NETCAP: Data-Plane Capability-Based Defense Against Token Theft in Network Access

Osama Bajaber
Virginia Tech
obajaber@vt.edu

Bo Ji
Virginia Tech
boji@vt.edu

Peng Gao
Virginia Tech
penggao@vt.edu

***Abstract*—Tokens play a vital role in enterprise network access control by enabling secure authentication and authorization across various protocols (e.g., JSON Web Tokens, OAuth 2.0). This allows users to access authorized resources using valid access tokens, without the need to repeatedly submit credentials. However, the ambient trust granted to all processes within an authorized host, combined with long token lifetimes, creates an opportunity for malicious processes to hijack tokens and impersonate legitimate users. This threat affects a wide range of protocols and has led to numerous real-world incidents.**

In this paper, we present NETCAP, a new defense mechanism designed to prevent attackers from using stolen tokens to access unauthorized resources in enterprise environments. The core idea is to introduce *unforgeable, process-level capabilities* that are bound to authorized processes. These capabilities are continuously embedded in the processes' network traffic to target resources for validation and are frequently refreshed. This binding between process identity and capability ensures that even if access tokens are stolen by malicious processes, they cannot be used to pass authentication without valid capabilities. To support the high volume of requests generated by processes in the network, NETCAP introduces a novel data-plane design based on programmable switches and eBPF. Through multiple optimization techniques, our system supports inline generation and embedding of capabilities, allowing large volumes of traffic to be processed at line rate with little overhead. Our extensive evaluations show that NETCAP maintains line-rate network performance across a variety of protocols and real-world applications with negligible overhead, while effectively securing these applications against token theft attacks.

I. INTRODUCTION

Token-based authentication is widely adopted for network access control across enterprise environments (e.g., cloud platforms, enterprise networks, data centers), and is used by various protocols and services (e.g., OAuth, Kerberos, HTTPS, FTP, and RDP). After a client is authenticated (e.g., through credentials such as username and password), the service issues an access token. This token is lightweight and allows the client application to access the requested resources without requiring repeated authentication. However, a major threat associated with token-based authentication is *access token hijacking*. In such attacks, an adversary can steal a valid access token and use it to access the services before the token expires.

This threat is broad and manifests in various forms across different environments. For instance, in cloud and web environments, stolen tokens from the browser cache can enable session hijacking attacks [1]. In enterprise networks, attackers may extract Kerberos ticket-granting tickets from memory in pass-the-ticket attacks [2], gaining unauthorized permissions to services. In remote administration settings, memory scraping techniques [3] can be used to extract in-memory SSH private keys, allowing attackers to execute arbitrary remote commands. This risk is further exacerbated in data centers and large-scale cloud deployments, where tokens are frequently passed between processes and stored in shared memory, dramatically increasing the attack surface for token theft. Such attacks have resulted in numerous security breaches in high-profile organizations [4], [5].

Fig. 1 illustrates this threat scenario. When a client machine attempts to access a server, the server application issues an access token to the client upon successful authentication. However, an attacker, operating a malicious process `P2` on the same client machine or `P3` on a different machine, can steal the access token issued to the legitimate process `P1` using various means. For example, an Apache server may issue a JSON Web Token (JWT) stored in the client browser's local storage. Attackers can steal such tokens using techniques like cross-site scripting [6], which injects malicious scripts into web pages to extract tokens from browser storage, or memory scraping [1], which retrieves tokens from application memory due to improper memory management. With a stolen token, the attacker can impersonate the legitimate client process and access protected server resources. In Section VII-A, we show how this threat materializes in real attacks against different protocols using various exploitation techniques.

This threat stems from **two fundamental issues** in current token-based schemes: (1) *Access tokens are not bound to specific authorized processes*. Authentication servers rely on ambient trust, granting all processes within an authorized host equal access to the token. Since access tokens typically encode only user-level information, they cannot differentiate between processes on the client. As a result, any process, including malicious ones, can reuse a valid token to access protected resources. (2) *Access tokens have a long lifetimes*, often ranging from hours to days (see Table I). This extended

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240273
www.ndss-symposium.org

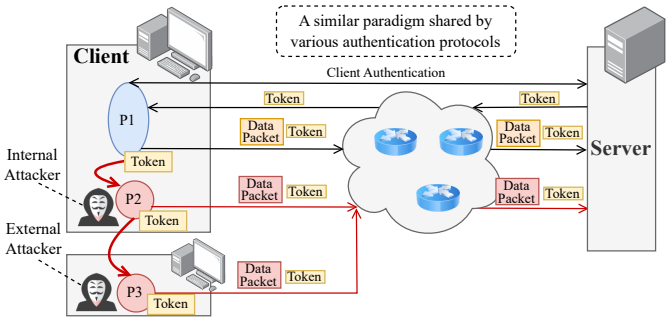


Fig. 1: Access token hijacking is a prevalent threat that manifests in different forms across various protocols (e.g., OAuth, Kerberos, SMTP, RDP, HTTPS).

validity window gives attackers ample opportunity to exploit a stolen token to cause damage before it expires.

Prior efforts have primarily focused on developing *protocol-specific* countermeasures by enhancing the security of authentication protocols. These include mechanisms for detecting token misuse [7], strengthening token validation procedures [8], [9], and hardening protocol implementations [10]. However, none of these solutions directly address the fundamental *ambient trust* issue, where access tokens are implicitly trusted by any process on a host once issued. Consequently, new zero-day attacks [11] continue to emerge that bypass these defenses, forcing defenders into an endless arms race of patching vulnerabilities after exploitation.

Goal & challenges. In this work, we aim to directly address the fundamental issue of ambient trust and distinguish processes making access requests. Our idea is to enable a form of lightweight authentication credential that is cryptographically bound to the identity of the requesting process. Without this credential, any access request will fail validation.

However, several key challenges need to be addressed: (1) Directly modifying existing access tokens to incorporate process identity would lead to a protocol-dependent solution, requiring substantial changes to each protocol and application. Such an approach lacks scalability and generalizability. (2) Supporting secure access requests from every process in the network results in substantial traffic volume, and continuously handling these requests presents significant performance challenges. Ideally, the defense system should verify each process-level access request in real time without adding latency or degrading overall network performance.

Contributions. We present NETCAP, a defense system designed to secure network access against access token hijacking. To address the ambient trust problem, we introduce the notion of a *capability*, a lightweight, unforgeable token that grants its holder access to a specific resource. During capability generation, we *cryptographically bind the capability to the identifier (i.e., PID) of the authorized process* using a secure hash function (see Eq. (1)). This binding ensures that the capability is unforgeable and enables NETCAP to distinguish authorized processes from unauthorized ones. The authorized process embeds its unique capability in outbound traffic to remote resources, while NETCAP continuously validates the capability along with relevant traffic metadata, ensuring that only requests from legitimate processes are allowed.

TABLE I: Token lifetime for different services

Token Type	Service	Token Lifetime
JWT	Web APIs	1 hour to 7 days [13]
OAuth 2.0	Mobile [14], web [15], and IoT [16]	Hours to months [17]
Service ticket	Kerberos authentication	10 hours [18]
SSH authentication keys	Secure Shell (SSH) Protocol	Indefinitely [19]

The key advantage of introducing this separate capability, alongside the original access token, is that even if the token is stolen, requests lacking a valid capability will be rejected. This design also adds an additional layer of protection while remaining protocol-independent, requiring *no modifications* to underlying protocols or applications and complementing existing protocol-specific enhancements.

To efficiently handle high volumes of access requests, NETCAP introduces a novel *data-plane design* that leverages a co-design between programmable switches and eBPF to implement the capability mechanism directly within the network data plane. Programmable switches offer data-plane programmability via the P4 language [12], enabling custom packet processing at Tbps line rate. This makes it feasible to design specialized packet formats that carry capabilities, and to inspect and validate them in real time. eBPF is an emerging kernel technology that enables sandboxed programs to run safely within the operating system (OS) kernel, without requiring modifications to the kernel source code, host OS, or user-space applications, simplifying deployment. We developed lightweight eBPF programs to efficiently trace processes, bind capabilities within hosts, and embed them into outbound traffic. Additionally, we designed a set of optimization techniques to support efficient capability generation, validation, refresh, and management.

We carefully designed our capability-based defense protocol and its data-plane realization to ensure robust coordination among components and strong resilience against different attack vectors, including capability replay attacks, IP/PID spoofing, and attempts to disrupt the normal operations of legitimate processes (see Section III-B).

We compared NETCAP's switch-based design with an eBPF-only design (i.e., without the switch data plane), which suffers from both security and performance limitations. As shown in Section VII-F, the eBPF-only design incurs latency that is four orders of magnitude higher than our switch-based design, incurring noticeable delays in user access. It also becomes easily saturated as packet rates increase (e.g., 20K packets/s), making it vulnerable to saturation attacks. In contrast, NETCAP's switch-based design sustains over 600K packets/s at line rate with negligible delays. Offloading enforcement to the switch data plane is not merely an optimization; it is essential for maintaining resilience against saturation attacks and preventing the host from becoming a bottleneck.

Evaluations. We deploy NETCAP in a physical testbed and use synthetic and real-world enterprise traces for evaluations. We extensively evaluate NETCAP in terms of defense effectiveness, scalability, system capacity, and overhead. The results demonstrate that: (1) NETCAP can protect a wide range of protocols from attackers attempting to hijack legitimate users' sessions while adding only a negligible 130 nanoseconds overhead to the packet processing time. (2) NETCAP can successfully scale to real-world enterprise traces from DARPA OpTC [20] and LANL Unified Host and Network [21] datasets, while achieving 99.9 Gbps throughput on the 100 Gbps (per-port) programmable switch, incurring negligible overhead to legitimate traffic. (3) NETCAP can seamlessly support different real-world applications (e.g., Apache Server, NodeJS Server, and FTP Server) without modifying the applications' code. (4) The eBPF-based programs add a negligible 3-9 microseconds to the total time of the monitored kernel events, imposing minimum overhead on the host performance.

NETCAP is the first work to realize capabilities within the data plane to mitigate token hijacking for secure network access. It introduces a capability-based defense that extends beyond a single host to enforce in-network, process-level authentication for remote resource access, while achieving high-throughput validation. NETCAP supports a wide range of protocols and can be seamlessly deployed without modifications to underlying protocols, applications, or kernel source code. We open-source the prototype of NETCAP at [22]

II. BACKGROUND

Capabilities-based access control. To enforce access control at the process level, one approach is identity-based access control (IBAC), where a central authority (e.g., an access control list) verifies access rights based on the identities of subjects (e.g., processes) and objects (e.g., resources) [23]. However, this approach is difficult to scale due to the large number of rules and the dynamic nature of process activities such as frequent spawning and termination, resulting in high management overhead. In contrast, capability-based access control [24] adopts a decentralized model. The capability itself serves as proof of access rights to specific resources, thereby simplifying access control management.

