IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 October 2023, accepted 31 October 2023, date of publication 7 November 2023, date of current version 13 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3330706

RESEARCH ARTICLE

ZTSFC: A Service Function Chaining-Enabled Zero Trust Architecture

LEONARD BRADATSCH[1], OLEKSANDR MIROSHKIN[2], AND FRANK KARGL[1], (Member, IEEE)
[1]Institute of Distributed Systems, Ulm University, 89077 Ulm, Germany
[2]Communication and Information Centre, Ulm University, 89077 Ulm, Germany

Corresponding author: Leonard Bradatsch (leonard.bradatsch@uni-ulm.de)

This work was supported in part by the Ministry of Science, Research and the Arts Baden-Wuerttemberg (MWK) through the bwNET2020+ Project, and in part by the Open Access Publishing Fund of the Ulm University.

ABSTRACT Recently, zero trust security has received notable attention in the security community. However, while many networks use monitoring and security functions like firewalls, their integration in the design of zero trust architectures remains largely unaddressed. In this article, we contribute with respect to this aspect a novel network security architecture called *Zero Trust Service Function Chaining* (ZTSFC). With ZTSFC, we achieve three main improvements over zero trust architectures: (1) the zero trust components can directly integrate other monitoring and security functions into their access decisions, (2) an efficient flow of information between zero trust components, monitoring, and security functions are achieved, and (3) ZTSFC improves the performance with respect to hardware load and user experience. As proof of concept, we implemented a publicly available ZTSFC prototype based on HTTPS and the policy language ALFA. Using this prototype, we demonstrate the achievement of all three improvements in representative use cases. In addition, our performance evaluation compares ZTSFC with a regular zero trust network without ZTSFC. The results indicate that ZTSFC can reduce CPU usage by 25% for specific monitoring and security functions in certain scenarios. Overall, we also observed a 30% decrease in the time it takes to access services with ZTSFC.

INDEX TERMS Network performance, network security, zero trust, access control, service function chaining.

I. INTRODUCTION

Ever since Google published their Zero Trust (ZT) approach *BeyondCorp* [2], ZT has been a much-discussed topic in network security. ZT is motivated by the shortcomings of perimeter security, which arise from its design assumption that valid resource accesses originate from an internal trusted network only and attacks always come from the outside. The limitations of perimeter security became apparent with issues like insider attacks and the growing trends of working from home and *bring your own device* (BYOD) policies [3]. These challenges introduce new ways resources are accessed, showing that the original design idea behind perimeter security is now outdated. One of ZT's core measures to address the shortcomings of perimeter security is to strictly enforce authentication and trust-based authorization (Auth*) for all resource access requests (RAR). These Auth* procedures are executed no matter who sends the RARs or where they are sent from. The foundational components of a ZT architecture typically comprise the policy enforcement point (PEP) located in the data plane, and the policy decision point (PDP) positioned in the control plane. They are responsible for the Auth* decisions where the PEP enforces the decisions made by the PDP in-line [4]. Such an Auth* decision can either result in allowed access, in which case the PEP forwards the RAR to the resource, or denied access, in which case the PEP rejects the RAR. Figure 1 shows a simple ZT architecture commonly used in the literature [3], [4], [5] that is composed of the components mentioned.

Additionally to strictly enforced Auth* decisions, the ZT paradigm requires the logging and inspection of all network traffic [6], [7]. In most ZT architectures, the PEP and PDP

The associate editor coordinating the review of this manuscript and approving it for publication was Aneel Rahim.

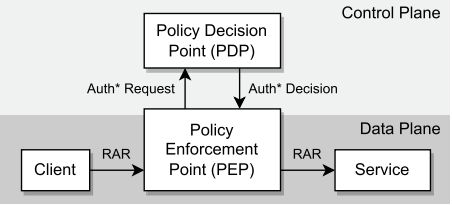

FIGURE 1. Example ZT architecture adapted from NIST [4] that consists of the basic ZT components PEP and PDP separated by a control plane and a data plane.

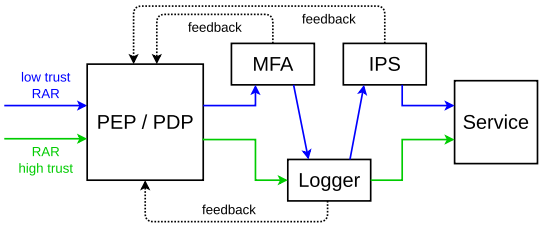

FIGURE 2. ZTSFC conceptual idea, which demonstrates traffic steering by the ZT components PEP and PDP based on trust.

alone do not provide all these features. Therefore, to fulfill the ZT paradigm, one needs to deploy additional monitoring and security (MaS) functions. These are for example intrusion prevention systems (IPS) for deep packet inspections (DPI), multi factor authentication systems (MFA) for prompting the client for additional authentication factors, or loggers for packet logging in the network [6]. However, as shown in Figure 1, most ZT architectures do not consider MaS functions in their architectural design. As we will discuss in detail in Section III, the integration of MaS functions into ZT networks for efficient and synergistic operation remains often unaddressed.

In this article, we introduce a novel ZT network architecture named *Zero Trust Service Function Chaining* (ZTSFC) that addresses the aforementioned lack of integration between ZT components and MaS functions. ZTSFC enables a dynamic and highly flexible orchestration and composition of MaS functions by ZT components. To achieve this, we combine the concept of ZT with service function chaining (SFC) [8]. SFC is a technique to dynamically steer traffic through a set of service functions, which can be any type of MaS functions. In addition, it is possible to instruct MaS functions to provide feedback to the PEP/PDP about the packets they process, such as suspicious findings in a packet header. Utilizing SFC, the ZT components have fine-grained control over how traffic is steered through the MaS functions and select the appropriate feedback, based on the level of trust in the RAR. This is conceptually illustrated in Figure 2. By this, we enable the ZT components to dynamically and fine-granularly decide when and to which traffic MaS functions are applied and which feedback they expect from the MaS functions. We presented initial concepts of the ZTSFC idea in [9]. In this article, we now present the refined concept, a detailed architecture, and evaluation of benefits and performance impact.

With ZTSFC we claim the following three main improvements over common ZT architectures:

- **Improvement 1 - The integration of MaS functions into the PDP's Auth* decision-making process:** Based on the established trust in the RAR, the PDP can obligate a set of MaS functions for this RAR and the corresponding response. Assuming a client initiating a RAR uses a managed device. Such managed devices typically offer strong device authentication and host-based intrusion detection systems with on-device packet inspection [10]. Together with other attributes, like the client's usual access time, this results in a high trust level. That leads to the RAR being permitted. In contrast, if the same client uses an unmanaged device, it lacks one authentication factor and on-device packet inspection, resulting in a lower trust level. With ZTSFC, utilizing MFA and IPS functions can make up for this trust deficit. The MFA might request an additional factor, such as a portable authentication token, to compensate for the absent device authentication. The IPS could perform a DPI on the RAR's packets to account for the missing on-device inspection. Thus, the PDP can make permitted access dependent on both IPS and MFA, steering the RAR's packets through these two MaS functions. By this, for example, necessary resource access for users following a BYOD policy can be permitted while ensuring security. As discussed in Section III, such a mechanism is hard to implement in a ZT architecture without MaS integration, while ZTSFC can accomodate this in a straight forward way.
- **Improvement 2 - The implementation of a direct and efficient information flow between MaS functions and PDP:** As previously noted, the Auth* decisions depend on the degree of trust placed in the RARs. This trust derives from various contextual information about the RAR. Such information encompasses details about the client that sent the RAR, such as its location and typical access times, as well as data like the protocol version used for the RAR [4]. A logger typically obtains the latter. Using ZTSFC, the PDP can steer packets through a logger function to collect such information, which the PDP subsequently uses for future RARs from the same source. This method establishes a direct flow of contextual information between MaS functions and PDP. As we will outline in Section III, how to ensure such an information flow in ZT architectures is mostly unaddressed so far.
- **Improvement 3 - Reduction of hardware load and improvement of user experience:** MaS functions such as an IPS are typically positioned at network choke points, where they process all packets that pass through them according to pre-configured settings. However, security features, like DPI, demand

significant computational resources. With the application of ZTSFC, the PDP can make dynamic decisions regarding traffic steering through specific MaS functions based on trust levels. As a result, if a high level of trust in a RAR is already established, certain resource-intensive functionalities, such as DPI, can potentially be saved. The IPS would then be reserved for RARs that do not meet a specified trust threshold. This approach can reduce the strain on the IPS. Concurrently, saving the IPS lowers the access times. That leads to an improved user experience. As we will discuss in Section III, this can reduce the hardware load while maintaining security considerations.

Contributions: Our goal in this article is to address the missing integration of MaS functions in existing ZT architectures. To this end, we make three key contributions:

- We describe in detail our novel ZT network architecture called ZTSFC.
- We provide a publicly available ZTSFC proof of concept based on an HTTPS approach.
- We discuss with detailed use cases how improvement one and two are achieved by ZTSFC. Regarding improvement three, we conducted a detailed performance analysis regarding the saved hardware load and increased user experience.

The remainder of the article is structured as follows: In Section II we provide the necessary technical background. Section III motivates the improvements based on the state of the art. Section IV describes the ZTSFC architecture. Subsequently, we introduce in Section VI a ZTSFC proof of concept based on an HTTPS SFC approach described in Section V. Based on this proof of concept, we present the implementations of the three improvements in Section VII. With regard to improvement three, we conducted a comprehensive performance evaluation in Section VIII. Subsequently, we discuss in Section IX various general aspects of ZTSFC. The conclusion in Section X gives a summary and discusses future work.

II. BACKGROUND

Before we discuss ZTSFC, we describe the core concepts of ZT and SFC in this section.

A. ZERO TRUST

ZT security recoins the well-known IT security paradigm *trust but verify* into *never trust, always verify* [6]. Trusted LANs, as managed in the perimeter security concept, are removed. Neither a device nor a user is attributed initial trust based on inherent properties such as network location. Instead, each RAR must be verified before it is permitted. In ZT security, verifying is a process consisting of strictly enforcing authentication, authorization and monitoring of all RARs [4]. In the context of this article, we limit the granularity of access decisions to service accesses, like web services, rather than individual resources within that service. Several ZT architectures have been proposed to ensure the strict Auth* of all RARs. For example, in Google's *BeyondCorp* publications [2], [11], [12], in Gilman and Barth's book *Zero Trust Networks* [3], or in the NIST publication *Zero Trust Architecture* [4].

Figure 1 depicts such a ZT architecture, where the NIST naming scheme is adopted. Here the PEP acts as an access control gateway for the clients. At the arrival of a client's RAR, the PEP requests the PDP for an Auth* decision whether the requested access should be permitted or rejected. The PDP's Auth* decision is then enforced in-line by the PEP, which either forwards or rejects the RAR.

1) AUTH* DECISIONS

The first step of the Auth* decision-making process is client authentication. That includes often prompting the client for multiple authentication factors. After the client authentication, the PDP proceeds to make an authorization decision. With ZT security, this is based on the trust attributed to a RAR. This trust is built from as many data sources as possible. That includes, for example, user and device attributes as well as history and threat intelligence data. The latter provides conclusions about the current security posture of the network and the hosted services. Access is only permitted if the trust in the RAR is sufficiently high. A widely-used approach to assess if the trust level is adequate is the criteria-based method, detailed in [4] and [5]. Criteria-based methods define a set of binary criteria that must all be true for access to be permitted; only then the trust is considered high enough. Listing 1 depicts an example criteria-based access policy expressed in the policy language ALFA.[1] The showed policy manages accesses to an example service. The keyword *target clause* in ALFA checks the policy's applicability for specific RARs. Here, the rule *getServiceAccess* inside of the *exampleServiceAccess* policy is applicable if the RAR wants to get access (line 5) to service *exampleService* (line 2). The conditions for permitted access require access within the time range 9 a.m. and 5 p.m., a two-factor authenticated user, and a RAR originated by a managed device. Each condition represents a criteria, which can be either true or false. Access is permitted if all conditions result in true. Furthermore, the example policy shown uses the obligation feature of ALFA. This feature adds additional requirements to policy rules, like logging permitted accesses as shown in the example. The PDP sends the obligation along with the authorization decision to the PEP, which enforces it, i.e., the PEP logs the access and forwards the RAR to the requested service. Other possible policy languages suitable for ZT were presented by Dimitrakos et al. [13], Colombo et al. [14], and Lazouski et al. [15]. They combine features of the policy languages XACML [16] and UCON$_{ABC}$ [17]. That enables granular and continuous trust-based authorization decisions.

