AirCloak: An App-Transparent Traffic Cloaking Middleware Against Wireless Fingerprinting Attack

Huafeng Bian, Jianfeng Li, Haodan Luo, Yixuan Liu, Xiaobo Ma, *Member, IEEE*, Zhenhua Li, *Senior Member, IEEE*, Jialong Zhang, Jigang Wang, and Wei Wang, *Senior Member, IEEE*

***Abstract*—Despite the traditional perception that Wi-Fi communications are secure and resistant to eavesdropping, recent advancements in wireless fingerprinting have challenged this view. Emerging wireless app fingerprinting techniques exploit side-channel traffic features to infer fine-grained, privacy-sensitive user behaviors. If left unresolved, this threat could expose nearly all Wi-Fi users to pervasive app fingerprinting attacks. While several defense strategies show potential, each suffers from inherent limitations. For example, in-app traffic obfuscation (*client-side*) can effectively conceal traffic patterns but requires modifying individual apps, making it impractical for rapid deployment at scale. On the other hand, encrypted proxy-based defenses (*cloud-side*) offer certain traffic shaping capabilities, but their encapsulation and multi-hop forwarding introduce additional latency and degrade overall performance. To address this pressing need, we pioneer an *edge-side* traffic obfuscation architecture, and develop `AirCloak`, an immediately deployable defense middleware that avoids app-level modification and minimizes performance overhead, making it especially practical for privacy-conscious users demanding high-security Wi-Fi environments. `AirCloak` relies on the cooperation between mobile terminals (e.g., smartphones) and the local Wi-Fi access point (AP), enabling real-time traffic cloaking while remaining transparent to the apps themselves. Despite its promise, `AirCloak` faces critical challenges: *traffic heterogeneity* and *defense evasion*. To overcome these challenges, a *harmonization shaping* obfuscation approach is proposed. It unifies traffic behaviors of structurally similar apps through adaptive generative synthesis, enabling efficient and resilient obfuscation that conceals discriminative fingerprints with minimal performance cost. We prototype `AirCloak` as a one-click installable app on mobile terminals and lightweight scripts on the AP. Extensive evaluations demonstrate `AirCloak`'s superior defensive effectiveness. It reduces the average F1-score of four WAF attacks from 0.77 to 0.19, while imposing negligible runtime overhead on both mobile terminals and the AP.**

***Index Terms*—Traffic analysis, fingerprinting defense, wireless network security.**

Received 4 November 2025; revised 2 March 2026; accepted 14 June 2026; approved by IEEE TRANSACTIONS ON NETWORKING Editor J. S. Sun. Date of publication 22 June 2026; date of current version 3 July 2026. This work was supported in part by the National Natural Science Foundation of China under Grant U23A20332, Grant 62272381, Grant 62441227, Grant 62572388, and Grant U25B2068; in part by the Key Research and Development Program of Shaanxi under Grant 2024PT-ZCK-91; in part by the Fundamental Research Funds for the Central Universities; in part by the Postdoctoral Science Foundation of China under Grant 2024T170722 and Grant 2023M732791; and in part by the Youth Innovation Team of Shaanxi Universities. *(Corresponding author: Xiaobo Ma.)*

Huafeng Bian, Jianfeng Li, Haodan Luo, Yixuan Liu, Jialong Zhang, and Wei Wang are with the MOE Key Laboratory for Intelligent Networks and Network Security and the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: t1071680825@stu.xjtu.edu.cn; jfli.xjtu@xjtu.edu.cn; LuoHaodan@stu.xjtu.edu.cn; 3113625391@stu.xjtu.edu.cn; jlzhang@xjtu.edu.cn; wei.wang@xjtu.edu.cn).

Xiaobo Ma is with the MOE Key Laboratory for Intelligent Networks and Network Security and the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China, and also with Shaanxi Province Key Laboratory of Computer Network, Xi'an 710049, China (e-mail: xma.cs@xjtu.edu.cn).

Zhenhua Li is with the School of Software and BNRist, Tsinghua University, Beijing 100084, China (e-mail: lizhenhua1983@gmail.com).

Jigang Wang is with ZTE Corporation, Shenzhen 518057, China (e-mail: wang.jigang@zte.com.cn).

Digital Object Identifier 10.1109/TON.2026.3705510

I. INTRODUCTION

MOBILE apps widely adopt encrypted communication. However, existing research has revealed that attackers can still infer app types and privacy-sensitive behaviors by analyzing side-channel traffic features, such as packet length, direction, and timing [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Among various attacks, wireless app fingerprinting (WAF) attacks are particularly concerning due to their ease of deployment and stealth. Without infiltrating the network, a WAF attacker can passively capture app traffic by deploying a sniffer within Wi-Fi coverage, enabling continuous user monitoring. This poses a serious threat to user privacy [13], [14], [15], [16], [17]. For instance, by analyzing traffic generated by healthcare apps, an attacker could infer medical information about the user. Researchers at NYU Tandon demonstrated this risk by showing that traffic from RouterSense, a health monitoring app, could reveal signs of cognitive decline in Alzheimer's patients [18]. Furthermore, as certain apps directly interface with IoT devices, attackers may extend cyber-level app attacks into the physical world. For instance, identifying traffic from a smart lock app operating in "away mode" could enable attackers to facilitate burglary or other malicious activities [19].

A straightforward defense against WAF attacks is for app developers to integrate traffic obfuscation mechanisms on both the client and server sides, thereby preventing the exposure of traffic features [20], [21], [22], [23]. However, this approach requires modification of each app, resulting in high deployment costs and limited scalability across the vast and heterogeneous app ecosystem [24]. An alternative defense is to deploy encryption proxies [25], [26], [27]. These proxies are designed to anonymize destinations by relaying traffic through intermediate servers. To reduce detectability, they also apply traffic shaping. However, such shaping often uses static parameters that leak side-channel features. For example, OpenVPN pads packets to advanced encryption standard (AES) block sizes, yielding lengths in multiples of 16 bytes. This creates clusters in packet length distributions, allowing attackers to infer destination fingerprints. Dynamic shaping can offer better protection but increases

2998-4157 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

system complexity and resource usage. Specifically, encapsulation and multi-hop forwarding introduce extra latency and degrade Quality of Service (QoS). Our evaluation (see § VI-E) shows that even lightweight proxies like Shadowsocks add about 19 ms RTT (↑ 40.43%). Dynamic strategies would incur even higher overhead, further harming user experience.

Consequently, deployable and effective defenses remain lacking, leaving mobile apps persistently exposed to WAF risks. To bridge this gap, there is a pressing need for a system that supports rapid deployment in real-world settings. At the core of such a system lie two essential objectives that must be met to ensure swift and effective integration into diverse app ecosystems, i.e., *app transparency* and *marginal overhead*. The former requires that the defense operates transparently to apps, necessitating no modifications to either the client or server. The latter mandates that the defense minimizes communication overhead and latency, ensuring a seamless user experience. These two foundational principles enable the defense to swiftly adapt to the dynamic and varied app landscapes.

To achieve the objectives above, we propose an *edge-side* traffic obfuscation architecture. This architecture leverages the cooperation between mobile terminals and the access point (AP) to perform real-time cloaking of app traffic. For mobile terminals, only a traffic-shaping app needs to be installed, without any modifications to the apps themselves. The traffic-shaping app allows users to select which apps to obfuscate, thereby offering WAF defense tailored to their needs. On the AP side, by operating at the network's edge within the same local area network (LAN) as the mobile terminals, cooperation with the traffic-shaping app ensures that the localized traffic processing mechanism eliminates the need for cross-domain forwarding. This approach significantly reduces communication latency and mitigates the risk of data leakage. Furthermore, since the AP only obfuscates traffic from a limited number of local devices, it maintains manageable computational overhead.

As a matter of fact, ensuring a defense with properties of app transparency (thus deployable on user demand) and marginal overhead is only part of the equation. To truly be effective, the defense must also demonstrate *resilience against sophisticated attacks*. In particular, it must withstand adversarial traffic analysis aimed at bypassing defenses. Only with this added robustness can the defense be deemed reliable and effective. However, implementing such resilience in the edge-side architecture still faces two challenges.

C1: Traffic Heterogeneity. Different apps exhibit distinct traffic patterns, and even the same app may produce significantly different patterns under varying operations. As a result, a fixed obfuscation strategy struggles to balance defense effectiveness and overhead. For instance, uniformly transforming the traffic of low-bandwidth apps into that of high-bandwidth ones may lead to network congestion. Therefore, an efficient defense must adaptively adjust its strategy to heterogeneous traffic, achieving effective obfuscation with minimal overhead.

C2: Defense Evasion. Attackers may continuously monitor app traffic both before and after defense. By leveraging this data, they can retrain fingerprint classifiers to recognize obfuscation patterns, filter them out, and ultimately circumvent the defense. Therefore, an effective defense should conceal static traffic characteristics and, at the same time, ensure resistance to adversarial or retraining attacks.

TABLE I
AIRCLOAK VS. EXISTING DEFENSES

Defenses	App Transparent	Lightweight	Robust Defense
AppMutation [20]	○	○	○
Chaddad et al. [22]	○	○	○
Cactus [23]	●	○	○
Random	○	○	○
GAN [28]	○	○	◐
Surakav [29]	○	◐	●
AirCloak (Our Method)	●	●	●

● Fully Supported ◐ Partially Supported ○ Not Supported

To address these challenges, we introduce a novel traffic obfuscation method called *harmonization shaping*, integrated into our edge-side architecture as a middleware solution, AirCloak. Harmonization shaping relies on the fact that apps with similar traffic patterns incur less overhead when obfuscated together than those with divergent patterns (see IV-D for details). AirCloak begins by segmenting app traffic into bursts (i.e., consecutive packets within short intervals) and creating structured feature matrices to characterize them. It groups heterogeneous apps into anonymity clusters based on matrix similarity. For each cluster, a dedicated traffic generation model obfuscates traffic to mask intra-cluster differences, addressing **C1**. To improve robustness, adversarial augmentation refines the obfuscation models. Before transforming real traffic, generated obfuscated traffic undergoes three refinement mechanisms: early sending, gap filling, and trace fine-tuning, ensuring consistency with real communication patterns and addressing **C2**.

With these innovations, AirCloak stands apart from existing defenses, as summarized in TABLE I. We prototyped AirCloak with two middleware components: a terminal-side module built upon the *VpnService* framework [30] and packaged as a one-click installable app, and an AP-side module implemented using the *Netfilter* framework. AirCloak was evaluated on public datasets and deployed in a real-world network. Extensive evaluations demonstrate that AirCloak effectively defends against three types of WAF attacks, and consistently outperforms existing defenses.

