eZTrust: Network-Independent Zero-Trust Perimeterization for Microservices

Zirak Zaheer[†] Hyunseok Chang[*] Sarit Mukherjee[*] Jacobus Van der Merwe[†]
University of Utah[†] *Nokia Bell Labs*[*]

ABSTRACT

Emerging microservices-based workloads introduce new security risks in today's data centers as attacks can propagate laterally within the data center relatively easily by exploiting cross-service dependencies. As countermeasures for such attacks, traditional perimeterization approaches, such as network-endpoint-based access control, do not fare well in highly dynamic microservices environments (especially considering the management complexity, scalability and policy granularity of these earlier approaches). In this paper, we propose eZTrust, a network-independent perimeterization approach for microservices. eZTrust allows data center tenants to express access control policies based on fine-grained workload identities, and enables data center operators to enforce such policies reliably and efficiently in a purely network-independent fashion. To this end, we leverage eBPF, the extended Berkeley Packet Filter, to trace authentic workload identities and apply per-packet tagging and verification. We demonstrate the feasibility of our approach through extensive evaluation of our proof-of-concept prototype implementation. We find that, when comparable policies are enforced, eZTrust incurs 2–5 times lower packet latency and 1.5–2.5 times lower CPU overhead than traditional perimeterization schemes.

CCS CONCEPTS

• **Networks** → **Data center networks**; *Programmable networks*; • **Security and privacy** → *Access control*;

KEYWORDS

Data center networks, Access control, Microservices, eBPF

SOSR '19, April 03–04, 2019, San Jose, CA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As common network security measures, data centers are traditionally protected at their borders, under the assumption that attacks originate externally via north-south traffic. This assumption is proving incorrect as data centers start to house more and more interdependent microservices [1], which in turn leads to increasingly dominant intra-data center east-west traffic (85% of total data center traffic [2]). This emerging application deployment trend poses new security risks as the infrastructure is not properly protected against its internal misbehavior, which allows threats from east-west traffic to propagate laterally across any number of microservices within data centers. In order to address the newly emerging security risks within data centers, the zero-trust security model [3] has been postulated with a guiding principle of "never trust, always verify" instead of the current operating model of "trust but verify." Under this model, every deployed tenant microservice must be secured with *fine-grained perimeterization policies* that scrutinize the traffic in and out of the microservice, as dictated by tenants.

In modern SDN centric data centers [4], where a centralized controller interconnects tenant microservices by pushing appropriate forwarding rules at the programmable software switches, a traditional way of realizing perimeterization is to define network-endpoint-based policy rules at these switches [5, 6, 7]. In this network based perimeterization, tenant policy intents, which are typically defined based on workload identities (e.g., only workload "X" can talk to workload "Y") need to be translated into corresponding network-endpoint policies (e.g., only <IP1:port1> can reach <IP2:port2>) to be enforced by data center operators at the network level. However, this semantic gap between tenant's policy intents and operator's policy enforcement infrastructure makes the resulting perimeterization potentially unreliable and error-prone. Network endpoint properties such as IP addresses and port numbers are not the *binding* properties of tenant workloads, but rather ephemeral attributes attached to them, which can dynamically be changed, either by tenants from microservice reconfigurations, or by middleboxes as part of network operations (e.g., address/port translation and load balancing), or even can be spoofed by malicious attackers.

Correctness aside, the network based perimeterization also introduces scalability challenge in policy rule management. First of all, the size of policy rule sets increases multiplicatively with the number of communicating microservices or their security zones, as well as the number of endpoint properties relied upon by policies. In addition, every time communication patterns change due to microservice creation, termination and migration events, policy rules provisioned for existing microservices need to be inspected and adjusted in a timely fashion to fulfill tenant policy intents. Considering the large-scale, highly dynamic microservice deployment nature of modern clouds [8], the task of maintaining and updating policy rule sets in such environments incurs significant resource overhead on the data center infrastructure [9].

Finally, the policy granularity of the network based access control is restricted to network endpoint level. On the other hand, emerging security risks increasingly necessitate more fine-grained perimeterization, where access is regulated based on detailed contexts associated with microservice workloads (e.g., application/user identity, protocol version, status of security patches). Such granular policies are useful to contain potential damage from newly found software vulnerabilities (e.g., POODLE attack against SSL, OpenSSL Heartbleed, Shellshock). However, enriching network-endpoint policies with granular contexts (e.g., SSL/OpenSSL/Bash versions) typically requires resource-heavy deep packet inspection (DPI) and intrusive guest introspection [10, 11].

In order to address these limitations of the existing network-endpoint-based perimeterization, we propose in this paper an alternative solution called *eZTrust*, where we shift perimeterization targets from network endpoints to *workload identities*. In this approach, we exploit the fact that microservices are typically packaged in lightweight containers. We are also motivated by the ongoing efforts to monitor detailed lifecycles of containerized microservice workloads [12, 13, 14]. Our approach is to repurpose the growing wealth of such monitoring data gleaned from deployed workloads for perimeterization. The key idea of eZTrust is as follows. Every packet generated by a microservice is stamped with a *tag* which encodes a fine-grained identity of the microservice. The fine-grained identity is defined as *a set of authentic contexts tied to the microservice workload*. Example contexts include application-level identity (e.g., application name/version), runtime environment-related signatures (e.g., kernel version, dynamically loaded library version, user identity) or deployment-specific metadata (e.g., geographic location, filesystem image tag). Some of these contexts are detected from the workloads themselves, while others are fetched from the centralized microservice orchestrator. Once the tagged packet is received, the receiver end extracts the tag, decodes it back to the sender-side contexts, and applies perimeterization policies based on the sender-side contexts as well as recipient's contexts, as instructed by a receiver-side tenant. In this manner, the whole perimeterization process is completely decoupled from underlying networks.

To realize eZTrust, we leverage eBPF [15], the extended Berkeley Packet Filter, which enables us to trace various contexts associated with microservice workloads as well as perform per-packet tagging and verification. Inspired by the flow cache design of Open vSwitch (OVS) [16], we adopt dual-path per-packet verification, where slow path via userspace is triggered to handle packets with unknown contexts, while fast path conducts eBPF-based in-kernel packet verification. To ensure correct packet verification in the presence of context changes, we leverage the notion of an epoch, which is used to detect context changes and invalidate caching on the fast path. We have prototyped eZTrust and conducted detailed evaluations to show its efficacy. We find that, when comparable policies are enforced, eZTrust incurs a factor of 2–5 lower packet processing latency and a factor of 1.5–2.5 lower per-packet CPU overhead than other state-of-the-art perimeterization schemes. Using realistic perimeterization scenarios such as OpenSSL Heartbleed vulnerability containment and control flow protection for a real-world e-commerce application, we demonstrate that eZTrust can support context-rich perimeterization policies efficiently.

We make the following specific contributions in this paper: (i) We design the eZTrust architecture which enables fine-grained context based perimeterization without relying on complex network-endpoint-based policies, nor requiring compute-intensive DPI for detailed context discovery. (ii) We implement a proof-of-concept prototype of the architecture using eBPF, and demonstrate motivational scenarios enabled by the prototype. (iii) We quantify the performance and resource overhead of the prototype, and compare it against alternative approaches.

2 RELATED WORKS

In the following, we discuss several alternative perimeterization approaches and their limitations.

Transport-level perimeterization. The First Packet Authentication [17] and Trireme [18] enforce perimeterization policies at the TCP layer. In these proposals, a cryptographically signed identity token is carried in a TCP SYN packet, and the rest of the TCP handshake proceeds only if access is granted based on the identity. Compared to network-endpoint-based approach, transport-level perimeterization is more reliable as the identity of a microservice is not tied to the underlying network, but cryptographically verified. These approaches, however, have several drawbacks. First, these TCP-specific schemes cannot be generalized to non-TCP based connection-less traffic (e.g., QUIC [19] over UDP).