Capabilities have been used to regulate access in cloud computing [25] and the Internet of Things [26]. However, these systems operate at a coarse granularity and lack the ability to distinguish between individual processes within a host. Other efforts have integrated capabilities into operating systems (OSes) [27], [28] to control process access to memory locations, but these solutions are confined to a single host. Moreover, unlike NETCAP, these systems require extensive modifications to the OS kernel to enforce capabilities. To date, no existing work enforces capabilities for processes that continuously access remote resources.

Programmable data planes. Programmable switches have gained widespread attention for their ability to support data-plane programmability through the P4 language [12], while achieving Tbps line-rate performance. Fig. 2 illustrates the Protocol Independent Switch Architecture [12]. A P4 program defines the packet headers and protocols, along with a programmable parser that extracts these headers. The headers are then processed through multiple hardware stages in the switch, each performing match/action operations. After processing, a programmable deparser reconstructs the packet headers before forwarding the packets. The main P4 processing takes place in a pipeline of match/action tables, where each table matches on specific keys (e.g., header fields) and applies actions such as arithmetic or bitwise operations. Each hardware stage includes Arithmetic Logic Units (ALUs) to support packet-level computations. Static RAM (SRAM) and Ternary Content Addressable Memory (TCAM) are used to store table entries, supporting exact and wildcard (range) matches, respectively. SRAM can also be configured as register arrays to maintain state across packets for stateful processing. However, *these memory resources are inherently limited*, with hundreds of MB of SRAM and tens of MB of TCAM available. This poses a major challenge when designing efficient data-plane solutions.

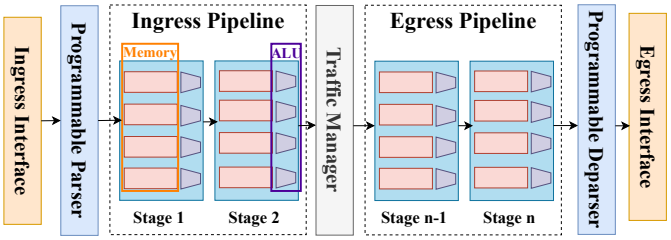


Fig. 2: Protocol independent switch architecture (PISA)

III. NETCAP SYSTEM OVERVIEW

Fig. 3 illustrates the high-level operational flow of NETCAP. Once a server authenticates a process, an eBPF program instructs the programmable switch to generate a capability and send it to the client host. On the client side, another eBPF program manages the capabilities issued to all authorized processes and appends them to their outbound traffic. The switch continuously monitors the traffic, inspects the attached capabilities, and performs in-line validation. If the capability is valid and not expired, the switch removes it and forwards the remaining data packets to the server for resource access.

Below, we provide a more detailed explanation of each step. Certain low-level details are intentionally omitted to aid understanding of the high-level flow. In Sections IV and V, we present the complete protocol and system design, detailing how the components are coordinated.

A. High-Level Operational Flow

Phase 1: Initial Setup: This phase describes how a new client joins the protection with proper information setup.

(Step 1) A new client installs the eBPF program to participate in NETCAP's protection mechanism.

(Step 2) A client process initiates authentication by sending user credentials (e.g., username and password) to

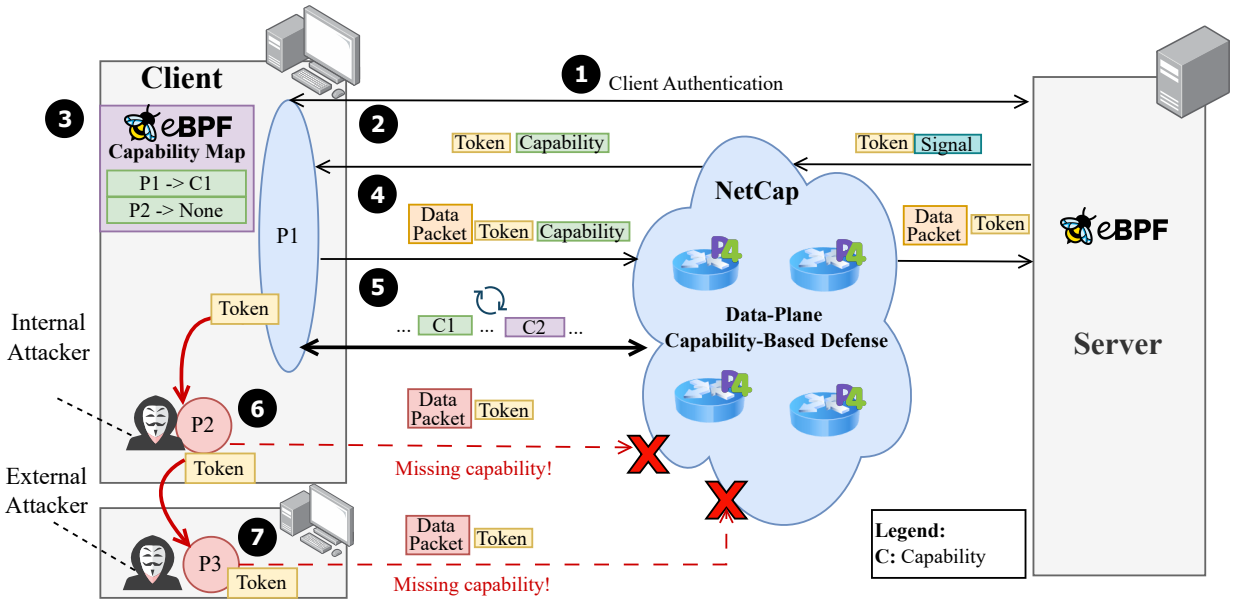


Fig. 3: NETCAP introduces a data-plane, capability-based defense leveraging programmable switches and eBPF to prevent malicious processes from using stolen tokens to gain unauthorized access. ❶ The client process first authenticates using valid credentials (e.g., username and password). ❷ If successful, the server-side eBPF program signals the switch to generate a capability for the authorized process. ❸ The client-side eBPF program binds the capability to the authorized process. ❹ The switch verifies the capability internally and forwards only the validated data packets to the server. ❺ Expired capabilities are continuously refreshed by the switch and resubmitted to the authorized process. ❻ Internal attackers cannot reuse stolen tokens without the matching process-bound capability. ❼ Similarly, if an attacker exfiltrates a token to an external machine, any traffic lacking the required capability will be dropped.

the server. At the same time, the eBPF program inserts the process ID (PID) into the first packet of the connection, signaling that the client process is requesting a new capability from NETCAP.

(Step 3) The programmable switch intercepts this packet and caches the PID and the flow's 5-tuple in a register. This information will later be used to generate an unforgeable capability. The switch then removes the PID and forwards the packet to the server.

(Step 4) Once the server authenticates the process, an eBPF program on the server detects this event. It then sends a crafted response packet back to the client, carrying a signal that indicates a successful authentication.

(Step 5) Upon receiving the signal packet, the programmable switch matches the destination network address with the previously cached 5-tuple. The switch then knows that the process associated with the cached PID has been successfully authenticated.

Phase 2: Capability Generation: This phase describes how NETCAP generates a unique, unforgeable capability for an authorized process and sends it to the client for use.

(Step 6) The switch knows that a new client from an unseen source has been authenticated and intends to join the protection. The switch then generates a unique Device ID to distinguish the client.

(Step 7) The switch uses the cached PID, Device ID, and other information (e.g., current timestamp) to generate a capability. To ensure unforgeability and strong security guarantees, we leverage Chaskey [29], [30], a lightweight and secure cryptographic function. The capability is computed following Eq. (1) with a secret key maintained in the switch's storage. This construction ensures that the capability is uniquely bound to the authorized process's PID and the target service, and is resilient against forgery.

(Step 8) The switch reuses the response packet that carried the authentication signal, replaces the signal with the generated capability and Device ID, and sends the modified packet back to the client.

(Step 9) Upon receiving the modified response packet, the eBPF program on the client extracts the capability and binds it to the authorized process by storing it alongside the process's PID in a BPF map (Fig. 5).

Phase 3: Capability Validation: This phase describes how the capability is incorporated into outbound traffic and used for continuous validation.

(Step 10) When the client process sends data packets to access resources, the eBPF program on the client constructs a custom capability packet (see Fig. 4). This packet includes the PID of the sending process, the corresponding capability retrieved from the BPF map, and the Device ID. The eBPF program transmits the capability packet alongside the data packets.

(Step 11) Upon receiving the capability packet, the switch recalculates the capability value using its stored secret key and the information in the packet (e.g., PID, service IP and port). Data packets are forwarded only if the recalculated value matches the received capability value.

In Section IV-E, we further describe how NETCAP periodically rotates keys and updates capabilities to enhance security.

B. Threat Model

Our threat model aligns with established network security research [31], [32], [33] and defenses leveraging programmable switches [34], [35], [36], [37], [38]. NETCAP is designed for controlled network environments (e.g., cloud providers, enterprise networks, data centers), where a central authority manages clients, servers, and switches. Our trusted computing base includes programmable switches, the control plane, and the eBPF-based programs. Attackers are assumed either on the victim's host or controlling a separate network node, aiming to steal access tokens through various techniques like memory scraping [1], XSS [6], and network sniffing [39]. We do not consider attackers who trick users into revealing credentials (e.g., usernames and passwords), as they can authenticate directly as legitimate users. We assume attackers do not have root access, as this gives unrestricted control, such as disabling the host program, terminating victim processes, injecting malicious code into processes, or altering process PIDs. This assumption is realistic and aligned with many prior host-level defenses [31], [32], [33]. Note that even under these assumptions, the attack surface remains substantial, with numerous ways for attackers to steal and misuse tokens, leading to many high-profile data breaches across organizations [4], [5]. NETCAP ensures that regardless of how the token was stolen, only authorized processes with valid user credentials can obtain capabilities to access the service.

We also consider *advanced attackers* who attempt to compromise NETCAP directly or exploit NETCAP to interrupt legitimate processes. In Section VI-A, we enumerate these advanced adversaries and explain how the design of NETCAP is resilient against them.

IV. IN-NETWORK CAPABILITY-BASED DEFENSE

In this section, we detail the capability design, the switch primitive, and the techniques we developed to support in-line, real-time capability generation, validation, and refresh.

