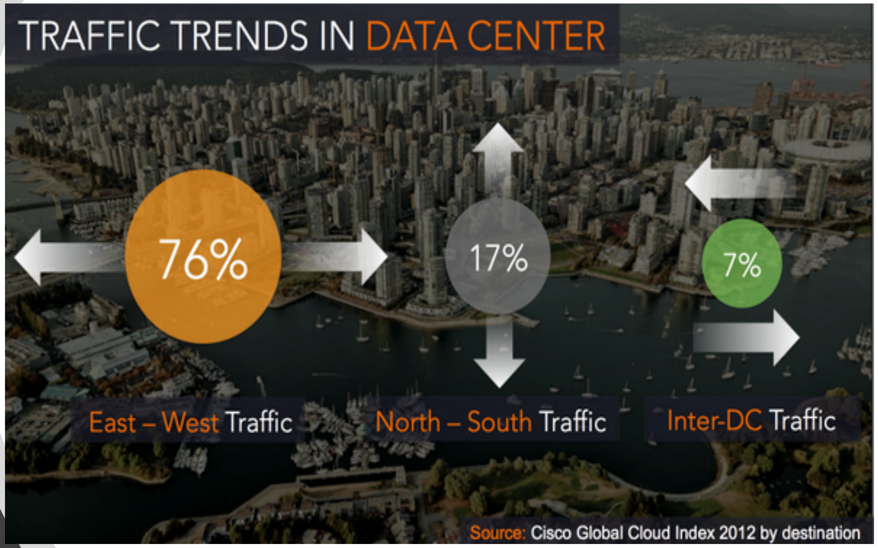
eZTrust: Network-Independent Zero-Trust Perimeterization for Microservices

Presenter: Zirak Zaheer (Facebook / University of Utah)

Collaborators: Hyunseok Chang (Nokia Bell Labs), Sarit Mukherjee (Nokia Bell Labs), Jacobus Van der Merwe (University of Utah)

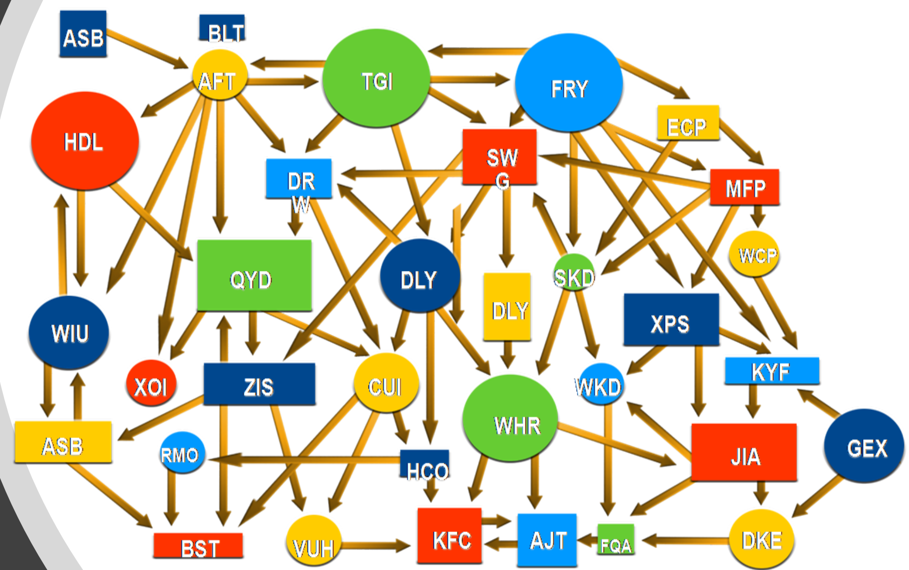
Evolution of Traffic Pattern in the Era of Microservices

- Data centers are housing more and more microservices
- Monolithic legacy apps are broken into and deployed as multiple “interdependent” microservices
- **East-west traffic accounts for 70-80% of traffic!**



Characteristics of Typical Microservices Environment

- High churn rate of microservices
  - ✓ Frequent updates to microservices (CI/ CD workflows)
  - ✓ Dynamic autoscaling
- Compared to legacy apps, # of microservice instances is much higher
- **New security threats exploit cross-service dependencies and propagate laterally via east-west traffic**



Securing East-West Traffic using Traditional Ways

- Network-based perimeterization
  - ✓ Access control rules based on network endpoints (IP/port)

- Access boundary for each workload or a group of workloads is enforced with:
  - ✓ Security group rules using software switches
  - ✓ IPtable rules

Problem #1: Reliability

Policy Intent

Only workload "front-end" can talk to workload "back-end"

Implemented Rule

Only IP_1:$port_1$ can reach IP_2:$port_2$

- Semantic gap between high-level policy intents and ephemeral network endpoints
- Network endpoints are not binding properties of microservices
  - ✓ Can change dynamically due to microservice reconfiguration or by middleboxes (NAT/PAT) as part of network operations
  - ✓ Can be spoofed by malicious tenants]

Problem #2: Scalability

- Highly dynamic communication patterns due to high churn rate of microservices (frequent re-deployments and dynamic autoscaling)
- Policy table grow with the number of communicating microservice instances and network attributes relied upon
- Hard to update and manage growing policy tables in a timely fashion

Problem #3: Granularity

- Emerging security attacks are often associated with newly found software vulnerabilities (e.g. OpenSSL Heartbleed, Shellshock)
- Necessitates fine-grained perimeterization and flexibility in policy definitions (e.g. policies based on application identity and version, status of security patches, etc)

Takeaway

We want more reliable, scalable and granular perimeterization for microservices!