Properties	Network-endpoint-based	Transport-level	Label-based	DPI-based	API gateway	eZTrust
Policy management complexity	Bad	Good	Good	Good	Good	**Good**
Reliability of policy attributes	Bad	Good	Bad	Good	Good	**Good**
End-server resource overhead	Good	Bad	Good	Bad	Bad	**Good**
Protocol/application dependency	Good	Bad	Good	Bad	Bad	**Good**
Policy granularity and dynamism	Bad	Bad	Bad	Good	Bad	**Good**

Table 1: Comparison of existing perimeterization approaches.

2 They also require heavy-duty cryptographic operations during TCP handshake, resulting in high per-connection computation overhead. This is problematic with a large number of short-lived TCP flows, which are common for microservices, or denial-of-service SYN attacks. 3 Finally, since access control is only on a per-flow basis during initial TCP handshake, they can be vulnerable to session hijacking attacks [20].

Label-based perimeterization. A label-based perimeterization approach called Cilium [21] is similar to the previous transport-level approach in that its policy enforcement is based on the "network-independent" identity of microservices. In Cilium, the microservice identity carried in network packets is defined by a set of key-value pair labels (e.g., role=frontend, user=joe) that are specified by its tenant. Unlike the transport-level approach, its policy enforcement is protocol-agnostic. Besides label-based policies Cilium also supports layer-7 API-aware policies. The problem of this approach is that tenant-defined labels for microservices are not their binding properties, thus not providing protection *across* tenants. When a label is assigned to a microservice by its tenant, other tenants blindly trust that label, which makes it vulnerable to malicious tenants who attempt to impersonate other microservices using their labels. Also, the labels are defined statically at the microservice launch time. Once associated, the labels remain with the microservice for the rest of its lifetime. Such static labels are problematic in dynamic policy environments.

DPI-based perimeterization. The aforementioned problems of the static label-based approach can be addressed by more dynamic context-aware schemes [10, 22, 23]. In this approach, individual end servers, where microservices are hosted, operate a DPI engine to actively extract layer-7 contextual information (e.g., application/protocol types, version, etc.) from packet payload, so that traffic can be filtered based on contextual attributes. While this approach allows fine-grained and genuine context based policies, the DPI processing takes a heavy toll on CPU resources, and sacrifices end-to-end packet delay. Besides, increasingly common encrypted traffic (e.g., HTTP/2) is not properly identified by DPI, and thus can bypass contextual policy filters.

API gateway/proxy-based perimeterization. In a distributed microservice architecture, API gateways or sidecar proxies [24, 25, 26] are often responsible for many critical management services for deployed microservices. As dedicated entry points to individual microservices, they can provide detailed API-level perimeterization using standard authentication/authorization techniques (e.g., OpenID, OAuth). However, this approach is only applicable to the cross-service communication designed for API gateways/proxies, but cannot properly regulate non-API traffic. Besides, it can be leveraged only for custom designed microservices with built-in support for OpenID/OAuth-capable APIs, but is not a general context-driven perimeterization solution. Many microservices are realized with existing open-source software, which is not originally developed for API gateways, and application-integrated API security is not a viable option for them.

Table 1 summarizes the pros and cons of different perimeterization approaches. eZTrust aims to address the limitations of these approaches.

Other network-independent packet processing. Authors of [27] propose a new type of policy routing that is based on process-level identifiers. While similar to eZTrust, this preliminary work does not provide detailed description on required data/control plane processing, and the performance of their prototype implementation is very limited.

3 ARCHITECTURE DESIGN

3.1 Threat Model

Before presenting the eZTrust architecture, we first describe the assumed threat model which motivated our design.

eZTrust is a policy driven perimeterization access control system for a containerized microservices environment. We assume that the infrastructure provider is trustworthy, and that the provider's infrastructure is free of vulnerabilities. We assume that the eZTrust framework is securely integrated with the provider's infrastructure for context collection, packet tagging and policy enforcement. This ensures that policies and contexts are securely stored and distributed, and that packets are tagged and verified without compromise. Unlike the infrastructure, we assume that tenant microservices are not to be trusted. Therefore, we trust contexts derived from microservices only if those contexts are collected from the trusted infrastructure (e.g., operating system kernel) or from untampered software packages (e.g., libraries from official Linux repositories or digitally signed software). We call these trusted contexts *authentic* contexts.

We assume that a malicious tenant attacker can compromise a microservice container by exploiting vulnerabilities in the container and can attempt to move laterally and get unauthorized access to microservices of other tenants running in the infrastructure. eZTrust's goal is to contain the attacker from moving laterally from one compromised workload or application to another.

3.2 Key Idea and Design Requirements

Next, we present the key idea of eZTrust and discuss associated design requirements. For better understanding of eZTrust, let's consider the following illustrative example scenario enabled by eZTrust, where two hypothetical microservices S1 (*HAproxy* load balancer) and S2 (*nginx* web server) are operated. S1 carries three contexts: app=HAproxy, appVersion=1.8, and location=US-West. The value "tag1" is mapped to these contexts. Similarly, S2 contains three contexts: app=nginx, appVersion=1.2, and loc=US-East, and the value "tag2" ($\neq$ tag1) is resolved to these contexts. S2's policy is defined as "*accept traffic only if it originates from HAproxy with version 1.8, and is destined to an nginx server in the east coast US.*" Under the eZTrust architecture, every packet generated by HAproxy on S1 is stamped with tag1. When the packet is received by S2, tag1 is converted to the sender-side contexts (app=HAproxy, appVersion=1.8, and loc=US-West). S2 then applies its defined policy based on the combination of the sender-side and receiver-side contexts, and accepts the packet. Tag2 is attached to the packets sent by S2 in a reverse direction.

As is clear from the above example, there are several important requirements to meet in order for the proposed architecture to become a reality.

- **(R1)** For each microservice, its associated contexts must be correctly determined without significant overhead. To make context discovery verifiable and lightweight, the contexts must be directly derived from the microservices, rather than arbitrarily assigned by a tenant like the static label-based approach, or separately mined with a heavy duty packet processing like the DPI-based approach.
- **(R2)** Some microservice (e.g., LAMP stack service or multi-container pod in Kubernetes) may run more than one applications in it, in which case, there will be *multiple* sets of contexts defined in the microservice (for different applications). Thus, when network packets are generated within a microservice (by any one of the applications running inside), the packets must be tagged with a *correct* set of application contexts.
- **(R3)** The mapping between a tag and a set of contexts must be globally unique and available for any arbitrary microservice to retrieve contexts from received packets with tags.

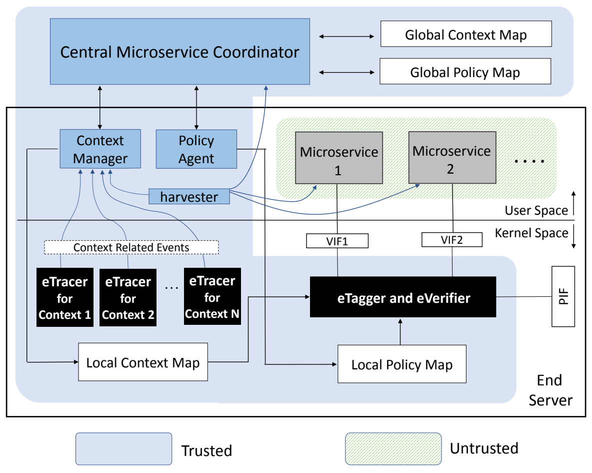


Figure 1: The eZTrust architecture.

- **(R4)** Whenever any context is changed in any microservice, the change must be reflected in the mapping and subsequent policy enforcement in a timely fashion.
- **(R5)** The per-packet access control process must be lightweight enough to handle line rate traffic.