A. Capability Design

The capability needs to be resilient against forgery to ensure the overall security of the protocol. Prior in-network defenses (e.g., for user anonymity [40] and DDoS mitigation [37]) primarily rely on built-in hash functions in programmable switches, such as CRC16 and CRC32. However, the CRC hash functions are non-cryptographic and insecure. Their short hash lengths make them vulnerable to chosen-plaintext attacks [41], allowing attackers to brute-force a captured CRC value and recover the key. Additionally, CRCs possess algebraic properties (e.g., $CRC(A) \oplus CRC(B) = CRC(A \oplus B)$) that can be exploited to expose the key without brute-forcing [42]. Such threats significantly undermine the security of the defenses.

To overcome these limitations, we adopt Chaskey [30], a secure and lightweight cryptographic function, to generate unforgeable capabilities. Chaskey uses a 128-bit secret key to process messages in 128-bit blocks and produces a message authentication code (MAC). It offers strong provable security against forgery and differential attacks, with formal proofs establishing its security bounds in the standard model [29].

Chaskey's 128-bit output ensures that capability collisions are cryptographically negligible. According to the birthday bound [43], even with 1 million active capabilities per switch, the collision probability remains extremely low, on the order of 10^{-27}. In contrast, CRC32 reaches $\sim$50% collision probability after only $\sim$65K capabilities, underscoring the need for a secure hash function like Chaskey. Despite its stronger security guarantees, Chaskey adds only $\sim$400 nanoseconds to the computation time compared to CRC in programmable switches, which is negligible given that typical RTTs are in the milliseconds. Chaskey was specifically developed for microcontrollers and resource-limited systems, making it well-suited for data-plane deployment on programmable switches.

Formally, a capability, denoted as `C`, is computed as follows:

$$C = \mathrm{Chaskey_K}(\mathrm{Device\_ID} \parallel \mathrm{IP_{svc}} \parallel \mathrm{Port_{svc}} \parallel \mathrm{Protocol} \parallel \mathrm{PID} \parallel \mathrm{TS} \parallel \mathrm{Nonce}) \quad (1)$$

We explain each component of the equation below:

- Device_ID: Source IP addresses are inherently unreliable and can be spoofed from user space. To address this, when the switch receives a capability request for the first time from the client, it assigns a unique Device ID to that client. Including the Device ID in the capability computation ensures that capabilities are bound to specific clients. That is, a different client with a different Device ID cannot replay the capability issued to a legitimate client and pass validation.
- $\mathrm{IP_{svc}}$, $\mathrm{Port_{svc}}$, Protocol: These fields form the 3-tuple identifier of the service-bound flow, where svc denotes the remote service. This ensures the capability is tied to a specific service endpoint. The client-side port ($\mathrm{Port_{client}}$) is omitted as it may vary across flows.
- PID: The process ID of the authorized process. Including it ensures that the capability `C` is bound to the specific authorized process, hence removing ambient trust.

Fig. 4: NETCAP's capability packet format with standard Layer 2-4 headers

- TS: The timestamp at which `C` is generated. This field is used to compute the capability's validity period and determine whether it has expired.
- Nonce: A random value generated and stored by the switch. It introduces entropy into the input, making it more difficult for attackers to reverse-engineer the capability.
- K: A 128-bit secret key used to securely compute `C`. The key is generated by the switch, stored in a match/action table, and is *never* shared with any host.

B. Capability Packet

One way for clients to transmit their capabilities and for the switch to return newly generated ones is to embed capabilities directly into data packets using a custom header. However, this design introduces significant performance overhead in two critical paths: (1) the switch data plane, which must compute capabilities using a cryptographic algorithm, and (2) the client's network stack, where modifying packet headers is CPU-intensive. As shown in our evaluations in Section VII-D, this design leads to significant throughput degradation.

To avoid impacting the performance of data packet processing, we adopt a decoupled design in which capability information is transmitted via separate control packets, referred to as *capability packets*, by both clients and the switch.

Fig. 4 shows our capability packet format. The switch distinguishes capability packets from other packets using a reserved bit set in the IP fragment field [44]. The Capability field stores the capability value of the current connection. The Device ID and PID verify the sender's identity. The Timestamp field indicates the time the capability was generated.

The Type field is used to distinguish between different types of capability packets in NETCAP:

- **(Type=0)** packets are sent by the switch to deliver a newly generated capability to the client (Step 8 in Section III-A).
- **(Type=1)** packets are sent from the client to the switch and carry the process's capability for validation to gain service access (Step 10 in Section III-A).
- **(Type=2)** packets act as ACKs from the client to the switch to ensure reliable delivery of newly generated capability. The switch retransmits the capability if no ACK is received.

Lastly, the Key Version field helps identify which secret key was used to generate the capability during key rotation. This allows the switch to validate a capability with the correct key version (more details in Section IV-E).

C. Capability Generation

When a client initiates a new connection, the client-side eBPF program inserts the sending process's PID into the checksum field of the first packet and sets the reserved IP fragment bit to 1. Upon receiving this packet, the switch extracts the PID and stores it in a hardware register using the flow's 5-tuple ($\mathrm{IP_{src}}$, $\mathrm{Port_{src}}$, $\mathrm{IP_{dst}}$, $\mathrm{Port_{dst}}$, Protocol) as the key and the PID as the value. The switch then recalculates the checksum and clears the reserved bit before forwarding the packet to its destination.

Once the client process completes successful authentication with the service, the switch generates a capability for the authorized process. A naive approach to detecting this event is to inspect the payload of packets sent from the server. However, many applications encrypt their traffic [45], preventing the switch from verifying authentication via deep packet inspection. Another approach is to analyze network traffic patterns between the client and server to infer successful authentication [46]. However, traffic volume is an unreliable indicator, as different applications exhibit varying patterns before and after authentication. For instance, FTP often shows a sharp increase in traffic post-authentication, whereas SSH may exhibit little change.

In contrast, we design a more reliable mechanism by developing an eBPF program on the server to detect successful authentication events. The detailed operations of this program are described in Section V-B.

Once the client process is authenticated, the server-side eBPF program signals the switch by encoding a flag in the reserved bit of the TCP header [47] of a client-bound data packet (Step 4 in Section III-A). Upon receiving this signal, the switch looks up the flow's 5-tuple in its hardware register to retrieve the associated PID. If the client is not seen before, NETCAP generates a unique Device ID to distinguish the client and stores this ID in a local hardware register. If the same client later requests another capability, NETCAP uses the client's IP address to retrieve the corresponding Device ID from the register.

With both the PID and Device ID available, the switch computes a new capability using Chaskey, following Eq. (1). To transmit the capability back to the client, the switch must work around the limitation that data planes cannot create new packets. To overcome this, the switch clones an existing data packet, repurposes it as a capability packet, and reprocesses it through the pipeline. The switch then embeds the computed capability into the capability packet, sets the appropriate header fields (i.e., **Type=0**, as described in Section IV-B), and forwards the capability packet to the client.

Once the client receives the capability, it replies with an ACK packet (**Type=2**) to confirm receipt. If the server does not receive this ACK, it retransmits the authentication signal to ensure reliable delivery. The connection is dropped if the switch does not receive a signal from the service within a configurable timeout period, which can be set through our APIs (i.e., `SetSignalTimeout` in Table II).

D. Capability Validation

To access the service, the client process must send a capability packet containing its current capability `C` for validation

by the switch. To verify C, the switch performs two checks: (1) C is valid, and (2) C has not expired. For the first check, the switch recomputes it following Eq. (1). If the received C matches the recomputed capability, its validity is confirmed. Otherwise, the connection is dropped. For the second check, the switch determines C's current lifetime by calculating the difference between the current timestamp and C's creation time carried in the capability packet. If the received C passes both checks, the process's data traffic is forwarded.

If C passes the first validity check but its lifetime exceeds the permitted duration, the switch generates a refreshed capability C' with a new timestamp and Nonce. The switch then transmits C' back to the process using a dedicated capability packet (**Type=0**) and requires an ACK from the client to confirm receipt (**Type=2**). This ACK-based mechanism not only ensures reliable delivery of capabilities but also prevents communication interruption during network failures. We describe failure handling in more detail in Section VI-B.

Cached capability decisions. For a client to keep accessing a service, NETCAP needs to ensure that each data packet is authorized to reach its destination. A naive approach would transmit a capability packet alongside every data packet. However, this imposes high processing overhead on clients due to high packet generation and saturates the network with a large volume of capability packets.

To improve efficiency, NETCAP employs an *in-network capability decision cache* using a customized software-defined switch register, called CapDec, which is implemented as a register array in the switch's hardware SRAM. CapDec maps H(Flow's 5-tuple ID, Device ID, PID) to a cached decision. Instead of sending a capability packet along with every data packet, our client-side eBPF program transmits a capability packet once every t seconds (see details in Section V-A), and the switch caches the resulting decision. Data packets from client processes do not carry capabilities. Instead, they include only the Device ID and PID, embedded in the IP checksum and TCP urgent pointer fields, respectively. A reserved bit in the IP version field is set to indicate that this data packet carries process-level metadata. When data packets arrive at the switch, the switch extracts the PID and Device ID and matches them against CapDec. If an entry is found with an "allow" decision, the switch recomputes the packet checksum before forwarding it. This ensures the correct delivery of the packets.

The interval t is configurable through our API (i.e., SetHostInterval in Table II). Our evaluations in Section VII-D show that sending one capability packet per second achieves a good balance between security and efficiency. This configuration enables timely capability validation while significantly reducing overhead. Overall, this approach maintains line-rate throughput with negligible network overhead.

E. Capability Refresh

Unlike existing access tokens that often have long lifetimes (see Table I), NETCAP adds an additional layer of security by issuing each capability with a limited lifetime (e.g., within seconds). When the switch detects that a received capability has expired (by comparing the current timestamp with the Timestamp field in the capability packet), it refreshes the capability using an updated timestamp and a new secret key through key rotation.

However, refreshing capabilities introduces a timing challenge. When the switch detects that a received capability has expired, it must generate and deliver a new one to the process. During this window (i.e., between detecting expiration and the client receiving the updated capability), the client might continue to send the expired capability packet to keep accessing the service. If each of these capability packets triggered a new refresh, the switch could generate multiple valid capabilities simultaneously for the same process and destination. This results in inconsistencies.

A naive mitigation strategy is to stall the client's traffic until the refreshed capability reaches the process. However, this causes delays and degrades application performance. To avoid this, NETCAP employs a *lightweight refresh coordination mechanism* using a temporary stateful cache in the switch. When a capability is detected as expired, the switch immediately generates a new one and records a temporary entry containing the expired capability, the client's Device ID, and the PID. This entry serves as a short-term marker to prevent redundant refreshes. If additional capability packets (not data packets) arrive from the same connection during the refresh process, they are dropped without triggering another refresh. Meanwhile, data packets from the same connection are allowed to continue flowing. Once the client receives the updated capability, it sends an *ACK packet* (**Type=2**) to confirm receipt. The switch then removes the temporary entry and resumes regular validation using the refreshed capability.