Ideas to solve these problems?

- Decouple perimeterization from network endpoints
- Instead of IP/port strongly identify microservices with a set of **authentic contexts** (derived from microservices)
- Enable fine-grained and granular policies on a per-packet level
- Policy rule tables should not grow as the number of microservice instances grow.

e.g: “accept traffic only if it originates from HAproxy with sslVersion 1.8, and is destined to an nginx server”

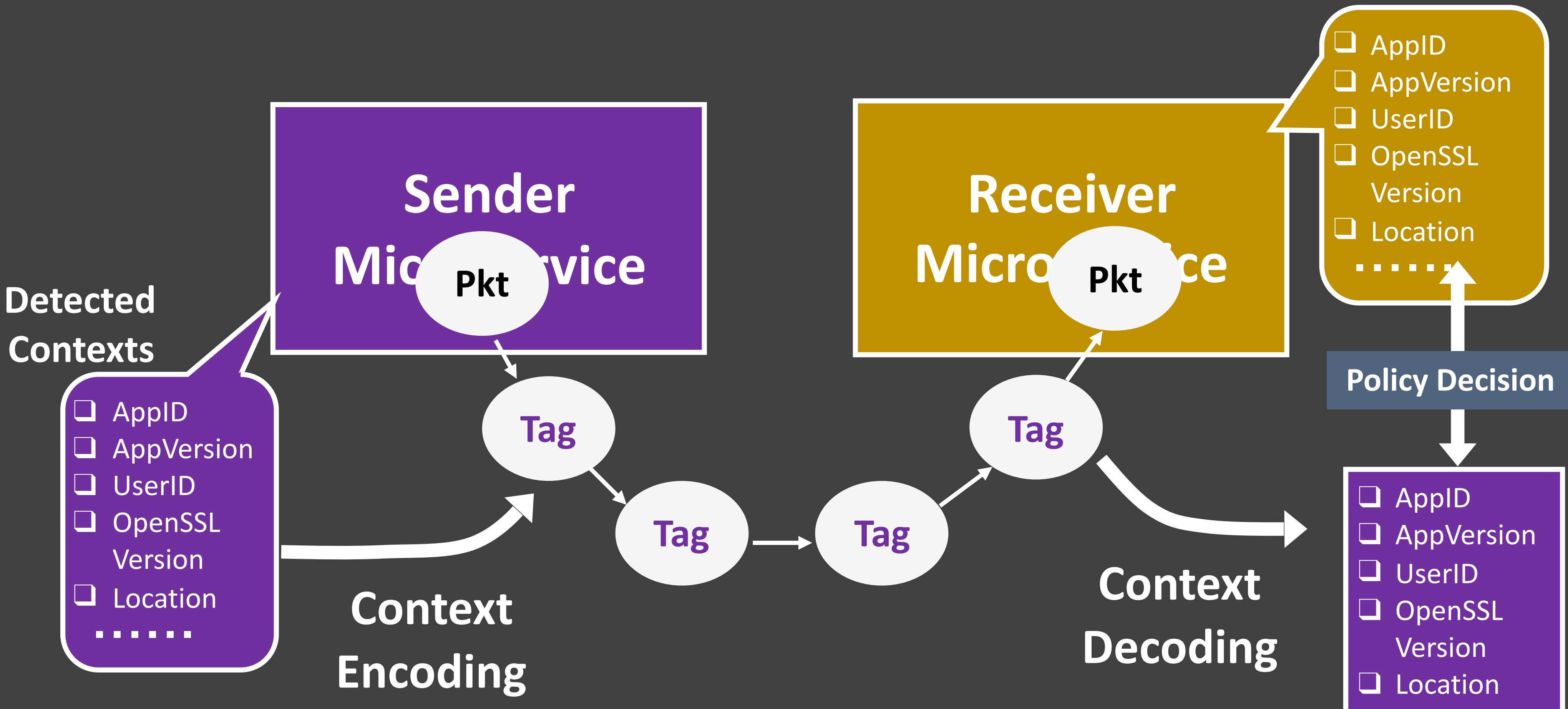
Threat Model

- We trust cloud provider's server OS/Kernel and central orchestrator and assume its free of vulnerabilities

- We trust contexts derived from deployed microservices if:
  - ✓ Their contexts are traced from the trusted infrastructure (OS kernel)
  - ✓ Their contexts are derived from untampered software packages (e.g. binaries/libraries from official Linux repos, digitally signed software)
  - ✓ Their contexts are retrieved from the trusted orchestrator (e.g. container image tag, geographic location)

Our Approach: eZTrust

Fine-grained context-driven perimeterization.