The requirements **(R1)** and **(R2)** are relevant to egress packet processing on the sender side, where packets are tagged based on contexts, while the requirements **(R3)**, **(R4)** and **(R5)** are needed for ingress packet processing on the receiver side, where received tags are resolved to sender-side contexts, and packets are verified based on them. In the rest of this section, we describe how we address these requirements in the eZTrust architecture. The overall architecture diagram of eZTrust, along with trust boundaries is shown in Fig. 1.

3.3 Egress Packet Processing

3.3.1 Context Discovery. To meet requirements **(R1)** and **(R2)**, we are motivated by the recent advance in the universal in-kernel virtual machine technology called eBPF [15]. As a mainline Linux kernel feature, eBPF allows user-defined bytecode programs to be dynamically attached to kernel hooks in order to trace and process various kernel events without any expensive instrumentation or kernel customization. Deployed in-kernel bytecode programs can process and report captured kernel events to userspace, and access allowed kernel memory regions (e.g., packet data or key-value maps for stateful processing) via available eBPF helper function APIs.

We leverage the eBPF-based tracing mechanism to monitor various microservice-related events, which can reveal authentic application contexts associated with deployed microservices **(R1)**. For example, a currently running application's identity (e.g., name and version) can be reliably determined by tracing process creation events and mapping the created PIDs to corresponding application identity (e.g., appID and version).[1] In case of multi-user applications like remote desktop services, the identity of a logged-in user can be found from the user ID of the detected login shell PIDs. The SSL version enabled in an application can be identified by tracing an SSL handshake library call and its arguments (e.g., `SSL_do_handshake()` in OpenSSL). We collectively call these eBPF programs deployed for tracing *eTracers*.

On top of the eTracer-driven event tracing, we rely on active probing via *harvester*, which is a privileged monitoring service tasked by the infrastructure with collecting additional runtime environment related contexts of microservices, either by querying the centralized microservice orchestrator, or by attaching itself to the namespaces of the target microservice. Example contexts so collected include geographic location, filesystem image tag, signer's identity for a digitally signed image, container capabilities, kernel version, etc.

A dedicated userspace daemon called *Context Manager* collects events and contexts from eTracers and the harvester, and stores the discovered contexts in the *context map* in the form of <*tag*, *a set of contexts*> tuples. A set of contexts stored in each tuple is essentially a dictionary, containing a list of key-value pairs (e.g., {*context1*:*value1*, *context2*:*value2*, *context3*:*value3*,...}). A tag, which is the key to the context map, is uniquely mapped to a set of contexts associated with a particular application instance running in a microservice. To ensure its global uniqueness, the tag is formed by concatenating a microservice ID (which is unique data center wide) and an application PID. The centralized policy orchestrator maintains the *global context map* for all deployed microservices, and its size scales with the number of microservices as the maximum number of contexts maintained for each service is fixed. Each end-server operates a *local context map*, which is a subset of the global context map for all locally running microservices, as well as some non-local microservices as part of slow path processing (see Section 3.4.1).

In order to identify a correct set of application contexts for each egress packet **(R2)**, we keep track of which network sockets are created by which PID in what network namespace. The port number information in network sockets can provide a link between packets and corresponding applications, while network namespace information can be used to disambiguate different microservices that happen to create sockets with the same port number (e.g., HTTP port 80). We trace the in-kernel socket binding events using eBPF, and store the tracing result in the *socket map* in the form of <*port number*, *network namespace*, *PID*> tuples.

[1] We assume there is an infrastructure-managed trusted database that maps the cryptographic hash of binary executables or interpreted application bytecode to corresponding appID and version. More thorough application integrity verification is possible [28], but is out of scope of this paper.

3.3.2 Per-Packet Tagging. In order to tag every egress packet generated by a microservice, we attach a separate eBPF program called *eTagger* to the microservice's virtual network interface (VIF). eTagger intercepts every egress packet in the form of in-kernel packet data structure (e.g., *sk_buff*), which carries raw packet data, as well as per-packet metadata such as network namespace information. From the captured packet data structure, source port number (from TCP header) and namespace (from packet metadata) are extracted, and using the socket map above, are mapped to a corresponding PID. This PID can be used to construct a tag that represents a correct set of application contexts for the packet. Once a tag is ready, it can be added to the existing packet header as part of an IPv4 option or a TCP option, or appended as a trailer to an IPv4 packet to prevent the tag from being modified accidentally by intermediate switches or middleboxes, or added as part of encapsulation protocol headers (e.g., VxLAN, VLAN). In case of IPv6, the tag can be carried in the 20-bit flow label field.

3.4 Ingress Packet Processing

In order to verify each ingress packet with respect to the intended receiver's policies, we attach a separate eBPF program called *eVerifier* to the physical NIC interface. eVerifier performs per-packet verification in four steps: (i) extract a tag from an incoming packet, (ii) resolve the tag to the sender's contexts, (iii) look up the intended receiver's contexts, and (iv) finally perform verification based on those obtained contexts. As we will see, the design of eVerifier is inspired by the multi-level flow caching in OVS, where packets are processed using OpenFlow tables on slow path and megaflow/microflow caches on fast path. The major difference in eZTrust is that packets are classified into flows not based on packet header fields, *but based on microservice contexts.* This makes eZTrust intrinsically more scalable than network-endpoint-based perimeterization, as traffic from distinct microservice instances carrying the same contexts (e.g., due to auto-scaling) can be processed as a single flow.

3.4.1 Slow Path Processing. The step (ii) to resolve a tag to a sender's contexts requires that the local context map on the receiver side be already populated with the extracted tag **(R3)**. However, the local context map is not expected to contain tags for all existing microservices running in the data center, due to scalability concerns and resource constraints. Inspired by the flow cache design of OVS, we instead populate the local context map *on demand* from the global context map via *slow path*. During ingress packet processing, the slow path is triggered when a tag extracted from an incoming packet is missing in the local context map. On the slow path, eVerifier punts the packet to the *Policy Agent* in userspace, which then fetches the contexts for the tag from the central microservice coordinator. Once the obtained contexts are populated into the local context map, the Policy Agent re-inserts the packet to eVerifier for subsequent verification. To prevent a burst of packets carrying the same missing tag from entering the slow path during this time, we maintain a simple in-kernel status table maintaining a list of tags for which slow path processing is underway. Any packet that carries the tag stored in the status table is simply dropped without entering the slow path. In practice, communication patterns among deployed microservices are well-defined, and thus contexts are highly likely to be used repeatedly on the same microservices during their already short lifetime, thereby making slow path processing an uncommon event.

The step (iii) after tag resolution is to look up the intended receiver's contexts. For this, eVerifier looks up the local socket map using the key constructed from the destination port number of the packet and network namespace associated with the VIF, and finds out the recipient PID for the packet. This PID is used to construct a receiver's tag, which in turn is mapped to the receiver's contexts from the local context map. Unlike the sender's tag, the receiver's tag is guaranteed to exist in the local context map, and thus no slow path processing is necessary.

3.4.2 Per-Packet Verification. The final step (iv) is to perform packet verification based on a set of sender's contexts as well as receiver's contexts. For verification, we maintain a *policy map* which holds the perimeterization policies for individual microservices in the form of <*microservice id*, *sender's contexts*, *receiver's contexts*, *policy decision*> tuples. Any context field can be wildcarded in the policies, and possible policy decisions are "accept" or "drop". Finding a match for a packet based on a set of sender/receiver contexts in the policy map is the classic packet classification problem. The difference is that packets are classified, not based on packet header fields, but based on a set of contexts. There are many efficient algorithms for packet classification, and we adopt a scheme similar to the tuple space search classifier [29], commonly employed by popular software switches (e.g., megaflow cache in OVS). It is simpler than the original tuple space search as it does not need to handle the longest prefix searching, but only exact match.