Our work has made the following contributions:

- To our best knowledge, AirCloak represents the first effort towards an edge-side defense middleware against WAF attacks. It overcomes two key challenges, namely traffic heterogeneity and defense evasion, and achieves effective and efficient cloaking of heterogeneous app traffic.
- We propose a novel harmonization shaping method to efficiently obfuscate heterogeneous traffic. To enhance defense robustness, we introduce an adversarial traffic generation model. Additionally, we design three traffic optimization strategies to balance defense overhead and effectiveness.
- Extensive evaluations on both offline datasets and online deployments demonstrate that AirCloak effectively

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

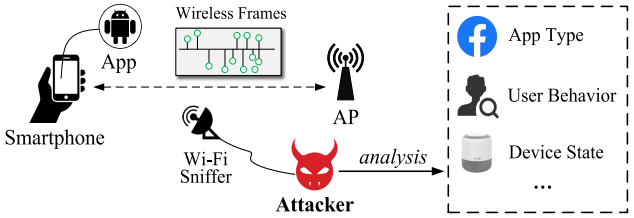


Fig. 1. WAF attack. An attacker deploys a Wi-Fi sniffer within the AP's coverage to capture and analyze wireless traffic.

mitigates WAF attacks. Specifically, it reduces the average F1-score of four WAF attacks from 0.77 to 0.19. Moreover, AirCloak imposes minimal overhead on mobile terminals, with only a 3% increase in CPU usage, an additional 3 ms in RTT, and a 5% rise in packet loss rate.

II. Background and Motivation

We present the fundamental principles of WAF attacks and the motivation behind our defense.

A. Background: Wireless App Fingerprinting Attack

Although mobile apps widely adopt encrypted communication, side-channel traffic features such as packet length, direction, and timing remain exposed. These features, intrinsically linked to an app's runtime logic, exhibit distinctive patterns. For example, Li et al. [1] demonstrated that different user activities on Twitter (such as following, notifications, and messaging) correspond to unique traffic patterns. Such attacks [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] can accurately identify app types and infer privacy-sensitive behaviors (e.g., video calls and financial transactions), thereby posing significant threats to user privacy.

The defense scenario considered in this work targets typical Wi-Fi environments, such as home and enterprise networks. As illustrated in Fig. 1, an attacker can passively capture 802.11 frames using a wireless sniffer within Wi-Fi coverage, without compromising the network or obtaining system-level privileges, and subsequently perform WAF attacks by exploiting exposed side-channel features.

B. Motivation for Our Defense

Existing WAF defenses predominantly rely on in-app traffic obfuscation and encryption proxies.

In-app traffic obfuscation requires modifying both the client and the server for each protected app. While effective, this defense incurs high deployment costs and lacks scalability across the vast and rapidly evolving app ecosystem.

Encryption proxy-based defenses leverage cloud servers for centralized traffic forwarding, avoiding modifications to apps. However, this centralized architecture has inherent limitations: i) concurrent device access greatly burdens proxy servers; ii) cloud forwarding introduces additional latency; and iii) transmitting sensitive data, especially in healthcare and finance, raises privacy concerns. Moreover, such proxies are designed to conceal destinations rather than obfuscate side-channel features, and thus provide limited defense against WAF attacks.

In contrast, AirCloak adopts an edge-side obfuscation architecture, in which defensive middleware deployed on terminals and the AP collaboratively perform traffic cloaking.

III. AirCloak Overview

We introduce our new defense, AirCloak, by outlining its design goal and presenting an overview of its workflow.

A. Design Goal

AirCloak is designed for app-transparent deployment, while providing effective and efficient WAF defense.

1) App-Transparent Deployment: AirCloak adopts an edge-side defense architecture that performs traffic obfuscation locally. Lightweight middleware is deployed on both the AP and the mobile terminal, communicating via an encrypted tunnel. Upon detecting traffic from a target app, the middleware triggers obfuscation and forwards the transformed traffic via the tunnel. The process remains fully transparent to apps, requiring no modifications to either the client or server.

2) Effective and Efficient Defense: Traditional obfuscation strategies, such as randomization and uniformization, have inherent limitations. Randomization may expose global features (e.g., peaks or overall volume) [2], [4], [5], [31], while uniformization incurs significant overhead when transforming heterogeneous traffic. To address these issues, AirCloak introduces *harmonization shaping*: it clusters apps with homogeneous traffic patterns and performs intra-cluster traffic obfuscation, thereby preventing attackers from identifying individual apps within the same cluster.

B. AirCloak Workflow

Fig. 2 illustrates the workflow of AirCloak. Apps on the mobile terminal continuously generate network traffic, which is captured in real time by the terminal-side middleware. The middleware applies harmonization shaping to the uplink traffic and forwards the obfuscated packets to the AP via an encrypted tunnel (see § V-A). The AP-side middleware then reconstructs the original traffic and forwards it to the Internet. Downlink traffic follows the reverse path, being processed at the AP before delivery to the terminal.

Specifically, AirCloak begins by continuously observing app traffic and processing it using the *Traffic Characterization* module (§ IV-A), which extracts burst-level features. These features are formulated as a structured matrix, termed *ATAM*. Multiple *ATAM* instances are collected per app and aggregated into a unified matrix (*HM*) to represent its overall traffic profile. The *Homogeneous App Clustering* module (§ IV-B) quantifies the similarity between *HM*s and groups apps with similar patterns into anonymity clusters. Subsequently, the *Obfuscation Traffic Generation* module (§ IV-C) leverages an adversarial model to learn cluster-level traffic distributions. Based on the learned models, the *Obfuscation Traffic Optimization* module (§ IV-D) produces coarse-grained obfuscation traffic, which is further refined using three runtime strategies (i.e., early sending, gap filling, and trace fine-tuning) to preserve temporal and structural consistency before traffic transmission.

IV. AirCloak Design

We elaborate on the four core modules of AirCloak: *Traffic Characterization* (§ IV-A), *Homogeneous App Clustering* (§ IV-B), *Obfuscation Traffic Generation* (§ IV-C), and *Obfuscation Traffic Optimization* (§ IV-D).

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

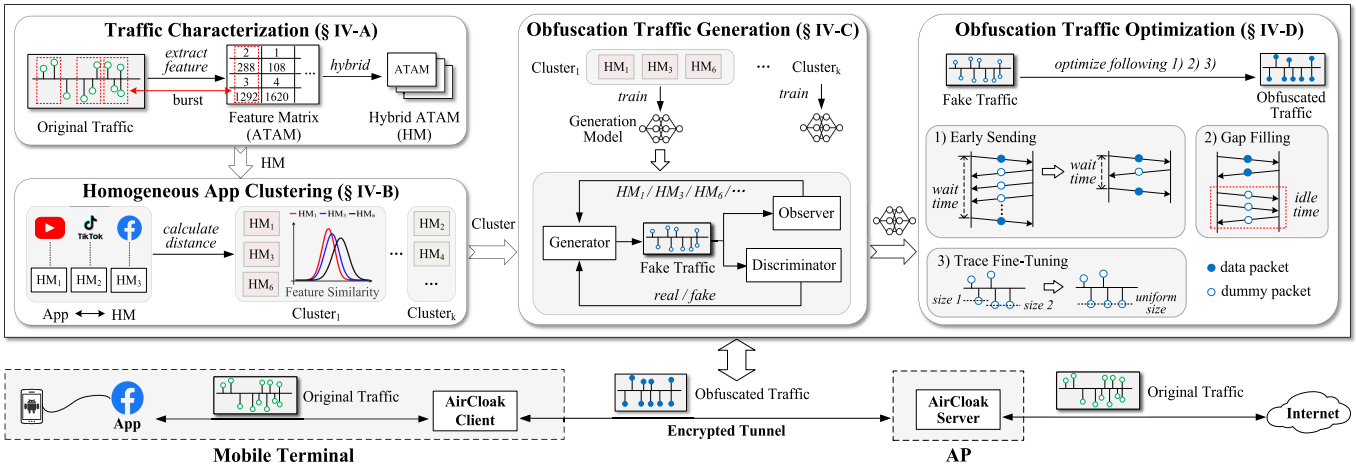
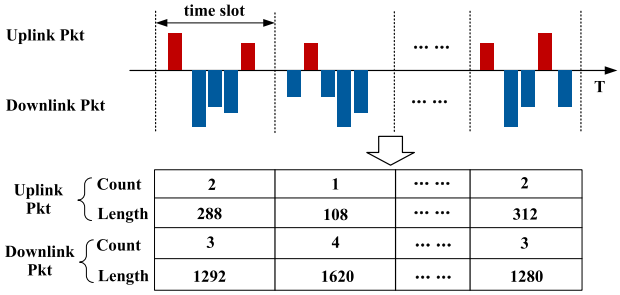


Fig. 2. AirCloak's workflow and overall architecture. AirCloak deploys client middleware on the mobile terminal and server middleware on the AP. Both ends capture original traffic, apply harmonization shaping, and transmit the obfuscated traffic through an encrypted tunnel. The obfuscation process consists of four steps: i) characterize traffic; ii) cluster homogeneous apps; iii) train a traffic generation model for each cluster; and iv) optimize obfuscated traffic for transmission.



Uplink Pkt	Count	2	1	… …	2
	Length	288	108	… …	312
Downlink Pkt	Count	3	4	… …	3
	Length	1292	1620	… …	1280

Fig. 3. *ATAM* characterizes traffic by calculating the count and length of uplink and downlink packets within each time slot and representing these features as a matrix.

A. Traffic Characterization

This module characterizes app traffic through two steps: constructing characterization matrices and aggregating them.

1) Construct Characterization Matrix: App traffic originates from user interactions and typically manifests as short packet bursts [10]. After deploying AirCloak, the obfuscated traffic is encapsulated within an encrypted tunnel, preventing attackers from observing flow-level bursts. To accurately model the packet-level characteristics visible to attackers and support targeted obfuscation, we introduce the App Traffic Aggregation Matrix (*ATAM*). As illustrated in Fig. 3, itATAM partitions time into N slots of duration t, recording in each slot the count and total length of uplink and downlink packets. These features form a matrix $M \in \mathbb{R}^{4\times N}$. Based on the experimental analysis in § B-A, we set $t = 50$ ms, $N = 100$.

2) Aggregate Characterization Matrix: App traffic varies with context and user interactions, making it difficult for a single $ATAM$ to fully characterize an app's overall behavior. Therefore, we construct a hybrid matrix by extracting stable features from multiple $ATAM$ instances. For each $M[i][j]$ (i.e., the i^{th} feature in the j^{th} time slot), we analyze its distribution in terms of central tendency (mean and extrema) and stability. To quantify representativeness, we adopt a variance-based weighting mechanism that assigns larger weights to positions with lower variance, as they exhibit higher consistency across samples, and smaller weights to positions with higher variance, which are less reliable. Based on this, we construct a hybrid *ATAM* (HM), computed as

$$HM[i][j] = \alpha \cdot \frac{\mu_{ij}}{\sigma_{ij}^2} + (1-\alpha) \cdot \frac{m_{ij}}{\sigma_{ij}^2}, \tag{1}$$

where α balances the contributions of the mean μ_{ij} and the maximum m_{ij}, while the variance σ_{ij}^2 reflects feature stability. This formulation jointly captures long-term average trends and short-term burst variations.