Key rotation. Secret keys stored in the switch are crucial for securing capabilities and preventing forgery. Although Chaskey is designed to resist forgery and differential attacks [29], implementing a key rotation mechanism provides an added layer of protection against practical risks such as accidental key leakage, control plane misconfigurations, or administrative mishandling.

In NETCAP, the switch rotates both the secret key and nonce at a configurable interval, adjustable via our APIs (Section VI-B). Even if an attacker leaks the current key, it becomes invalid quickly, significantly reducing the attack window. However, directly replacing an old key with a new key can cause inconsistencies. For example, a secret key $K1$ is updated to $K2$ after a capability is sent to an authorized process. When the process sends back the capability, NETCAP will not be able to validate it with the current key. To prevent this, NETCAP temporarily retains the old key for an additional 5 seconds to validate expired capabilities and generate refreshed ones. This period accounts for typical wide-area network (WAN) delays and packet retransmissions under moderate congestion, ensuring active connections can refresh their capabilities before the old key is discarded [48].

V. Lightweight eBPF Programs

We design lightweight eBPF programs for the client and server to manage process capabilities and send authentication signals, respectively. eBPF [49] provides a secure sandbox for running user-defined programs within the kernel. These programs attach to kernel hooks (e.g., network events and system calls), enabling custom functionality *without modifying the kernel or loading new modules*. This allows seamless deployment and broad applicability of our defense system.

A. Client-Side Program

Fig. 5 illustrates the architecture of our client-side eBPF program, which operates at ingress and egress hooks to support capability extraction, trace intra-host process activities, and facilitate packet transmission.

Kernel hook points. To achieve high-performance packet processing, we attach a custom eBPF program to the XDP (eXpress Data Path) hook [50], enabling our logic to run directly in the network stack. When a capability packet arrives with the reserved IP fragment bit set, our XDP program extracts the capability packet fields into a structure we call `Cap`, which contains three fields: `Capability`, `Timestamp`, and `Key_Version`.

To accurately associate each capability with its process, we investigated several binding designs. One design is to maintain a dynamic process tree in BPF maps of all active processes and use their ancestry information to derive a unique process identifier by hashing the parent PIDs. However, continuously updating and synchronizing this tree across thousands of concurrent processes introduced significant lookup and synchronization overhead in the kernel, especially as processes frequently spawn and terminate.

Instead, NetCap uses a lightweight and robust PID-binding mechanism that tracks process system calls through kernel tracepoints. When a capability packet is received, NetCap extracts the PID of the process receiving the network connection. This is done by tracing the system call used to accept the connection as the packet reaches the transport layer of the kernel network stack. NetCap then binds the extracted capability to the receiving process by storing it in a BPF map, `CapMap(PID,IP`$_{svc}$`,Port`$_{svc}$`->Cap)`.

To ensure the binding is safely removed when a process terminates, we attach an eBPF program to the exit path of the kernel process termination tracepoint `sched_process_exit`. This program triggers upon process termination and immediately deletes the corresponding entry from `CapMap` before the kernel allocator reuses the PID. This design maintains accurate PID bindings, ensuring that the corresponding binding is removed before the PID is reassigned to a new process.

To identify the sending process of each packet, we trace system calls that initiate network connections and extract the process's PID and destination address (IP and port). These values serve as keys for lookup in `CapMap`. If a match exists, the associated `Cap` is copied into a second map: `OutCap(sport->[Cap, last_TS])`. To transmit capability packets, we attach an egress eBPF program at the TC hook. It checks whether the outgoing packet's source port is marked in `OutCap`. If so, it constructs and transmits the capability packet.

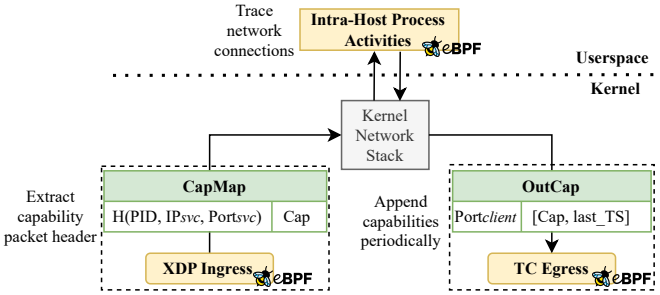


Fig. 5: Architecture of the client-side eBPF program

Appending capability information directly to every data packet is CPU-intensive. As shown in Section VII-D, this can cause a 30% throughput drop. To avoid this, the program sends capabilities in separate capability packets, as mentioned in Section IV-B. Since eBPF cannot generate new packets due to the lack of dynamic memory allocation, we use the BPF helper function `bpf_clone_redirect`. This allows the egress program to *clone* an outgoing data packet, repurpose it as a capability packet, and send it to the switch. Note that cloned packets are not part of the main application data stream. Since these packets are sent independently, they do not affect the application's sending rate or degrade its performance.

In-kernel capability management. To store capabilities associated with processes, one way is to use a BPF map keyed by PID. However, this method fails when a process communicates with multiple destinations, each requiring a distinct capability. Nested maps could store multiple capabilities per PID. However, eBPF limits map sizes to fixed allocations, which may be insufficient for processes requiring more capabilities or result in wasted space for processes with few entries.

To address this challenge, we design a specialized mapping structure that supports a *variable* number of capabilities per process. Specifically, we update our `CapMap` to use a hash key: `CapMap(H(PID, IP`$_{svc}$`, Port`$_{svc}$`)->Cap)`. This hash-keyed map distinguishes capabilities by both the process and destination, supporting efficient lookup.

Timed capability transmission. As discussed in Section IV-D, sending capabilities with every data packet introduces noticeable traffic overhead. To mitigate this, our client-side program sends one capability packet every t seconds. It tracks the last transmission time (`last_TS`) in the `OutCap` map and sends a new capability when the elapsed time exceeds t. The decision is then cached in the switch's `CapDec` register.

B. Server-Side Program

The server needs to signal to the switch whether a process has been successfully authenticated, so that the switch will generate the corresponding capability. A strawman solution is to modify server applications to explicitly coordinate with

NETCAP. However, this approach requires *extensive changes* to existing applications or protocols, significantly reducing the compatibility and scalability of the defense. An alternative is to inspect outgoing data packets from the server using an eBPF program hooked at the egress point. However, this is also impractical because application-layer data is often encrypted (e.g., HTTPS, TLS), preventing eBPF programs from accessing or interpreting the actual payload.

To facilitate broad applicability, it is preferable to design a protocol-independent solution that does not require modifying applications or their underlying authentication protocols. To achieve this, our current implementation detects successful authentications by monitoring application logs. Such logs contain the whole network 5-tuple ID and login status, making it ideal to infer when a client process successfully authenticates. This approach avoids intrusive instrumentation of the running services. Application logs have been extensively used in prior research efforts [51] and enterprise-grade security tools [52] to trace internal application behavior.

Specifically, we design a lightweight server-side program that monitors application logs and signals the switch to generate capabilities. It combines a user-space component with an eBPF program for efficient operation. The user-space component monitors application logs using a lightweight parser, and matches log entries against patterns for successful authentications using regular expressions. When a match is found, the app extracts the client's network flow ID. An eBPF program on the TC egress path then receives the flow ID and marks matching client-bound packets with the successful signal. A key advantage of this approach is its scalability. Adding new logs to monitor or updating the patterns for detecting successful authentication entries is straightforward. Administrators can easily update the user-space component with the new log file path and revised regular expressions, without any changes to the running service code.

Application logs may be targeted by attackers seeking to cover their traces. Many existing defense mechanisms, such as intrusion detection based on host audit logs [51], [33] or lateral movement detection using application logs [53], implicitly assume that logs are part of the trusted computing base and remain untampered. Ensuring tamper-evident logging to support different defense applications is a parallel and active area of research. Recent efforts have proposed techniques such as encrypted access logging [54] to harden the logs. NETCAP can benefit from these solutions.

VI. HANDLING PRACTICAL REQUIREMENTS

A. Resilience Against Advanced Attackers

We provide a security analysis of how NETCAP counters various *advanced attackers* who are aware of our defense design and attempt to bypass it.

Replay capability packets. An attacker may attempt to replay captured packets containing valid capabilities, substituting a malicious payload directed at the destination server. However, our client-side program adds the sending process's PID to all connections in the kernel space, which is beyond the attacker's control. Since the PID is included in the capability computation (Eq. (1)), the switch can detect when the incoming packet's PID does not generate a capability that matches the replayed one, and consequently drops the connection. Additionally, a malicious user may try to misuse their own legitimate capability to access an unauthorized service. However, since NETCAP incorporates the service's network address into the capability computation, the switch can easily detect any mismatch in the intended service address.

IP and PID spoofing. IP addresses are unreliable [55] and can be spoofed from user space. Advanced man-in-the-middle (MiTM) attackers may replay captured capability packets from another client with a *spoofed victim's IP*. To prevent this, NETCAP assigns a unique Device ID to each client when it joins the network. Consequently, our client-side program attaches the received Device ID to outgoing capability and data packets. Because the Device ID is included in the capability computation (Eq. (1)), spoofing the IP is ineffective. The switch detects that the Device ID in the packet does not match the one used to generate the capability, and drops the connection. Since NETCAP stores the Device ID in protected kernel memory and appends it via the client program in the kernel space, attackers cannot tamper with or spoof the Device ID. Any attempt to reuse a stolen Device ID from another host results in a capability mismatch and is dropped by the switch.

An attacker on the same victim's host may attempt *PID spoofing*. However, this approach fails as the OS enforces unique PIDs for active processes. If the attacker instead uses the same PID on a different host, the attack is still detected. As described above, since the Device ID is included in the capability computation (Eq. (1)), the switch detects a mismatch between the attacker's Device ID and the original Device ID used to generate the capability, dropping the connection.

Denial-of-service. NETCAP remains resilient even under Tbps-scale attacks by processing all packets *entirely in the data plane* at line rate (as shown in our evaluations in Section VII-D). However, attackers can launch a denial-of-capability attack by flooding the switch with many new capability requests, overwhelming the switch register used for caching capability requests. This could evict legitimate process requests from the cache register, denying them the ability to receive capabilities. To counter this, NETCAP employs a rate-limiting strategy that caps the number of requests from a specific PID on a client in a given time frame. This allows NETCAP to restrict malicious processes without disrupting benign processes. Also, NETCAP periodically evicts inactive entries to free up switch memory.

