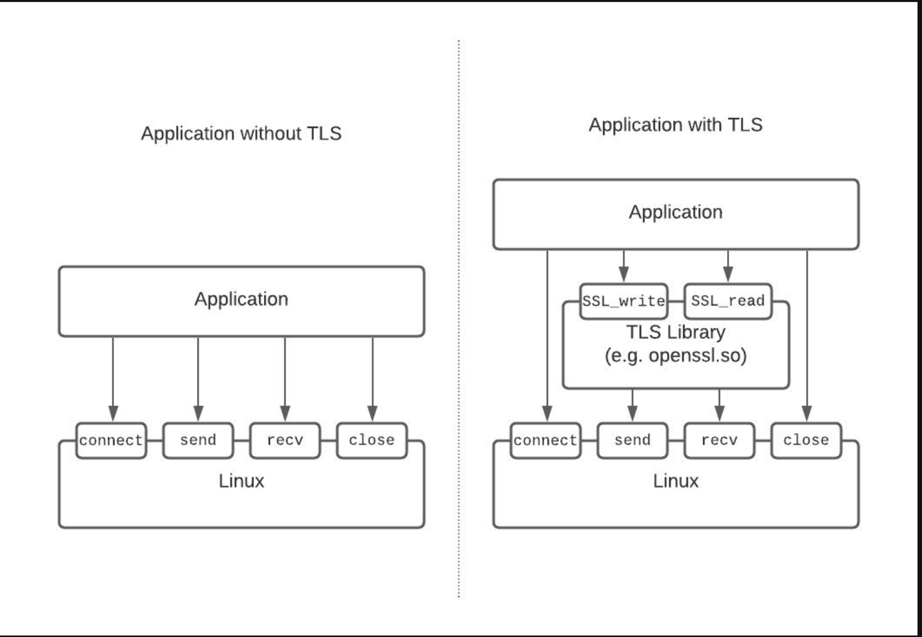
Tracing SSL traffic with kprobes doesn't work either.

- Data is already encrypted

Uprobes come to the rescue

- Trace the SSL library instead

BCC has a similar tool: sslsniff



https://github.com/pixie-io/pixie/blob/main/src/stirling/source_connectors/socket_tracer/bcc_bpf/openssl_trace.c

Uprobes on TLS Libraries

There is a simple mapping of kprobes to uprobes

Kprobe function	OpenSSL API function	Golang crypto/tls library
read/recv	SSL_read	crypto/tls.(*Conn).Read
write/send	SSL_write	crypto/tls.(*Conn).Write

Uprobes on SSL API push to same perf buffer as syscall probes

- No changes to user-space code :)

SSL Tracing Observations

While uprobes have the scalability problem, it's not so bad with SSL

- The number of popular SSL libraries is small.
- By tracing a public API, we get good probe stability across versions.

One interesting exception: node.js

- Uses OpenSSL in an asynchronous manner (via libuv).
- Makes it hard to correlate the traced data with a FD.

Requires additional node.js specific uprobes :(

Summary

Pixie is a Kubernetes observability platform.

- Protocol tracer provides instant visibility on K8s clusters.
- No user instrumentation: powered by eBPF.

Pixie is now an open-source CNCF sandbox project

- https://github.com/pixie-io/pixie
- Contributions are welcome!

PIXIE

Thank you!...Questions?