B. Homogeneous App Clustering

Mobile apps inherently exhibit heterogeneous traffic patterns. When trained jointly across all apps, traffic generation models tend to overfit frequent or structurally simple patterns, thereby reducing defense effectiveness. Thus, we propose a harmonization shaping strategy that clusters apps with similar traffic characteristics and performs modeling within each cluster. This design mitigates cross-app bias and reduces the distinguishability of apps within the same cluster.

1) Calculate Inter-HM Distances: As described in § IV-A, each app is represented as $HM \in \mathbb{R}^{4\times N}$, spanning N time slots and four traffic features (uplink/downlink packet counts and lengths). To measure the similarity between HM_m and HM_n, we employ a Dynamic Time Warping (DTW)-based distance metric, which enables elastic temporal alignment and accommodates runtime-induced variations (e.g., network fluctuations). Specifically, we construct a matrix $D \in \mathbb{R}^{(N+1)\times(N+1)}$, where $D[i][j]$ denotes the minimum cumulative distance required to align the first i time windows of HM_m with the first j time windows of HM_n. D is initialized as

$$\begin{cases} D[0],[0] = 0, \\ D[i][0] = \infty, & \text{for } i > 0 \\ D[0][j] = \infty, & \text{for } j > 0 \end{cases} \tag{2}$$

The local distance at a given time slot, denoted as $d(HM_m[:, i-1], HM_n[:, j-1])$, is computed using the

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

Euclidean distance. The cumulative distance $D_{mn}[i][j]$ is updated iteratively as

$$D_{mn}[i][j] = d(HM_m[:, i-1], HM_n[:, j-1]) + \min(D[i-1][j], D[i][j-1], D[i-1][j-1]). \quad (3)$$

Subsequently, $D_{mn}[i][j]$ is normalized to remove the influence of alignment path length, thereby ensuring comparability across HMs. The normalized distance is

$$D^*_{mn}[i][j] = \frac{D_{mn}[i][j]}{N}. \quad (4)$$

Finally, D^*_{mn} denotes the normalized distance between HM_m and HM_n, where smaller values indicate a higher similarity in their traffic patterns.

2) Cluster HMs: To satisfy evolving defense requirements, particularly the emergence of new apps, we adopt an extensible K-Means algorithm to cluster HMs in an unsupervised manner. Given a predefined cluster number k, we randomly initialize k cluster centers $\{C_1, C_2, \cdots, C_k\}$. For each HM, its distance to each C_i is computed using Eqn. (4) and it is assigned to the nearest cluster. When a new HM joins cluster S_i, the cluster center is updated by recalculating the centroid C_i^{new} as

$$C_i^{\text{new}} = \frac{1}{|S_i|} \sum_{HM \in S_i} HM. \quad (5)$$

Sample assignment and centroid updates iterate until convergence or the maximum number of iterations is reached. In practice, clusters with very few samples may exhibit distinctive patterns and remain vulnerable to WAF attacks. To mitigate this issue, we introduce cluster-size regularization. Clusters below a predefined threshold are identified, their distances to other centroids are computed, and they are merged into the nearest sufficiently large cluster. When multiple small clusters are in close proximity, they are merged preferentially.

3) Update HMs: Mobile app updates may introduce traffic pattern drift, potentially invalidating prior cluster assignments. To this end, we design a traffic drift detection and online cluster update mechanism. Let the current traffic representation of app a be HM_a, and let C_c denote its assigned cluster centroid. The distance between them is defined as $\Delta_a = d(HM_a, C_c)$. If this distance exceeds a threshold τ for Γ consecutive windows, traffic drift is declared. Here, τ is set as the 95th percentile of natural intra-app fluctuation, and $\Gamma = 100$. Once drift is detected, distances between the updated HM_a and all centroids C_i are computed, and the app is reassigned to the nearest cluster. HM_a then inherits the corresponding cluster-level generative model for subsequent traffic obfuscation.

The clustering procedure is detailed in § A-A.

C. Obfuscation Traffic Generation

For each cluster, `AirCloak` trains a dedicated generative model to produce obfuscation traffic aligned with the cluster's traffic characteristics. An adversarial training mechanism is then employed to iteratively refine the generated traffic, improving its realism and diversity.

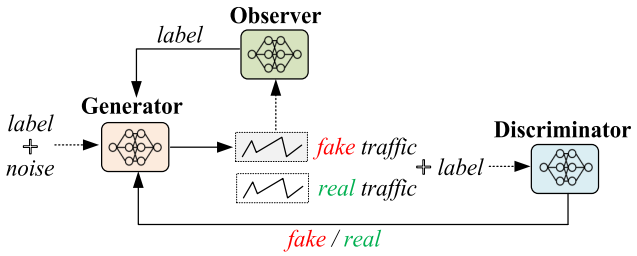


Fig. 4. Architecture of the traffic generation model.

1) Model Architecture: We design a conditional Generative Adversarial Network (cGAN) model to generate app-specific traffic. As illustrated in Fig. 4, the model consists of a generator, a discriminator, and an observer.

Generator learns authentic app traffic patterns and generates app-specific obfuscated (fake) traffic. It adopts an LSTM-based architecture that takes an intra-cluster label c and random noise z as input, and outputs the obfuscated traffic $G(z)$.

Discriminator distinguishes between authentic and obfuscated traffic. It takes a labeled traffic sample as input and outputs a scalar indicating the probability that the sample is real.

Observer enhances semantic consistency between the obfuscated traffic and the target app. It is a pre-trained classifier that takes traffic samples that have successfully fooled the discriminator as input and outputs the predicted app label.

We train a dedicated model per cluster to generate obfuscated traffic aligned with cluster-level characteristics. This design avoids applying a uniform obfuscation strategy to heterogeneous apps, thereby minimizing defense overhead (e.g., unnecessary high-density obfuscation for sparse traffic). Moreover, the cluster-specific design increases feature overlap among apps within the same cluster, further enhancing indistinguishability. The detailed cGAN architecture is provided in § A-B.

2) Model Training: Model training involves three steps: training the discriminator, the observer, and the generator.

The *discriminator* distinguishes whether a traffic sample conforms to the real traffic distribution. It is trained to maximize the probability assigned to real samples and minimize that of generated samples, thereby strengthening its discriminative capability. Its loss function is

$$L_D = -\mathbb{E}_x \left[\log D(x)\right] - \mathbb{E}_z \left[\log\left(1 - D(G(z))\right)\right], \quad (6)$$

where the first term denotes the cross-entropy loss on real traffic x, and the second penalizes classifying generated traffic $G(z)$ as real.

The *observer* classifies generated traffic and guides the generator by minimizing the discrepancy between predicted and true labels, thereby preserving the target app's semantic characteristics. Its loss function is

$$L_O = -\sum_i c_i \log \hat{c}_i, \quad (7)$$

where $\hat{c}_i$ is the predicted label.

The *generator* aims to produce traffic that is both realistic and app-specific. It is optimized via joint gradient feedback from the discriminator and the observer. The loss function is

$$L_G = -\mathbb{E}_z \left[\log D(G(z))\right] - \lambda \mathbb{E}_z \left[\sum_i c_i \log \hat{c}_i\right]. \quad (8)$$

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

The global optimization objective is formulated as

$$\min_G \max_D \mathcal{L}_{\text{total}} = \mathbb{E}_x \left[\log D(x)\right] + \mathbb{E}_z \left[\log(1 - D(G(z)))\right] - \lambda \mathbb{E}_z \left[\sum_i c_i \log \hat{c}_i\right]. \quad (9)$$

During training, the generator, discriminator, and observer are optimized alternately. As the generator initially produces low-quality traces, introducing the observer too early may hinder convergence. Thus, the observer loss is initially set to 0 and progressively introduced as generation quality improves.

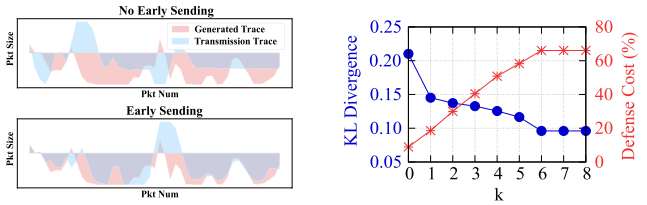
Algorithm 1 Traffic Transmission

Input: sending buffer ψ, generated sequence ξ
Output: transmission sequence κ
Initialization: $p \leftarrow$ first packet in ψ, $q \leftarrow$ first packet in ξ;
while $\psi \neq \emptyset$ ***and*** $\xi \neq \emptyset$ **do**
  retrieve p, q;
  if $len(p) > len(q)$ **then**
    split p into p_1, p_2, s.t. $len(p_1) = len(q)$;
    send p_1;
    return p_2 to ψ;
  else if $len(p) < len(q)$ **then**
    if $\psi \neq \emptyset$ **then**
      update p;
      retrieve p' from ψ;
      merge p, p' into p;
      send p;
    else
      pad p to $len(q)$;
      send p;
    end
  else if $len(p) = len(q)$ **then**
    send p;
  end
  update p, q;
end
while $\psi = \emptyset$ ***and*** $\xi \neq \emptyset$ **do**
  forge p;
  send p;
end

D. Obfuscation Traffic Optimization

This module uses the trained cGAN generator to produce obfuscation traffic, jointly enforced by the mobile terminal and the AP. Both sides establish dedicated send buffers, capturing raw packets from their network interfaces and forwarding them to these buffers. At each time slot, the generator outputs a sampled traffic trace. The terminal and AP then retrieve packets sequentially and apply transformations (such as fragment, reassemble, pad, and forge, as detailed in § V-B) to match the sampled trace. The detailed steps are presented in Algorithm 1. To further enhance realism, we introduce three runtime mechanisms: early sending, gap filling, and trace fine-tuning to optimize the generated obfuscation traffic.

1) Early Sending: In real-world networks, app traffic evolves continuously. A traffic generation model trained on historical data may not fully capture runtime variation, leading to mismatches between generated and authentic traffic, particularly in packet direction and timing. For example, if an app needs to send a 100-byte uplink packet from the terminal, but the generator produces a downlink trace (e.g., $[\cdots, -200, -300, \cdots]$), the AP must first forge two downlink packets (200 and 300 bytes) before allowing the uplink transmission. Although this strategy enforces conformity between generated traces and observed behavior, it inevitably incurs additional defense overhead and communication latency.



(a) Early sending. (b) The impact of k on KL divergence and defense cost.

Fig. 5. Subfigure (a) compares the generated traffic with the transmission traffic after applying early sending. Subfigure (b) shows the impact of k on the KL divergence between the generated and transmission traffic, as well as the defense cost.

To address this, we introduce an early sending mechanism. When k consecutive real packets fail to align with the generated trace, `AirCloak` temporarily suspends forced packet injection and allows immediate transmission of real packets. Fig. 5(a) shows that enabling this mechanism significantly improves structural similarity between generated and actual traces. Fig. 5(b) quantifies its impact under different k values using two metrics: KL divergence, which measures the statistical discrepancy between transmission and generated traces, and defense cost, which reflects the additional injected data. As k increases, `AirCloak` prioritizes trace consistency by continuing to inject expected packets despite mismatches, albeit at the cost of higher overhead.