Secret key compromise. An attacker may attempt to brute force all possible secret keys to derive the one used by the switch to generate capabilities. However, NETCAP uses the secure Chaskey algorithm [29]. Its high computational complexity, coupled with the 128-bit key, makes brute-forcing computationally infeasible. Keys may also leak due to admin

TABLE II: NETCAP's configuration APIs

Configuration API	Description
AddService(IP, port)	Add a new service to NETCAP
AddPort(switch_id, port)	Enable NETCAP on switch port
SetLifetime(switch_id, lifetime)	Set key lifetime for a specific switch
SetHostInterval(IP, interval)	Adjust host's capability packet sending interval
SetSignalTimeout(duration)	Set the timeout for receiving signals from services

errors or control plane misconfigurations. As discussed in Section IV-E, NETCAP incorporates an additional protection layer by updating keys frequently (e.g., every 1-5 seconds), further reducing the potential damage.

B. Operational Deployment

Programmable switches provide in-network programmability at line rate while matching the cost and performance of legacy switches ($12K for a Tofino 6.5Tbps switch [37]). They are increasingly adopted by major enterprises (Meta [56], Google [57], etc.), ISPs (AT&T [58], SK Telecom [59], etc.), and switch manufacturers (Cisco [60], Juniper [61], etc.). NETCAP builds on this momentum to protect services in cloud and enterprise networks. A natural deployment is to run NETCAP on Top-of-Rack (ToR) or edge switches. This applies to cloud data centers and enterprise networks, as both environments share similar architectures (e.g., racks of servers connected to ToR switches and client nodes connected to edge switches to reach internal services). On hosts, NETCAP uses eBPF, which is widely adopted in cloud-native environments and data centers for network management and security monitoring [49]. Our eBPF-based programs are lightweight, require no changes to the OS or application code, and are loaded once at system startup. They impose minimal overhead (see Section VII-D), making them practical for real-world deployment.

Incremental deployment. NETCAP supports partial network deployment without requiring changes to the entire network. The defense remains effective as long as at least one NETCAP-enabled switch exists between the client and server. Such a design enables the gradual adoption of NETCAP in non-P4 environments. NETCAP also allows environments where clients with and without our client-side program can coexist. This is configurable via our API, `AddPort`, to run NETCAP on selected switch ports, while forwarding all other traffic. Clients without our eBPF programs can still connect, but they will not benefit from NETCAP's protection.

Failure handling. We designed robust mechanisms in NETCAP to handle common network and system failures. To address packet loss, NETCAP uses ACKs between switches and hosts to confirm delivery of control packets, such as capability packets and authentication signals (refer to Section IV for more details). The sender retransmits if an ACK is not received. For communication between the controller and the switches, NETCAP uses UDP with a timeout-based mechanism to exchange secret keys. While UDP is lightweight and efficient, it does not guarantee delivery, so the controller resends secret key updates if ACKs are not received within a predefined window. In the event of a switch failure or reboot, any cached capability decisions stored in the switch's local registers are lost. Clients may experience brief disruptions during this recovery window, as data packets relying on missing capabilities could be dropped. However, because clients periodically resend their capabilities (every 1-2 seconds), these cached entries are quickly reconstructed with minimal delay, and the switch can refresh these capabilities if needed.

Configuration APIs. NETCAP provides a set of APIs (shown in Table II) through the switch control plane. It allows administrators to easily customize capability requirements to meet their networks' needs with *little effort and no low-level data plane programming*. They can specify which services require capabilities by defining IP addresses and ports, set secret key lifetimes based on client risk levels (e.g., shorter lifetimes for high-risk clients), and configure default lifetimes for services without specific settings. Administrators can also control how frequently client programs send capabilities and define timeouts for how long switches wait for service authentication signals before dropping connections. Once defined, the control plane automatically translates API calls into the corresponding data plane configurations (e.g., match/action table entries and associated actions) to orchestrate the defense.

VII. EVALUATION

We implemented NETCAP using ∼2500 lines of code. The switch program is developed in $P4_{16}$, and the controller is written in Python. The client and server programs are implemented in eBPF using BCC 0.23.0. To monitor application logs, we developed the user-space application using ∼150 lines of code in Python without modifying the service's code.

We aim to answer the following key research questions on the defense effectiveness, scalability, real-world application support, and NETCAP's overhead, through extensive evaluations on our physical testbed (see Section VII-A) and using real-world enterprise workload datasets.

(RQ1) How effective is NETCAP's capability-based scheme in protecting various protocols and services?

(RQ2) How efficiently does NETCAP process capability requests within the network?

(RQ3) What impact does NETCAP have on network and host performance?

(RQ4) How well does NETCAP support complex real-world server applications?

(RQ5) How does NETCAP's switch-based design compare with an alternative eBPF-only design?

A. Testbed Setup & Attack Implementations

Testbed. Our testbed is consistent with prior programmable network works [36], [37], [35], [34]. It mirrors real-world deployments, where a P4-enabled Top-of-Rack (ToR) switch runs NETCAP to protect machines within a rack. The testbed

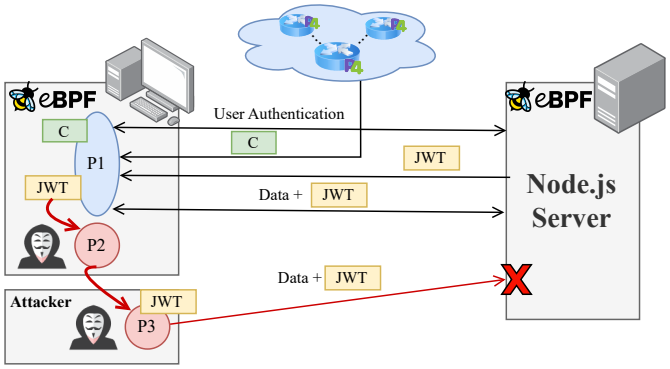


Fig. 6: Preventing JWT theft and usage from an attacker's machine

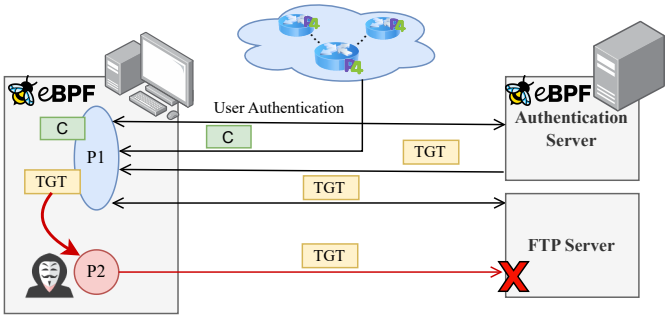


Fig. 7: Denying the usage of a valid TGT to access the target server

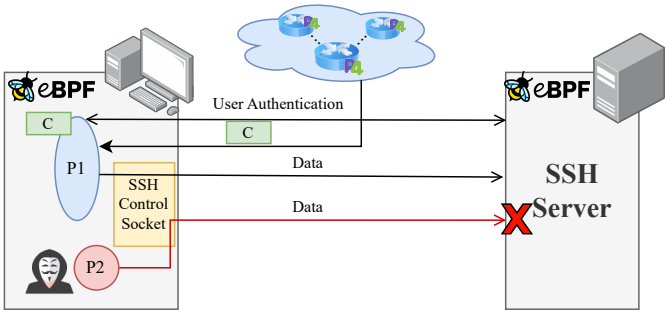


Fig. 8: Preventing SSH session hijacking

consists of three servers connected to a physical Tofino P4 switch with 32×100 Gbps ports. The servers are equipped with Intel Xeon E5-2430 CPUs (2.20 GHz) and 64 GB RAM, running Ubuntu 20.04 with kernel version 5.15.0. One server acts as the client, attempting to authenticate and access various services hosted on the second server. The third server is set up to act as an external attacker, aiming to gain unauthorized access to the running services. We deploy three real-world services that require authentication and generate tokens for authorized clients. In addition, we evaluate NETCAP using two real enterprise workloads: the LANL Unified Host and Network dataset [21] and the DARPA OpTC dataset [20]. The LANL dataset contains activities from 17K hosts, while the DARPA OpTC dataset contains activities across 1K hosts.

We present three representative attacks targeting widely used protocols: (1) OAuth 2.0 session hijacking, (2) Kerberos pass-the-ticket attacks, and (3) SSH lateral movement. These scenarios reflect real-world threats across cloud platforms, enterprise identity systems, and remote administration settings. However, NETCAP's protocol-agnostic design enables broader applicability beyond these specific examples.

Attack 1: Web application session hijacking. OAuth 2.0 is a widely used authorization protocol in web services [62]. Upon client authentication, the server generates an access token, often in the form of a JWT. We set up a Node.js server running an OAuth 2.0 service that generates and validates JWTs. On the client machine, we use a terminal process to initiate a curl command that sends valid credentials to the web server and receives a JWT. This JWT is stored in the client's file system. We then initiate a wget process to retrieve a confidential file from the web server using the JWT. Concurrently, we set up an external attacker that exploits a vsftpd service running in the client's system with a malicious backdoor [63]. Using Metasploit [64], we gain access and exfiltrate the client's JWT to the attacker via scp. Using the stolen JWT, we retrieve the confidential file without requiring valid credentials. Fig. 6 illustrates the JWT theft attack.

Attack 2: Kerberos pass-the-ticket attack. Kerberos is a widely used authentication service, particularly in identity management systems (e.g., Windows Active Directory) [65]. It issues a Ticket Granting Ticket (TGT), which is a token that authenticates clients to request access to specific services. We use a client machine to authenticate with the Kerberos server and receive a TGT for FTP server access. Using a malicious user account on the client system, we exploit a vulnerability in the vsftpd service [63] running on the system to gain access to the victim's account. This allows the attacker to access the legitimate client's memory and harvest the TGT from the Kerberos ticket cache. We then utilize Impacket [66] with the stolen TGT to gain the needed permissions to access the FTP server. Fig. 7 illustrates the TGT theft attack.

Attack 3: Lateral movement via SSH multiplexing. SSH is a critical tool for secure remote connections and system administration [9]. It supports SSH multiplexing, which allows a client to create multiple SSH sessions over a single network connection using a control socket. This socket is created after the client successfully authenticates with their SSH keys. Consider the scenario (illustrated in Fig. 8) where we deploy a malicious attacker with access to the client's machine. We run an SSH client that uses its private SSH keys to authenticate to an SSH server. We set up the attacker to exploit SSH multiplexing by reconfiguring the SSH configuration file to enable this feature. Consequently, when the legitimate SSH process establishes a connection and authenticates to the server via SSH keys, a control socket is opened. This allows the attacker to create their own SSH processes and establish sessions to the server without authentication.