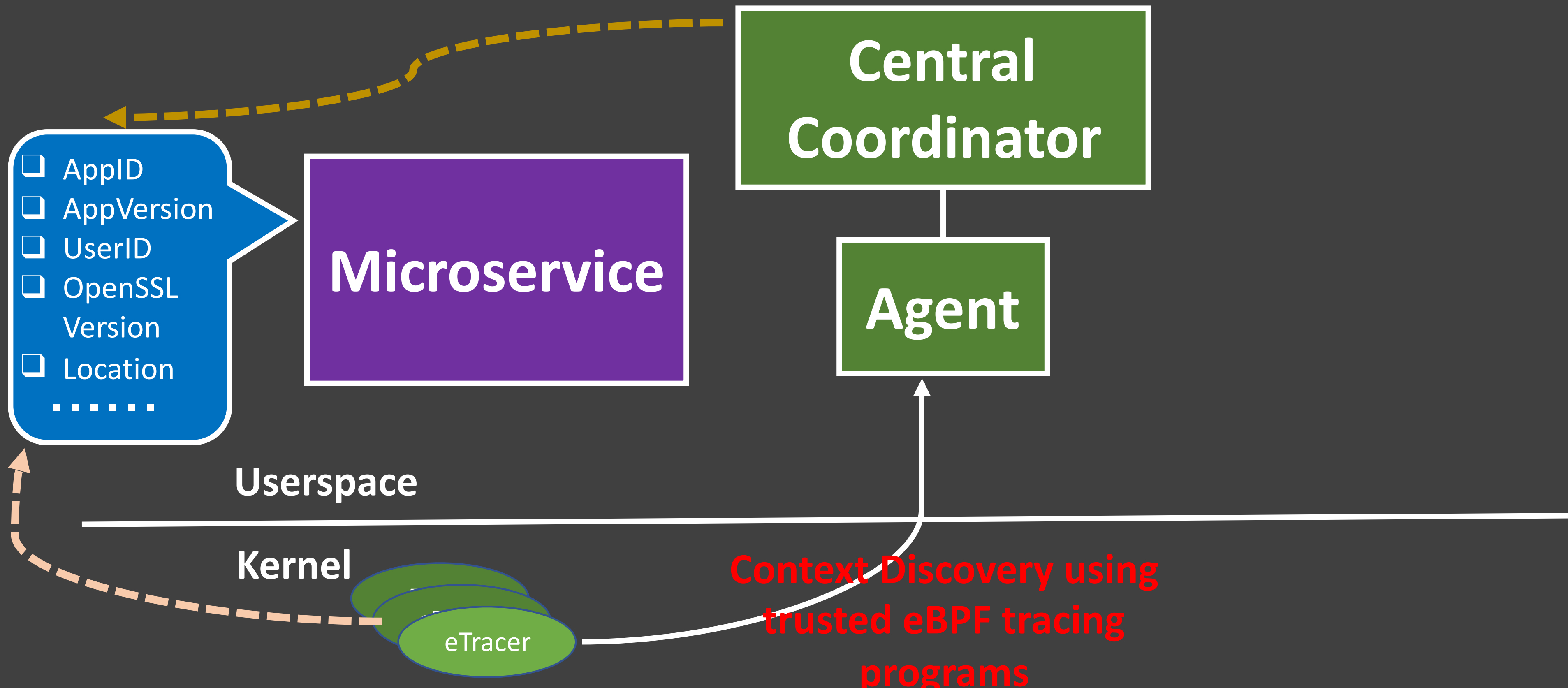


Challenges (Sender-side)

- Contexts of each microservice must be correctly determined without significant overhead
- Packets generated by each microservice must be correctly associated with the microservice in the dataplane

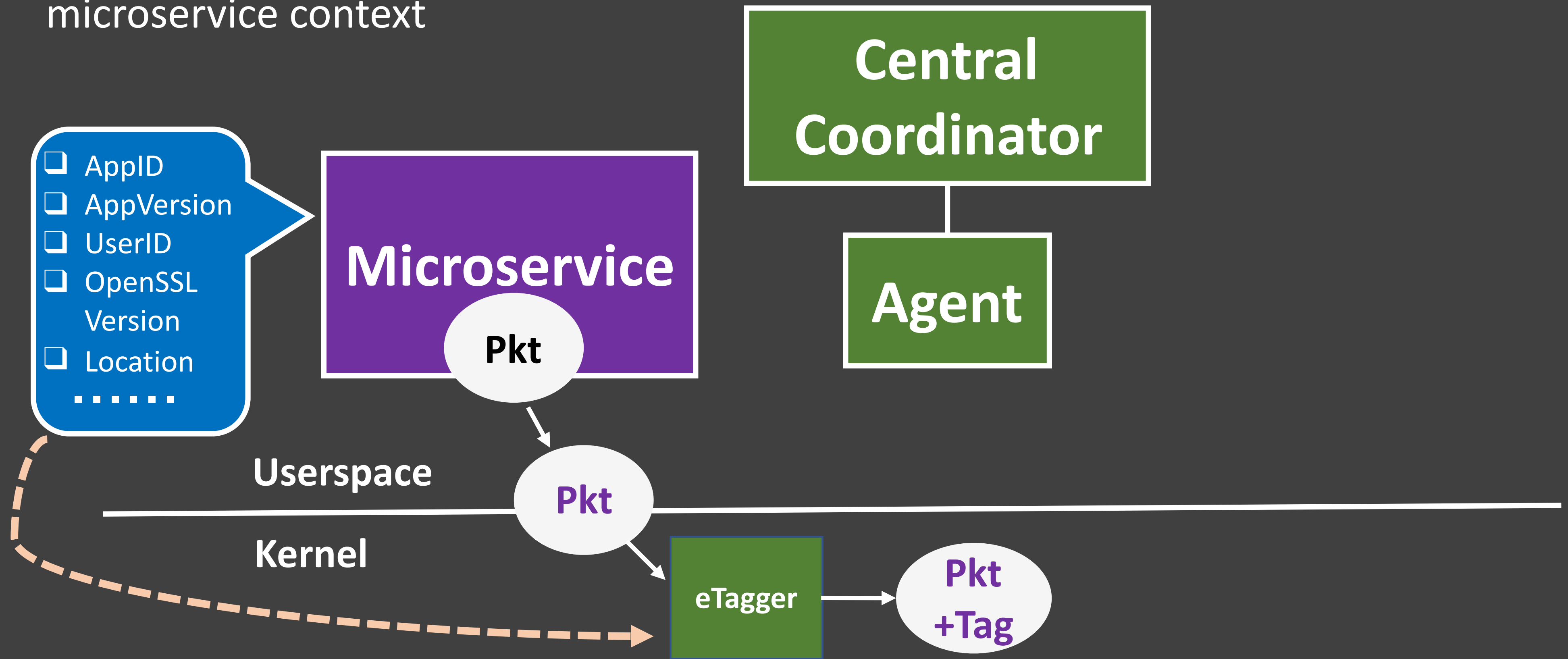
Sender-Side Challenge #1

- Reliably determining context for each service



Sender-Side Challenge #2

- Packets generated by each microservice must be correctly associated with the microservice context