In this scheme, we define a *policy template* for each microservice, which is an array of <*subset of sender's context keys*, *subset of receiver's context keys*> tuples. The policy template of a microservice indicates which subsets of sender-side/receiver-side contexts are used to define its policies. If a tenant installs multiple policies for her microservice, each based on distinct subsets of contexts, the policy template of the microservice would contain more than one tuples. For example, if two policies are defined for a microservice:

Algorithm 1 Procedure for generating a policy decision.

```
/* array of all available contexts */
struct context_t {
    uint32 context[MAX_CONTEXT]
}
/* each boolean field tells if a given context is considered in the policies */
struct template_t {
    bool srcContext[MAX_CONTEXT]
    bool dstContext[MAX_CONTEXT]
}
procedure GENERATE_POLICY_DECISION(src, dst, T)
input:  context_t src,     /* sender's contexts */
        context_t dst,     /* receiver's contexts */
        template_t [] T    /* array of receiver's policy templates */
output: ACCEPT or DROP
    for each t in T do
        uint32 key ← 0
        /* generate a policy key from template t */
        for i ← 0 to MAX_CONTEXT-1 do
            if t.srcContext[i] then
                key ← compute_hash (key, src.context[i])
            end if
            if t.dst[i] then
                key ← compute_hash (key, dst.context[i])
            end if
        end for
        /* look up policy map with the key */
        result ← lookup_policy_map (key)
        if result = null then        /* not found */
            return DROP
        end if
        return result        /* ACCEPT or DROP */
    end for
end procedure
```

"*accept traffic only if it originates from HAproxy and is destined to nginx*", and "*drop traffic if it is generated by an application located in US-West*", its policy template would look like: [<$appID_{src}$, $appID_{dst}$>, <$Location_{src}$>]. When eVerifier looks up the policy table for an incoming packet, it iterates over the policy template of the intended receiver, forms all possible keys to the policy table, and performs policy table lookup. Upon finding the first successful lookup, it stops the iteration, and returns the decision. In case policy prioritization needs to be supported, eVerifier can complete the full iteration and chooses the policy decision with the highest priority among multiple matches. With no match, a packet is processed based on a default action. The full procedure for step (iv) is described in Algorithm 1.

3.4.3 Dynamic Context Handling. So far in these packet verification steps, it is assumed that contexts remain unchanged both at the sender side and at the receiver side. However, contexts associated with a microservice can dynamically change for various reasons. For example, a microservice can be migrated geographically. Multi-user services such as remote desktop services or HPC applications can be accessed by different users at different time. In addition, mission critical production environments often benefit from dynamic software updates [30, 31], where any critical security patches or software upgrade are applied live without incurring downtime. This can affect the contexts (e.g., software version) associated with any active microservices.

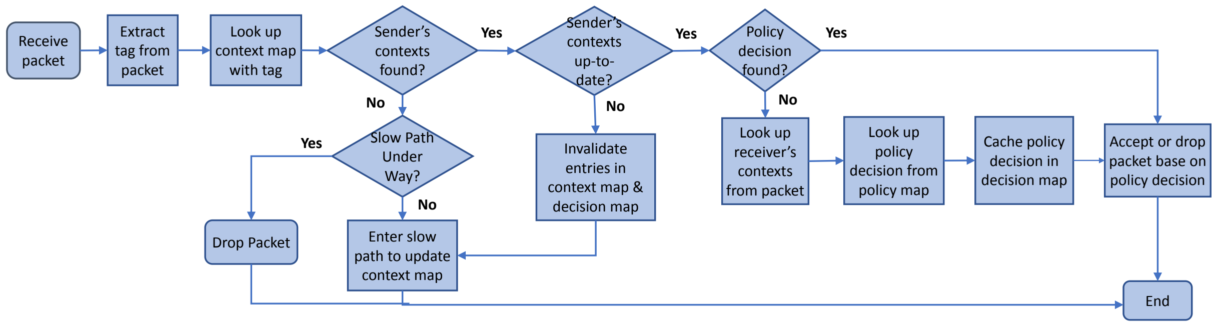

Receive packet
Extract tag from packet
Look up context map with tag
Sender's contexts found?
Yes
Sender's contexts up-to-date?
Yes
Policy decision found?
Yes
No
Slow Path Under Way?
Yes
No
No
Invalidate entries in context map & decision map
No
Look up receiver's contexts from packet
Look up policy decision from policy map
Cache policy decision in decision map
Accept or drop packet base on policy decision
Drop Packet
Enter slow path to update context map
End

Figure 2: eVerifier's packet verification procedure.

In order to detect and handle resulting potential context changes during the packet verification steps, we introduce the notion of an "epoch" in the contexts **(R4)**, which indicates the up-to-dateness of the detected contexts. An epoch is a simple counter that is incremented and wraps around when it reaches its maximum. The entries in the context map are now expanded to include an epoch: <*tag*, *a set of contexts*, ***epoch***>. Whenever any context is changed in any microservice, the corresponding entries in the context map have their epoch incremented. In addition, each egress packet carries not only a tag, but also its corresponding epoch. When a tag and its associated epoch are received by the other end, the receiver can detect whether the entry stored in the local context map for the tag is outdated or not by comparing the epoch in the entry against the received epoch. If the entry is detected as outdated, it is evicted from the context map, and the receiver goes through the slow path to re-populate the context map for the tag against the latest epoch.

Note that the entire per-packet verification procedure described so far requires multiple independent map lookups (i.e., context map, socket map, and policy maps), even without considering one-time slow path through userspace. As an optimization to speed up the multi-step verification operation **(R5)**, we cache the final policy decision obtained from the step (iv) in a separate table called *decision map*, which stores the mapping <*sender's tag*, *receiver's tag*, *policy decision*>. Subsequent packets with the same tags can be verified with a single lookup of the decision map. This design is somewhat similar to microflow caching in OVS. Whenever any sender-side context change is detected from the epoch of the received tag, the corresponding entries in the decision map are invalidated, and verification for the packet falls back to the original multi-step procedure. Any receiver-side context change also invalidates associated entries in the decision map. The overall packet verification steps are summarized as the flowchart in Fig. 2.

4 PROTOTYPE IMPLEMENTATION

We implement the eZTrust prototype in Python/C and integrate it with Docker runtime environment. In this section we highlight key implementation details of the prototype.

Context management. The userspace Context Manager is implemented in Python (600 LoC), using bcc library [32] to interact with eBPF-based eTracers, and Docker SDK [33] to listen on container events. Context Manager collects contexts of deployed containers either by attaching eTracers to `kprobes` and `uprobes`, or by invoking a harvesting routine in a target container's namespace. For example, eTracers attached to *sys_clone()* and *sys_execve()* derive the identity of an application process from the `md5sum` of its binary executable (for compiled applications) or its application bytecode (for interpreted applications such as Java/Python apps). The association between network sockets and application processes is found by tracing *inet_csk_accept()* and *tcp_v4_connect()* for TCP INET sockets. See Table 2 for a list of collected contexts. The obtained information is written as container identities to local eBPF maps (socket/context maps), as well as distributed to the global context map realized with Redis [34].