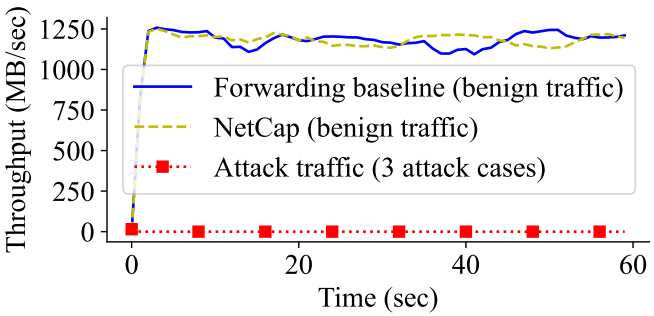
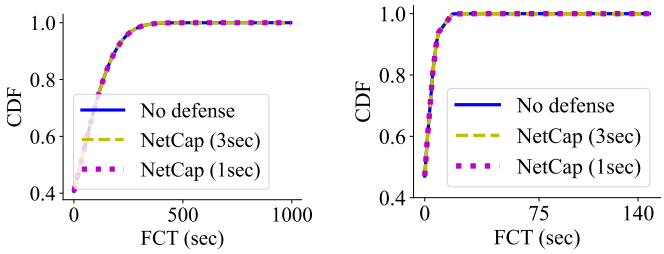


Fig. 9: NETCAP blocks all session hijacking attempts while imposing negligible overhead on benign traffic



(a) FCT in LANL **(b)** FCT in DARPA OpTC

Fig. 10: NETCAP imposes negligible overhead under real-world enterprise workloads

B. RQ1: Defense Effectiveness

We evaluate the effectiveness of NETCAP in countering the three attacks we constructed (Section VII-A). To quantify the network performance, we measure the *throughput* using iperf3. We compare the throughput achieved under NETCAP to the forwarding baseline, which just forwards packets without our defense scheme. We manually assign a valid capability to the iperf3 process on the client machine. We set the lifetime of capabilities to 10 seconds and configure the client-based program to send a capability packet every 1 second.

When the attacker steals tokens (Attack 1, 2) or hijacks SSH sessions (Attack 3), they initiate traffic via malicious processes. Since these processes lack the necessary capabilities, NETCAP drops their traffic. As shown in Fig. 9, NETCAP successfully *blocks all* unauthorized traffic across three scenarios, resulting in 0 throughput for malicious attempts. Meanwhile, we can observe that the throughput of benign traffic is similar to that of the forwarding baseline. The results demonstrate that NETCAP effectively blocks malicious connections with negligible network overhead.

C. RQ2: Scalability with Real-World Workloads

We evaluate NETCAP's *scalability* using the LANL Unified Host and Network dataset [21] and the DARPA OpTC dataset [20]. Both datasets contain real-world traces from enterprise networks. We replay the traces of both datasets to the physical switch using the Distributed Internet Traffic Generator (D-ITG) [67]. We modify the server-side eBPF program to send a signal to the switch for every new network flow. This allows NETCAP to generate and validate capabilities for all network flows. We configure the capability lifetime at 10 seconds, with the client-side program sending a capability packet every 3 seconds in one experiment run and 1 second in a different run. To quantify the latency added by NETCAP to network flows, we measure the flow completion time (FCT) of both experiment runs under the forwarding baseline and NETCAP.

Fig. 10 illustrates the cumulative distribution function (CDF) of the FCT across both datasets while NETCAP is running. As NETCAP validates and refreshes capabilities inside the hardware, we observe no significant changes in FCT across both datasets, whether the client-side program sends a capability packet every 3 seconds or 1 second. This confirms that NETCAP scales with real-world enterprise traces while performing an effective fine-grained capability validation.

Maximum number of active connections. We leverage a feature of the P4 compiler to measure NETCAP's capacity for handling active connections. The P4 compiler ensures that compiled programs fit within the switch's hardware pipeline and are guaranteed to run at line rate. Programs that exceed available hardware resources are rejected at compilation time. Using this feature, we gradually increase the maximum number of active connections in the switch's match/action table until the P4 compiler rejects the program and record the largest number supported. We find that up to 200K concurrent connections can be supported at the same time. This is more than the number of active connections found in Facebook frontend clusters, which range between 10K-100K [68].

D. RQ3: System Overhead

Capability traffic overhead. We analyze the traffic overhead of generating capability packets at varying intervals. We set up a client-side eBPF program that sends a 64-byte capability packet at different intervals. To simulate real-world workloads, we generate 100K connections through the switch, which is the typical number of active connections found in enterprise clusters [68]. Fig. 11 shows total capability traffic over 60 seconds. Sending a capability packet every 1 second adds 366 MB of traffic overhead, but ensures frequent capability refreshes. Conversely, sending capability packets less frequently (every 10-15 seconds) significantly reduces traffic overhead by 93%, while still refreshing capabilities within seconds. These results indicate that the interval at which capability packets are sent significantly impacts the traffic volume.

Separate vs. appended capability packet headers. We evaluate the performance of NETCAP when appending capabilities to data packets (NETCAP-appended) versus sending them separately (NETCAP-separate). We measure the network throughput under the two configurations using the iperf3 tool. As shown in Fig. 12, embedding capabilities directly into data packets (NETCAP-appended) results in a *noticeable drop* in throughput. This drop is attributed to the added latency incurred by (1) the eBPF program inserting the capability

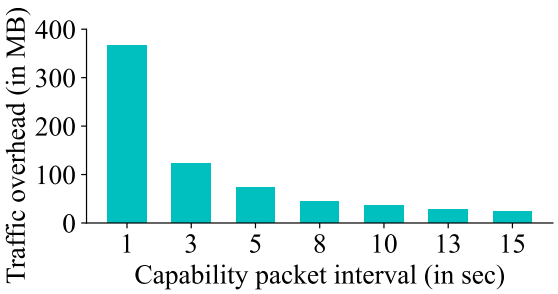


Fig. 11: Traffic overhead from generating capability packets at different intervals

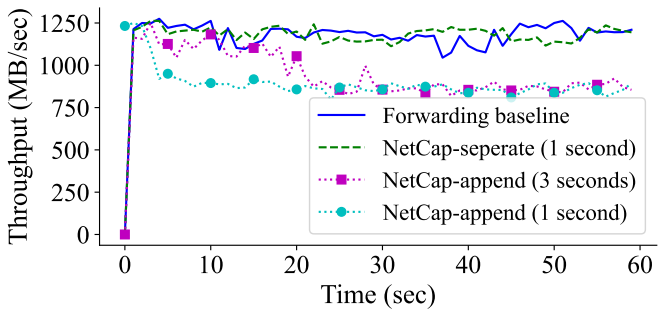


Fig. 12: Network performance of separate capability packet stream compared to directly modifying data packets

header at the TC hook and (2) the switch cloning packets to construct capability packets. In contrast, transmitting capabilities in a separate stream of capability packets (NETCAP-separate) *maintains* throughput levels similar to the forwarding baseline, as capability packets are processed independently from the data packets. These observations are consistent with our testbed experiments in Section VII-B.

eBPF-based programs overhead. We evaluate the client-side program's overhead by measuring the average latency of the eBPF programs over 10K runs. The client-side program incurs an additional 7-8 microseconds to craft capability packets in the egress TC hook. Since these packets are not part of the data stream, they do not impact the throughput of the active connections. For the ingress eBPF program, the client-side program adds a negligible 2-3 microseconds to parse capability packets. For the server-side program, our TC egress program incurs a one-time 7-8 microseconds overhead to append the signal to a single packet in a connection, which is negligible.

We also measure the storage overhead of maintaining the `CapMap` in kernel memory. With 4GB of recommended kernel memory for eBPF programs, a 32-bit hash key, and a 20-byte capability struct value, we can store up to ∼167M capabilities inside the kernel memory. This number of capabilities is sufficient considering the typical number of processes in a Linux system of 65,536 (each process can have up to 2.5K capabilities at the same time).

Switch resource utilization. We retrieve the switch resource utilization for NETCAP via the Intel P4 Insight tool. As reported, NETCAP consumes only a small amount of the shared storage resources (8% of SRAM and 0.1% of TCAM). As NETCAP offloads capabilities to client machines, it reduces its impact on the memory resources inside the switch. Additionally, NETCAP consumes 80% of the HashUnit capacity to generate and validate capabilities. Current HashUnit utilization remains low, allowing NETCAP to run alongside more feature-rich P4 programs (e.g., tna_simple_switch.p4), as HashUnits are generally used less by other network programs.

Throughput and latency. We evaluate NETCAP's impact on network throughput and latency by comparing its capability operations to a basic P4 program that only forwards packets. The comparison results are shown in Fig. 14. After successful compilation, the pipelined nature of the switch hardware ensures that NETCAP's P4 program operates at a rate of 99.9Gbps per port, matching the forwarding baseline. As for the latency, NETCAP introduces an additional 110-130 nanoseconds to match data packets with the capability decision. This increase is negligible considering that the RTT in typical enterprise networks spans several milliseconds.

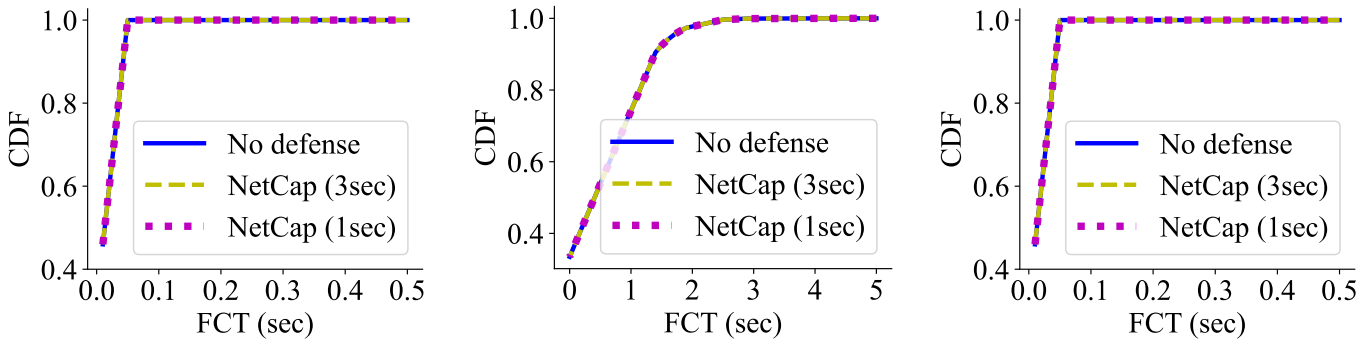
E. RQ4: Support for Complex Applications

To evaluate NETCAP's transparency, we assess how our capability-based scheme supports different complex server applications. We compare the network performance of data transfers under NETCAP and the forwarding baseline across Apache, Node.js, and FTP servers. Each server is configured with JWT-based authentication to generate tokens. We generate 1K client requests with capability lifetimes of 3 seconds and 1 second. Fig. 13 shows the CDFs of the FCT for the applications under both lifetimes. NETCAP *maintains line rate performance* for all applications while processing all requests.