[1] https://www.oasis-open.org/committees/download.php/55228/alfa-for-xacml-v1.0-wd01.doc

```
policy exampleServiceAccess {
  target clause request.service == exampleService
  apply firstApplicable
  rule getServiceAccess {
    target clause request.action == getAccess
    condition currentTime>"09:00:00":time or currentTime<"17:00:00":time and
      user.authentication == authentication.password and
      user.authentication == authentication.oneTimePassword and
      device.type == "managed"
    permit
    on permit {
      obligation notify {
        action = log_access
        log_information = user, device, access time, accessed service
      }
    }
  }
}
```

LISTING 1. Example ZT criteria-based policy that manages requested accesses to an example service.

B. SERVICE FUNCTION CHAINING

Where ZT aims to increase service security, SFC wants to achieve a more dynamic and flexible deployment of network service functions (SFs) such as firewalls or NAT routers. In a network without SFC SFs are tied to the underlying network topology, which severely limits the flexibility and dynamics. If an SF lies on the topological path of a packet, it can be applied to it. In contrast, it is a non-trivial task to apply network functions to packets traversing a network path in the topology that does not contain that function. To address this limited flexibility, RFC 7665 [8] describes a standardized SFC architecture. Figure 3 depicts a simplified SFC architecture based on RFC 7665.

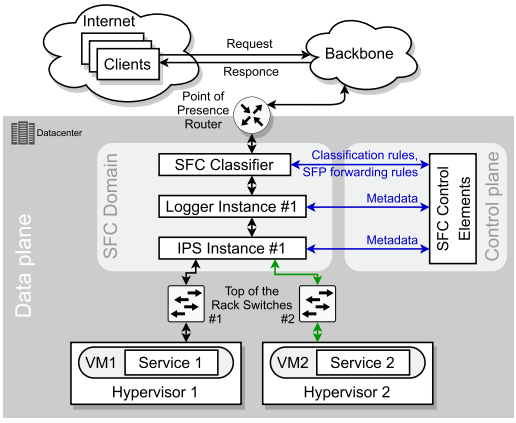

FIGURE 3. SFC architecture based on RFC 7665 [18] that is separated into a data plane and a control plane. The data plane consists of an SFC classifier and SFs. The control plane connects the abstract control elements.

The depicted architecture defines an SFC classifier as the entry point into an SFC domain. It classifies incoming traffic according to classification criteria such as network packets 5-tuple (destination and source IP, destination and source port, and the next protocol header). Based on this criterion, the classifier determines which SFs must be applied to these packets. All SFs defined in this way form an SFC. However, the defined SFC only describes which SFs must be applied and in which order. The SFC control elements derive a concrete service function path (SFP) from this abstract SFC. The SFP contains the actual used SF instances and defines the network path the packets traverse. All affected packets are forwarded through the network according to this SFP until they reach the destined service. In Figure 3, an exemplary SFC <Logger, IPS> could contain the logger SF and the IPS SF. From this SFC an SFP such as <Logger Instance #1, IPS Instance #1> is derived. Packets classified by the SFC classifier are then forwarded through the defined SF instances before leaving the SFC domain and eventually arriving at the destination (either Service 1 or Service 2). Studies such as those by Iffländer et al. [19] and Shameli-Sendi et al. [20] are concerned with creating the most efficient SFCs and SFPs possible.

The SFC control components described abstractly in [18] define the classification criteria and specify the associated SFCs and SFPs. This information is communicated with the SFC classifier. In return, function-relevant data, such as the SF's load, or feedback about the processed packets is received from individual SFs. Based on this input from the SFs, the control components can adjust the affected SFPs when necessary.

III. MOTIVATION AND STATE OF THE ART

As discussed in the introduction, the ZT components PDP and PEP usually do not provide all the security functionalities necessary for satisfying the ZT paradigm, such as packet inspection. Therefore, the efficient and synergistic integration of other MaS functions that provide these missing functionalities, such as an IPS, into a ZT architecture is necessary for our understanding. However, such an integration remains often times unaddressed in the literature. That motivated the design of ZTSFC, which addresses this lack of integration. With ZTSFC, we achieve three main improvements over ZT architectures. This section discusses the integration in current ZT architectures and motivates the three improvements based on the state of the art.

Motivation for Improvement 1 - The integration of MaS functions into the PDP's Auth* decision-making process: In the introduction, we outlined a scenario where applying a well-defined set of MaS functions (here: MFA and IPS) to a RAR can compensate for specific attributes (here: managed device) that are not fulfilled. Let us assume an employee is on a business trip without having a managed device with her but still needs urgent access to a service. So the employee is missing device authentication and a host-based intrusion detection system with on-device packet inspection [10]. In such a case, we can permit access by applying the MaS functions MFA and IPS to her RAR. The MFA prompts for an additional authentication factor compensating for the device authentication. The IPS performing a DPI compensates for the missing on-device packet inspection. By doing this, we can keep up her productivity while ensuring security. The trust deficit created by the lack of a managed device can be rebuilt through the targeted use of MFA and IPS.

Such a scenario requires two things. First, the PDP implements suitable policies that cover the integration of MaS functions in the policy rule set. And second, the PDP can ensure at the time it makes the Auth* decision, that the set of MaS functions specified in the policy is later applied to the RAR's packets before they reach the service.

The first requirement is the integration of MaS functions into the policy rule sets. This could be implemented, for example, by using the obligation feature supported by most of ZT-suitable policy languages such as ALFA, XACML, or UCON. In the literature, Dimitrakos et al. [13], Colombo et al. [14], and Lazouski et al. [15] introduce powerful ZT-suitable policy languages. However, the integration of MaS functions is out of the scope of their work.

The second requirement, ensuring that specific sets of MaS functions are applied to packets, is difficult to achieve in conventional network solutions. To illustrate this, Figure 4 shows a simplified example enterprise network. In this example network, an IPS is deployed at a network choke point where all traffic from external requestors such as client 1 flows through. However, since ZT eliminates intrinsic trust from internal requestors like client 2, equivalent security measures are also imperative here. To enable the PDP to apply MaS functions, including the IPS, to all packets-even those from client 2-the IPS must be integrated into all network paths. This leads to a challenging and cumbersome task for network administrators. Either costly IPS hardware must be deployed to all network paths or time-consuming rewiring is required [21]. Standard software-defined networking (SDN) solutions also do not provide the necessary granularity as they work with criteria such as the 5-tuple. The checking for a managed device or specific authentication factors is oftentimes not supported out of the box. It should be noted, however, that SDN solutions are suitable for implementing ZTSFC. In ZT grey literature like in [4] or [5], it is abstractly described that MaS findings, as part of threat intelligence, could be used in the PDP's Auth* process. However, it remains unaddressed how the PDP can dynamically apply specific sets of MaS functions to RARs if needed. Previous academic studies have also examined different methods that

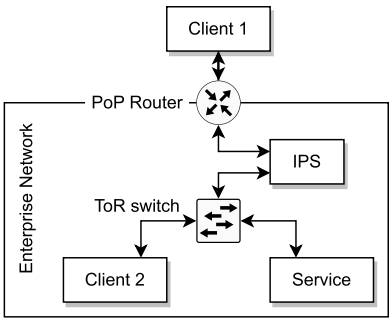

FIGURE 4. Example enterprise network encompassing a point of presence (PoP) router as network entry point and a top of rack (ToR) switch connecting client and services internally, and an IPS right behind the PoP router.

enable the PDP to orchestrate other security functions, but often with a different focus. Eidle et al. [22] present a ZT approach that enables an orchestration server to coordinate an access gateway and firewall. This server can instruct the firewall to block suspicious IP addresses as part of a DoS defense. Vanickis et al. [23] introduce a new risk-based policy enforcement framework that leverages firewalls to enforce access decisions. Both works focus on firewall security functions. However, both described approaches do not directly consider the firewall in the trust establishment. Also, it must be given that a firewall is physically integrated on all packet paths so that the approaches work. Ghate et al. [24] and Ramezanpour et al. [25] introduce new ZT architectures in different domains. They consider input from MaS functions as passive input for the trust evaluation. However, the ability of the PDP to orchestrate the MaS functions and in this way increase the trust or compensates for missing attributes is out of the scope of the work. In the area of software-defined networking, Yu et al. [21] and Chen et al. [26] introduce adaptable SDN security architectures. They allow the dynamic application of MaS functions depending on the context. However, the integration of ZT components is out of the scope of their work.

To the best of our knowledge, there is currently no approach that integrates MaS functions directly into the Auth* decision-making process. That allows the ZT components, for example, to actively apply selected sets of MaS functions to packets to compensate for a trust deficit of the RAR.

Motivation for Improvement 2 - The implementation of a direct and efficient information flow between MaS functions and PDP: With ZT, Auth* decisions are trust-based. Each RAR must proof the PDP that it is trustworthy. The goal is to make Auth* decisions as accurately as possible. That means ensuring that no unauthorized RARs are allowed and that as few legitimate RARs as possible are rejected. To make an accurate decision, the PDP needs comprehensive information about the RAR and its context. That ensures that the PDP can accurately determine the trust level of the RAR. As stated in [3], [4], [27], and [5], important sources of information for establishing trust in a RAR are threat intelligence data and network logs both provided by MaS functions. Therefore, a direct and efficient flow of information between MaS functions and PDP providing this data is desirable.

There are several publications on this topic in the field of gray literature. Forrester, in their ZT studies [6], [27], [28], [29], [30], suggests logging all network traffic by mirroring it at a central gateway. However, they have not fully explained how they handle encrypted traffic and how the results are leveraged for trust-based decisions. Google's BeyondCorp solution [2], [11], [12], [31], [32], [33] discusses a mature ZT framework, but the specifics of monitoring and data collection were not the focus of their publications. Both [4] and [34] highlight the need for security monitoring and threat analysis. [35] emphasizes the importance of seeing

everything on the network for making trust decisions, though implementation details are not discussed. The Qi An Xin Group [36] and VMware [37] introduce their own proprietary solutions that use network information for trust decisions. However, the authors leave room for details, like how they handle encrypted traffic. In academic papers, Lee et al. [38] discuss situational factors, such as enterprise security risks, that influence decisions but the data collection methods were not the focus of their paper. Tao et al. [39] have a ZT model for big data and state the monitoring of all network traffic, but they do not go into detail about how they use this in the Auth* process or if they decrypt traffic. Yao et al. [40] introduce a system that can log network traffic. Mehraj et a. [41] mention the importance of monitoring everything in the ZT system. However, they leave the technical specifics for subsequent studies.

To the best of our knowledge, there is currently no approach that provides all the necessary detail on how to ensure a direct and efficient information flow between PDP and MaS functions. The information provided by MaS functions should be dynamically adaptable depending on the situation. If the PDP is sure that the RAR is trustworthy, for example, only access times and accessed services need to be logged. If the RAR is less trustworthy, it may be necessary to log the entire packet to ensure through packet analysis that the RAR is indeed benign. Permanent logging of the entire packet would place an unnecessarily high load on the MaS components. The information collected about packets should therefore be fine-granularly customizable on the packet flow level. It should also be possible for MaS functions to handle encrypted traffic in order to have access to the payload. In addition, the PDP must have direct access to all this collected data in order to be able to include it in the Auth* process.

Motivation for Improvement 3 - Reduction of hardware load and improvement of user experience: In our discussion regarding improvement one, we talked about how in conventional networks, the IPS is deployed at choke points in the network. At these points, it checks all the packets that pass through. As part of the motivation for the second improvement, we highlighted the importance of a direct information flow between MaS functions and PDP. However, if we constantly log all packet details or apply security functionalities like DPI to every packet, it leads to a high load on the servers running the MaS functions. This issue is amplified due to the continuous growth in network speeds and the significant computational demands of security functionalities like DPI. Servers with weaker hardware are particularly affected by this issue, as it is especially important here to save computing resources here [42]. Saving computing resources often leads to security features being disabled to maintain performance [43].

With eZTrust, Zaheer et al. [44] introduce a network-independent way of parameterization of microservices. By tagging packets with all contextual information that is necessary for making Auth* decisions on a host basis, they were able to implement these decisions efficiently. It leads to a 1.5-2.5 times lower CPU overhead on the service hosting servers compared to traditional parameterization schemes such as those that are DPI-based. However, it was out of scope to investigate the impact on other MaS functions. It may be also not applicable for some networks to abandon the network-based MaS functions.

To the best of our knowledge, there are currently no concrete approaches within the area of ZT to reduce the load on the servers hosting the MaS functions through more fine-grained packet handling. At the same time, excluding MaS functions from packets' paths where they are not necessary has a positive impact on the user experience by keeping the RAR latency as low as possible. This exclusion is possible when the trust in the RAR is already sufficiently high.