Policy enforcement. Its implementation is split between (1) the userspace Policy Agent written in Python (350 LoC) and (2) two in-kernel eBPF programs written in C; eTagger (70 LoC) and eVerifier (370 LoC). eTagger is attached to the ingress TC classifier [35] of each container's VIF for outgoing packet tagging, while eVerifier is added to the egress TC classifier of a physical NIC interface to verify incoming packets. For this prototype, we make use of the 12-bit VID field in VLAN header to carry per-packet tags (10 bits) and epoch (2 bits). eTagger and eVerifier use *bpf_skb_vlan_push()* and *bpf_skb_vlan_pop()* eBPF APIs to add or remove a VLAN header. Other types of encapsulation protocols (e.g., VxLAN, Geneve) are possible with eBPF [36] to support realistic large-scale deployments. The Policy Agent is responsible for slow path handling, receiving raw packets from eVerifier via `perf-event` interface, and re-inserting them via a `tap` interface, to which eVerifier is also attached. In eVerifier, which implements Algorithm 1 in iteration loops, we

Collector	Context	Method
eTracers	AppID, App version	Trace *sys_clone()* and *sys_execve()* using `kprobes`.
	Operating system userID	Use *bpf_get_current_uid_gid()* in the process context captured above.
	TCP socket	Trace *inet_csk_accept()*, *tcp_v4_connect()* and *tcp_v6_connect()* using `kprobes`.
	SSL version	Trace *SSL_do_handshake()* and its arguments using `uprobes`.
	MySQL user	Trace *connection__start()* and its arguments using `uprobes`.
Harvester	Microservice ID, geographic location, filesystem image tag and signer's identity (for a digitally signed image), capabilities	Query the microservice orchestrator.
	Kernel version	Probe the host operating system with *sys_uname()*.
	OpenSSL version	Search for `SSLEAY_VERSION` string in the OpenSSL library.

Table 2: Microservice context collection.

leverage eBPF tail calls to get around eBPF's 4K bytecode size limit, which helps increase maximum template count from 20 to 200.

Slow path processing. During slow path, the Policy Agent interacts with the global context map to retrieve up-to-date contexts of a remote container. However, this northbound interaction is complicated by potential race conditions. Suppose a remote container C_{remote} is newly launched, immediately opening a TCP connection to a local container C_{local}. Then by the time C_{local} initiates slow path processing for the first SYN packet from C_{remote}, the global context map may or may not have been populated with the contexts of C_{remote}. In the latter case, the first SYN packet would be rejected, causing TCP timeout and significantly delaying TCP connection establishment. A similar race condition can occur when C_{remote} changes its context; by the time C_{local} sees epoch change from C_{remote}, the global map may or may not have updated new contexts for C_{remote}. In the latter case, C_{local} would fail to detect context change in C_{remote}. To avoid the first race condition (due to traffic from a new container), we perform global map lookup *upto N times* in case of lookup failure. We observe that global map lookup most often succeeds with $N = 2$ for new containers. To avoid the second race condition (due to traffic with epoch change), we perform *staged* epoch update on the sender-side as follows. Whenever a context change is detected on the sender, we update the global context map with the new context, but *without* incrementing the epoch at this point. Only after the new context is successfully committed to the global map, do we increment the epoch. That way, a receiver will be guaranteed to obtain the updated context with an epoch change.

5 MOTIVATIONAL USE CASES

In this section, we describe a few practical use case scenarios that can be enabled by the eZTrust prototype.

Vulnerability-driven perimeterization. Although unpatched software vulnerabilities are a common source of security breaches, software patches are often neglected due to other pressing tasks or postponed for integrity testing [37, 38]. Official software patches may not even be available at the time of zero-day attacks. To minimize potential damage while critical software patches are phased in, a data center operator can use eZTrust to quickly deploy data center wide contingency policies, where traffic to vulnerable application binaries is either blocked or alerted depending on tenant requirements. Note that alternative container image scanning approaches (e.g., Clair [39]) are not only time-consuming but also insufficient due to live container updates [40].

Control flow integrity. In a distributed microservice architecture, interdependencies of microservices can be highly complex (see a sock shop example in Fig. 8). On top of that, each microservice can scale in and out independently. This makes the network-endpoint-based regulation of cross-service interaction extremely challenging. On the other hand, eZTrust makes it easy to express policies for acceptable control flows solely based on microservice identities. As the identities are derived from the fingerprint of application executables or bytecode (e.g., JAR), such policies remain unchanged with microservice auto-scaling.

User identity based firewall. Consider a remote desktop service deployment, where multiple users log in to the same remote desktop frontend service, and from there access different backend services (e.g., read documents hosted in a remote file storage service, or open a remote SSH terminal). In such an environment, eZTrust can enable user identity base perimeterization, where remote desktop traffic generated by different users' login shells is tagged differently, so that the traffic is selectively allowed or blocked at different backend services. This network-independent approach does not require complex user-to-IP address mapping like other commercial firewall solutions [10].

6 EVALUATION

In this section, we evaluate the eZTrust prototype implementation to answer the following key questions.

- What is the overhead of eZTrust's packet processing on slow path and fast path, and what is the impact of policy complexity on the overhead?
- How does eZTrust compare against alternative schemes in terms of performance and overhead?
- Does eZTrust work correctly as expected in dynamic environments?

Our testbed is set up on CloudLab [41], and encompasses two machines with dual Intel Xeon E5-2640 v4 processors (10 cores at 2.40GHz) and 64GB DDR4 memory. The two machines are back-to-back connected via 10G NIC interface (Mellanox ConnectX-4 LX), each running on Ubuntu 18.04.1 LTS, kernel 4.15.0, with eBPF JIT flag enabled.

6.1 Packet Latency

Slow path vs. fast path. eZTrust adopts dual-path (slow/-fast path) ingress packet processing to speed up packet verification in a resource efficient fashion (Section 3.4.1). In the first experiment, we evaluate the implication of this design as the complexity of policies varies. We deploy two containers across two back-to-back connected servers, install appID-based perimeterization policies for each container, and measure network latency between them in two separate experiments. In one experiment, we measure round trip delay by launching a dummy container, which opens a one-time TCP connection to the other container and reports connection establishment delay incurred from TCP SYN/ACK exchange. The initial TCP SYN/ACK packets go through slow path as the local context map on either server is not yet populated for the other remote container. We launch the dummy container multiple times to obtain average connection establishment latency. In the other experiment, we deploy `netperf` (in TCP_RR mode) on both containers, which reports average round trip latency by generating multiple request/response transactions over a single long-lived TCP connection. The `netperf` traffic in this case is handled via fast path.

Fig. 3 shows the packet latencies of slow path and fast path as we adjust the number of installed policy templates, which represents the *complexity* of policies. A higher number of templates imply that more diverse policies (based on different contexts) are installed. In case of slow path, packet latency increases with the number of policy templates because more iterations are required for policy map lookup (see Algorithm 1). Note that multiple polices based on the *same* template still do not increase slow path delay due to constant time policy map lookup. In case of fast path, packet latency is not affected by policy templates as the policy decision for an initial packet is cached in the decision map, and subsequent packets carrying the same contexts can be verified with a single decision map lookup.

Fast path packet latency. Next, we evaluate the packet latency of eZTrust on fast path. Using the same `netperf`-based deployment as previously explained, we measure average round trip latency, with and without perimeterization policy. As suggested in the previous experiment, the overhead of ingress packet processing on fast path in eZTrust does not depend on the number of policy templates nor the

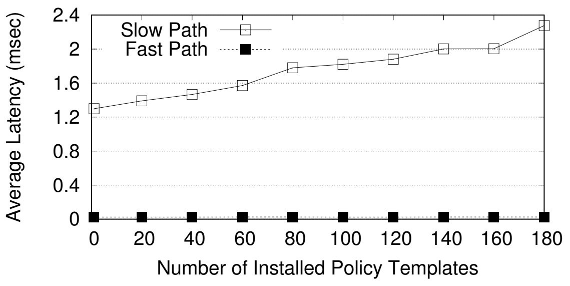


Figure 3: Latency: slow path vs. fast path.