Challenges (Sender-side)

- Contexts of each microservice must be correctly determined without significant overhead
- Packets generated by each microservice must be correctly associated with the microservice

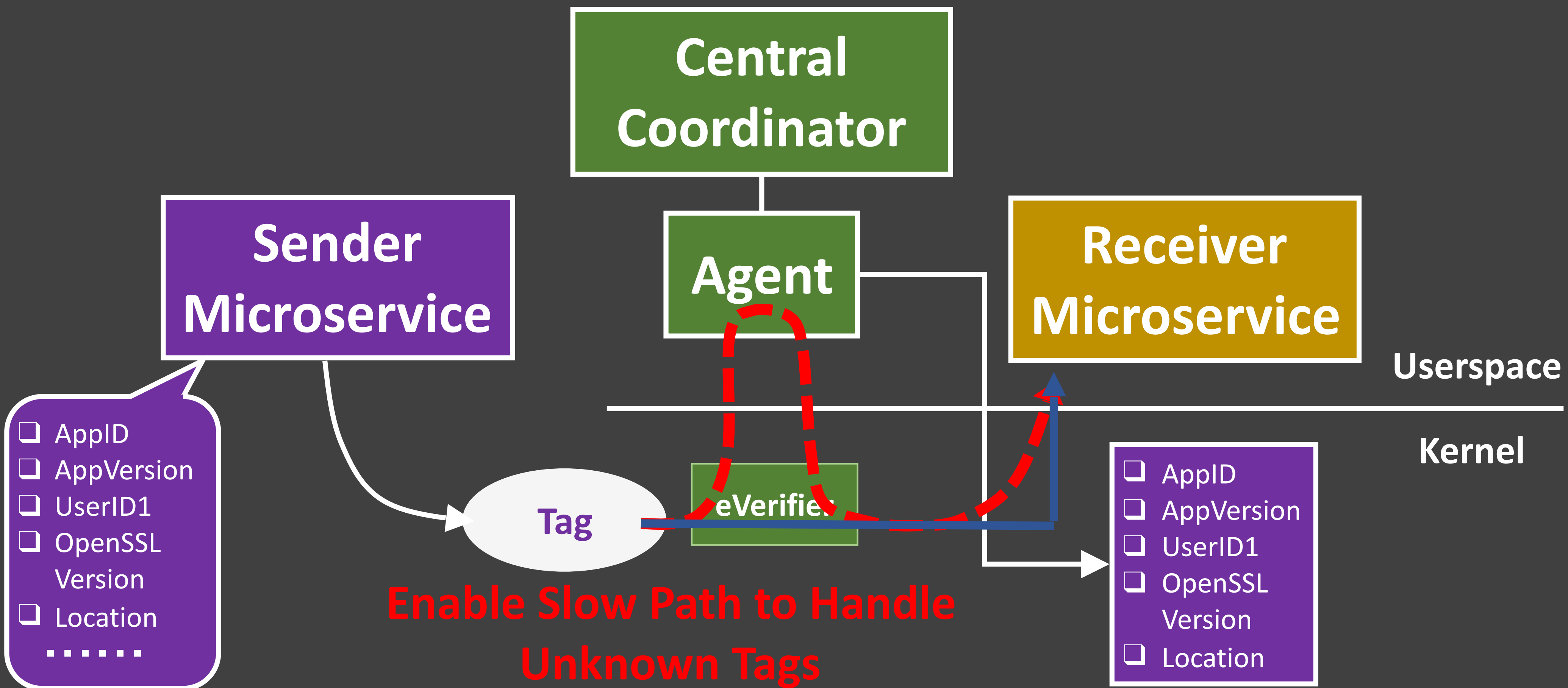
- Approach: Leverage the **eBPF framework**
  - **Trace context events:** process creation events, userspace events (SSL handshake, MySQL connection)
  - **Trace socket events:** identify which packets are generated by which sockets in which namespaces

Challenges (Receiver-side)

- Perform context decoding when tags are unknown to receiver
- Perform context decoding during dynamic context changes