IV. ZTSFC ARCHITECTURE

In this section, we describe ZTSFC, all its components, and its workflow. With ZTSFC, we make an attempt to integrate MaS functions into ZT architectures and achieve all three previously discussed improvements. The ZTSFC architecture is depicted in Figure 5 and is rooted in the proven and widely adopted XACML 3.0 flow model.[2]

A. OVERVIEW

In the architecture shown, each white rectangle represents a ZTSFC component, such as the PDP. We designed ZTSFC modularly. The components can run on the same physical machine but can also run distributed. Different implementations of a ZTSFC component are interchangeable as long as they provide the necessary interfaces for communication with other ZTSFC components. The interfaces can be realized via REST APIs, for example. Basic processes, such as authentication or authorization, are handled by individual components within ZTSFC to enhance its modular structure. Additionally, each component must be scalable so that multiple replicas of a single component instance can share the work. That allows for adjustments to handle varying workloads and helps prevent single points of failure. This ability to scale is visually shown using overlapping rectangles. The architecture enables one PEP to be responsible for any number of services or multiple PEPs for one service. Each PEP works independently from the other PEPs in the network. However, other components such as the databases need to ensure consistency between replicas. This and scalability between replicas of distributed components is sufficiently discussed in the literature, such as in [45]. The individual components are presented abstractly to allow flexibility in a concrete implementation. The architecture describes their task and the interfaces they must offer.

The architecture is divided into a data plane and a control plane. The data plane handles all user data whereas the control plane is responsible for all management communication. All

[2]http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf

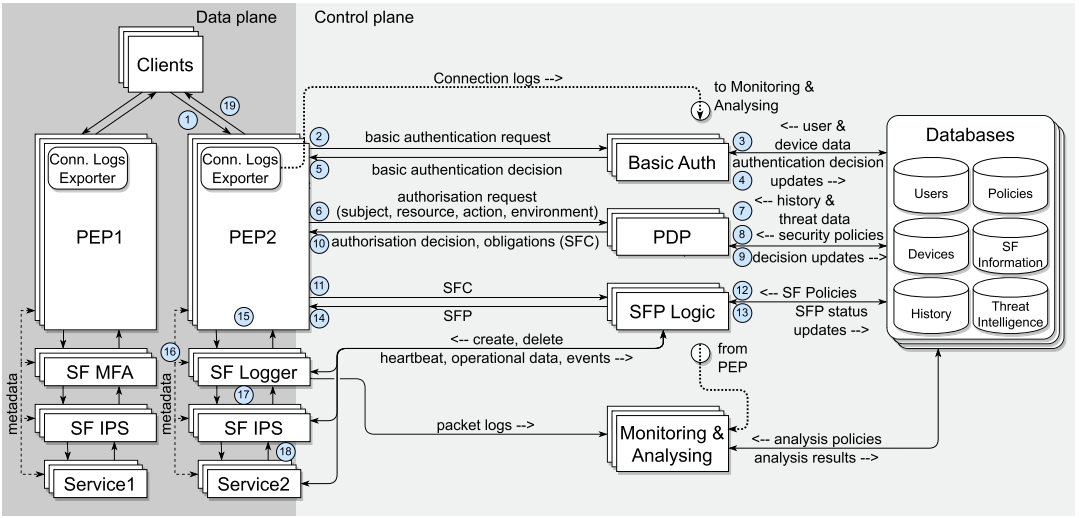

FIGURE 5. ZTSFC architecture with all its components and their communication with each other numbered in the order of their processing. Overlapping rectangles symbolize the possibility to scale the components to any number of replicas. PEP, SFs and services are part of the data as well as of the control plane. The components Basic Auth, PDP, SFP logic, Monitoring&Analysing and all Databases only communicate within the control plane.

depicted communication channels must provide confidentiality, authenticity, integrity, and perfect forward secrecy (PFS). PFS is a security feature that describes that the leakage of a secret key does not affect past encrypted traffic. These security goals can be achieved by using mTLS 1.3, for example. However, this requires the management of X.509 certificates and impacts server performance. Additionally, the integrity of all ZTSFC components must be checked regularly using security tools such as malware scanners. MaS functions are realized in the ZTSFC approach as SFs. Here it must be ensured that the SFs implement a proof of transit, a technique that proves that all SFs in an SFC are traversed [46]. In the context of this work, we assume that the components are not compromised and work correctly.

B. COMPONENTS

Below we will explain all the components of ZTSFC.

PEP: The PEP can be considered as core component of the architecture. It acts as an access gateway for the services and coordinates the complete workflow. In addition, the PEP logs certain connection information that is not available for SFs later in the network path, e.g., information about the TLS handshake performed with a client. These logs are exported to the Monitoring&Analysing component (Conn. Logs Exporter in Figure 5).

Basic Auth: It is responsible for client authentication. Clients are authenticated on a per-session basis. A session could, for example, include all packets with the same valid session token. Basic Auth processes corresponding authentication requests from the PEP. This type of request contains the client's claimed identity including the authentication factor(s) to be verified.

PDP: The PDP performs an authorization for each RAR. For this, it checks compliance of the RARs with trust-based policies based on the subject (client) that is requesting a specific action (service access) on a resource (service) in a given environment (e.g., date and time of the RAR). As in the background chapter, we will explain the authorization procedure using criteria-based access control. The PDP can also implement a score-based approach. He et al. [42] provide an overview of possible trust score algorithms. However, attribute-based policy languages such as ALFA, XACML, and UCON+ [13] are well suited to express criteria-based policies. Each policy has a set of criteria that all need to be satisfied to grant access. Listing 2 provides an example of these criteria in ALFA format, where criteria are called conditions.

The PDP then determines if all these criteria are met using data from the databases as well as the RAR attributes such as the access time. The RAR is permitted in case all criteria are met. Then, the PDP extracts the SFC to apply to the RAR

```
policy service2Access {
  target clause request.service == service2
  apply firstApplicable
  rule getService2Access {
    target clause request.action == getAccess
    condition currentTime>"09:00:00":time or currentTime<"17:00:00":time and
      user.authentication == authentication.password and
      device.type == "managed"
    permit
    on permit {
      obligation notify {
        action = apply_sfc
        sfc = sf_logger
      }
    }
  }
}
```

LISTING 2. ZTSFC example policy expressed in ALFA. The policy consists of a rule that handles the access to ZTSFC Service 2 as depicted in Figure 4. The obligation included describes the application of an SFC to the RAR.

from the obligation part of the policy evaluated for this RAR. An example obligation is shown in Listing 2. As described in the introduction, the SFs included in the SFC could be chosen to compensate for missing attributes. The concretely used policy language, set of criteria, and applied SFs as well as the SFC creation strategy depend on the particular application scenario and must be specified by the responsible security officer. A detailed use case describing the SFC strategy is given in Section VI. Finally, the PDP informs the PEP about the authorization decision and the specified SFC the PEP must enforce.

SFP Logic: The SFP logic manages all SF instances in the network. It creates and deletes instances if necessary. For example, the SFP logic initiates a new instance if the workload on the currently running instances of a type is too high. To get all the necessary information for this job, it receives heartbeat and operational data from all active SF instances. These data provide information about the current health status of the SF instances. Additionally, it processes specific events, such as security alerts from an IPS SF, and pushes those events to a threat intelligence database. This behavior is inspired by [19]. In addition to the SF management, the SFP logic derives concrete SFPs from the abstract SFCs it receives from the PEP. This process is based on the information about the available instances and the SF policies. In the latter, further requirements for the arrangement of the SFs are defined, such as specific SF orders. Several concrete approaches to deploy and manage SFs are discussed in existing literature, e.g., in [47] or [48]. The SFP derived in this way is then sent back to the PEP.

Databases (DBs): The DBs provide input for all performed authentication, authorization, and SFP logic decisions. A user and device DB must be available so that clients can be authenticated. In addition, a policy database must be available for authorization and SFC creation. For storing information about SFs, an SF information DB must be used. There should also be a threat intelligence and history DB that store the analysis results from the Monitoring&Analysing components. Any additional DB as further input for authorization is possible.

Service Functions (SFs): SFs can be any network function that is SFC-ready as described in the background section. With ZTSFC, SFs are mainly represented by MaS functions. Any arbitrary amount of different SFs can be implemented and managed. In Figure 5, all depicted SFs are based on the ZTSFC concept shown in Figure 1. The actions the SFs apply to the packets are determined by the PEP via metadata inserted into the packet headers. For example, network service headers (NSH) [49], MPLS labels [50], or HTTP headers (see Section V) can be used for metadata transfer. Similarly, SFs can send feedback to the PEP in the form of metadata. This feedback can be inserted into the service's response packets.

SF Logger: The logger function logs detailed information about processed packets on its path. The level of detail captured about processed packets is communicated to the SF via metadata from the PEP.

SF Multi Factor Authentication (MFA): It prompts the user for additional authentication factors. The factors to check is included in the metadata. It checks for factors in addition to the ones used by the Basic Auth component.

SF IPS: A function that can perform, among other actions, a DPI on processed packets. It can send threat alerts to the SFP Logic. The IPS can also drop packets if something malicious is found in them. In this case, the IPS also sends feedback about this to the PEP, which the PEP can then provide to the client for better decision transparency.

Monitoring&Analysing: It monitors and analyses all log data exported from, e.g., the PEP or the logger. Depending on the scenario, arbitrary analysis approaches can be applied. The results are then pushed to the respective DBs (here: threat intelligence) where they can then be included into the Auth* process.

C. WORKFLOW

The ZTSFC workflow is described based on the numbered steps depicted in Figure 5. In the case of an incoming client RAR, the components work together as follows: Before a client's RAR is permitted, all its packets are processed by the PEP beforehand (1). First, the PEP authenticates the client's RAR. For this purpose, the PEP sends an authentication request to the Basic Auth component (2). The Basic Auth component then extracts the claimed client identity. To verify the client's identity, it queries the corresponding data from the user and device DB (3). According to the verification, the Basic Auth pushes the result (successful or not) to the history DB (4). Finally, the Basic Auth component informs the PEP about the authentication decision (5). Depending on this decision, the client's RAR is rejected or is further processed. Next, the client's RAR is authorized. To achieve this, the PEP sends an authorization request to the PDP (6). This authorization request includes all data about the RAR, namely who is requesting which action on which resource in which environment. The PDP is enriching this data with information from the history and threat intelligence DBs (7). This information includes, for example, the usual access times for this client or analysis results from the Monitoring&Analysing component. According to the enriched data, the PDP queries the matching policy from the policy DB (8). In addition, this enriched data serves the PDP as a trust source for trust establishment in the client's RAR. All criteria defined in the matching policy are evaluated for the authorization of the RAR. The authorization decision resulting from this evaluation, either allowed if all criteria are met or denied, is then pushed to the history DB (9). Additionally, depending on the policy's obligation, an SFC and instructions about how each SF must process the packets belonging to this RAR are defined by the PDP. The PDP then sends the authorization decision and the SFC obligation to the PEP (10). This obligation contains the defined SFC and the metadata

that defines the actions the SFs apply to the affected packets. Again, the PEP is either rejecting the client's RAR or further processing it. For the latter, it sends the SFC information to the SFP Logic (11), which then loads the corresponding SF policies for the included SFs (12) and transforms the sent SFC into an SFP. This process includes the setting up of all necessary SF instances if not already available. This process involves creating any required SF instances if they aren't already set up. For traceability, this SFP is first saved to the history DB (13). After that, the SFP is shared with the PEP (14). The last job of the PEP is then the preparation of the RAR's packets (15). This includes PDP's metadata for the SFs (16) and the enforcement of the forwarding rules to the packets. The SFs included in the SFP process then forward all packets according to the included metadata (17). In case an SF does not successfully process a packet, e.g., when the IPS detects malicious payload in it, the packet is dropped and the corresponding feedback is sent back to the PEP. Eventually, the packet arrives at the requested service. The service response is taking the same way back (18) through the SFs and PEP to the client (19).

V. HTTPS-BASED ZTSFC

For the PoC implementation of our ZTSFC architecture presented in IV, we rely on an approach to SFC that is based on HTTPS [9]. Figure 6 depicts the conceptual idea behind the HTTPS-based approach, which we will describe in detail in this section.

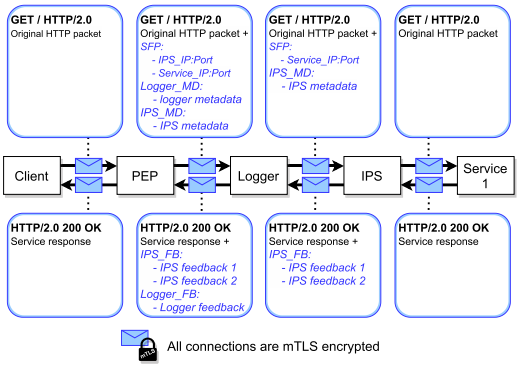

FIGURE 6. HTTPS-based ZTSFC approach showing an example SFP consisting of a logger SF and an IPS SF with Service 1 as destination. Next hop information (*SFP*), instructional metadata (*Logger_MD*, *IPS_MD*) and SF feedback (*IPS_FB*, *Logger_FB*) are stored in the HTTP header as custom header fields.