To balance traffic alignment and overhead, we propose a history-aware adaptive strategy that adjusts k in real time based on prior observations. At time slot t, the matching metric is

$$\zeta_t = \frac{1}{l} \sum_{i=1}^{l} \left| X_i - \hat{X}_i \right|, \quad (10)$$

where X is the transmission trace, $\hat{X}$ is the generated trace, and l is the trace length. The update strategy for k is

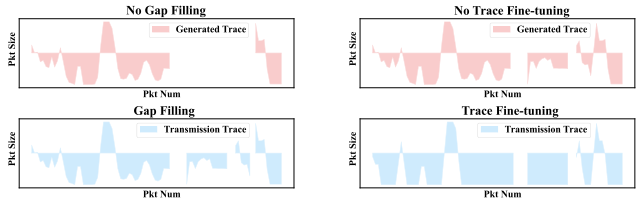
$$k_t = k_0 + \chi \left(\frac{1}{N} \sum_{i=t-N+1}^{t} \zeta_i - \gamma \right), \quad (11)$$

where k_0 is the initial value, γ is the matching threshold, N is the number of historical time slots, and χ is the adjustment rate. Based on experiments on the Android150 [9] and AppTrace50 datasets, we set $k_0 = 3, \gamma = 0.9, N = 5$, and $\chi = 0.1$.

2) Gap Filling: App traffic is strongly driven by user interactions. During inactivity, pronounced idle gaps emerge in the traffic stream. Such intervals may act as distinctive fingerprints, as attackers can exploit their frequency, duration, and timing to identify the source app.

The gap filling mechanism injects perturbations into authentic traffic, thereby obscuring distinctive idle intervals. When `AirCloak` detects consecutive idle time windows, it maintains a counter; if the counter exceeds a threshold ω, the segment is treated as a potential fingerprinting risk and gap filling is triggered. To avoid reverse engineering, ω is randomized following an exponential distribution, i.e., $\omega \sim \text{Exp}(\cdot)$. Once

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.



(a) Gap filling. (b) Trace fine-tuning.

Fig. 6. Subfigure (a) compares the generated traffic with the transmission traffic after applying gap filling, and subfigure (b) shows the results after applying trace fine-tuning.

triggered, the mobile terminal and the AP collaboratively inject dummy packets based on the generated trace, without waiting for real traffic. As shown in Fig. 6(a), prominent idle segments in the original traffic trace are replaced by forged packets of varying directions and lengths, creating the appearance of continuous activity.

3) Trace Fine-Tuning: Statistical analysis of the Android150 [9] and AppTrace50 datasets reveals a consistent pattern: over 70% of packets are server responses, typically close to the Maximum Transmission Unit (MTU). This reflects the common practice of segmenting large resources (e.g., images or media) into near-MTU-sized packets to optimize throughput. Consequently, response packets significantly smaller than the MTU may appear statistically abnormal and serve as potential fingerprints.

We propose a trace fine-tuning mechanism that modifies downlink packet sizes to obscure real server transmission patterns. For each generated trace, `AirCloak` inspects downlink packets and uniformly resizes them toward the MTU. Fig. 6(b) illustrates the resulting traffic transformation. We evaluate the mechanism using two metrics: i) defense cost, packet resizing increases total transmitted bytes, with an average overhead of approximately 12%. ii) obfuscation effectiveness, measured by KL divergence between original and obfuscated traffic. After trace fine-tuning, KL divergence decreases from 0.156 to 0.098, a reduction of 37.2%, indicating a more concentrated distribution and reduced information leakage.

V. AirCloak Implementation

`AirCloak` establishes an encrypted tunnel between the mobile terminal and the AP, and performs traffic obfuscation before forwarding app traffic through the tunnel.

A. Encrypted Tunnel

As shown in Fig. 7, `AirCloak` establishes a UDP-based encrypted tunnel between the mobile terminal and the AP. The terminal-side middleware transforms uplink traffic, while the AP-side middleware processes downlink traffic.

1) Terminal-Side Middleware: The terminal-side middleware is implemented using Android's native framework, *VpnService* [30], which creates a virtual network interface to relay all traffic. App traffic is redirected through this interface, allowing `AirCloak` to capture raw IP packets via the *vpnInput* API for parsing and processing. For TCP traffic, `AirCloak` maintains per-connection state to support protocol-level operations such as handshake, retransmission, and connection termination (e.g., the handshake phase is labeled *SYN_SENT*, and the data transfer phase *ESTABLISHED*). Intercepted packets are then transformed through fragmentation, reassembly, padding, and forgery to match the generated trace, encapsulated in a UDP tunnel, and forwarded to the AP. Similarly, downlink packets from the AP follow the reverse path: they traverse the tunnel, are reconstructed at the terminal, and are finally delivered to the target app through the *vpnOutput* API.

Moreover, `AirCloak` offers customizable defense policies to accommodate diverse privacy requirements. It identifies apps by their unique $package_name$s and displays app information in a graphical interface (see Fig. 8), allowing users to selectively activate protection. Once authorized, traffic from selected apps is intercepted and obfuscated, while all other traffic bypasses the middleware and is forwarded directly to external networks. Implementation details are provided in § A-C.

2) AP-Side Middleware: The AP-side middleware is built on the Linux *iptables* and *Netfilter* frameworks. *iptables* rules redirect uplink obfuscated traffic to user space, where a *Netfilter* Queue buffers and processes the encapsulated packets. Each packet is then decapsulated, re-injected into the kernel, and forwarded via the standard routing path to the destination server. Similarly, response packets from remote servers are redirected to another *Netfilter* Queue through dedicated *iptables* rules. After processing, the AP encapsulates the transformed packets and sends them back to the terminal via the UDP tunnel.

B. Packet Obfuscation Operations

`AirCloak` implements four fundamental obfuscation operations (i.e., fragment, reassemble, pad, and forge) to modify packet size and direction, thereby masking side-channel patterns. To support correct reconstruction at the tunnel endpoint, each transformed packet is prepended with a 2-byte obfuscation header and encapsulated in a UDP packet. The header uses a fixed encoding format: the first 2 bits specify the operation type, and the remaining 14 bits encode operation-specific metadata.

1) Fragment: The original packet is split into two sub-packets, each carrying an obfuscation header. To enable correct reassembly, a globally unique fragment identifier is encoded in header bits 2–14, while bit 15 denotes the fragment's position within the pair (0 for the first fragment, 1 for the second).

2) Reassemble: `AirCloak` combines two consecutive packets into one, prepending an obfuscation header. To conceal their boundary, individual lengths are omitted from the payload; instead, the first packet's size is encoded in header bits 2–15.

3) Pad: `AirCloak` appends a random padding string to the original packet and prefixes the result with an obfuscation header. Header bits 2–15 encode the original packet length, ensuring accurate payload recovery during reconstruction.

4) Forge: `AirCloak` generates dummy packets independently of original traffic. In these packets, header bits 2–15 are filled with a randomly generated bit sequence.

VI. Evaluation

Our evaluation aims to answer four research questions:

- **RQ1:** How effective is `AirCloak` in defending against WAF attacks compared with existing defenses?
- **RQ2:** How effective is `AirCloak` in defending against WAF attacks compared with encryption proxies?
- **RQ3:** How well does `AirCloak` protect unknown apps?
- **RQ4:** What is the runtime overhead of `AirCloak`?

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

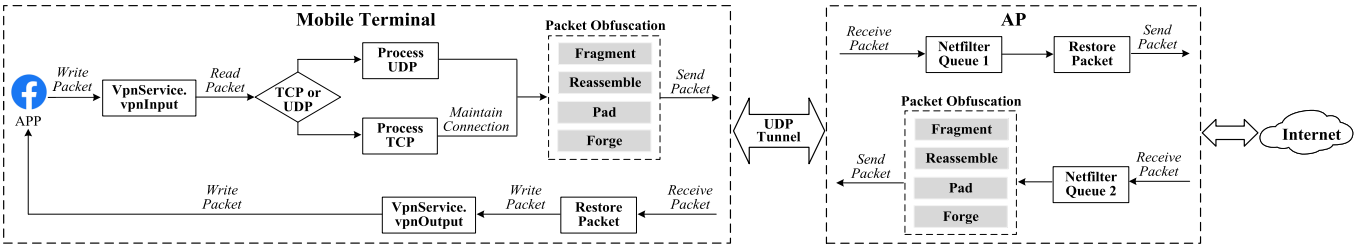
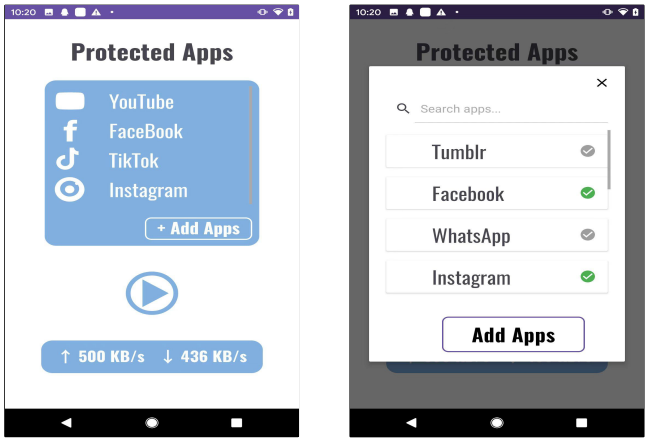


Fig. 7. Implementation of the encrypted tunnel between the mobile terminal and the AP.



(a) Main interface. (b) Custom defense.

Fig. 8. User interface of the terminal-side `AirCloak`.

A. Testbed and Datasets

To answer these questions, we conduct extensive evaluations of `AirCloak` using two datasets:

Android150 [9] is a public dataset containing traffic data from 150 of the most popular apps on the Google Play Store, such as Facebook, Instagram, and Airbnb.

AppTrace50 is collected using our AppTrace testbed and contains traffic from 50 popular Google Play apps spanning categories such as social media, streaming, news, music, and lifestyle. User interactions were simulated on Google Pixel 2 devices using the *Monkey* framework [32], and traffic was extracted by filtering packets based on each app's *package_name*.

B. App Fingerprinting Attacks

Existing WAF attacks exploit three types of traffic features: sequence structures, spatiotemporal features, and statistical features. We evaluate four representative WAF attacks covering all three types.

AppScanner [5] extracts bursts from TCP flows and computes statistical features for each burst, including the maximum, minimum, mean, standard deviation, variance, skewness, and quantiles of packet lengths.

Aceto et al. [33] extract features such as packet length, direction, and inter-arrival time, and employ a Markov model to capture their temporal dependencies.

AppSniffer [9] adopts a two-stage approach for app classification. First, it distinguishes normal traffic from VPN traffic based on the number of TCP flows. Then, it uses a stacked ensemble model built on LightGBM and FastAI to learn the spatiotemporal characteristics of packets.

MCS [2] analyzes bursts in TCP flows and extracts statistical features such as packet length, inter-arrival time, uplink/downlink packet ratios, and direction changes. It then combines the outputs of multiple classifiers, including Bayesian and Random Forest, to make the final decision.