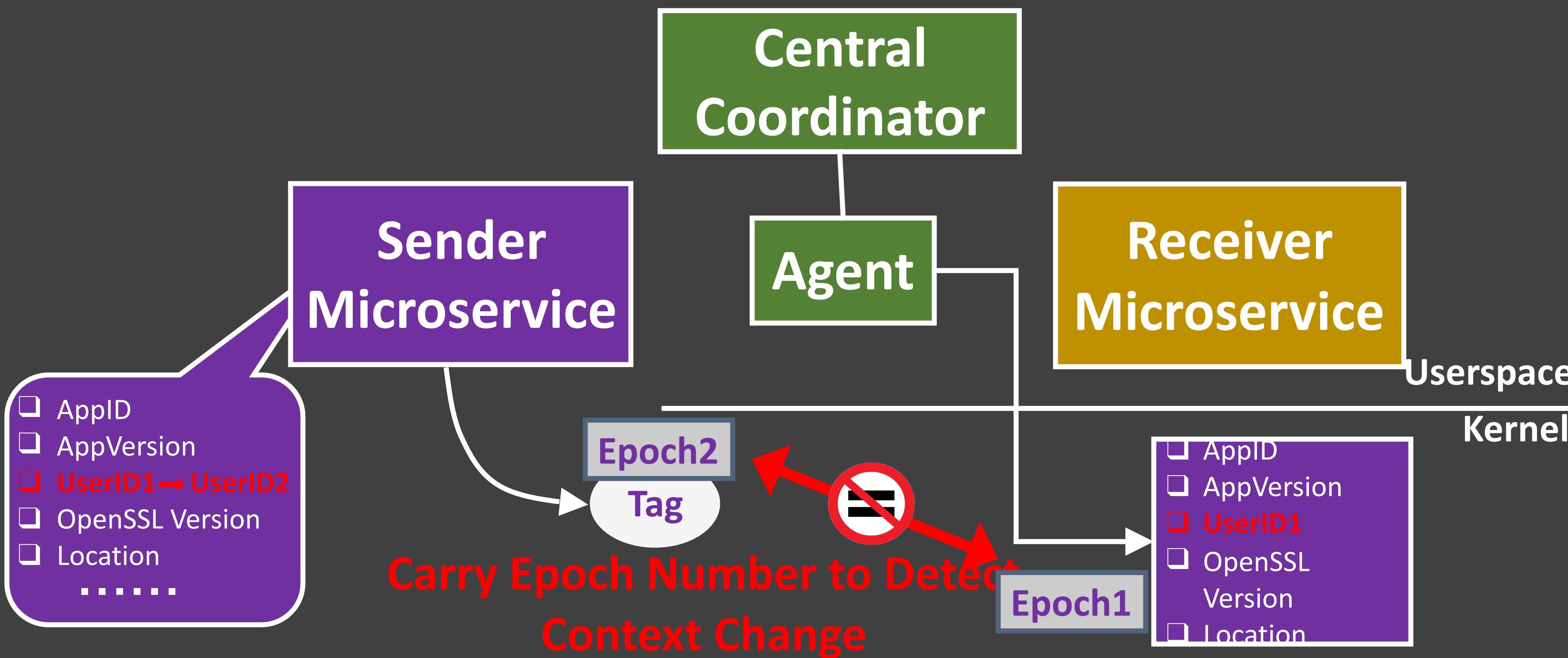
Receiver-Side Challenge #1

- Context decoding when tags are unknown to receiver.



Receiver-Side Challenge #2

- Context decoding during dynamic context changes.



Challenges (Receiver-side)

- Perform context decoding when tags are unknown to receiver
- Perform context decoding during dynamic context changes

- Approach:
  - **Dual path packet processing:** slow path to resolve unknown tags in userspace, and fast path to process cached tags in kernel
  - **Epoch counter in the tag:** epoch counter increment to detect context change and invalidate caching on fast path

How eZTrust solves the problems highlighted

- **Reliability:** eBPF-driven real-time tracing of authentic microservice contexts
- **Scalability:** Ruleset size scales only with # of distinct contexts, regardless of # of network endpoints
- **Granularity:** Policies are defined based on an extensible list of fine-grained workload contexts

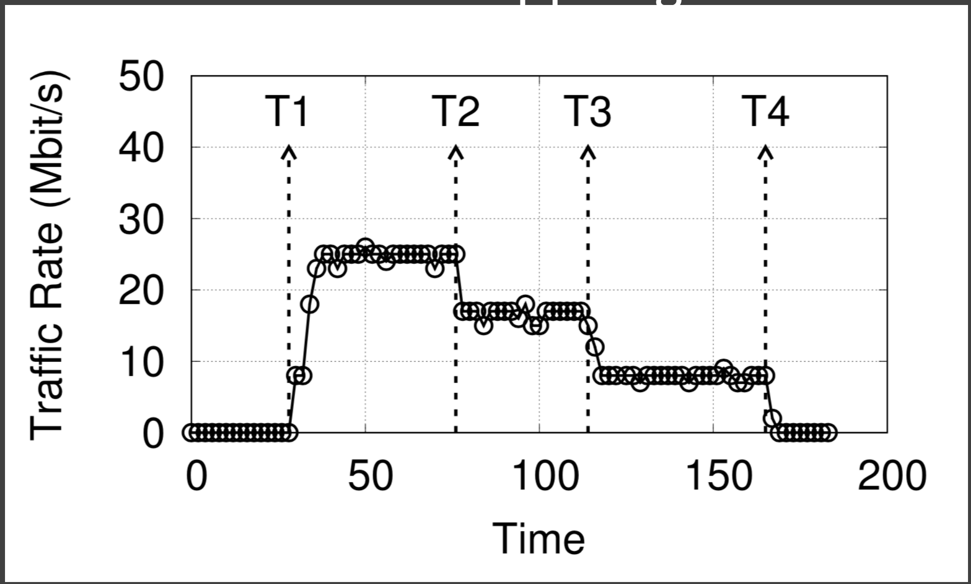
Perimeterization policy decisions are completely decoupled from underlying networks

Evaluations

- Implemented proof-of-concept prototype.
- Evaluated performance and correctness of the system (see paper)
- Compared our approach to other related works (see paper).
- Tested prototype with dynamic context changes and realisitic microservice deployment.