For HTTPS-based SFC, the PEP leverages custom HTTP header fields to pass forwarding and metadata information to the SFs contained in the SFP. The first SF in the SFP is directly addressed by the destination address and port fields in the IP and TCP header, respectively. Destination addresses and ports of the remaining SFs and the service are stored in a new *SFP* HTTP custom header field. Additional custom header fields are used to pass metadata such as operational instructions to the respective SFs. All SFs can then extract their next hop information and the metadata from the respective HTTP header fields. In the current implementation, the next hop information in the *SFP* header field is implemented as a list. The first element in the list represents the next hop address. The SF, to which the next hop address belongs, removes this field after it has processed it.

Figure 6 shows an exemplary SFP consisting of a logger SF and an IPS SF with service 1 as the destination. Here, the PEP modifies a RAR packet by adding the *SFP* header field to it. This header field includes the destination information of the IPS as well as from service 1. In addition, the PEP adds the metadata fields *Logger_MD* and *IPS_MD* for the respective SFs to the packet. After the preparation, the PEP sends the packet directly to the logger. After the packet gets there, the logger first extracts the *Logger_MD* metadata that pertains to it and then deletes it. After that, it takes the next hop information of the IPS from the *SFP* HTTP header field, removes this list entry and, proxies the packet to the IPS. The IPS repeats this procedure by extracting the respective *IPS_MD* metadata and service 1's address. In case the SF (here: IPS) that processes the packet is the last in the SFP, it removes the *SFP* header field completely. This allows the original client RAR to be restored before it is proxied to the eventual destination (here: service 1). The proxy chain created in this way is also used to forward the service's response back to the PEP and then eventually to the client. On the way back, each SF can modify the response packets by adding feedback information as a new HTTP custom header field (here: *IPS_FB* and *Logger_FB*) that can then be processed by the PEP. This feedback header is used, for example, to implement proof of transit. To achieve this, an SF includes a security token in the feedback header, signed using the same private key as for mTLS connections. The PEP then validates this token using the associated public key. After processing the feedback, the PEP removes the SF HTTP custom header feedback fields to restore the original response before forwarding it to the client. In the case the RAR got denied, the PEP can send the client feedback about the reasons for the denial.

All connections depicted in Figure 6 are mTLS encrypted, which ensures confidentiality, integrity, mutual authentication, and PFS. By placing the metadata and SFC information in the HTTP header, this data is also sent confidential and integrity protected. By fine granular certificate management, network components (here: PEP, SFs, and Service 1) only communicate with a specifically defined group of other network components. Thus, logical micro-segmentation can be implemented.

Using HTTPS-based SFC brings several advantages. First, it is rooted in TLS, the most common approach for providing traffic security [51]. Basing the SFC approach on hop-by-hop established TLS enables all SFs the option to look into the packets' payload. That capability is crucial for several SFs, such as an IPS that performs a DPI. Second, this approach offers location independence for all ZTSFC components as long as all components are reachable via IP/TCP. That provides a flexibility advantage over SFC approaches using

TABLE 1. Hardware composition of the servers the testbed consists of.

Device	Motherboard	CPU(s)	RAM (DDR4)	NIC
srv1	Supermicro® X10DRW-i	2x Intel® Xeon® E5-2630 v3	7x 16GB 2133MHz	100G Mellanox® MCX516A -CCAT
srv0,2,3,4			8x 16GB 2133MHz	
srv5	Supermicro® X11SCW-F	Intel® Xeon® E-2186G	4x 16GB 2667MHz	40G Mellanox® MCX456A -FCAT
srv8				
epyc	Supermicro® H11DSi-NT	2x AMD® EPYC® 7281	16x 8GB 2667MHz	10G Intel® X550-T2

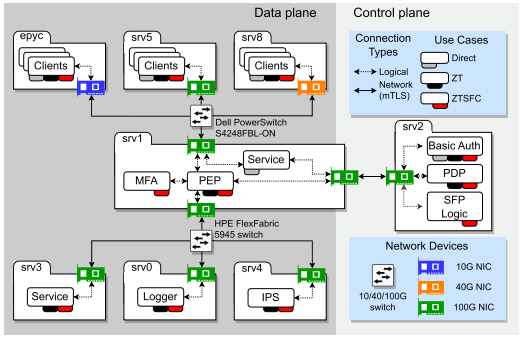

FIGURE 7. Overview of the ZTSFC PoC as well as the ZT and direct access prototypes. The hardware included in each use case is marked with colored labels (grey, black and red). All included servers and implemented components are separated into data and control planes.

lower ISO/OSI layers such as MPLS [52] for passing the next hop and metadata information. Third, TLS and HTTP are widely adopted, e.g., in BeyondCorp [2]. Fourth, a HTTPS-based prototype can be implemented in the application layer completely. Finally, such a prototype can run on almost every off-the-shelf computer without having the need for specialized and expensive hardware. On the other hand, the HTTPS-based approach is limited to web-based services. Additionally, all SFs must be ZTSFC-aware to be able to process the custom HTTP header fields accordingly.

VI. PROOF OF CONCEPT

Based on the described HTTPS-based SFC approach, a publicly available[3] PoC of the introduced ZTSFC architecture is presented in this section. Furthermore, we describe implementations for a ZT architecture as well as for a direct service access scenario. Both are used in the evaluation in Section VIII to compare the performance of the ZTSFC PoC against them.

A. TESTBED

Figure 7 depicts the hardware testbed in which the PoC is implemented. The hardware testbed consists of eight servers listed in Table 1. The operating system on all servers is Ubuntu 22.04.2 with kernel version 5.15.0-76. All processes that are not necessary for running the PoC were stopped. The maximum number of open files is increased via *ulimit -n* from 1021 to 100000. That is necessary to not run out of available file descriptors during the test runs. Additionally, the setting *net.ipv4.tcp_tw_reuse* is turned on. This allows the system to reuse TCP connections that are currently in the *TIME_WAIT* state, which would typically prevent the associated port from being used. All servers are connected via two 100G layer 2/3 switches. The system's data and control operations are managed in separate network subgroups. Unless mentioned differently, all components described below are compiled with Golang version 1.20.1.[4]

B. ZTSFC

In Figure 7, the hardware that makes up the ZTSFC PoC is marked with red labels. Except the Monitoring&Analysing and the threat intelligence, the reference architecture described in Section IV is completely implemented. Details about the components are described below:

PEP: The PEP implements all functions described in the architecture. The multi-threaded *Server* type from the Golang *net/http* package implements the client connections. The Server enforces mTLS 1.3. For the communication with Basic Auth, PDP, and SFP Logic, pre-established mTLS 1.3 connections are used. These connections stay established for the whole run-time. Due to this, only one TLS handshake per connection during the initialization process of the PEP is performed. For connections to the SFs as well as to the service, the *ReverseProxy* type from the package *net/http/httputil* is used. One new reverse proxy instance is set up for each client RAR. A pre-establishment of these proxy instances is not easily possible, since the PEP first learns the actual next hop SF instance to use from the SFP logic and this could change for each RAR in a productive setup.

MFA: The MFA is implemented as a PEP module. The communication happens via function calls. The MFA can prompt the user for a Passkey, a public key authentication method.

Basic Auth: For the Basic Auth component, we use the OpenLDAP docker image from rroemhild.[5] At the arrival of a RAR, the PEP checks if the RAR is already part of a valid session. This check is based on Java Web Tokens using the *jwt-go* package.[6] If a valid token is present, the authentication is not performed again. In any other case, the PEP prompts the client for its user and password via a POST form and asks the Basic Auth to authenticate the client's RAR. The optional certificate is provided during the TLS handshake between PEP and the client.

PDP: Similar to the PEP, the PDP implements the Server type from Golang's net/http package for all connections to

[3]https://github.com/vs-uulm
[4]https://golang.org/doc/devel/release
[5]https://github.com/rroemhild/docker-test-openldap
[6]https://github.com/dgrijalva/jwt-go

```
policy serviceAccess {
  target clause request.service == service
  apply firstApplicable
  rule getServiceAccess {
    target clause request.action == get
    condition user.role == "service_administrator" and
      currentTime>"09:00:00":time or currentTime<"17:00:00":time and
      user.authentication == authentication.password and
      device.authentication == authentication.certificate and
      device.type == "unmanaged" and
      rar.number < 25 and
      rar.failed < 3
    permit
    on permit {
      obligation notify {
        action = apply_sfc
        sfc = sf_ips
        sf_ips_action = dpi
      }
    }
  }
}
```

LISTING 3. Example PoC policy.

the PEP. The authorization requests happen in HTTP via a REST API. The PDP authorizes all possible resource access (here: GET and POST actions for the implemented service). One policy consisting of specific rule sets is defined for each possible RAR. Each policy rule represents a criteria (in ALFA conditions) and obligation combination that leads to permitted access. The following criteria are evaluated: user, used authentication factors, the requested service and action (GET or POST), device type, access time, the number of RARs for this day, and the number of failed attempts. The values for the criteria evaluation are extracted from the RAR and forwarded by the PEP to the PDP. In the PoC, we implemented criteria-based policies in ALFA. Listing 3 shows an example policy how it is implemented in the PoC.

In this example, the RAR originates from an unmanaged but authenticated device. To compensate for this, the obligation part includes the IPS in the SFC and instructs it to perform a DPI on the packets. All used criteria-based policies implemented in the PoC are described in the evaluation section. The RAR is permitted if all criteria are met. After the authorization process, the PDP sends the decision and the SFC extracted from the policy's obligation via HTTP to the PEP.

SFP Logic: For the SFP logic, the HTTPS server is implemented in the same way as for the PDP. It receives the SFC from the PEP embedded into an HTTP custom header field. Here the SFP logic derives the SFP from this given SFC by replacing each SF with a known IP:Port combination for this SF. In the prototype we use, one instance for each SF. The determined SFP is then sent to the PEP in an HTTP custom header field.

IPS: Server and reverse proxy functionalities are implemented in the same way as in the PEP. The IPS performs two attack detections: Path traversal and SQL injection detection. The first one is a simple string pattern matching. The latter consists of regexp-based pattern matching. The regexp functionality is implemented by Golang's *regexp* package. Both types of attack detection are applied to all processed packets. We chose an IPS as a representative for security functions because it is a very widely used function and the *regexp* pattern matching performed by it is at the same time a very CPU-consuming action [53]. The latter allows us to clearly investigate the performance impact that saving security functions has on the server load and the user experience. For the performance evaluation, the IPS performs the string pattern matching as well as the regexp-based pattern matching on all processed packets.

Logger: The logger implements server and reverse proxy functionalities in the same way as the PEP. During the test runs, all available TLS as well as HTTP data including the payload are logged for each processed packet. For the logging itself, the *logrus* package from[7] is used. All collected data are written to a Samsung SSD 850 EVO. We have chosen the logger as a representative for monitoring functions, as packet logging is probably the most widely used monitoring function. The task of forwarded packets logging leads to high usage of disk devices to save the logged data [54]. For the performance evaluation, the Logger logs the whole TCP payload of all processed packets.

Service: The service is a Golang multi-threaded HTTPS server using the *net/http* package for all connections. The service can work in two different access modes: it can serve a web resource of 1 *Byte* payload or provide a 1 *GB* sized file.

C. ZT

In Figure 7, the hardware involved in the ZT implementation is marked with black labels. Structure-wise, the ZT implementation is quite similar to the ZTSFC PoC. It differs in two aspects. First, the PDP does not specify an SFC but just makes an authorization decision. Second, the SFP Logic is not included into the workflow. All packets are always forwarded through the Logger and the IPS, if the Auth* decision was positive. Thus, it represents a ZT approach with Auth* process and legacy MaS functions that process all packets flowing through them.

D. DIRECT SERVICE ACCESS

The direct service access prototype is depicted in Figure 7 with grey labels. In this scenario, service access times can be measured without any security features and functions except a basic authentication. For this, the service that is part of the ZTSFC PoC is extended by basic authentication functionalities. It performs the same JWT checks for a valid user session, checks whether a X.509 certificate is present, prompts the client for its username and password and uses the same LDAP image for authentication.