Notably, we assume that only one foreground app is active and that WAF attacks are trained and tested on its clean traffic. This setting provides attackers with an idealized scenario. In practice, however, concurrent app execution significantly complicates fingerprint extraction [10]. Thus, strong defense performance under clean traffic conditions implies even greater robustness in mixed-app scenarios.

C. Baseline Defenses

We compare `AirCloak` with six traffic obfuscation-based defenses and three encryption proxies.

AppMutation [20] computes the packet length distribution of the source app and strategically pads packets to reshape it toward a target app's distribution.

Chaddad et al. [22] propose reshaping a source app's packet length distribution to mimic a target app. Unlike *AppMutation*, their method selects the optimal target through mathematical modeling to minimize transformation overhead.

Cactus [23] reshapes app traffic by leveraging inherent TCP mechanisms. It alters uplink traffic via chaff packet injection and packet fragmentation, and adjusts downlink traffic by modifying TCP window sizes.

Building on the encrypted tunnel established by `AirCloak`, we also compare several traffic generation methods.

Random randomly alters the length of each packet. If the new length is larger, padding is added; otherwise, the packet is split accordingly.

GAN [28] employs a generator to produce traffic traces from noise and a discriminator to distinguish between generated and real ones. Once trained, the generator is used to produce adversarial traffic traces for obfuscation. The generator includes a fully connected layer with 512 neurons, followed by two LSTM layers with 1,024 hidden units each, and an output projection layer. The discriminator consists of a hidden layer with 256 neurons and a single-neuron output layer.

Surakav [29] is built on a conditional WGAN to capture the burst-level distribution of real traffic and generate diverse obfuscation traces. During online deployment, Surakav employs the generated traces as transmission templates, integrating a tunable burst-interval alignment mechanism and stochastic inter-burst sampling. It further regulates dummy packet injection via a probabilistic skipping strategy, thereby

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

TABLE II
CONFIGURATION OF THREE ENCRYPTION PROXIES

Proxy	Encryption	Obfuscation	Encapsulation
Shadowsocks	chacha20	–	TCP
ShadowsocksR	chacha20	TLS	TCP
V2Ray	VMess	WebSocket	WebSocket

embedding real traffic into the learned generative distribution space.

Encryption proxies are generally classified into two types: i) lightweight proxies offering basic encryption and forwarding, and ii) highly covert proxies employing advanced traffic camouflage. We evaluate three representative proxies: the lightweight *Shadowsocks* [34], and the covert *ShadowsocksR* [35] and *V2Ray* [36]. Their configurations are detailed in TABLE II.

D. AirCloak Vs. Baseline Defenses

1) Experimental Setup: Since baseline defenses operate exclusively in offline mode, we evaluate both AirCloak and these defenses in an offline setting. Using the Android150 [9] and AppTrace50 datasets (split 7:3 for training and testing), we assess four WAF attacks: AppScanner [5], Aceto et al. [33], AppSniffer [9], and MCS [2]. We assume attackers can monitor traffic long-term and retrain their models on defended traces. Evaluation metrics include precision, recall, and F1-score of WAF attacks, as well as defense cost, defined as the relative increase in packet count and size compared to original traffic.

2) Results: As shown in TABLE III, AirCloak outperforms all baseline defenses, reducing the average F1-score of four WAF attacks from 0.77 to 0.19. Without defense, the average F1-scores of the four WAF attacks on the Android150 and AppTrace50 datasets are 0.82 and 0.71, respectively. With AppMutation defense, the F1-scores are 0.76 and 0.56; with Chaddad et al., the F1-scores are 0.78 and 0.73; and with Cactus, the F1-scores are 0.75 and 0.63. These baselines remain limited because they cannot flexibly transform bidirectional traffic. AppMutation and Chaddad et al. reshape packet length distributions to mimic those of other apps; however, attackers can simply retrain on defended traces to circumvent them. Cactus obfuscates bidirectional traffic via TCP header modifications, yet its changes are constrained ($< 10\%$) to avoid network delays, which substantially limits its effectiveness.

Answer to RQ1. AirCloak demonstrates strong efficacy in mitigating WAF attacks, surpassing all six traffic obfuscation-based defenses. It reduces the average F1-score of WAF attacks from 0.77 to 0.19.

In contrast, defenses combined with our encrypted tunnel, such as Random, GAN, and Surakav, exhibit significantly improved effectiveness. On the Android150 and AppTrace50 datasets, Random reduces the average F1-scores of four attacks by 43.90% and 49.30%, respectively; GAN reduces them by 41.46% and 57.75%, and Surakav reduces them by 65.85% and 61.97%. These gains, however, come at the cost of substantially higher overhead. For example, on the AppTrace50 dataset, GAN increases the packet number by 289% and packet size by 178%, whereas Surakav increases them by 298% and 100%, respectively. By comparison, AirCloak achieves the highest defense effectiveness with lower overhead, reducing F1-scores by 78.05% and 71.83% on the Android150 and AppTrace50 datasets, respectively, outperforming all baseline defenses.

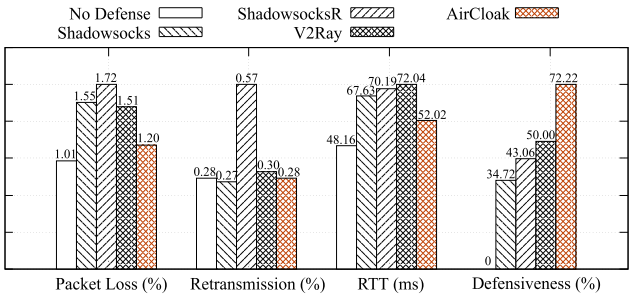


Fig. 9. Comparison of AirCloak and encryption proxy-based defenses in packet loss, retransmission, RTT, and defensiveness.

E. AirCloak Vs. Encryption Proxies

1) Experimental Setup: We use Google Pixel 2 smartphones as mobile terminals and a Raspberry Pi as the AP. The online evaluation is conducted on our AppTrace testbed, comparing AirCloak with three encryption proxies: Shadowsocks, ShadowsocksR, and V2Ray. Each proxy client is deployed on a Google Pixel 2 smartphone, with the proxy server hosted on the same cloud server. The apps used in this experiment are consistent with those in the AppTrace50 dataset.

Answer to RQ2. AirCloak outperforms all three encryption proxy–based defenses, reducing the average F1-score of WAF attacks from 0.71 to 0.20 while incurring minimal impact on app network communication.

2) Results: As shown in TABLE IV, AirCloak outperforms the three encryption proxy-based defenses in effectiveness. It reduces the average F1-score of WAF attacks by 71.83% while imposing minimal impact on smartphone network services. In the absence of defense, the four WAF attacks yield an average F1-score of 0.71. Deploying the three encryption proxies reduces attack effectiveness to varying degrees: Shadowsocks lowers the average F1-score by 33.80%, while ShadowsocksR and V2Ray achieve reductions of 42.25% and 49.30%, respectively. This limitation stems from the fact that encryption proxies prioritize traffic camouflage over genuine obfuscation. For instance, V2Ray encapsulates traffic within WebSocket to mimic normal behavior; however, it does not modify the underlying traffic structure, thereby limiting its defensive effectiveness. In contrast, AirCloak transforms traffic into diverse generated traces, achieving robust obfuscation and reducing the F1-score by 71.83%.

AirCloak provides strong defense while exerting minimal overhead on app network performance. As shown in Fig. 9, without defense, packet loss and retransmission rates are 1.01% and 0.28%, respectively, with an average RTT of 48.16 ms. In contrast, encryption proxies significantly increase network overhead: packet loss rate rises by 50%

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

TABLE III
EVALUATING AIRCLOAK AND BASELINE DEFENSES

Dataset	Attacks	AppScanner	Aceto et al.	AppSniffer	MCS	Defense Cost (Pkt Num, Size)	Defense Effectiveness
	Metric	Pre/Rec/F1	Pre/Rec/F1	Pre/Rec/F1	Pre/Rec/F1		
Android150	No Defense	0.91/0.78/0.84	0.84/0.74/0.76	0.87/0.88/0.87	0.82/0.82/0.82	0%, 0%	/
	AppMutation	0.92/0.71/0.80	0.80/0.75/0.76	0.67/0.71/0.64	0.84/0.83/0.83	100%, 100%	↓7.32%
	Chaddad et al.	0.98/0.84/0.90	0.83/0.79/0.80	0.45/0.54/0.43	0.97/0.97/0.97	33%, 2%	↓4.88%
	Cactus	0.93/0.76/0.83	0.76/0.63/0.67	0.60/0.70/0.61	0.88/0.88/0.88	36%, 5%	↓8.54%
	Random	0.75/0.35/0.46	0.61/0.60/0.59	0.40/0.44/0.27	0.51/0.51/0.51	91%, 9%	↓43.90%
	GAN	0.80/0.42/0.54	0.52/0.54/0.52	0.17/0.41/0.24	0.61/0.60/0.60	240%, 55%	↓41.46%
	Surakav	0.57/0.17/0.25	0.38/0.29/0.32	0.39/0.42/0.25	0.28/0.28/0.28	211%, 86%	↓65.85%
	AirCloak	**0.49/0.08/0.13**	**0.27/0.16/0.14**	**0.26/0.35/0.26**	**0.19/0.18/0.18**	**47%, 48%**	**↓78.05%**
AppTrace50	No Defense	0.75/0.65/0.68	0.76/0.62/0.67	0.78/0.78/0.77	0.73/0.72/0.72	0%, 0%	/
	AppMutation	0.65/0.45/0.52	0.65/0.54/0.58	0.59/0.64/0.60	0.54/0.54/0.53	103%, 112%	↓21.13%
	Chaddad et al.	0.97/0.78/0.86	0.78/0.70/0.70	0.49/0.54/0.40	0.97/0.96/0.96	63%, 7%	↑2.82%
	Cactus	0.84/0.59/0.68	0.66/0.49/0.56	0.49/0.60/0.49	0.79/0.79/0.78	21%, 6%	↓11.27%
	Random	0.62/0.29/0.38	0.36/0.46/0.37	0.25/0.50/0.33	0.37/0.39/0.37	50%, 44%	↓49.30%
	GAN	0.38/0.22/0.27	0.39/0.26/0.31	0.28/0.53/0.37	0.22/0.25/0.23	289%, 178%	↓57.75%
	Surakav	0.33/0.19/0.23	0.39/0.18/0.24	0.30/0.50/0.37	0.22/0.24/0.22	298%, 100%	↓61.97%
	AirCloak	**0.36/0.09/0.14**	**0.35/0.15/0.23**	**0.28/0.38/0.31**	**0.09/0.10/0.09**	**127%, 68%**	**↓71.83%**