Dynamic Context Change

- Setup: 3 wget containers, 1 Nginx container
- At T1 insert Policy => allow “nginx server” to talk to “app=wget with status=healthy”
- At T2, T3, T4 status changes to “compromised”
  - eZTrust drops connections



Realistic Microservice Deployment
(on single host)

14 distinct Microservices

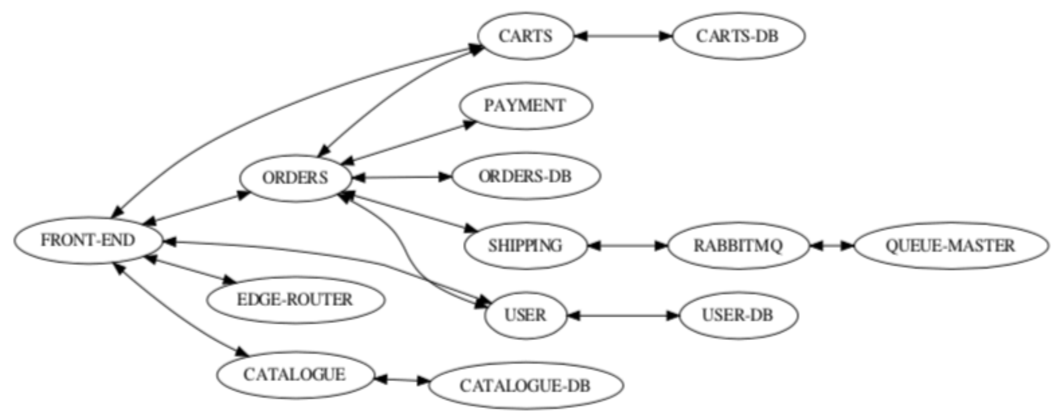


Figure 7: Microservice control flow in Sock Shop.

Source: https://microservices-demo.github.io/

Latency Comparison

- Installed policies to allow communication between microservices
- Generate http traffic
- eZTrust performs slightly better to enforce policy 5-15%
- Performance benefits of eBPF, encoding and decoding is lightweight

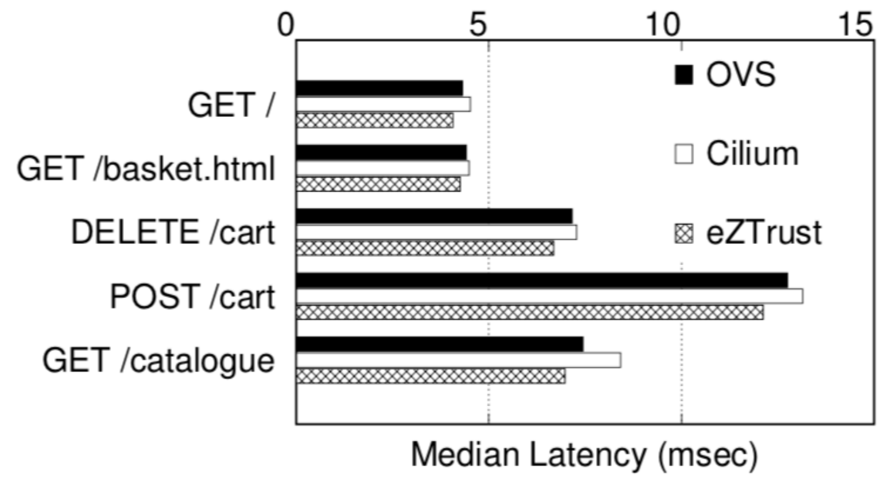


Figure 8: End-to-end latencies of Sock Shop.

Discussion

- **Tag granularity:** trade-off between policy granularity and slow path overhead (per-microservice, per-process, per-connection)
- **Tag anonymization:** Can be avoided if raw sockets are not allowed in the infrastructure
- **Smart NIC offload:** transparent eBPF offload is available with some smart NIC (e.g. Agilio) to partly offload eZTrust processing
- **Platform compatibility:** can co-exist with network endpoint based perimeterization by extending supported contexts

Conclusion

- eZTrust, a network-independent perimeterization solution for microservices, where we shift perimeterization targets from network endpoints to fine-grained, context-rich microservice identities.
- We tap into the growing wealth of tracing data of microservices made available by eBPF, and repurpose them for perimeterization.
- We adopt OVS-like flow-based packet verification, where packets are classified into flows not based on packet header fields, but based on microservice contexts.

Questions?