E. CLIENTS

For simulating realistic clients that continuously request the service, the load testing tool *k6*[8] version 0.32.0 is used. In *k6*,

[7] https://github.com/sirupsen/logrus
[8] https://k6.io/docs/

the option *noVUConnectionReuse* is set to true. Thus, for each simulated client and each service access a new TLS connection is established. For all packets that are part of one service access, the same connection is reused. One complete service access represents one session. Client data that are evaluated during the Auth* process such as managed device information are sent to the PEP by HTTP custom header fields.

VII. IMPLEMENTATION OF THE IMPROVEMENTS

With ZTSFC, we make an attempt to integrate MaS functions into ZT architectures effectively. This section explains how we achieved the three improvements claimed in the introduction. We discuss all improvements based on the presented PoC.

Implementation of Improvement 1 - The integration of MaS functions into the PDP's Auth* decision-making process: ZTSFC allows ZT networks to integrate MaS functions in their Auth* decision-making process. The PDP can apply MaS functions in a detailed manner to RARs based on the trust level of that RAR. As mentioned in Section III, this inclusion in the Auth* decision-making process needs two steps: First, the PDP must manage policies that consider MaS functions. Second, the PDP needs to make sure that these MaS functions are applied to the RARs. To illustrate this, we use the example of the employee abroad from Section III to show how ZTSFC, specifically the introduced ZTSFC PoC, achieves the integration of MaS functions.

Regarding the first step, we need to create policy rules for two scenarios: one for an employee using a managed device and another for an employee accessing the service with an unmanaged device. Listing 4 displays the ALFA policy in our ZTSFC PoC covering both scenarios. The first policy rule (line 4) covers the scenario where the employee is in her office and uses her managed device to access the service. In this case, strong device authentication and on-device packet inspection are ensured. Therefore, the trust level is already sufficient to get access. The second policy rule is for when an employee is abroad and uses an unmanaged device to make a RAR and, consequently, the device authentication and on-device packet inspection are not in place. Compensating for this trust deficit, the second policy rule (line 14) includes an obligation that makes permitted access dependent on applying a specific SFC to the RAR (line 25). This SFC (line 26) contains an IPS for performing a DPI (line 28) and MFA for prompting for an additional hardware token (line 27) to the RAR. If neither of these two policy rules is applicable, the request is automatically denied (line 32). Once all the set criteria (expressed as conditions in ALFA) are evaluated, the PDP sends the authorization decision and the obligation to the PEP.

The next step is to ensure that the obligation is enforced. The PEP takes charge of this in ZTSFC. The PEP sends the SFC "MFA" and "IPS" to the SFP Logic. This is done to get the actual SFP that provides the specific IP address and port details for both the IPS and MFA, as well as

```
1  policy serviceAccess {
2    target clause request.service == service
3    apply firstApplicable
4    rule getServiceAccessWithManagedDevice {
5      target clause request.action == GET
6      condition user.role == "employee" and
7        currentTime>"09:00:00":time or currentTime<"17:00:00":time and
8        user.authentication == authentication.password and
9        device.type == "managed" and
10       rar.number < 25 and
11       rar.failed < 3
12     permit
13   }
14   rule getServiceAccessWithUnmanagedDevice {
15     target clause request.action == POST
16     condition user.role == "service_administrator" and
17       currentTime>"09:00:00":time or currentTime<"17:00:00":time and
18       user.authentication == authentication.password and
19       device.type == "unmanaged" and
20       rar.number < 25 and
21       rar.failed < 3
22     permit
23     on permit {
24       obligation notify {
25         action = apply_sfc
26         sfc = sf_mfa, sf_ips
27         sf_mfa_action = prompt_for_hardware_token
28         sf_ips_action = dpi
29       }
30     }
31   }
32   rule defaultDeny {
33     deny
34   }
35 }
```

LISTING 4. Sample ALFA policy with three rules that demonstrates how to decide which MaS functions to apply to a RAR.

the sequence in which they should be applied. It is reasonable to use the MFA first. This is because if the employee can not provide the required hardware token, the request is denied. This way, we can avoid using the resource-intensive IPS. In response, the SFP Logic provides the requested SFP {"MFA":"127.0.0.4:443","IPS":"127.0.0.5:443"}. Taking this SFP, the PEP enforces the obligation by leveraging the HTTPS approach described in Section V. All necessary metadata are embedded into the HTTP header as custom header fields. The process tailored for the employee's scenario is shown in Figure 8.

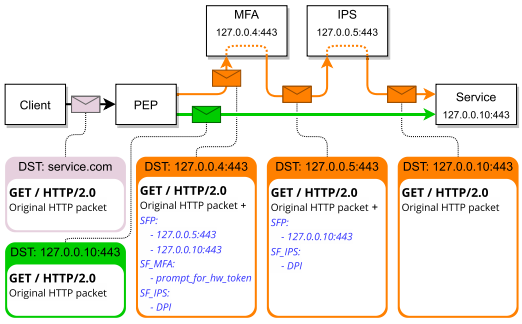

FIGURE 8. Sample HTTPS-based ZTSFC workflow of a client with managed device (green) and an unmanaged device (orange).

The example above illustrates how both employee access scenarios, access with a managed device represented by the green path and access with an unmanaged device and

```
1  policy serviceAccess {
2    target clause request.service == service
3    apply firstApplicable
4    rule getServiceAccessWithinOfficeHours {
5      target clause request.action == GET
6      condition user.role == "employee" and
7        currentTime>"09:00:00":time or currentTime<"17:00:00":time
8      permit
9      on permit {
10       obligation notify {
11         action = apply_sfc
12         sfc = sf_logger
13         sf_logger_action = packet_header
14       }
15     }
16   }
17   rule getServiceAccessOutsideOfOfficeHours {
18     target clause request.action == GET
19     condition user.role == "employee" and
20       currentTime<"09:00:00":time or currentTime>"17:00:00":time
21     permit
22     on permit {
23       obligation notify {
24         action = apply_sfc
25         sfc = sf_logger
26         sf_logger_action = packet_full
27       }
28     }
29   }
30 }
```

LISTING 5. Sample ALFA policy with two rules demonstrating how to decide what information the logger collects.

obligation highlighted with the orange path, are covered. This demonstrates how ZTSFC can integrate MaS functions in its Auth* decision-making. It is possible, to react to a trust deficit caused by the non-fulfillment of specific criteria by applying specific SFs to the affected RAR. ZTSFC is versatile, allowing for a variety of scenarios. Policies can be defined so that requests from specific locations always go through a firewall, or any access after 8 pm gets logged by a logging SF. It is important to note that for each unmet criterion, it is necessary to consider carefully how to react to it. For instance, performing a DPI is not a suitable substitute for a missing authentication factor.

Implementation of Improvement 2 - The implementation of a direct and efficient information flow between MaS functions and PDP: In Section III, we discussed the need for a direct and efficient flow of information between MaS functions and ZT components. ZTSFC allows us to achieve this. Depending on the level of trust of a RAR (in our context, the criteria met), the PDP can determine what details the MaS functions should gather about that RAR, which can be expressed in policies. Listing 5 presents an example of such a policy in ALFA, as used in our ZTSFC Proof of Concept. This example has two policy rules. The first rule (line 4) addresses access during regular office hours (from 9 a.m. to 5 p.m.). Since access during these hours is typical, the associated logger function only logs the packet headers. The second rule (line 17) deals with access outside regular hours, which might occur in urgent cases but is less common. Here, the rule specifies that the logger should log the entire packet. This approach allows for precise decisions on what details to collect about RARs. The rules, which include the SFC and actions of the SFs as obligations, are then enforced by the PEP, as discussed in the part about improvement one.

In Figure 9, we illustrate the two methods of information flow in our ZTSFC Proof of Concept. First, the MaS functions can add custom HTTP headers to the service's response packets. That sends direct feedback to the PEP. For instance, to confirm that a particular MaS function has processed the request, a proof of transit is used. This proof is a signature from the MaS function, signed with its private key, which is also used for TLS connections. Second, the MaS functions forward their data, like the collected packet headers, to the Monitoring&Analysis component using a REST API. This component then analyses the data and sends the results to the appropriate databases. The PDP can fetch this data from the database, giving it access to the MaS functions' outputs and findings. This information is then utilized in the Auth* decision-making process.

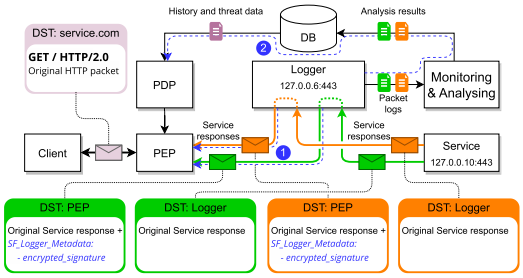

FIGURE 9. Sample HTTPS-based ZTSFC information flow between MaS functions and ZT components. First, information can send as feedback in service response (blue line 1). Second, information flow from logger through Monitoring&Analysis, DB to PDP (blue line 2).

As described above, the PDP can efficiently determine what information is collected about the RARs based on the trust level. This information flows back to the ZT components in two ways. By this, ZTSFC can ensure an efficient and direct flow of information between MaS functions and ZT components.

Implementation of Improvement 3 - Reduction of hardware load and improvement of user experience: In a ZT network, all packets flow through the MaS functions deployed on their network path. ZTSFC makes it possible to decide which MaS functions are applied to packets in a more fine-grained way than before. Furthermore, it can be decided just as finely granular which operations the MaS functions perform. These decisions are made based on the trust level of a RAR and expressed in the PDP policies. That can prevent too many packets from being forwarded through MaS functions due to too coarse-granular treatment, e.g., when a MaS function is deployed at a choke point and all packets flow through them. As we will show in Section VIII, this reduces the load on the servers hosting the MaS functions. At the same time, the access time for users is only increased by MaS functions if the established trust level is too low. This leads to an enhanced user experience for trusted users. ZTSFC can thus reduce the hardware load and improve the user experience.

VIII. EVALUATION

We now evaluate the ZTSFC PoC with regard to improvement three. Therefore, the main objective of the evaluation is to find out the impact of ZTSFC on the load of the servers that host MaS functions as well as on the user experience. We measure user experience by how long it takes for a user to access a resource (resource access time). We also measure at what point a higher number of such accesses simultaneously lead to increased access times (saturation). In addition, we measure the network throughput with which the users can download a resource in the scenario of increasing numbers of parallel accesses (average throughput). The load impact on the testbed servers is quantified by measuring CPU usage, interrupt rate and context switch rate (server load). The metrics mentioned above are described in Subsection VIII-B.

A. USE CASES

The measured performance of the ZTSFC PoC is then compared to that of the ZT network implementation and to that of the direct service access scenario. Whereby, ZT and direct service access each represent their own use case. The ZTSFC scenario is split up into six separate use cases. In all six use cases, the MFA prompts the client for a client certificate and is therefore not mentioned any further. All cases are described below:

- For the **zt** use case, the type of client used has a trust level high enough to be allowed to access the service. However, as no ZTSFC is available that would allow flexible path reconfiguration, all packets run through the logger and IPS in this order.
- The use case **direct** is used to measure the performance of the direct service access implementation. All clients are successfully authenticated and access the service directly without PEP and SFs.
- **no_sf** describes the ZTSFC use case in which only clients with a trust level high enough to access the ZTSFC service with no SF included in the path.
- The ZTSFC **logger** use case includes only one type of client with a trust level that results in a PDP decision to include the logger into the packet's path before the service can be accessed.
- In the ZTSFC **ips** use case we see only clients that trigger an SFP consisting of the IPS SF.
- **logger_ips** describes a ZTSFC use case with only clients with a trust level resulting in an SFP that consists of first logger and then IPS.
- For comparison reasons between different SF orders in a SFP, ZTSFC **ips_logger** is used. Here all client packets are forwarded first through the IPS and then through the logger.
- The ZTSFC **mixed** use case represents the most realistic use case. It includes 25% of clients from the use cases: *no_sf*, *logger*, *ips* and *logger_ips*.

The policies representing the ZTSFC use cases are appended in Appendix.