TABLE IV
EVALUATING AIRCLOAK AND ENCRYPTION PROXIES

Attacks	AppScanner	Aceto et al.	AppSniffer	MCS	Packet Loss/ Retransmission	RTT (ms)	Defense Effectiveness
Metric	Pre/Rec/F1	Pre/Rec/F1	Pre/Rec/F1	Pre/Rec/F1			
No Defense	0.75/0.65/0.68	0.76/0.62/0.67	0.78/0.78/0.77	0.73/0.72/0.72	1.01% / 0.28%	48.16	/
Shadowsocks	0.55/0.21/0.30	0.64/0.54/0.56	0.64/0.64/0.62	0.39/0.41/0.40	1.55% / 0.27%	67.63	↓33.80%
ShadowsocksR	0.54/0.28/0.35	0.45/0.36/0.37	0.48/0.53/0.49	0.40/0.42/0.41	1.72% / 0.57%	70.19	↓42.25%
V2Ray	0.53/0.21/0.28	0.49/0.36/0.39	0.42/0.48/0.43	0.35/0.35/0.35	1.51% / 0.55%	72.04	↓49.30%
AirCloak	**0.38/0.09/0.13**	**0.35/0.15/0.23**	**0.26/0.34/0.29**	**0.15/0.15/0.15**	**1.20% / 0.28%**	52.02	**↓71.83%**

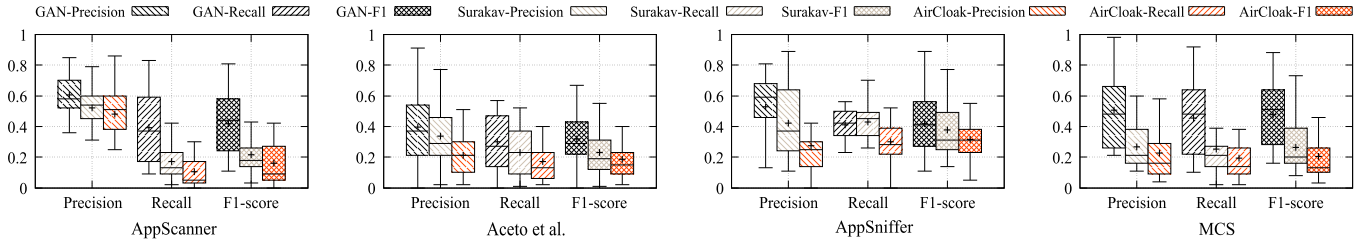


Fig. 10. Evaluating AirCloak and other defenses in the open world scenario.

to 70%, retransmission rate increases by up to 103%, and RTT increases to 67.63 ms, 70.19 ms and 72.04 ms, respectively, noticeably impacting user experience. By comparison, AirCloak has negligible effect on retransmission, introducing only an 18% increase in packet loss and an 8% increase in RTT, which represents substantially lower overhead than encryption proxies.

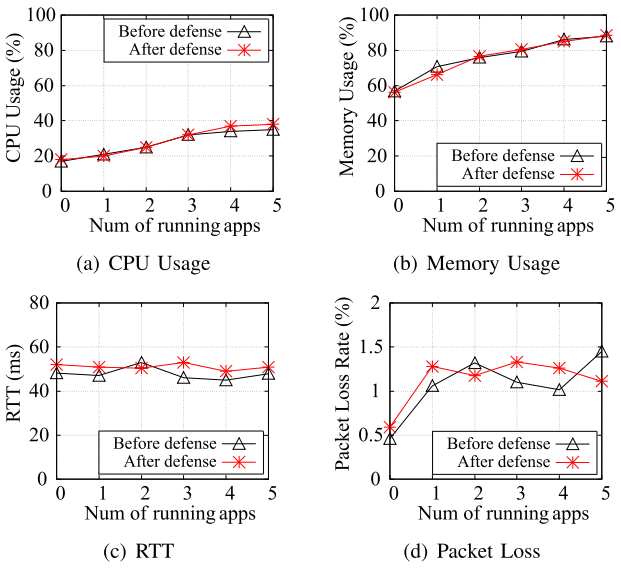
F. Open-World Evaluation

1) Experimental Setup: We evaluate AirCloak in protecting previously unseen apps by training on AppTrace50 and testing on Android150. AppMutation and Chaddad et al. rely on packet length distributions of known apps and are thus inapplicable in open-world settings. Cactus and Random apply generic random modifications and exhibit no meaningful difference between known and unknown apps, so they are also excluded. Accordingly, we compare AirCloak with GAN and Surakav. GAN trains a generative model on training apps and applies it uniformly to all test apps. Surakav uses a conditional generator to synthesize app-specific traffic. In contrast, AirCloak clusters training apps and trains a dedicated generator per cluster, assigning test apps to the appropriate cluster.

Answer to RQ3. In the open-world setting, AirCloak effectively protects previously unknown apps, reducing the average F1-score of WAF attacks by 73.17%.

2) Results: Fig. 10 shows the defense effectiveness of GAN, Surakav, and AirCloak against four WAF attacks in the open-world scenario. AirCloak reduces the average F1-score of attacks by 73.17%, outperforming GAN (50%)

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.



(a) CPU Usage (b) Memory Usage (c) RTT (d) Packet Loss

Fig. 11. System overhead of AirCloak on the terminal.

and Surakav (65.85%). Meanwhile, it increases defense cost (packet size) by only 38%, which is significantly lower than that of GAN (276%) and Surakav (197%). This performance gap is mainly due to GAN and Surakav generating traces for individual apps, leading to substantial mismatches when defending against unseen apps. In contrast, AirCloak generates matching traces at the cluster level and further refines them through three mechanisms: early sending, gap filling, and trace fine-tuning.

G. Runtime Overhead

1) Experimental Setup: We deploy the AirCloak client on a Google Pixel 2 smartphone and assess its impact on device performance with and without the defense enabled. Evaluation metrics include CPU usage, memory usage, RTT, packet loss rate, and retransmission rate. CPU and memory usage are collected using the Android debugging tool, *dumpsys* [37], specifically via the *cpuinfo* and *meminfo* commands. RTT, packet loss rate, and retransmission rate are obtained through *tcpdump*. Additionally, we deploy the AP-side middleware on an Ubuntu 20.04 virtual machine and evaluate its computational overhead. We measure CPU usage, memory usage, and I/O load, which are obtained using *pidstat*, *top*, and *iostat*, respectively.

2) Results: As shown in Fig. 11, AirCloak imposes negligible overhead on the terminal. When no apps are running, the system remains idle with no extra CPU cost. Fig. 11(a) shows that CPU usage scales proportionally with the number of running apps. With five apps active concurrently, AirCloak raises CPU usage only slightly from 35% to 38%. As shown in Fig. 11(b), AirCloak has virtually no impact on memory usage, owing to the lightweight traffic interception and processing mechanisms provided by Android's *VpnService* framework. RTT results in Fig. 11(c) show a modest average latency increase of approximately 3 ms, mainly due to traffic interception and transformation at the virtual interface before forwarding to the physical interface. Finally, Fig. 11(d) demonstrates that the packet loss rate increases by only approximately 5%, with minimal influence on packet retransmissions.

Answer to RQ4. AirCloak incurs minimal overheadon the terminal, increasing CPU utilization by 3%,RTT by 3 ms, and the packetloss rate by approximately 5%. Its impact on memory consumption and packetre transmission rate is negligible.

AirCloak also introduces minimal runtime overhead on the AP. When the AP is idle, CPU (user-space) utilization and memory usage are 5.4% and 22.7%, respectively. With one terminal running the defense middleware connected to the AP, CPU utilization increases to 7.9%, while memory usage rises to 23.4%. When five devices are connected, CPU utilization reaches 11.8% and memory usage increases to 25.6%. The I/O load remains stable across all evaluated conditions.

VII. Discussion

A. Broader Protection Scope

WAF attacks not only compromise user privacy but may also threaten physical safety. By analyzing traffic from IoT control apps, attackers can infer the types and operational states of IoT devices (e.g., a door lock set to "away mode"), thereby enabling targeted intrusions [38], [39], [40], [41], [42]. Fortunately, AirCloak effectively mitigates such threats. In § B-B, we evaluated 20 representative IoT devices (e.g., smart plugs, cameras, speakers, and sensors) along with their corresponding control apps. For each app, we generated real interaction traffic by manually simulating user operations. Using the collected traffic, we assessed the performance of WAF attacks before and after deploying AirCloak. The results demonstrate that AirCloak significantly hinders IoT device identification based on app traffic, reducing the average F1-score of WAF attacks to the level of random guessing.

B. More Sophisticated Attacks

While WAF attacks are widely adopted due to their ease of execution, advanced attackers may compromise the AP to analyze TCP traffic. AirCloak remains resilient against such attacks, as its AP-side middleware can also be deployed on cloud servers. In this mode, uplink traffic is tunneled to the cloud for forwarding, so a compromised AP only exposes obfuscated traffic. Cloud deployment, however, incurs additional latency and resource consumption (see VI-E). In practice, users can flexibly select the deployment mode according to resource and privacy needs.

C. Compatible Platforms

Currently, the terminal-side middleware of AirCloak is implemented for Android devices, while the AP-side middleware operates on Linux systems. We intend to extend AirCloak to other OSs, such as iOS and Windows, to facilitate broader deployment. On iOS, the Network Extension framework provides components such as *NEPacket Tunnel Provider* [43] and related tunnel management modules. This framework is functionally analogous to *VpnService*, as both enable user-space interception, processing, and forwarding of IP traffic. On Windows, equivalent functionality can be realized via mechanisms such as *WinDivert* [44] or the Windows Filtering Platform.

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

VIII. Related Work

We review the research progress on app fingerprinting attacks and discuss existing defenses.

A. App Fingerprinting Attacks

App fingerprinting attacks typically rely on four types of traffic features, including i) IP addresses, ii) sequence structure, iii) spatiotemporal features, and iv) statistical features of packets.

IP addresses [8], [45], [46]. Van Ede et al. [8] proposed FlowPrint, which automatically identifies temporal correlations among destination-related features in network traffic and uses these correlations to generate fingerprints. Pham et al. [45] treated IP addresses and port numbers as graph nodes, with communication correlations forming the edges, and applied deep graph convolutional networks to model app behaviors.

Sequence structure [1], [10], [33], [47], [48]. Li et al. [10] segmented traffic based on short-term packet patterns and extracted structural features linked to specific apps. Aceto et al. [33] constructed a Markov model using packet length, direction, and inter-packet intervals to capture sequence features to support both packet-level and message-level app identification.

Spatiotemporal features [3], [9], [49], [50], [51]. Oh et al. [9] proposed a two-stage identification model for VPN traffic: first identifying VPN traffic, and then using LightGBM and FastAI to learn spatiotemporal dependencies in packet sequences.

Statistical features [2], [4], [5], [6], [31], [52], [53], [54], [55]. Taylor et al. [5] designed AppScanner, which classifies app fingerprints by extracting statistical features from TCP flow bursts, while Aceto et al. [2] further improved classification performance using a multiclass system based on burst-level statistics.

B. Existing Defenses

App fingerprinting defenses typically rely on in-app traffic obfuscation and encryption proxies.

In-App traffic obfuscation defenses [20], [21], [22], [23], [56], [57], [58], [59], [60], [61] modify traffic by padding or splitting packets, inserting dummy packets, or altering packet length distributions. Although theoretically effective at disrupting traffic features, these approaches face deployment challenges: they require customized modifications to each app's client and server; otherwise, the transformed traffic cannot be correctly parsed. `AirCloak` addresses these issues by establishing an encrypted tunnel between the AP and the mobile terminal, thereby enabling flexible traffic modification without disrupting normal communication.