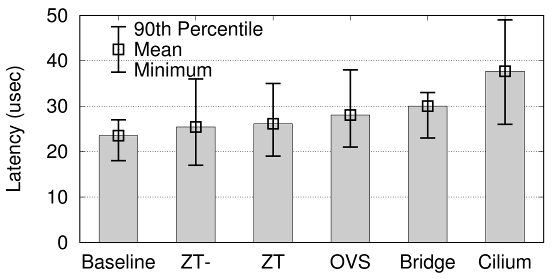


Figure 4: Fast path packet latency. "Baseline" indicates server-to-server latency. "ZT" and "ZT-" refer to eZTrust with and without policy rules, respectively.

number of policies installed for each template. For perimeterization, we set up a simple application-aware policy (i.e., accept traffic if $appID_{src}$=ID_NETPERF and $appID_{dst}$= ID_-NETPERF). In case of no policy, we disable tagging and verification in eZTrust, and let it forward traffic using the eBPF's packet redirection capability. We compare policy-enabled eZTrust against other alternative schemes: (1) OVS (v2.10.0) with flow rules, (2) Linux bridge with `iptables` rules, and (3) Cilium (v1.2.4 in native routing mode[2]) with label-based rules. In OVS deployment, we prepend a flow table at the ingress pipeline, in which we add flow rules allowing only `netperf` traffic between the two containers. In Linux bridge setup, we use the `physdev` extension of `iptables` to define similar filtering rules on individual ports of the bridge. In Cilium deployment, we configure policies based on two labels (e.g, accept traffic if $label_{src}$="joe" and $label_{dst}$="alice").

From Fig. 4, we make two observations. First, eZTrust's end-to-end rule processing adds only marginal latency overhead to the baseline forwarding (comparing second/third bars). In fact, we observe similar latency overhead in the rule enforcement of the other alternative schemes (not plotted here). This means that eZTrust's per-packet tagging and fast path verification are *not* any more heavy duty than other schemes. When baseline server-to-server latency is discounted from end-to-end latency, eZTrust incurs a factor

[2]Cilium's container network plugin interconnects containers across multiple servers via either native routing or VxLAN overlay. Native routing is chosen for fair comparison.

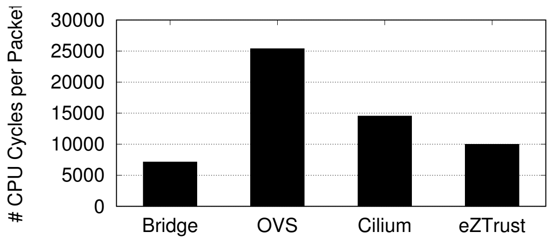


Figure 5: Per-packet CPU usage for policy processing.

of 2–5 lower latency on fast path than the other schemes. We believe that this latency advantage mainly comes from eBPF-based packet forwarding, which was also reported in [42].[3]

6.2 CPU Resource Usage

Per-packet CPU resource overhead. Next, we shift focus to the CPU overhead. Here we are interested in the *per-packet CPU overhead of perimeterization*, which captures the CPU usage incurred by policy enforcement only, but discounts packet forwarding overhead. For this, we measure the difference in CPU usage of eZTrust with and without policies. In this experiment, we deploy two containers on one server, pin them to fixed CPU cores, and generate 60-byte UDP packets between them bidirectionally using `iperf`, in a fixed packet-per-second rate ($\mathcal{R}$) for a fixed time period ($\mathcal{T}$). During $\mathcal{T}$, we count the total number of CPU cycles (C) incurred on the server using `perf-stat`. For each scheme (eZTrust, Linux bridge, OVS, Cilium), we repeat this experiment with and without perimeterization policies, and obtain two counters (C_1 and C_2), respectively. The average per-packet CPU overhead for perimeterization ($C_{\mathcal{P}}$) is obtained from $\frac{C_1 - C_2}{\mathcal{R} \cdot \mathcal{T}}$. We use the same perimeterization policies used in the previous experiment except that `netperf` is replaced with `iperf`. Fig. 5 compares $C_{\mathcal{P}}$ for different schemes with $\mathcal{T}$ being 1 min. It shows that Linux bridge with `iptables` rules exhibits the lowest CPU overhead. However, `iptables`-based perimeterization is already well known for its inability to support a large number of rules due to sequential processing [43]. Barring Linux bridge, the other schemes do not suffer from this problem as they rely on hash table lookups for perimeterization. eZTrust is the most CPU-efficient among them (by a factor of 1.5–2.5).

DPI-based vs. eZTrust. Next, we compare eZTrust against DPI-based context-aware perimeterization in terms of CPU efficiency. In this experiment, we consider a perimeterization policy that protects against the Heartbleed vulnerability [44], which targets particular versions of the OpenSSL cryptographic library. We run two containers, one `nginx`/https server, and the other `curl` client, deployed across two hosts.

[3]Compared to [42], we achieve further speedup in eBPF-based forwarding by avoiding packet cloning.

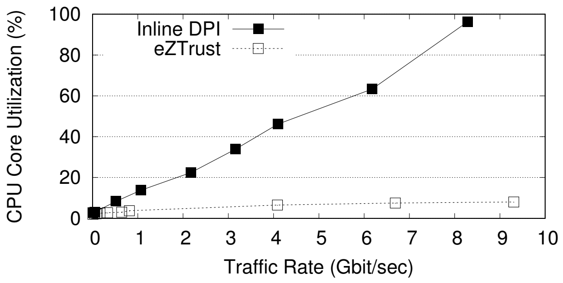


Figure 6: Protection against Heartbleed vulnerability: DPI-based vs. eZTrust.

As DPI-based perimeterization, we set up between them inline mode Snort [45] with the officially vetted Heartbleed Snort signature [46] as the only rule loaded. In case of eZTrust, we configure on both ends policies based on four contexts $< appID_{src}, openSSLVersion_{src}, appID_{dst}, openSSLVersion_{dst} >$, which are detected as described in Table 2. We install multiple rules allowing traffic between `nginx` and `curl` with Heartbleed-safe OpenSSL versions only (e.g., 1.0.1g through 1.1.2). Fig. 6 plots the CPU utilization of two approaches as a function of injected traffic rate. We measure CPU utilization on the host where `nginx` is running. In case of DPI approach, Snort is deployed on this host as well. The reported CPU utilization on y-axis is the host-wide CPU usage minus the CPU load attributed to `nginx` processes, thus capturing perimeterization overhead only. The figure shows that compared to DPI-based approach, eZTrust can achieve OpenSSL-based policy control with a factor of 8 to 10 smaller CPU overhead. This experiment illustrates the high CPU efficiency of eZTrust to support context-aware policies.

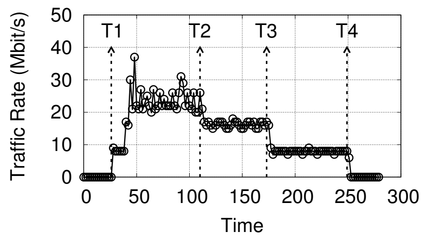
6.3 Dynamic Policies and Contexts

The evaluations thus far focus on static environments where policies and contexts remain fixed. In the next experiment, we consider dynamic deployment environments where policies or contexts change over time, and demonstrate that eZTrust can handle such dynamic environments correctly. In scenario #1, contexts remain fixed while policies are updated by tenants. In scenarios #2 and #2b, policies remain unchanged while contexts are altered at runtime. For latter scenarios, we define a hypothetical context called *status* (maintained by the data center operator) to indicate the health of a microservice. Table 3 describes the timeline of events we introduce in these scenarios.