B. METRICS MEASUREMENT

All test runs to measure the metrics were conducted in the testbed depicted in Figure 7. Each run lasted 1 minute and was repeated 10 times. The data from the first 5 seconds ramp-up phase were discarded to get the results from the steady state of the systems. Stated average values are calculated over the whole test duration excluding the ramp-up phase. Each run was executed with a stable number of clients. Every client is continuously requesting the web resource. The resource is then received completely before the next RAR is sent without a waiting time. The actual client numbers are stated in the following:

- **Service access time** (SAT) [55], [56], [57] represents the time it takes for a client to download a dummy web resource of size 1 *Byte*. It starts with the first client RAR and ends when the web resource is completely transmitted. Each RAR passes through the Auth* process, the SFP logic and the defined SFP; to the extent they are implemented in the respective use case. The SAT is measured by *k6* via the metric *iteration*. One iteration corresponds to one completed web resource access. The metric is collected for different numbers of concurrent clients, starting from 1. Thereafter, starting at 10, the number of clients is increased by 10, up to a maximum number of 250.
- The **saturation** [55] metric is measured to identify the point at which an increase of concurrent web resource accesses no longer leads to an increase in processed resource accesses per second. The saturation test runs were the same as the SAT runs. The metric itself, *accesses per second* (*a/s*), was calculated based on the *k6* results.
- The goal of the **average throughput** [55], [58] test case is to compare the effects of dynamic security chaining in the case of ZTSFC to the use cases *direct* and *zt*. To measure the average throughput metric, the service provides a 1 *GB* file via HTTP download. The metric is measured for the use cases *zt*, *direct* and *mixed*. Over the test period of one minute, the file is downloaded as often as possible.This is repeated for 1, 5, 10, 15 and 20 concurrent processes on srv5 and srv8. The reached average throughput from both servers is then aggregated. The average throughput is measured with *k6*.
- The **server load** test comprises of the metrics CPU usage [58], interrupt rate and context switch rate [59]. It is measured on the servers that host the PEP, the logger and the IPS. The goal is to determine the influence of leaving out different SFs on the load of the servers. The load was measured during the SAT test runs for the *zt* and ZTSFC *mixed* use cases. For this, *dstat*[9] has exported the data over the whole test runs with an reporting interval of 1 second.

[9]https://linux.die.net/man/1/dstat

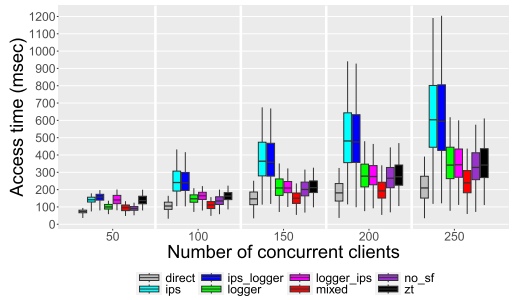

FIGURE 10. Dispersion (minimum, maximum as well as the 25th, 50th, and 75th percentiles) of the measured SATs for all evaluated use cases.

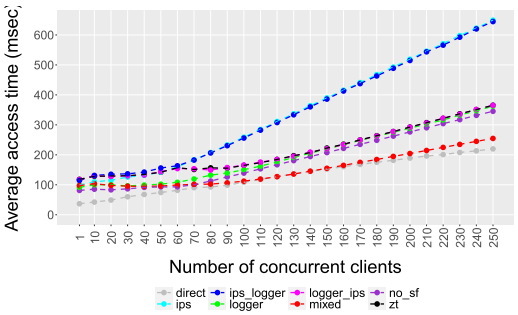

FIGURE 11. The average service access times for all evaluated use cases.

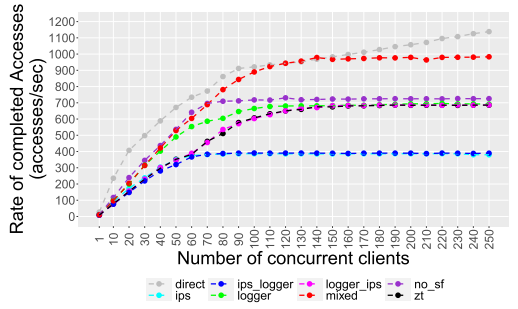

FIGURE 12. Saturation based on completed accesses per second for all use cases.

C. SERVICE ACCESS TIME & SATURATION

Figure 10 shows the measured SATs based on their dispersion (minimum, maximum as well as the 25th, 50th, and 75th percentiles). The measured average values for all evaluated numbers of concurrent clients are depicted in Figure 11. The saturation results during the same test runs are plotted in Figure 12.

First, the performance of the six ZTSFC use cases is evaluated. Use case *ips* shows the worst performance. The saturation point is reached at 70 concurrent clients with 381 *a/s* and an average SAT of 183 *ms*. This performance can be explained by profiling the IPS's CPU times during the test runs. For the profiling, Golang's package *runtime/pprof* was used. At 70 concurrent clients, the CPU spends 56% of the time doing cryptographic operations related to TLS. Most of the remaining time was spent performing regexp string matching (23%) and malloc system calls (21%) due to regexp expanding operations. The latter one leads to context switch intensive heap restructurings including heap expansions. At the saturation point, the IPS caused 180*k* context switches per second which leads to the bottleneck. More concurrent clients do increase the access times but not the amount of processed accesses per second. This result is as expected, since the IPS is the most computationally intensive function.

The use case *ips_logger* shows similar behavior since in this case the first SF in the SFP, the IPS, slows down the logger.

The *logger* use case, on the other hand, provides lower SATs than the use cases *ips*, *ips_logger* and also *logger_ips*. Furthermore, it saturates later at 150 concurrent clients with an average SAT of 219 *ms*. It processes a maximum of 681 accesses per second. The bottleneck is here the CPU. At the saturation point, the logger reaches 99% CPU usage.

Up to a number of 30 concurrent clients, the use case *logger_ips* shows higher average SATs (130 *ms*) and a lower amount of accesses per seconds (229 *a/s*) than use case *ips* (126 *ms* and 237 *a/s*). However, starting from 40 concurrent clients, the *logger_ips* case performs better although it includes one more SF. To see the reason for this, we recorded the global run queue (GRQ) of the Golang runtime every 10 *ms* during the tests. The GRQ represents the amount of jobs that ask for CPU time but are not yet assigned to a virtual Go processor and a local run queue (LRQ) of such a virtual processor. Every new incoming RAR results in such a job in the GRQ. For the *logger_ips* case, we see a much shorter and a less bursty GRQ for the IPS process. This means that the RARs arrive more evenly distributed and therefore the individual RARs have to wait less time in the GRQ until they are assigned to an LRQ. The GRQ is shown for a 5 *s* time window in Figure 13. The logger, which can process more accesses per second, in front of the IPS smooths the arrival of the RARs at the IPS. This also results in a much lower amount of context switches on the IPS server. At the saturation point of 150 concurrent clients, the *logger_ips* case shows around 120k compared to 200k in the *ips* case. For a higher number of clients, the level of context switching remains stable.

Starting from 150 concurrent clients, *logger_ips* behaves the same as the *logger* case because the logger now becomes the bottleneck.

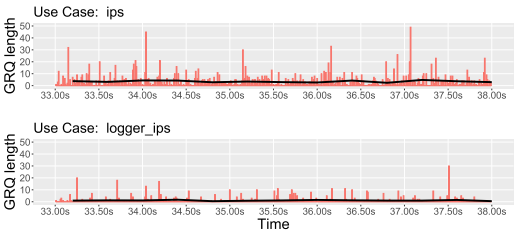

FIGURE 13. GRQ length for the IPS during a 5 *s* time window and measured every 10 *ms* during the test runs for the *ips* and *logger_ips* use cases. The black line represents the floating average of the GRQ length over the last 400 *ms*.

Looking again at Figures 5 and 2, except *mixed*, the *no_sf* use case performs best among the ZTSFC use cases. This is as expected since no SFs are included into the packets' path. At 70 concurrent clients it saturates with 694 *a/s* and an SAT of 100 *ms*. The main reasons for this is mainly the high number of context switches on srv2. The main cause for this is explained in more detail during the comparison with the *mixed* case.

The most representative ZTSFC use case *mixed* saturates at 180 concurrent clients with 973 *a/s* and an average SAT of 184 *ms*. Starting from 80 concurrent clients, *mixed* is the best performing ZTSFC use case. In terms of performance, from this point it also overtakes the use case *no_sf*. This is unexpected since 75% of the RARs in the *mixed* case are forwarded at least through one SF. In the *no_sf* case, the PEP forwards all packets directly to the service, which results in a bursty RAR arrival pattern at the service. In the *mixed* case, the packets are distributed over different SFPs. This causes a smoothness effect in terms of packet arrivals at the service. Mentioned burstiness for the *no_sf* case is directly reflected in the length of the GRQ. Figure 15 shows the GRQ progression over time for an exemplary time window of 5 *s*. These bursty arrivals result then in an inefficient handling of the high amount of RARs arriving at the same time. This inefficiency is caused by the high number of context switches per second (153710) for 180 concurrent clients that are necessary to deal fairly with all these RAR jobs that demand CPU time at the same time. The smoothness effect in the *mixed* case leads to a significantly lower number of context switches per second (14977) at 180 concurrent clients. The progression of the rates of context switches are depicted in Figure 14.

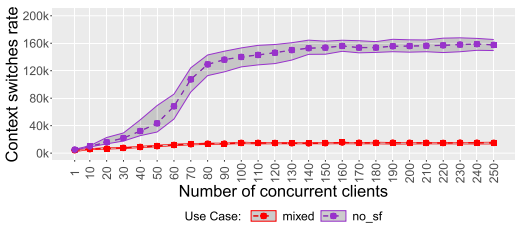

FIGURE 14. Measured context switches per second (incl. 95% confidence intervals) on the server hosting the service in use cases *mixed* and *no_sf*.

Among all use cases, the *direct* case performs best. This results from the fact that it does not include any SFs or PEP. It saturates at 510 concurrent users with 1710 *a/s* and an average access time of 297 *ms*.

Use case *zt* shows a comparable performance to case *logger_ips*. In the ZT scenario all packets also run through logger and IPS in the same order. The difference to the *logger_ips* case here lies in the absence of the SFP Logic. But since the PEP's query to the SFP Logic only shows a RTT of 0.363 *ms* on average, this is hardly noticeable. However, this RTT depends on the complexity of the implemented SFP Logic and whether a standing TLS connection to the PEP is used or not.

Overall, the case with direct service access was the best performing case in all metrics, but at the same time it is the most insecure without ZT and SFC functionalities. However, the realistic ZTSFC use case *mixed* could outperform the ZT case. At 140 concurrent clients, which is the saturation point of the *mixed* case, it has 30% lower SATs and processes also around 30% more RARs per second. This is mainly due to a traffic smoothing effect explained previously. This effect is most pronounced when the distribution of packets across different SFPs is most uniform. It also shows that having only clients of one type leads to worse service access times even when no SFs are applied. This illustrates the complex performance interrelationship between different parts of a network architecture. What has been described also applies to normal SFC. For ZTSFC, this means that the trust thresholds used for services and the chaining decisions made not only have a decisive effect on security, but also on performance and thus on the user experience.

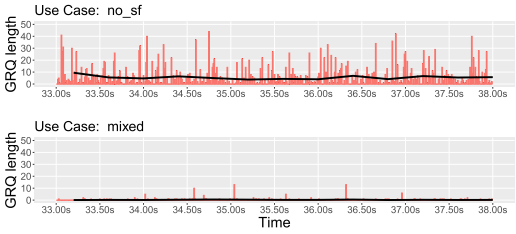

FIGURE 15. GRQ length for the Service during a 5 *s* time window and measured every 10 *ms* during the test runs for the *no_sf* and *mixed* use cases. The black line represents the floating average of the GRQ length over the last 400 *ms*.

D. AVERAGE THROUGHPUT

The average throughput results are shown in Figure 16. The values in the direct case are significantly higher. ZT and ZTSFC show both similar results with an advantage for ZTSFC. At 40 concurrent clients, a cumulative throughput of 18.4 *Gbps* in the ZTSFC case and 17.6 *Gbps* in the ZT case is reached. The reason that *direct* is faster and ZT and ZTSFC show similar results is the fact that the PEP here represents a bottleneck. During the test with 2 concurrent clients, it spent 48% of the time decrypting and encrypting TLS records. In addition, 20% of the threads had to wait for network interrupts to be processed. On the PEP server, we measured 86144 interrupts per second. It thus slows the packet rate down in a way that all following SFs in the SFPs can also process it at that speed. An increase in the number of clients was not possible due to RAM limitations on the used hardware. All received data had to be buffered in RAM.

In summary, the PEP limits the maximum throughput. This limitation results from the computational intensive cryptographic operations related to traffic encryption.

E. LOAD

Measured on the servers that host logger, PEP and IPS, the load results for *mixed* and *zt* are shown in Figure 17.

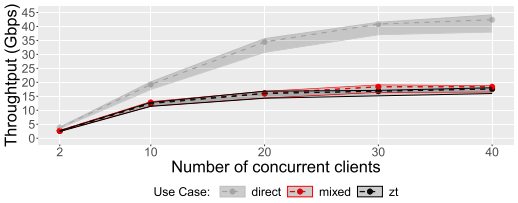

FIGURE 16. Measured average throughput (incl. 95% confidence intervals). The number of clients is evenly distributed between srv5 and srv8. The measured values are then accumulated.