F. RQ5: Comparison with eBPF-Only Design

We compare NETCAP's switch-based design against an eBPF-only design (i.e., without the switch data plane) that we implement, which represents NETCAP's logic running entirely on the server. Specifically, we move NETCAP's switch-side capability generation and validation logic into an ingress eBPF program. This program validates capability packets, refreshes expired ones, and stores security decisions in BPF maps, which are later matched against incoming data packets. This comparison allows us to evaluate (1) the responsiveness of capability validation, and (2) the resilience to saturation attacks under heavy capability refresh loads.

Capability validation responsiveness. The eBPF-only design faces serious limitations in both performance and security. We measure the time to validate a capability and install a security decision. The eBPF-only design suffers from 12 milliseconds *delays* to process a single capability packet. These delays stem primarily from frequent kernel-user space context switches and the overhead of accessing BPF maps. Executing secure cryptographic MACs (i.e., Chaskey) on the server CPU further increases this overhead, causing bottlenecks in packet processing and reducing network throughput by up to 40%.



(a) Apache server FCT **(b)** FTP server FCT **(c)** Node.js server FCT

Fig. 13: NETCAP achieves native network performance across various complex real-world applications (Apache, FTP, Node.js) while validating capabilities at the process-level

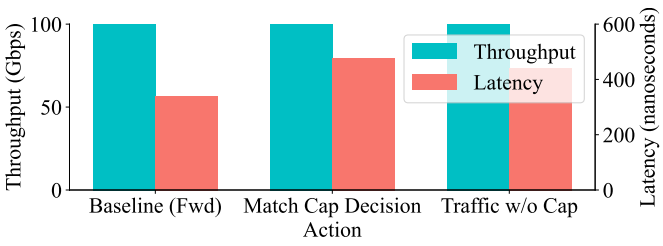


Fig. 14: Impact of NETCAP on throughput and switch processing latency compared to the forwarding baseline

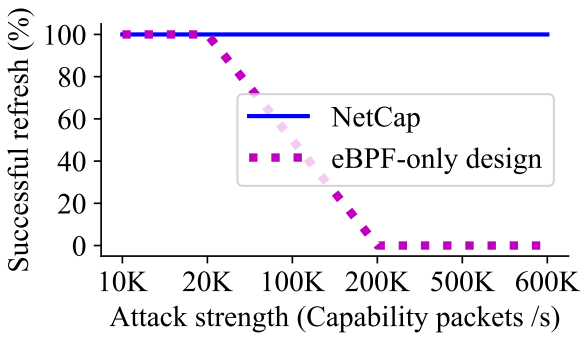


Fig. 15: Resilience against saturation attacks in server-based approach compared to NETCAP's in-network approach

Such latency is too high for practical deployments, causing noticeable delays in user access. In contrast, NETCAP's in-network implementation validates capabilities *in the data plane*, reducing delays to 2 microseconds in our Tofino 1 switch while processing packets at line rate (99.9 Gbps per port). *This represents a performance improvement of four orders of magnitude over the eBPF-only design.*

Resilience against saturation attacks. Beyond performance implications, the eBPF-only design introduces a single point of failure that can be easily exploited. We simulate an attacker flooding both designs with expired capabilities to be refreshed. In the eBPF-only design, the server CPU handles every refresh, allowing the attacker to quickly overwhelm the server and exhaust its processing capacity. As shown in Fig. 15, the eBPF-only design begins to saturate at an attack rate of 20K packets/s, and drops 99% of capability requests once the attack rate reaches 200K packets/s. At this point, legitimate clients could no longer receive refreshed capabilities, losing access to their services. In contrast, NETCAP processes capabilities within the data plane, *maintaining a stable performance for legitimate clients* and refreshing 100% of capabilities regardless of attack rate. These results show that offloading capability enforcement to the switch data plane is not just an optimization, but a necessity for resilience against saturation attacks and for preventing the server from becoming a bottleneck.

VIII. DISCUSSION

Capability delegation. Complex modern applications often involve a parent process delegating tasks to child processes. For example, an `update-manager` process responsible for managing software updates on a system may spawn a child `curl` process to download software updates. In this scenario, the child process receives capabilities from NETCAP, but the parent process will lack the capability to communicate with the server without re-authenticating itself. To relax this constraint, our client-side program treats the parent as equally trusted and assigns a new capability to both the authorized child and its parent. This enables seamless capability reuse strictly between parent and child processes without introducing an additional attack surface. We should note that supporting on-demand capability delegation between arbitrary processes would require careful eBPF-based designs for secure delegation and revocation, which we leave for future work.

Offloading eBPF programs to SmartNIC. Currently, eBPF programs can only attach to the TC egress hook, where they access the packet's socket buffer [38]. This leads to slower processing as the TC hook requires modifying the socket buffer's metadata. As a result, adding extra headers to every outgoing packet at the TC hook will degrade network performance.

NETCAP avoids this by using separate capability packets, avoiding costly operations on data packets. Alternatively, the eBPF program can run on a SmartNIC [69] (assuming the client is equipped with a SmartNIC). This approach enables faster packet processing within the NIC, reducing the overhead of real-time packet manipulation. NETCAP can leverage this to design a system that appends capability headers directly to data packets. However, this solution requires careful coordination between the SmartNIC and kernel eBPF programs that trace process activity. We will leave the integration of process-level visibility on SmartNICs for future work.

eBPF security. eBPF provides a secure sandbox environment for running user-defined programs in the kernel, enforced by a strict verifier to prevent unsafe or malicious operations that could crash or compromise the kernel. While several recent works [70], [71] have exploited bugs in the verifier to accept unsafe eBPF programs, allowing arbitrary reads and writes, many opposing works [72], [73] have proposed techniques to enhance safety verification. Additionally, further efforts aim to harden eBPF against compromised eBPF programs [74] through fine-grained BPF privileges [75]. Such efforts further bolster the security of our eBPF programs.

TEE. Trusted execution environments (TEEs) and NETCAP address fundamentally different threat surfaces. TEEs protect code and data confidentiality within enclaves and provide attestation to prove execution integrity. However, they protect tokens only while inside the enclave. Token thefts that occur outside enclaves (e.g., man-in-the-middle attacks, vulnerable applications) remain largely unprotected. In contrast, NETCAP's in-network, process-level authentication ensures only authorized processes can access remote resources, regardless of how or where a token is stolen.

IX. RELATED WORK

In-network programmability. Programmable switches are widely used in data centers to offload various networking tasks [76], [77], [78]. Another line of works developed defense primitives inside the network to defend against covert channels [35], cross-host attacks [36], distributed denial-of-service [37], [38], privacy threats [40], and RDMA vulnerabilities [79]. Unlike NETCAP, these works do not continuously authenticate processes to mitigate token hijacking.

Leveraging programmable switches and eBPF, P4Control [36] prevents cross-host lateral movements by performing in-network decentralized information flow control (DIFC). It assigns identical labels to processes within hosts to track attacker movement and stop cross-host attacks. In contrast, NETCAP addresses a fundamentally different problem: preventing attackers from exploiting stolen tokens to gain unauthorized network access. It achieves this by performing in-network, process-level authentication using capabilities that are cryptographically bound to process identifiers. NETCAP further introduces new techniques for secure capability creation, refresh, and process binding in the data plane, along with optimizations for high-throughput validation. Together, systems like NETCAP and P4Control highlight a promising *research direction* toward co-designing programmable switches and eBPF for end-to-end security enforcement across hosts and the network.

eBPF for cybersecurity. Existing efforts leveraged eBPF to develop offensive tools and eBPF-based malware [80], [81]. Other works explored new attacks that exploit eBPF programs to compromise cloud containers [74]. On the defensive side, several works [82], [83] utilized eBPF to enhance OS logging. NETCAP employs eBPF in a new way to enhance authentication schemes and enable process-level capabilities within the OS without kernel modifications.

Device-level continuous authentication. Traditional continuous authentication techniques have centered on verifying the user at the device level, particularly for smartphones and virtual/augmented reality (VR/AR) systems. These approaches authenticate users through biometric data (e.g., fingerprints and voice) [84], [85] or behavioral patterns (e.g., keystroke dynamics) [86]. While these methods effectively verify users on devices, they still implicitly trust all processes on an authenticated device. In contrast, NETCAP is the first to enforce fine-grained, network-level access control by associating access capabilities with individual processes rather than assuming device-level trust.

X. CONCLUSION

NETCAP is a protocol-independent, capability-based system designed to prevent attackers from using stolen tokens to gain unauthorized access to remote resources. It is the first solution to leverage programmable data planes for deploying a line-rate, capability-based defense that enables continuous access validation with negligible overhead.

There are several promising directions for future work. First, NETCAP's capability-based system can be extended to restrict memory access within the OS without excessive changes to the underlying kernel. This can be achieved by leveraging our eBPF programs to enforce fine-grained security decisions directly within the kernel. Second, we can extend our eBPF programs to support dynamic capability delegation and revocation across processes in different hosts to enable more complex workflows. Third, NETCAP can leverage SmartNICs to offload our eBPF programs onto the NICs, enabling faster packet modification and integrating process-level visibility directly at the network interface.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and our shepherd for their constructive comments and suggestions. This work is supported in part by the National Science Foundation under grant CNS-2442171 and the Google Academic Research Award. Any opinions, findings, and conclusions made in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] MITRE, "Steal application access token," https://attack.mitre.org/techniques/T1528/, 2024.

[2] R. Becwar, "Use alternate authentication material: Pass the ticket," https://attack.mitre.org/techniques/T1550/003/, 2023.

[3] A. Pingios, "Remote service session hijacking: Ssh hijacking," https://attack.mitre.org/techniques/T1563/001/, 2020.

[4] G. Rosen, "Security update," https://about.fb.com/news/2018/09/security-update/, 2018.

[5] R. Lakshmanan, "Nearly 100,000 npm users' credentials stolen in github oauth breach," https://thehackernews.com/2022/05/nearly-100000-npm-users-credentials.html, 2022.

[6] NIST, "Cve-2017-5638 detail," https://nvd.nist.gov/vuln/detail/cve-2017-5638, 2024.

[7] L. Roy, S. Lyakhov, Y. Jang, and M. Rosulek, "Practical Privacy-Preserving authentication for SSH," in *USENIX Security*, 2022, pp. 3345–3362.