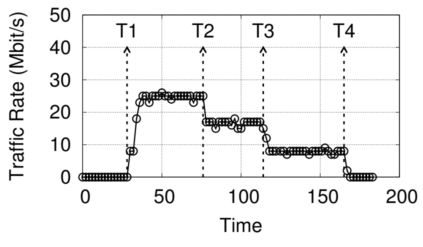
Fig. 7 shows timing diagrams indicating how the total transfer rate or HTTP response delay changes as a result of these events. The T2/T3 events in scenario #1 introduce policy changes on `nginx` servers. This in turn instructs the policy agent to update the local policy map and invalidate the decision map, so that subsequent packets can be re-inspected according to the updated policy map. This has the effect of

Scenario #1: dynamic policies for nginx container	**Scenario #2: dynamic contexts for wget container**	**Scenario #2b: dynamic contexts for web client**
T1: Install policies allowing traffic from `wget` & `curl` with OpenSSL version `X`.	**T1:** Install policies allowing traffic from `wget` with status `HEALTHY`.	**T1:** Install policies allowing traffic from client with status `HEALTHY` or `PATCHED`.
T2: Remove the `curl` policy.	**T2:** Change the status of `wget` container 1 to `COMPROMISED`.	**T2:** Change the status of client from `HEALTHY` to `PATCHED`.
T3: Change the `wget` policy to allow traffic from `wget` with OpenSSL version `Y`.	**T3:** Change the status of `wget` container 2 to `COMPROMISED`.	**T3:** Change the status of client from `PATCHED` to `HEALTHY`.
T4: Remove the `wget` policy.	**T4:** Change the status of `wget` container 3 to `COMPROMISED`.	

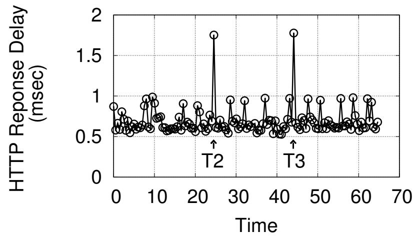
Table 3: Timeline of experimental scenarios. In scenario #1, two `wget` clients and one `curl` client on host1 download files from three `nginx`/https servers on host2 with 1MByte/sec rate limit respectively. In scenario #2, three `wget` clients on host1 download files from three `nginx`/https server on host2 with 1MByte/sec rate limit respectively. In scenario #2b, a web client on host1 fetches index.html from `nginx` server on host2 periodically for a delay measurement purpose.



(a) Scenario #1



(b) Scenario #2



(c) Scenario #2b

Figure 7: eZTrust in action in dynamic environments.

blocking further traffic to `curl` clients due to bidirectional nature of TCP. The T2/T3/T4 events in scenario #2 indicate the change in *status* context in `wget` clients one by one, which causes the monitoring agent on `wget` clients to increment the epoch for `wget`'s contexts accordingly. These epoch updates are then detected by eVerifier on `nginx` servers, which in turn invalidates the local context map as well as decision map for `wget` clients, and triggers slow path. As slow path is completed for each `wget` client, context/decision maps for `nginx` server side get fully populated, eventually blocking further traffic to the corresponding `wget` client. Finally, the slow path triggered at T2/T3 in the scenario #2b affects user-perceived HTTP response delay. The observed increase in delay is consistent with slow path latency shown in Table 3. The number of policy templates used in this setup is 60.

6.4 Real-World Application: Sock Shop

In the final experiment, we deploy a real-word microservices-based application on eZTrust. We choose the Sock Shop [47], a distributed e-commerce demo application composed of 14 different microservices. The control flow among these microservices is visualized in Fig. 8. In eZTrust, we set up microservice-aware policies based on this control flow. As a comparison, we also deploy the Sock Shop application in OVS-based and Cilium-based perimeterization environments, with equivalent flow-rule-based and label-based policies, respectively. Using `Locust`-base load generator, we inject identical workloads (e.g., retrieving product pages, posting orders, accessing shopping cart, etc.) in three deployments, and compare user-perceived end-to-end latencies. Fig. 9 plots end-to-end latencies for several Sock Shop APIs. Compared to OVS and Cilium, eZTrust reduces the latencies by 3–6% and 5–15%, respectively. These reduced end-to-end latencies in eZTrust are attributed to its lower packet processing latency previously shown in Fig. 4.

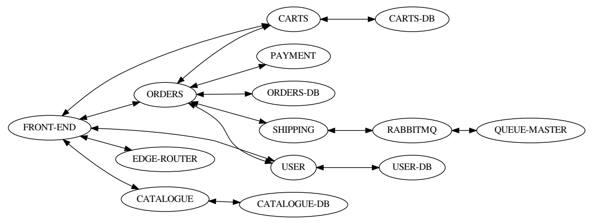


Figure 8: Microservice control flow in Sock Shop.

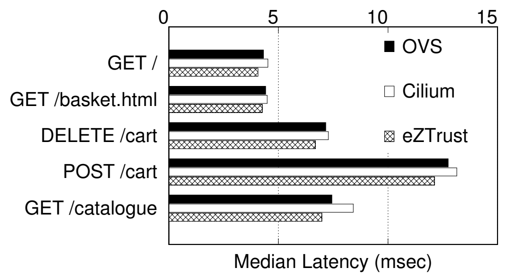


Figure 9: End-to-end latencies of Sock Shop.

7 DISCUSSION

In this section, we discuss possible extensions of eZTrust.

Tag granularity. In the current design, per-packet tags are instantiated at the process granularity (i.e., distinct tags per process). Alternatively, tags could be defined at coarser (microservice/app-level) or finer (transport connection-level) granularities. The implication of varying tag granularities is two fold. On one hand, coarser-grained tags would not support fine grained policies based on detailed contexts. For example, per-microservice/per-app tags would not support policies that regulate remote desktop traffic generated by different login shells. On the other hand, finer-grained tags would incur more frequent slow path processing. For example, with connection-level tags, which can carry session-level contexts (e.g., user context per database session), every single connection opened by a process would now trigger slow path processing on the receive end. In a sense, the epoch adopted by eZTrust can be considered a way to encode finer-grained contexts than process-level attributes at the cost of additional slow path processing.

Given this inherent trade-off in tag granularity, an alternative way to improve policy granularity while minimizing slow path handling is to introduce *prefix matching* in the context map lookup. With prefix matching, tag space is no longer flat but hierarchically defined (e.g,. delineated into microservice ID, process ID and port number fields), and the context map will contain wildcarded tags as keys. Then depending on the granularity of contexts that is needed for policy enforcement, the Context Manager can push appropriately wildcarded tags into context map during slow path processing, so that any subsequent traffic from the same microservice can avoid the slow path. Prefix match for eBPF maps is already supported (`BPF_MAP_TYPE_LPM_TRIE`).

Tag anonymization. We assume that a generated tag is placed in a well-known packet header field as plaintext. This can be justified if tenant microservices communicate with one another only through the end servers' network stack, which is controlled by the infrastructure, and is not compromised according to our threat model. In other words, tenant microservices cannot artificially inject or modify the tags in their traffic (e.g., by using raw sockets to bypass the network stack). One way to prevent such tag forgery or impersonation is to deny tenant microservices access to raw sockets as an infrastructure-wide policy. This is in fact one of the standard microservice security practices recommended to prevent packet spoofing [48]. If such restriction is not an option for any reason, one can *anonymize* the tags using traditional approaches such as shared secret based symmetric encryption [49]. Note that in this case, not the entire packet payload, but only the small tag needs to be encrypted. Such tag encryption/decryption can be done efficiently with modern SIMD instruction sets (e.g., SSE/SSE2, AVX/AVX2). Secrets can be shared with the existing orchestrators' secret management and distribution interfaces [50, 51].

Smart NIC offload. In order to minimize performance overhead introduced by per-packet operations for tagging/verification and possible encryption/decryption, one can leverage smart NICs. As eBPF is embraced as a mainline kernel feature, next-generation smart NICs (e.g., Netronome Agilio [52]) have already started to support eBPF offload, allowing unmodified eBPF programs along with maps to be transparently offloaded to the NICs [53]. As of this writing, however, eBPF offload is still experimental, supporting only ingress packet processing and a limited set of eBPF helper APIs. For example, slow path handling via `perf-event` interface cannot be offloaded. We plan to explore full potential of eBPF offload as the support improves.