Starting at 140 concurrent clients at which both cases saturate at the latest, the CPU usage on srv0 (logger) could be reduced by 25% in the ZTSFC *mixed* scenario. The same trend applies to interrupts and context switches which are in the ZTSFC case around 5% and 50% lower, respectively.

The same effect does not occur on the server that hosts the IPS. Here the *mixed* case causes similar interrupt rates and CPU usage. At the same time the number of concurrently processed RARs at the IPS is higher as it can be seen by the higher number of context switches. For the *mixed* case and with 140 clients, 191095 context switches per second are recorded, which is 42% higher compared to *zt*.

For the *mixed* case, the CPU usage on the PEP is due to the higher amount of processed accesses per second higher than in the *zt* case. This is as expected. Also the higher rate of interrupts results from the higher RAR rate but also the network interrupts caused by the additional SFP logic communication. At the same time, the PEP can work more efficiently due to the lower number of context switches. This results from the smoothness effect already explained in the access time & saturation part. As the packets are distributed over different SFPs, the arrival of the service responses at the PEP are less bursty.

These observations show that it is not a trivial task to reduce the load on the servers in general. Certain interrelationships between SFs such as the smoothness effect lead to an overall better performance in terms of SAT and saturation. This in turn results in better utilized servers and thus higher loads.

F. RESULTS

In the evaluation, we compared three different network architectures. The first architecture represented direct service access without MaS functions. As expected, this architecture performed best in terms of performance. However, due to the lack of security functions, we will not discuss this architecture further. The second architecture was a ZT architecture with security functions installed at network choke points. The third architecture we evaluated was our novel ZTSFC architecture.

With ZTSFC, network traffic is classified based on the trust established in a RAR. By this, we can achieve lower SATs for clients with a high level of trust. That was shown in the mixed test runs in Figure 10. With 150 concurrent clients, the SATs for the 25% of the clients having a trust level sufficient to be directly forwarded to the service was

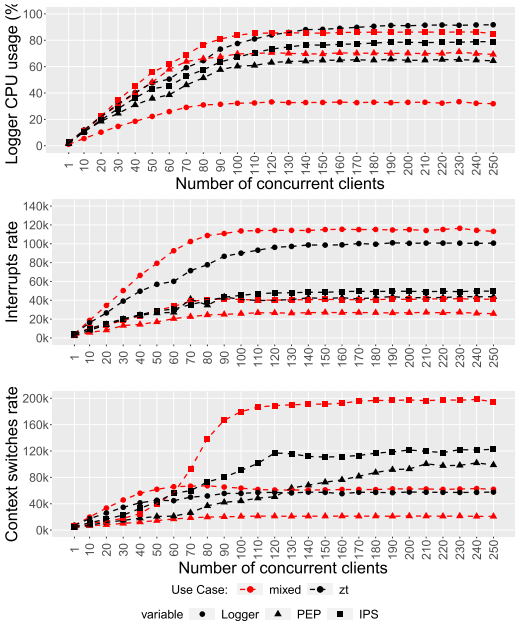

FIGURE 17. Load test results based on measured CPU usage as well as context switches and interrupts per second.

between 54 ms and 152 ms. In comparison, clients experience SATs between 97 ms and 341 ms in the case of a ZT network. This good result for RARs with high trust levels worsens with more similar trusted RARs, as can be seen in the *no_sf* case with SATs between 67 ms and 328 ms. The smoothness effect shows a diminishing return the more clients of one kind are requesting the service. With ZTSFC, clients with a high level of trust can access a service directly or have shorter SFCs. Individual SFCs can be used to meet just the required level of trust for service access and thus affect the user experience only to the minimum level necessary. At the same time, different chaining decisions such as different orders of MaS functions can have unforeseen effects on traffic and sometimes even lead to better performance. That can be seen when comparing use cases *logger_ips* and *ips_logger* in Figure 11, where *logger_ips* achieves significantly better SATs. The best results are achieved in a network with RARs with different trust levels and thus different SFPs.

In summary, ZTSFC allows incoming RARs to be classified in a fine-granular manner. This allows the network to react dynamically to different trust levels. RARs with a high trust level experience low access times. On the other hand, we can avoid saving demanding MaS actions such as DPI because of too high server load by selectively using these security functions only where it is necessary due to a insufficient trust level.

IX. DISCUSSION

After presenting ZTSFC and detailing its improvements, we will discuss various other aspects of the ZTSFC architecture in this section.

We presented ZTSFC as an architecture to integrate the MaS functions into a ZT architecture. We leverage SFC to achieve this. All MaS functions implemented by this can run in a distributed system. However, depending on the particular network architecture, the same MaS functions can also run on a single physical machine and communicate over a local Linux network namespace, such as implemented in Docker. However, inter-process communication, e.g., through Linux pipes, is also possible. A scenario in which all functions run together with the PEP on potent hardware is therefore just as conceivable as all functions and the ZT components realized in virtual machines distributed in a cloud.

We implemented our ZTSFC PoC with the help of HTTPS. As discussed in Section V, this HTTPS-based approach has clear advantages, but also some disadvantages. It could be arguably integrated into Google's BeyondCorp architecture [2], for example, because BeyondCorp also works with HTTP. However, ZTSFC can also be implemented at lower layers, e.g., with multiprotocol label switching, as shown in the work of Bradatsch et al. [7] or with software-defined networking similar to Yu et al. [21]. Both describe techniques to steer traffic in a network on network layers two and three, respectively.

ZTSFC decides which MaS functions should be used for each RAR, based on how much trust is associated with that RAR. In our ZTSFC PoC, we use a criteria-based approach, similar to what is described in NIST's ZT white paper [4]. For access to be permitted, a RAR must satisfy all these criteria. Once these criteria are evaluated, the PDP then decides which MaS functions and actions are applied to that RAR. There are alternative methods for making this decision, like using score-based algorithms. These algorithms calculate a trust score by considering attributes like the access time or which authentication factors were used. In works such as those by Tao et al. [39], Tian et al. [60], and da Silva et al. [61], score-based methods have been examined. Based on the trust score, it is decided which MaS functions should be applied. ZTSFC is compatible with criteria-based as well as score-based approaches.

For our criteria-based policies we used the ALFA policy language. ALFA shows the advantage of easy readability while having the same expressivness as XACML [16]. However, in ZTSFC it is equally possible to write the criteria-based policies in UCON [17], UCON+ [13] or similar policy languages.

Throughout this paper, we have used the example of an employee trying to access a service with an unmanaged device. To compensate for not having a managed device, we relied on two MaS functions: MFA and IPS. Deciding on the right MaS functions to use when certain criteria are not met, or based on a specific trust score, can be challenging. This holds true whether unmet criteria are being addressed or a trust score is being interpreted. A systematic way to choose the right MaS functions for each situation needs to be established. Creating this systematic approach is a main goal for our future studies.

While ZTSFC allows for greater granularity and flexibility in access decisions by incorporating SFs, this also introduces additional complexity to the creation of access control policies. To prevent unintended access decisions, careful consideration is necessary when integrating SFs into the decision-making process. Master policies, as described by Google's BeyondCorp [2], which take priority over all other policies, can help ensure that new policies incorporating SFs do not unintentionally grant access if certain master conditions are unmet.

X. CONCLUSION

In this paper, we introduced, implemented, and evaluated a novel network security architecture called ZTSFC. It is based on a concept from a previous work of ours [9]. ZTSFC integrates MaS functions and ZT components. In this way, we achieve three main improvements over regular ZT architectures: (1) The integration of MaS functions into the PDP's Auth* decision-making process, (2) the implementation of a direct and efficient information flow between MaS functions and PDP, and (3) the reduction of hardware load and improvement of user experience.

To prove the feasibility of ZTSFC, we implemented a ZTSFC prototype. That prototype leverages ALFA's obligation feature to specify under which conditions certain MaS functions are applied to RARs. Traffic steering is implemented with HTTPS-based SFC. In our discussion, we argue that this is particularly useful in environments where predominantly web-based apps and services are used, as in Google's BeyondCorp architecture. Our HTTPS-based approach allows MaS functions to access packet payloads and ensures confidentiality, authentication, integrity, and perfect forward secrecy for any communication. Using signed tokens embedded in HTTP custom headers, proof of transit for MaS functions is proven.

Using this PoC, we detailed the first and second improvements with representative use cases. To assess the third improvement, we compared the ZTSFC architecture to a standard ZT architecture, looking at both hardware load and user experience. While accessing services directly is faster, ZTSFC achieves a balance between user experience and security. It considers the needed security for a ZT approach but can skip some MaS functions when the RAR is already sufficiently trusted.

Our evaluation also points out the complex performance interrelationships between different parts of a network architecture where SFs modify and smooth traffic. Different chaining orders of SFs have a significant impact on network performance. That should not only be valid for ZTSFC but for SFC in general.

In terms of future work, we see a number of open challenges. For example, how to bring continuous authorization to ZTSFC, as described in [13]. That enables us to adjust the chosen SFC/SFP even while the service access is in progress. To address this challenge, we currently examine various strategies for how the involved control

```
policy serviceAccess {
  target clause request.service == service
  apply firstApplicable
    rule use_case_no_sf {
      target clause request.action == GET
      condition user.role == "service_administrator" and
        currentTime>"09:00:00":time or currentTime<"17:00:00":time and
        user.authentication == authentication.password and
        device.authentication == authentication.certificate and
        device.type == "managed" and
        rar.number < 25 and
        rar.failed < 3
      permit
    }
    rule use_case_logger {
      target clause request.action == POST
      condition user.role == "service_administrator" and
      currentTime>"09:00:00":time or currentTime<"17:00:00":time and
        user.authentication == authentication.password and
        device.authentication == authentication.certificate and
        device.type == "managed" and
        rar.number < 25 and
        rar.failed < 3
      permit
      on permit {
        obligation notify {
          action = apply_sfc
          sfc = sf_logger
          sf_logger_action = logging
        }
      }
    }
    rule use_case_ips {
      target clause request.action == GET
      condition user.role == "service_administrator" and
        currentTime>"09:00:00":time or currentTime<"17:00:00":time and
        user.authentication == authentication.password and
        device.authentication == authentication.certificate and
        device.type == "unmanaged" and
        rar.number < 25 and
        rar.failed < 3
      permit
      on permit {
        obligation notify {
          action = apply_sfc
          sfc = sf_ips
          sf_ips_action = dpi
        }
      }
    }
    rule use_case_logger_ips_and_ips_logger {
      target clause request.action == POST
      condition user.role == "service_administrator" and
        currentTime>"09:00:00":time or currentTime<"17:00:00":time and
        user.authentication == authentication.password and
        device.authentication == authentication.certificate and
        device.type == "unmanaged" and
        rar.number < 25 and
        rar.failed < 3
      permit
      on permit {
        obligation notify {
          action = apply_sfc
          sfc = sf_ips, sf_logger
          sf_ips_action = dpi
          sf_logger_action = logging
        }
      }
    }
    rule default_deny {
      deny
    }
}
```

LISTING 6. ALFA policy used for the ZTSFC use cases in the evaluation.

plane components can trigger and communicate such re-authorizations. As mentioned in the discussion section, we see the need for a systematic approach to decide which combinations of MaS functions can make up for specific criteria or attributes that are not. Efforts are underway to implement such an approach. Furthermore, in the introduced ZTSFC architecture, MaS functions need to be arranged behind the PEP. An open challenge is to enable the option to orchestrate MaS functions that are logically positioned in front of the PEP. For this, we investigate related research works such as [22]. Lastly, another starting point for future research is the integration of third-party service functions that do not run in the company's own network domain, for example, and how their trustworthiness can be ensured.

APPENDIX
PROOF OF CONCEPT - EVALUATION POLICIES

For all ZTSFC use cases, we defined ALFA policies used in the evaluation. The set of policy rules is shown in Listing 6.

ACKNOWLEDGMENT

The authors alone are responsible for the content of this article. For the creation of Section VII, the language tool Speak [1] was used for support.

REFERENCES

[1] (2023). *Speak*. Accessed: Oct. 3, 2023. [Online]. Available: https://www.speak.com

[2] R. Ward and B. Beyer, "BeyondCorp: A new approach to enterprise security," *Login*, vol. 39, no. 6, pp. 6–11, Dec. 2014.

[3] E. Gilman and D. Barth, *Zero Trust Networks: Building Secure Systems in Untrusted Networks*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Jun. 2017.

[4] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, *Zero Trust Architecture*, NIST document 800–207, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Aug. 2020.

[5] J. Garbis and J. W. Chapman, *Zero Trust Security*. Berlin, Germany: Springer, 2021.

[6] J. Kindervag, S. Balaouras, and L. Coit, "No more chewy centers: Introducing the zero trust model of information security," Forrester, Cambridge, MA, USA, Tech. Rep. RES56682, 2016.