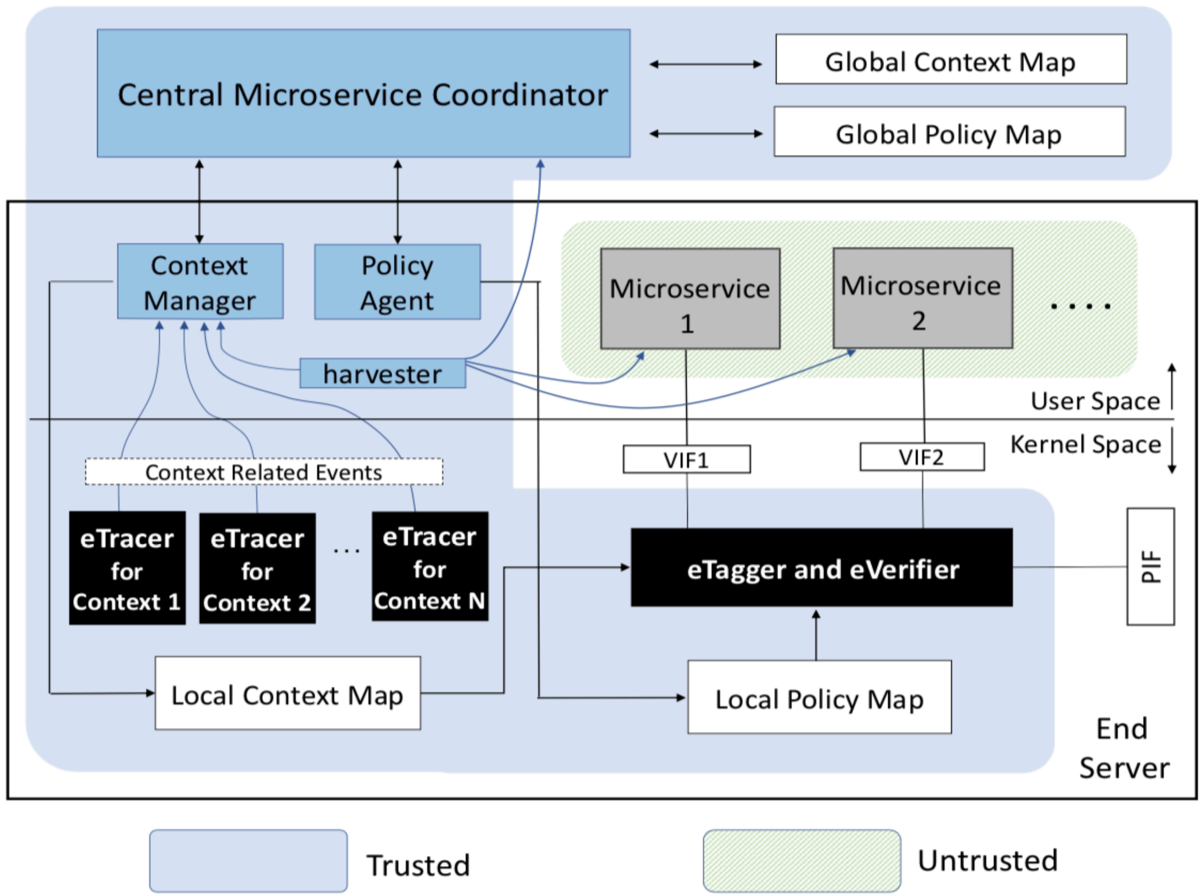
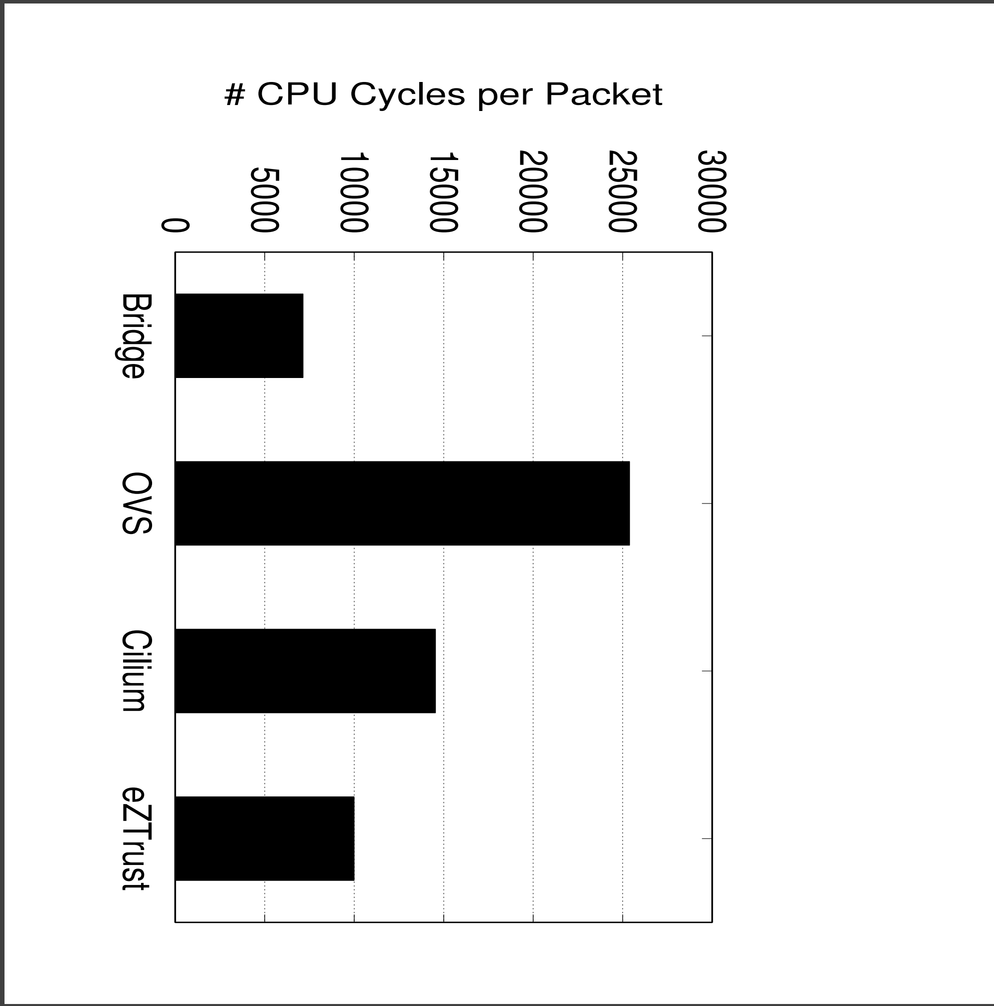


Figure 1: The eZTrust architecture.