Platform compatibility. The network-independent approach taken by eZTrust does not necessarily make it incompatible with the existing network-endpoint-based perimeterization. The extension of eZTrust to support network-endpoint-based polices is straightforward by treating packet header fields as additional application contexts and enabling prefix match in policy map lookup. Alternatively, if context-based and network-endpoint-based policies are not intertwined with complex priorities, one can separate out these two policy sets, and deploy a hybrid solution, where eZTrust co-exists with network-endpoint-based perimeterization, and the former is only responsible for context-aware policies at the first ingress point in a bump-in-the-wire fashion. Similar hybrid solutions are also possible with other complementary approaches. For example, eZTrust can be deployed in tandem with API gateways for additional API-level access control.

8 CONCLUSION

In this paper, we present eZTrust, a network-independent perimeterization solution for microservices, where we shift perimeterization targets from network endpoints to fine-grained, context-rich microservice identities. To this end, we tap into the growing wealth of tracing data of microservices made available by eBPF, and repurpose them for perimeterization. While doing so, 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. As such, we believe that eZTrust can further benefit from advances in microservice tracing technologies as well as ongoing software switch development efforts.

Acknowledgements. We would like to thank the anonymous SOSR reviewers, Robert Ricci and our shepherd, Eric Keller, for their feedback on earlier versions of this paper. This work is supported in part by the National Science Foundation under grant number 1642158.

REFERENCES

[1] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.

[2] 2018. Cisco Global Cloud Index: Forecast and Methodology 2016–2021. White Paper. Cisco Systems, Inc. (2018).

[3] Doug Barth and Evan Gilman. 2017. *Zero Trust Networks*. O'Reilly Media, Inc.

[4] Justin Pettit, Jesse Gross, Ben Pfaff, Martin Casado, and Simon Crosby. 2010. Virtual Switching in an Era of Advanced Edges. In *Proc. DC CAVES Workshop*.

[5] 2018. VMware NSX. http://www.vmware.com/products/nsx.html. (2018).

[6] 2018. OVSDB:Security Groups - OpenDaylight Project. https://wiki.opendaylight.org/view/OVSDB:Security_Groups. (2018).

[7] Cheng Jin, Abhinav Srivastava, and Zhi-Li Zhang. 2016. Understanding Security Group Usage in a Public IaaS Cloud. In *Proc. IEEE INFOCOM*.

[8] Bingwei Liu, Yinan Lin, and Yu Chen. 2016. Quantitative Workload Analysis and Prediction using Google Cluster Traces. In *Proc. IEEE INFOCOM Workshop on Big Data Sciences, Technologies and Applications*.

[9] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. 2013. Scalable Rule Management for Data Centers. In *Proc. USENIX NSDI*.

[10] Stijn Vanveerdeghem. 2018. VMware NSX - Context-Aware Micro-segmentation. https://blogs.vmware.com/networkvirtualization/2018/02/context-aware-micro-segmentation-innovative-approach-application-user-identity-firewall.html. (2018).

[11] Roie Ben Haim. 2016. NSX Identity Firewall - Deep Dive. http://www.routetocloud.com/2016/11/nsx-identity-firewall-deep-dive/. (2016).

[12] 2018. Sysdig. https://sysdig.com. (2018).

[13] 2018. Prometheus. https://prometheus.io. (2018).

[14] 2018. Lumogon. https://github.com/puppetlabs/lumogon. (2018).

[15] 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/. (2017).

[16] Ben Pfaff et al. 2015. The Design and Implementation of Open vSwitch. In *Proc. USENIX NSDI*.

[17] Casimer DeCusatis, Piradon Liengtiraphan, Anthony Sager, and Mark Pinelli. 2016. Implementing Zero Trust Cloud Networks with Transport Access Control and First Packet Authentication. In *Proc. IEEE International Conference on Smart Cloud*.

[18] 2018. Trireme. https://github.com/aporeto-inc/trireme-lib. (2018).

[19] Adam Langley et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proc. ACM SIGCOMM*.

[20] Oliver Zheng, Jason Poon, and Konstantin Beznosov. 2009. Application-based TCP hijacking. In *Proc. ACM European Workshop on System Security*.

[21] 2018. Cilium. https://cilium.io. (2018).

[22] 2018. vArmour DSS Distributed Security System. https://www.varmour.com/pdf/data-sheet/vArmour-DSS-Data-Sheet.pdf. (2018).

[23] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and T.V. Lakshman. 2014. Application-aware Data Plane Processing in SDN. In *Proc. ACM HotSDN*.

[24] Chris Richardson and Floyd Smith. 2016. Microservices: From Design to Deployment. Nginx, Inc. (2016).

[25] 2018. Istio. https://istio.io. (2018).

[26] 2018. Consul. https://www.consul.io. (2018).

[27] Oliver Michel and Eric Keller. 2016. Policy Routing using Process-Level Identifiers. In *Proc. IEEE International Symposium on Software Defined Systems*.

[28] Luigi Catuogno and Clemente Galdi. 2015. Ensuring Application Integrity: A Survey on Techniques and Tools. In *Proc. International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*.

[29] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification using Tuple Space Search. In *Proc. ACM SIGCOMM*.

[30] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2014. Kitsune: Efficient, General-Purpose Dynamic Software Updating for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36, 4.

[31] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, and Taesoo Kim. 2016. Instant OS Updates via Userspace Checkpoint-and-Restart. In *Proc. USENIX ATC*.

[32] 2015. IO visor bcc. https://github.com/iovisor/bcc. (2015).

[33] 2018. Docker-SDK. https://docker-py.readthedocs.io/en/stable/. (2018).

[34] 2018. Redis. https://redis.io. (2018).

[35] Daniel Borkmann. 2016. On getting tc classifier fully programmable with cls bpf. In *Proc. NetDev 1.1*.

[36] Daniel Borkmann. 2016. Advanced programmability and recent updates with tc's cls bpf. In *Proc. NetDev 1.2*.

[37] Eric Rescorla. 2003. Security holes... Who cares? In *Proc. USENIX Security Symposium*.

[38] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. 2002. Timing the Application of Security Patches for Optimal Uptime. In *Proc. USENIX LISA*.

[39] 2018. Clair: Vulnerability Static Analysis for Containers. https://github.com/coreos/clair/. (2018).

[40] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. 2017. Understanding Security Implications of Using Containers in the Cloud. In *Proc. USENIX ATC*.

[41] 2018. CloudLab. https://cloudlab.us. (2018).

[42] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: In-Kernel Distributed Network Virtualization for DCN. *ACM SIGCOMM Computer Communication Review*, 46, 3.

[43] Thomas Graf. 2018. Why is the Kernel Community Replacing Iptables with BPF? https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/. (2018).

[44] 2017. The Heartbleed Bug. http://heartbleed.com. (2017).

[45] 2018. Snort. https://snort.org. (2018).

[46] 2014. FBI Snort Signatures (Heartbleed). https://ics-cert.us-cert.gov/UPDATE-FBI-Snort-Signatures-Heartbleed-April-2014. (2014).

[47] 2017. Sock Shop – A Microservices Demo Application. https://microservices-demo.github.io. (2017).

[48] 2018. Docker Security. https://docs.docker.com/engine/security/security/. (2018).

[49] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. 2017. Report on Lightweight Cryptography. https://doi.org/10.6028/NIST.IR.8114. NIST. (2017).

[50] Ying Li. 2017. Introducing Docker Secrets Management. https://blog.docker.com/2017/02/docker-secrets-management/. (2017).

[51] 2018. Distribute Credentials Securely Using Secrets. https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/. (2018).

[52] 2018. Netronome Agilio CX. https://www.netronome.com/products/agilio-cx/. (2018).

[53] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. In *Proc. NetDev 1.2*.