[7] L. Bradatsch, M. Haeberle, B. Steinert, F. Kargl, and M. Menth, "Secure service function chaining in the context of zero trust security," in *Proc. IEEE 47th Conf. Local Comput. Netw. (LCN)*, Sep. 2022, pp. 123–131.

[8] J. Halpern and C. Pignataro, *Service Function Chaining (SFC) Architecture*, document RFC 7665, RFC Editor, Oct. 2015.

[9] L. Bradatsch, F. Kargl, and O. Miroshkin, "Zero trust service function chaining," in *Proc. Conf. Networked Syst.* (Electronic Communications of the EASST), vol. 80, 2021, pp. 1–3.

[10] M. Walker, *CEH Certified Ethical Hacker Practice Exams (All-in-One)*. New York, NY, USA: McGraw-Hill, 2021.

[11] B. Osborn, J. McWilliams, B. Beyer, and M. Saltonstall, "BeyondCorp: Design to deployment at Google," *Login*, vol. 41, no. 1, pp. 28–34, 2016.

[12] L. Cittadini, B. Spear, B. Beyer, and M. Saltonstall, "BeyondCorp—Part III: The Access Proxy," *Login*, vol. 41, no. 4, pp. 28–33, 2016.

[13] T. Dimitrakos, T. Dilshener, A. Kravtsov, A. La Marra, F. Martinelli, A. Rizos, A. Rosetti, and A. Saracino, "Trust aware continuous authorization for zero trust in consumer Internet of Things," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 1801–1812.

[14] M. Colombo, A. Lazouski, F. Martinelli, and P. Mori, "A proposal on enhancing XACML with continuous usage control features," in *Grids, P2P and Services Computing*, F. Desprez, V. Getov, T. Priol, and R. Yahyapour, Eds. Boston, MA, USA: Springer, 2010, pp. 133–146.

[15] A. Lazouski, F. Martinelli, and P. Mori, "A prototype for enforcing usage control policies based on XACML," in *Proc. 9th Int. Conf., Trust, Privacy Secur. Digit. Bus. (TrustBus)*. Cham, Switzerland: Springer, Sep. 2012, pp. 79–92.
[16] E. Rissanen. (Jan. 2013). *eXtensible Access Control Markup Language (XACML) Version 3.0*. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html
[17] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 128–174, Feb. 2004.
[18] M. Boucadair. (Oct. 2016). *Service Function Chaining (SFC) Control Plane Components & Requirements*. Internet Engineering Task Force, Internet-Draft. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-control-plane-08
[19] L. Iffländer, L. Beierlieb, N. Fella, S. Kounev, N. Rawtani, and K.-D. Lange, "Implementing attack-aware security function chain reordering," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. Companion (ACSOS-C)*, Aug. 2020, pp. 194–199.
[20] A. Shameli-Sendi, Y. Jarraya, M. Pourzandi, and M. Cheriet, "Efficient provisioning of security service function chaining using network security defense patterns," *IEEE Trans. Services Comput.*, vol. 12, no. 4, pp. 534–549, Jul. 2019.
[21] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan, "PSI: Precise security instrumentation for enterprise networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.
[22] D. Eidle, S. Y. Ni, C. DeCusatis, and A. Sager, "Autonomic security for zero trust networks," in *Proc. IEEE 8th Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, Oct. 2017, pp. 288–293.
[23] R. Vanickis, P. Jacob, S. Dehghanzadeh, and B. Lee, "Access control policy enforcement for zero-trust-networking," in *Proc. 29th Irish Signals Syst. Conf. (ISSC)*, Jun. 2018, pp. 1–6.
[24] N. Ghate, S. Mitani, T. Singh, and H. Ueda, "Advanced zero trust architecture for automating fine-grained access control with generalized attribute relation extraction," *IEICE Proc. Ser.*, vol. 68, nos. 1–5, p. C1-5, Dec. 2021.
[25] K. Ramezanpour and J. Jagannath, "Intelligent zero trust architecture for 5G/6G networks: Principles, challenges, and the role of machine learning in the context of O-RAN," 2021, *arXiv:2105.01478*.
[26] H. Chen, D. Zou, H. Jin, S. Xu, and B. Yuan, "SAND: Semi-automated adaptive network defense via programmable rule generation and deployment," *Sci. China Inf. Sci.*, vol. 65, no. 7, Jul. 2022, Art. no. 172102.
[27] S. Turner, D. Holmes, C. Cunningham, J. Budge, P. McKay, A. Cser, H. Shey, and M. Maxim, "A practical guide to a zero trust implementation," Forrester, Cambridge, MA, USA, Tech. Rep. RES157736, Mar. 2021.
[28] J. Kindervag, S. Balaouras, and L. Coit, "Build security into your network's DNA: The zero trust network architecture," Forrester, Cambridge, MA, USA, Tech. Rep. RES57047, Nov. 2010.
[29] J. Kindervag, S. Balaouras, A. Spiliotes, and P. Dostie, "Five steps to a zero trust network," Forrester, Cambridge, MA, USA, Tech. Rep. RES120510, Jul. 2016.
[30] C. Cunningham, "The zero trust extended (ZTX) ecosystem," Forrester, Cambridge, MA, USA, Tech. Rep. RES137210, Jan. 2018.
[31] V. Escobedo, B. Beyer, M. Saltonstall, and F. Zyzniewski, "BeyondCorp 5: The user experience," *Login*, vol. 42, no. 3, pp. 38–43, 2017.
[32] M. Janosko, H. King, B. Beyer, and M. Saltonstall, "BeyondCorp 6: Building a healthy fleet," *Login*, vol. 43, no. 3, pp. 24–30, 2018.
[33] B. Beyer, C. Beske, J. Peck, and M. Saltonstall, "Migrating to BeyondCorp: Maintaining productivity while improving security," *Login*, vol. 42, no. 2, pp. 49–55, 2017.
[34] U. Kuederli, L. Neher, and F. Faistauer, "Zero trust architecture: A paradigm shift in cybersecurity and privacy," White Paper, 2023. Accessed: Nov. 7, 2023. [Online]. Available: https://www.pwc.com/sg/en/publications/assets/page/zero-trust-architecture.pdf
[35] N. F. Doherty, C. Ashurst, and J. Peppard, "Factors affecting the successful realisation of benefits from systems development projects: Findings from three case studies," *J. Inf. Technol.*, vol. 27, no. 1, pp. 1–16, Mar. 2012.
[36] S. Riley, N. MacDonald, and L. Orans, "Zero trust architecture and solutions," Gartner, Stamford, CT, USA, Tech. Rep. G00386774, Apr. 2019.
[37] P. Bjork, G. Gordon, A. Lanusse, S. Navare, H. Lantinga, and C. Arakelian. (Nov. 2019). *Zero Trust Secure Access to Traditional Applications with VMware*. [Online]. Available: https://techzone.vmware.com/resource/zero-trust-secure-access-traditional-applications-vmware
[38] B. Lee, R. Vanickis, F. Rogelio, and P. Jacob, "Situational awareness based risk-adaptable access control in enterprise networks," in *Proc. 2nd Int. Conf. Internet Things, Big Data Secur.*, 2017, pp. 400–405.
[39] Y. Tao, Z. Lei, and P. Ruxiang, "Fine-grained big data security method based on zero trust model," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2018, pp. 1040–1045.
[40] Q. Yao, Q. Wang, X. Zhang, and J. Fei, "Dynamic access control and authorization system based on zero-trust architecture," in *Proc. 1st Int. Conf. Control, Robot. Intell. Syst.* New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 123–127.
[41] S. Mehraj and M. T. Banday, "Establishing a zero trust strategy in cloud computing environment," in *Proc. Int. Conf. Comput. Commun. Informat. (ICCCI)*, Jan. 2020, pp. 1–6.
[42] Y. He, D. Huang, L. Chen, Y. Ni, and X. Ma, "A survey on zero trust architecture: Challenges and future trends," *Wireless Commun. Mobile Comput.*, vol. 2022, Jun. 2022, Art. no. 6476274.
[43] (Oct. 2014). *McAfee Report Reveals Organizations Choose Network Performance Over Advanced Security Features*. [Online]. Available: https://www.businesswire.com/news/home/20141028006800/en/
[44] Z. Zaheer, H. Chang, S. Mukherjee, and J. van der Merwe, "EZTrust: Network-independent zero-trust perimeterization for microservices," in *Proc. ACM Symp. SDN Res.* New York, NY, USA: Association for Computing Machinery, 2019, pp. 49–61, doi: 10.1145/3314148.3314349.
[45] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 179–196, doi: 10.1145/3335772.3335934.
[46] F. Brockners, S. Bhandari, T. Mizrahi, S. Dara, and S. Youell. (Nov. 2020). *Proof of Transit*. Internet Engineering Task Force, Internet-Draft. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-proof-of-transit-08
[47] G. Sun, Y. Li, H. Yu, A. V. Vasilakos, X. Du, and M. Guizani, "Energy-efficient and traffic-aware service function chaining orchestration in multi-domain networks," *Future Gener. Comput. Syst.*, vol. 91, pp. 347–360, Feb. 2019.
[48] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, and B. Akbari, "Joint energy efficient and QoS-aware path allocation and VNF placement for service function chaining," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 1, pp. 374–388, Mar. 2019.
[49] P. Quinn, U. Elzur, and C. Pignataro, *Network Service Header (NSH)*, document RFC 8300, IETF, Fremont, CA, USA, Jan. 2018.
[50] A. Farrel, S. Bryant, and J. Drake, *An MPLS-Based Forwarding Plane for Service Function Chaining*, document RFC 8595, IETF, Fremont, CA, USA, Jun. 2019.
[51] E. Rescorla and B. Korver, *Guidelines for Writing RFC Text on Security Considerations*, document RFC 3552, Network Working Group, Jun. 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3552
[52] A. Viswanathan, E. Rosen, and R. Callon, *Multiprotocol Label Switching Architecture*, document RFC 3031, Network Working Group, Jan. 2001. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3031
[53] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. Gódor, G. Szabó, and T. Westholm, "Deep packet inspection tools and techniques in commodity platforms: Challenges and trends," *J. Netw. Comput. Appl.*, vol. 35, no. 6, pp. 1863–1878, Nov. 2012.
[54] K. Yao, G. B. de Pádua, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Log4Perf: Suggesting and updating logging locations for web-based systems' performance monitoring," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 488–531, Jan. 2020.
[55] S. Sharifian, S. A. Motamedi, and M. K. Akbari, "A content-based load balancing algorithm with admission control for cluster web servers," *Future Gener. Comput. Syst.*, vol. 24, no. 8, pp. 775–787, Oct. 2008.
[56] A. Bestavros, "Using speculation to reduce server load and service time on the WWW," in *Proc. 4th Int. Conf. Inf. Knowl. Manage. (CIKM)*. New York, NY, USA: Association for Computing Machinery, 1995, pp. 403–410.

[57] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Web server performance measurement and modeling techniques," *Perform. Eval.*, vol. 33, no. 1, pp. 5–26, Jun. 1998.
[58] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "CPU demand for web serving: Measurement analysis and dynamic estimation," *Perform. Eval.*, vol. 65, nos. 6–7, pp. 531–553, Jun. 2008.
[59] A. Kattepur and M. Nambiar, "Service demand modeling and performance prediction with single-user tests," *Perform. Eval.*, vol. 110, pp. 1–21, Apr. 2017.
[60] X. Tian and H. Song, "A zero trust method based on BLP and BIBA model," in *Proc. 14th Int. Symp. Comput. Intell. Design (ISCID)*, Dec. 2021, pp. 96–100.
[61] G. R. da Silva, D. F. Macedo, and A. L. dos Santos, "Zero trust access control with context-aware and behavior-based continuous authentication for smart homes," in *Proc. Anais 21st Simpósio Brasileiro Segurança Informação Sistemas Computacionais (SBSeg)*, Oct. 2021, pp. 43–56.

LEONARD BRADATSCH received the B.Sc. and M.Sc. degrees in computer science from Ulm University, Germany, where he is currently pursuing the Ph.D. degree with the Institute of Distributed Systems. His current research interests include zero trust security and network security.

OLEKSANDR MIROSHKIN received the master's and Ph.D. degrees from Donetsk National Technical University, Ukraine, in 2007 and 2013, respectively. His current research interests include network security and high-performance simulation environments.

FRANK KARGL (Member, IEEE) is currently a Tenured Professor and the Director of the Institute of Distributed Systems, Ulm University. His current research interests include security and privacy in networked systems with a special focus on automotive and the IoT systems. One aspect of his research is in the area of trust in networking and particularly zero trust security architectures.