Encryption proxy-based defenses [25], [26], [27] set up a proxy server between the client and the destination server to forward traffic in both directions, thereby offering protection. However, these solutions have notable drawbacks. All traffic must pass through the proxy server, introducing unavoidable latency and computational overhead. Moreover, forwarding based solely on domain names or IP addresses lacks the granularity required for precise per-app protection. In contrast, `AirCloak` performs traffic transformation within the LAN, ensuring low and controllable defense overhead. Leveraging Android's `VpnService`, `AirCloak` enables fine-grained, user-level traffic interception and supports customizable defense strategies.

IX. Conclusion

This paper presents `AirCloak`, an edge-side defense middleware against WAF attacks. `AirCloak` operates transparently to apps, enabling rapid and scalable deployment across diverse app ecosystems. It employs a harmonization shaping method to adaptively reshape heterogeneous app traffic, thereby achieving efficient and robust protection. Moreover, `AirCloak` features a lightweight design and supports a one-click installable app on mobile terminals. Extensive evaluations on public datasets and real-world networks demonstrate that `AirCloak` effectively mitigates various WAF attacks and outperforms existing defenses. As the first edge-side defense middleware designed for large-scale deployment, `AirCloak` lays the foundation for a more secure app ecosystem.

Appendix A
Details of Models and Algorithms

A. HM Clustering

The HM clustering process is illustrated in Algorithm 2.

Algorithm 2 HM Clustering

Input: S: a set of HMs, k: desired number of clusters, ϖ: threshold for small clusters, ζ: maximum number of iterations;
Output: a set of clusters $\{C_1, C_2, ..., C_m\}$;

$\{\mu_1^0, \mu_2^0, \ldots, \mu_k^0\} \leftarrow$ randomly select k HMs from S;
$t \leftarrow 0$;
for $t < \zeta$ **do**
  for *each* $HM \in S$ **do**
    $i^* \leftarrow \arg\min_{1 \le i \le k} d(HM, \mu_i^t)$;
    assign HM to $C_{i^*}^t$;
  end
  for *each* $i \in \{1, 2, ..., k\}$ **do**
    if $C_i^t \neq \emptyset$ **then**
      $\mu_i^{t+1} \leftarrow \frac{1}{|C_i^t|} \sum_{x \in C_i^t} x$;
    else
      $\mu_i^{t+1} \leftarrow \mu_i^t$;
    end
  end
  $t \leftarrow t + 1$;
end
for *each* $i \in \{1, 2, ..., k\}$ **do**
  if $|C_i^t| < \varpi$ **then**
    $j^* \leftarrow \underset{\substack{1 \le j \le k \\ |C_j^t| \ge \text{min_size}}}{\arg\min} d(\mu_i^t, \mu_j^t)$;;
    merge c_i^t into $C_{j^*}^t$;
  end
end

B. Design Details of cGAN Model

We detail the architecture and parameter configuration of the generator, discriminator, and observer in the cGAN model.

The **generator** is a multi-layer LSTM network that takes random noise z and the target app label c as input, and outputs a synthetic burst trace t of length l. As shown in Fig. 12, it consists of three blocks. Block 1 concatenates z and c, then projects them into a 512-dimensional latent space via a fully connected layer followed by LeakyReLU. Block 2 is a two-layer LSTM that models sequential packet dependencies, with hidden dimensions of 1024 and 2048, respectively. Block 3 applies a fully connected layer with Tanh activation to constrain outputs to $[-1, 1]$, producing the final traffic trace t.

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

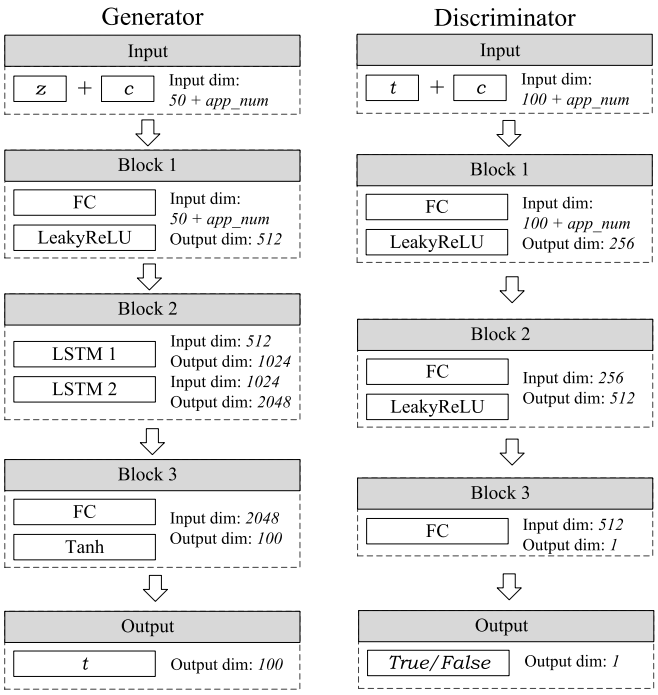


Fig. 12. The architecture of our cGAN model.

The **discriminator** is an MLP that takes a traffic trace t and its label c as input and outputs a binary decision (real or fake). As shown in Fig. 12, it comprises three blocks. Block 1 concatenates t and c, and applies a fully connected layer with LeakyReLU, mapping the input of dimension l plus label size to 256. Block 2 uses the same structure, increasing the output dimension to 512. Block 3 is a fully connected layer with a single output neuron for real/fake prediction.

The **observer** is a pre-trained MLP that takes a traffic trace t as input and predicts the corresponding app label c. Its architecture mirrors the discriminator, differing only in the output layer. It comprises three blocks: Blocks 1 and 2 each consist of a fully connected layer with LeakyReLU activation, producing 512- and 256-dimensional features, respectively. Block 3 is a fully connected layer whose output dimension equals the label size, yielding the predicted label c.

C. Customizable Defense Configuration

Unlike global defense mechanisms or IP/domain-based filtering, `AirCloak` supports customizable app-level protection, allowing users to selectively protect specific apps while leaving other traffic unaffected. As shown in Fig. 13, this functionality is implemented in three steps. First, `AirCloak` queries the *package manager* to retrieve the names and package identifiers of installed apps and displays them to the user through the user interface. Next, users select the apps to be protected and authorize `AirCloak` access to their traffic, which triggers an update of the defense rules. Finally, once activated, `AirCloak` captures packets in real time and identifies their source app: packets generated by protected apps are redirected to a virtual network interface for transformation before forwarding, whereas those from unprotected apps bypass this process and are transmitted directly through the physical interface to the external network.

```
/* Retrieve the names and corresponding package names of installed apps */
List<Pair<String, String>> getInstalledApps() {
    List<Pair<String, String>> appList = new ArrayList<>();
    for (ApplicationInfo appInfo : getPackageManager().getApplications(0)) {
        String appName = appInfo.loadLabel(getPackageManager()).toString();
        String packageName = appInfo.packageName;
        appList.add(new Pair<>(appName, packageName));
    }
    return appList;
}

/* UI action, user grants authorization to protect an app */
List<String> selectAppsUI() {
    List<Pair<String, String>> availableApps = getInstalledApps();  // retrieve app list
    List<String> selectedPackages = showAppSelectionUI(availableApps);  // user authorization
    return selectedPackages;
}
```

(a) Step 1: The user authorizes `AirCloak` access to the target app's traffic.

```
/*  Update defense rules */
AppDefenseConfig updateDefense(List<String> selectedPackages, boolean FLAG) {
    AppDefenseConfig vNT = new AppDefenseConfig();
    if (FLAG) {
        vNT.selectedApps = selectedPackages;    // user-authorized app list
        Map<String, Boolean> defenseRules = new HashMap<>();
        for (String app : selectedPackages) {
            defenseRules.put(app, true);    // add authorized app to defense rules
        }
        vNT.defenseRules = defenseRules;    // update defense rules
    } else {
        return defaultConfig();
    }
    return vNT;
}
```

(b) Step 2: `AirCloak` updates the defense rules.

```
/*  Run defense and process traffic */
VpnService runDefense() {
    List<String> vUserApps = selectAppsUI();    // authorized apps
    AppDefenseConfig vTmp = updateDefense(vUserApps, true);
    Map<String, Boolean> vConfig = vTmp.defenseRules;   // retrive defense rules
    VpnService vpnService = startVpnService();
    processTraffic(vConfig, vpnService);
    return vpnService;
}

void processTraffic(Map<String, Boolean> defenseRules, VpnService vpnService) {
    while (true) {
        Packet packet = vpnService.readPacket();
        if (packet == null) continue;
        String vPkg = getPackageName(packet);
        if (defenseRules.containsKey(vPkg) && defenseRules.get(vPkg)) {
            redirectTraffic(packet, vpnService);    // intercept and redirect
        } else {
            forwardTraffic(packet, vpnService);    // normal forward
        }
    }
}
```

(c) Step 3: `AirCloak` activates the defense.

Fig. 13. Scripts for customizable defense configuration.

APPENDIX B
SUPPLEMENTARY MATERIAL OF EVALUATIONS

A. Parameter Settings for ATAM

We systematically evaluated the influence of two key *ATAM* parameters, namely the time slot length t and the number of consecutive time slots N, on app traffic characterization. By varying t and N, we extracted traffic features under different temporal configurations and constructed the corresponding *ATAM* matrices. The resulting matrix samples were then split into training and testing sets at a 7:3 ratio and fed into the observer module of the cGAN model for performance evaluation.

TABLE V presents the impact of different combinations of t and N on app traffic characterization, with the classification accuracy of the observer used as the evaluation metric. The results show that when the time slot is too short, the limited number of packets captured per slot, or even missing packets in some slots, negatively affects the observer's classification

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

TABLE V
IMPACT OF ATAM PARAMETERS ON CHARACTERIZATION

t (ms)	*N*	Precision/Recall/F1-score
10	10	0.25/0.12/0.10
	20	0.33/0.23/0.22
	50	0.29/0.25/0.22
	100	0.27/0.26/0.23
20	10	0.47/0.34/0.33
	20	0.55/0.46/0.44
	50	0.60/0.48/0.45
	100	0.66/0.46/0.54
50	10	0.70/0.63/0.62
	20	0.79/0.75/0.74
	50	0.81/0.81/0.81
	100	**0.83/0.83/0.82**
100	10	0.79/0.68/0.67
	20	0.85/0.71/0.79
	50	0.81/0.71/0.79
	100	0.78/0.80/0.78

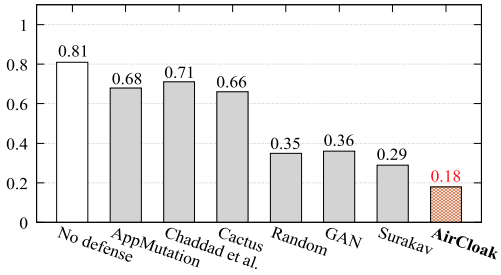


Fig. 14. The average F1-score of WAF attacks in IoT scenarios.

performance. Conversely, overly long time slots may obscure burst characteristics in the traffic and reduce the granularity of information representation. Based on these factors, we selected the parameter configuration that best balances observer performance: a time slot length t of 50 ms and the number of consecutive time slots N of 100.