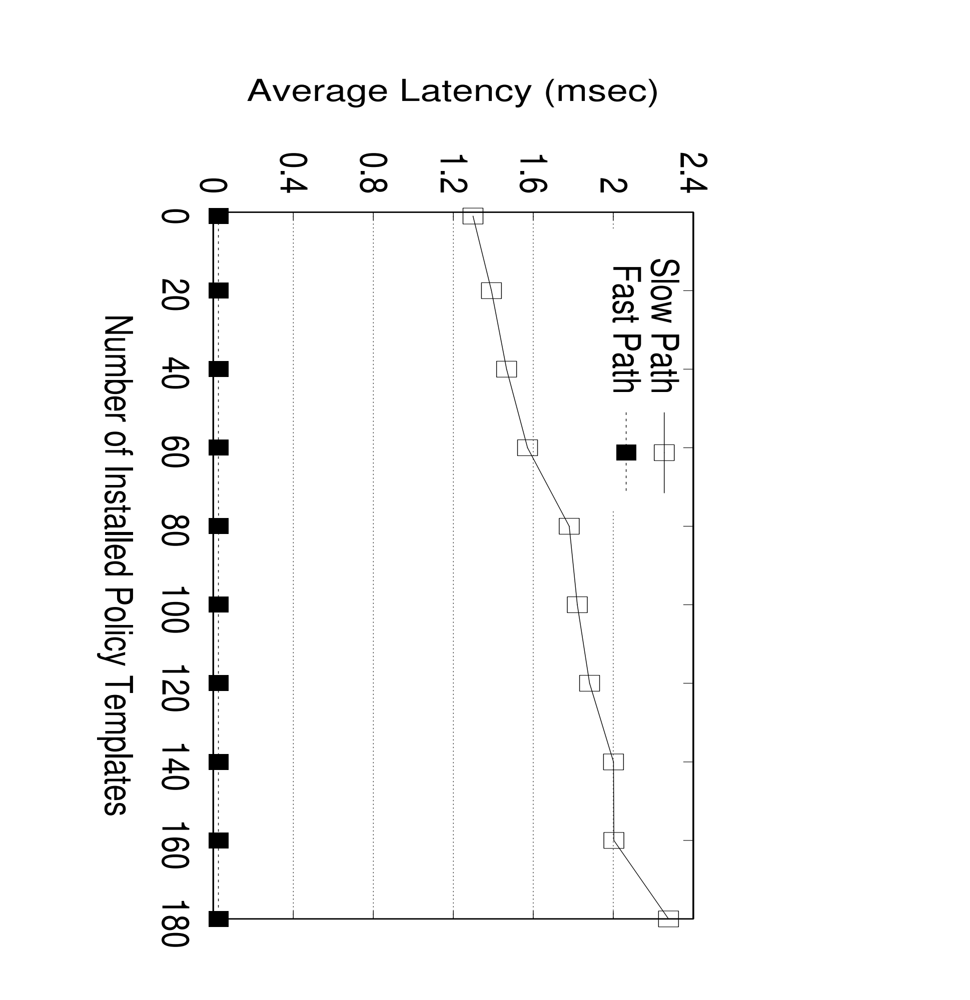
Per-packet CPU resource overhead

- Captures CPU usage incurred by policy enforcement only
- Two containers on one Server pinned to fixed CPU cores
- 60 byte UDP packets using iperf
- eZTrust per packet overhead is minimum compared to similar approaches



Slow Path vs Fast Path Latency

- Slow path latency grows as the Policy template table size grows
- Fast Path latency is unaffected by the Policy Template table size



Context Based Policy
DPI vs eZTrust

- DPI Policy: Accept traffic if it originates from OpenSSL version X.
- eZTrust enforces similar policy with drastically lower CPU

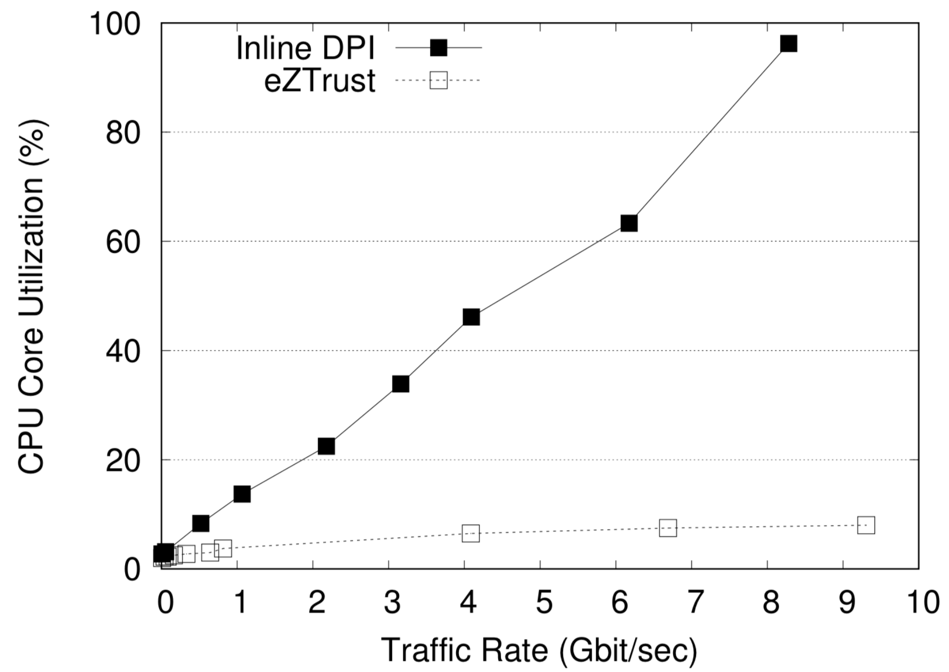


Figure 5: DPI-based perimeterization vs. eZTrust.