[8] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *ACM CCS*, 2014, p. 892–903.

[9] S. K. Singh, S. Gautam, C. Cartier, S. Patil, and R. Ricci, "Where the wild things are: Brute-Force SSH attacks in the wild and how to stop them," in *USENIX NSDI*, 2024, pp. 1731–1750.

[10] T. A. Rahat, Y. Feng, and Y. Tian, "Cerberus: Query-driven scalable vulnerability detection in oauth service provider implementations," in *ACM CCS*, 2022, p. 2459–2473.

[11] "Recent github supply chain attack traced to leaked spotbugs token," https://www.bleepingcomputer.com/news/security/recent-github-supply-chain-attack-traced-to-leaked-spotbugs-token/.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," in *ACM SIGCOMM CCR*, 2014, p. 87–95.

[13] Okta, "What is the lifetime of okta minted json web tokens(jwt)," https://support.okta.com/help/s/article/What-is-the-lifetime-of-the-JWT-tokens?language=en_US, 2023.

[14] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of oauth implementations in android applications," in *ACSAC*, 2015, p. 61–70.

[15] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of oauth sso systems," in *ACM CCS*, 2012, p. 378–390.

[16] S. Sciancalepore, G. Piro, D. Caldarola, G. Boggia, and G. Bianchi, "Oauth-iot: An access control framework for the internet of things based on open standards," in *IEEE ISCC*, 2017, pp. 676–681.

[17] Microsoft, "Refresh tokens with oauth 2.0," https://learn.microsoft.com/en-us/linkedin/shared/authentication/programmatic-refresh-tokens, 2023.

[18] ——, "Maximum lifetime for service ticket," https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-10/security/threat-protection/security-policy-settings/maximum-lifetime-for-service-ticket, 2017.

[19] K. Patil, "Four reasons why ssh key management is challenging," https://www.appviewx.com/blogs/four-reasons-why-ssh-key-management-is-challenging/, 2024.

[20] A. W. and O. M., "Operationally transparent cyber datase," https://github.com/FiveDirections/OpTC-data, 2020.

[21] M. J. M. Turcotte, A. D. Kent, and C. Hash, "Unified host and network data set," in *World Scientific*, 2018, pp. 1–22.

[22] "NetCap source code," https://github.com/peng-gao-lab/netcap.

[23] R. S. Sandhu and P. Samarati, "Access control: principle and practice," in *IEEE Communications Magazine*, 1994, pp. 40–48.

[24] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," in *Commun. ACM*, 1966, p. 143–155.

[25] A. Birgisson, J. G. Politz, U. Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *NDSS*, 2014.

[26] S. Gusmeroli, S. Piccione, and D. Rotondi, "Iot access control issues: A capability based approach," in *IEEE IMIS*, 2012, pp. 787–792.

[27] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "Capstone: A capability-based foundation for trustless secure memory access," in *USENIX Security*, 2023, pp. 787–804.

[28] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE S&P*, 2015, pp. 20–37.

[29] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: an efficient mac algorithm for 32-bit microcontrollers," in *SAC*, 2014, pp. 306–323.

[30] M. Francisco, B. Ferreira, M. V. Fernando, E. Ramos, and S. Marin, "P4Chaskey: An efficient MAC algorithm for PISA switches," in *IEEE ICNP*, 2024.

[31] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable Cross-Host attack investigation with efficient data flow tagging and tracking," in *USENIX Security*, 2018, pp. 1705–1722.

[32] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *USENIX ATC*, 2018, pp. 113–125.

[33] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *NDSS*, 2020.

[34] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable In-Network security for context-aware BYOD policies," in *USENIX Security*, 2020, pp. 595–612.

[35] J. Xing, Q. Kang, and A. Chen, "Netwarden: Mitigating network covert channels while preserving performance," in *USENIX Security*, 2020, pp. 2039–2056.

[36] O. Bajaber, B. Ji, and P. Gao, "P4control: Line-rate cross-host attack prevention via in-network information flow control enabled by programmable switches and ebpf," in *IEEE S&P*, 2024, pp. 146–146.

[37] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *USENIX Security*, 2021, pp. 3829–3846.

[38] S. Yoo, X. Chen, and J. Rexford, "Smartcookie: Blocking large-scale syn floods with a split-proxy defense on programmable data planes," in *USENIX Security*, 2024.

[39] MITRE, "Capec-102: Session sidejacking," https://capec.mitre.org/data/definitions/102.html, 2018.

[40] T. Datta, N. Feamster, J. Rexford, and L. Wang, "SPINE: Surveillance protection in the network elements," in *USENIX FOCI*, 2019.

[41] S. Yoo and X. Chen, "Secure keyed hashing on programmable switches," in *SPIN*, 2021, p. 16–22.

[42] G. Ewing, "Reverse-engineering a crc algorithm," https://www.csse.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html.

[43] D. R. Stinson, *Cryptography: Theory and Practice*. Chapman and Hall/CRC, 2005.

[44] "The security flag in the ipv4 header ietf rfc3514," https://www.ietf.org/rfc/rfc3514.txt.

[45] E. Papadogiannaki and S. Ioannidis, "A survey on encrypted network traffic analysis applications, techniques, and countermeasures," in *ACM Comput. Surv.*, 2021.

[46] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian, "Eavesdropping on Fine-Grained user activities within smartphone apps over encrypted network traffic," in *USENIX WOOT*, 2016.

[47] IBM, "Tcp header field definitions," https://www.ibm.com/docs/en/aix/7.2?topic=protocols-tcp-header-field-definitions.

[48] "Computing tcp's retransmission timer," https://datatracker.ietf.org/doc/html/rfc6298.

[49] "ebpf official website," https://ebpf.io/.

[50] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *CoNEXT*, 2018, p. 54–66.

[51] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran *et al.*, "Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation." in *NDSS*, 2021.

[52] "What is an access log?" https://www.crowdstrike.com/en-us/cybersecurity-101/observability/access-logs/.

[53] G. Ho, A. Sharma, M. Javed, V. Paxson, and D. Wagner, "Detecting credential spearphishing in enterprise settings," in *USENIX Security*, 2017, pp. 469–485.

[54] C. O. Pérez, A. Daffalla, T. Ristenpart, and C. Tech, "Encrypted access logging for online accounts: Device attributions without device tracking," in *USENIX Security*, 2025.

[55] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in Software-Defined networks," in *USENIX Security*, 2017, pp. 415–432.

[56] "Facebook engineering. disaggregate: Networking recap," https://engineering.fb.com/2017/01/30/data-center-engineering/disaggregate-networking-recap/.

[57] "Onf, in collaboration with microsoft, google and intel, brings sdn to sonic™," https://p4.org/onf-in-collaboration-with-microsoft-google-and-intel-brings-sdn-to-sonic/.

[58] "Making the switch: Disruptive telecom white box collaboration accelerates and opens the platform, powering unprecedented network performance and insights," https://about.att.com/story/white_box_collaboration.html.

[59] "Using programmable chip and open source sw toward disaggregated network packet broker and 5g upf," https://opennetworking.org/wp-content/uploads/2020/12/10_350pm_Chris_Park.pdf.

[60] "Cisco joins the p4 consortium as a premier member in 2025," https://p4.org/cisco-joins-the-p4-consortium-as-a-premier-member-in-2025/.

[61] "Juniper advancing disaggregation through p4 runtime integration," https://blogs.juniper.net/en-us/engineering-simplicity/juniper-advancing-disaggregation-through-p4-runtime-integration.

[62] D. Hardt, "The oauth 2.0 authorization framework," https://datatracker.ietf.org/doc/html/rfc6749.

[63] NIST, "Cve-2011-2523 detail," https://nvd.nist.gov/vuln/detail/CVE-2011-2523, 2021.

[64] Rapid7, "Metasploit framework," https://docs.rapid7.com/metasploit/msf-overview/, 2008.

[65] Microsoft, "Kerberos authentication overview," https://learn.microsoft.com/en-us/windows-server/security/kerberos/kerberos-authentication-overview.

[66] fortra, "Impacket," https://github.com/fortra/impacket.

[67] A. Botta, A. Dainotti, and A. Pescapè, "A tool for the generation of realistic network workload for emerging networking scenarios," *Comput. Netw.*, pp. 3531–3547, 2012.

[68] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM*, 2017, p. 15–28.

[69] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hXDP: Efficient software packet processing on FPGA NICs," in *USENIX OSDI*, 2020, pp. 973–990.

[70] NIST, "Cve-2023-39191 detail," https://nvd.nist.gov/vuln/detail/CVE-2023-39191, 2023.

[71] ——, "Cve-2020-8835 detail," https://nvd.nist.gov/vuln/detail/CVE-2020-8835, 2020.

[72] L. Nelson, J. V. Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel," in *USENIX OSDI*, 2020, pp. 41–61.

[73] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *ACM PLDI*, 2019, p. 1069–1084.

[74] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, "Cross container attacks: The bewildered ebpf on clouds," in *USENIX Security*, 2023, pp. 5971–5988.

[75] J. Corbet, "Finer-grained bpf tokens," https://lwn.net/Articles/947173/.

[76] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR*, 2016.

[77] A. Devraj, L. Wang, and J. Rexford, "Redact: Refraction networking from the data center," in *ACM SIGCOMM CCR*, 2021, p. 15–22.

[78] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow," in *USENIX ATC*, 2018, pp. 823–835.

[79] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen, "Bedrock: Programmable network support for secure RDMA systems," in *USENIX Security*, 2022, pp. 2585–2600.

[80] G. Fournier, S. Baubeau, and S. Baubeau, "ebpf, i thought we were friends," in *DEFCON*, 2021.

[81] J. Dileo, "Evil ebpf: Practical abuses of an in-kernel bytecode runtime," in *DEFCON*, 2019.

[82] R. Sekar, H. Kimm, and R. Aich, "eaudit: A fast, scalable and deployable audit data collection system," in *IEEE S&P*, 2024, pp. 87–87.

[83] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *ACM SoCC*, 2021, p. 518–532.

[84] S. Eberz, K. B. Rasmussen, V. Lenders, and I. Martinovic, "Evaluating behavioral biometrics for continuous authentication: Challenges and metrics," in *ACM AsiaCCS*, 2017, p. 386–399.

[85] H. Feng, K. Fawaz, and K. G. Shin, "Continuous authentication for voice assistants," in *MobiCom*, 2017, p. 343–355.

[86] S. P. Banerjee and D. L. Woodard, "Biometric authentication and identification using keystroke dynamics: A survey," *Journal of Pattern recognition research*, pp. 116–139, 2012.