B. Assisting in Protecting IoT Devices

Beyond exposing app usage and user privacy, WAF attacks may threaten physical safety by revealing IoT device types and operational states. Fortunately, `AirCloak` also provides effective protection for IoT devices. As shown in TABLE VI, we evaluated 20 types of IoT devices, including plugs, cameras, speakers, lamps and sensors. To simulate real-world usage, we controlled these devices through their companion apps and manually performed typical user operations. As shown in Fig. 14, the WAF attack achieves an average F1-score of 0.81 without defense. Existing defenses, including AppMutation, Chaddad et al., and Cactus, remain ineffective, particularly when attackers retrain their classifiers on defended traffic. In contrast, `AirCloak` outperforms all other methods, reducing the average F1-score of WAF attacks from 0.81 to 0.18.

C. Ablation Study

1) Experimental Setup: To quantify the independent contribution of each core module to `AirCloak`'s overall performance, we conduct an ablation study by isolating the clustering, optimization, and adversarial training modules.

TABLE VI
IOT DEVICES AND THEIR CONTROL APPS

App Name	IoT Device	Function
Mijia	Mijia Plug	Plug
EBOX	Ebo Robot	Camera
Huawei Smart Life	Huawei Speaker	Speaker
Xiaodu	DuSmart Speaker	Speaker
Lenovo Smart Living	Lenovo Camera	Camera
XiaoJingYu	Hemu Plug	Plug
Smart Life	CO_2 Detector	Sensor
Xiaodu	King Kong Speaker	Speaker
Mijia	Water Leak Sensor	Sensor
Xiaodu	Smart Alarm Clock	Clock
Mijia	Xiaomi Camera	Camera
Tuya Smart Life	Tuya Temperature Detector	Sensor
EZVIZ Cloud Video	EZVIZ Camera	Camera
Mijia	AlmLbs Smart Plug	Plug
Tuya Smart Life	Tuya Door/Window Alarm	Sensor
Mijia	OPPLE Lamp	Lamp
OKOK	HiLink Smart Scale	Sensor
Huawei Smart Life	DALEN Smart Lamp	Lamp
Mijia	Linptech Human Sensor	Sensor
Huawei Smart Life	Huawei Smart Plug	Plug

TABLE VII
ABLATION STUDY RESULTS

Setting	Precision	Recall	F1-score
`AirCloak` (full)	0.25	0.18	0.19
A1	0.32	0.32	0.30
A2	0.76	0.69	0.71
B1	0.26	0.18	0.20
B2	0.28	0.25	0.26
B3	0.55	0.42	0.47
C1	0.40	0.41	0.36
C2	0.31	0.28	0.27

The experiments are conducted on the *AppTrace50* dataset. `AirCloak` (full) denotes the complete system. For the clustering module, we consider two settings: *A1*: a single cluster containing all apps; *A2*: each app forms its own cluster. For the optimization module, we evaluate three configurations: *B1*: disabling early sending; *B2*: disabling gap filling; *B3*: disabling trace fine-tuning. For the adversarial training module, we examine two variants: *C1*: replacing the cGAN with arbitrary perturbations; *C2*: replacing the cGAN with a standard GAN.

2) Results: The modules contribute both individually and jointly to defense effectiveness. As shown in TABLE VII, the full `AirCloak` achieves the best performance, reducing the average F1-score of four WAF attacks to 0.19. Using a single cluster (A1) increases the F1-score to 0.30, as excessive aggregation introduces substantial inter-app traffic heterogeneity. Conversely, modeling each app individually (A2) raises the F1-score to 0.71, suggesting that over-segmentation preserves app-specific structural patterns exploitable by attackers. For the optimization module, disabling Early Sending (B1) has only a marginal impact (F1-score = 0.20), implying that it primarily serves runtime overhead control rather than core obfuscation. In contrast, disabling Gap Filling (B2) and Trace Fine-tuning (B3) increases the F1-score to 0.26 and 0.47, respectively, demonstrating that these mechanisms effectively mitigate statistical feature leakage. For the adversarial training module, replacing the cGAN with random perturbation (C1) or

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

a standard GAN (C2) increases the F1-score to 0.36 and 0.27, respectively. This result indicates that naive perturbation or unconditional generation fails to construct stable and semantically consistent anonymous traffic distributions, leaving them vulnerable to being relearned by the attacker.

REFERENCES

[1] J. Li et al., "FOAP: Fine-grained open-world Android app fingerprinting," in *Proc. USENIX Secur.*, 2022, pp. 1579–1596.

[2] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescape, "Traffic classification of mobile apps through multi-classification," in *Proc. GLOBECOM - IEEE Global Commun. Conf.*, Dec. 2017, pp. 1–6.

[3] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning," in *Proc. Netw. Traffic Meas. Anal. Conf. (TMA)*, Jun. 2018, pp. 1–8.

[4] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on Android apps via traffic analysis," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2015, pp. 297–304.

[5] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "AppScanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Mar. 2016, pp. 439–454.

[6] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 1, pp. 63–78, Jan. 2018.

[7] M. Tian, P. Chang, Y. Sang, Y. Zhang, and S. Li, "Mobile application identification over HTTPS traffic based on multi-view features," in *Proc. 26th Int. Conf. Telecommun. (ICT)*, Apr. 2019, pp. 73–79.

[8] T. van Ede et al., "FlowPrint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[9] S. Oh, M. Lee, H. Lee, E. Bertino, and H. Kim, "AppSniffer: Towards robust mobile app fingerprinting against VPN," in *Proc. ACM Web Conf.*, Apr. 2023, pp. 2318–2328.

[10] J. Li et al., "Packet-level open-world app fingerprinting on wireless traffic," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022, pp. 1–18.

[11] J. Li et al., "Robust app fingerprinting over the air," *IEEE/ACM Trans. Netw.*, vol. 32, no. 6, pp. 5065–5080, Dec. 2024.

[12] X. Yang, Y. Huang, J. Guo, D. Zhang, and Q. Wang, "Eavesdropping mobile apps and actions through wireless traffic in the open world," in *Proc. Springer ICIC*, 2024, pp. 3–15.

[13] T. Ni et al., "Uncovering user interactions on smartphones via contactless wireless charging side channels," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 3399–3415.

[14] T. Ni, G. Lan, J. Wang, Q. Zhao, and W. Xu, "Eavesdropping mobile app activity via radio-frequency energy harvesting," in *Proc. USENIX Secur.*, 2023, pp. 3511–3528.

[15] J. Baek et al., "Targeted privacy attacks by fingerprinting mobile apps in LTE radio layer," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2023, pp. 261–273.

[16] J. Varmarken, J. Al Aaraj, R. Trimananda, and A. Markopoulou, "FingerprinTV: Fingerprinting smart TV apps," *Proc. Privacy Enhancing Technol.*, vol. 2022, no. 3, pp. 606–629, Jul. 2022.

[17] M. Shen et al., "Machine learning-powered encrypted network traffic analysis: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 25, no. 1, pp. 791–824, 1st Quart., 2023.

[18] D. Y. Huang. (2025). *RouterSense Passively Tracks Digital Behaviors to Detect Potential Health Conditions Without Wearables or Surveys*. [Online]. Available: https://engineering.nyu.edu/news/nyu-tandon-researchers-adapt-cybersecurity-tool-monitor-health-through-smartphone-traffic

[19] C. Osborne. (2020). *Smart Locks Opened With Nothing More Than a MAC Address*. [Online]. Available: https://www.zdnet.com/article/smart-locks-opened-with-nothing-more-than-a-mac-address

[20] L. Chaddad, A. Chehab, I. H. Elhajj, and A. Kayssi, "App traffic mutation: Toward defending against mobile statistical traffic analysis," in *Proc. IEEE INFOCOM Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2018, pp. 27–32.

[21] L. Chaddad, A. Chehab, I. H. Elhajj, and A. Kayssi, "Mobile traffic anonymization through probabilistic distribution," in *Proc. 22nd Conf. Innov. Clouds, Internet Netw. Workshops (ICIN)*, Feb. 2019, pp. 242–248.

[22] L. Chaddad, A. Chehab, I. H. Elhajj, and A. Kayssi, "Optimal packet camouflage against traffic analysis," *ACM Trans. Privacy Secur.*, vol. 24, no. 3, pp. 1–23, Aug. 2021.

[23] R. Xie et al., "Cactus: Obfuscating bidirectional encrypted TCP traffic at client side," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 7659–7673, 2024.

[24] (2025). *Mobile App Download Statistics & Usage Statistics (2025)*. [Online]. Available: https://buildfire.com/app-statistics

[25] A. Alshehri, J. Granley, and C. Yue, "Attacking and protecting tunneled traffic of smart home devices," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2020, pp. 259–270.

[26] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart (ER) IoT traffic shaping," in *Proc. PoPETs*, 2019, pp. 128–148.

[27] Q. Li, K. Yu, D. Chen, M. Sha, and L. Cheng, "TrafficSpy: Disaggregating VPN-encrypted IoT network traffic for user privacy inference," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Oct. 2022, pp. 145–153.

[28] T. Hou, T. Wang, Z. Lu, Y. Liu, and Y. Sagduyu, "IoTGAN: GAN powered camouflage against machine learning based IoT device identification," in *Proc. IEEE Int. Symp. Dyn. Spectr. Access Netw. (DySPAN)*, Dec. 2021, pp. 280–287.

[29] J. Gong, W. Zhang, C. Zhang, and T. Wang, "Surakav: Generating realistic traces for a strong website fingerprinting defense," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 1558–1573.

[30] (2025). *Android VpnService*. [Online]. Available: https://developer.android.com/reference/android/net/VpnService

[31] L. Zhai, Z. Qiao, Z. Wang, and D. Wei, "Identify what you are doing: Smartphone apps fingerprinting on cellular network traffic," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Sep. 2021, pp. 1–7.

[32] (2025). *Monkey*. [Online]. Available: https://developer.android.com/studio/test/monkey

[33] G. Aceto, G. Bovenzi, D. Ciuonzo, A. Montieri, V. Persico, and A. Pescapé, "Characterization and prediction of mobile-app traffic using Markov modeling," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 907–925, Mar. 2021.

[34] (2025). *Shadowsocks*. [Online]. Available: https://shadowsocks.org/

[35] (2025). *ShadowsocksR*. [Online]. Available: https://github.com/ShadowsocksR-Live/ssrWin

[36] (2025). *V2Ray*. [Online]. Available: https://www.v2ray.com/

[37] (2025). *Dumpsys*. [Online]. Available: https://developer.android.com/tools/dumpsys

[38] L. Fan et al., "AutoIoT: Automatically updated IoT device identification with semi-supervised learning," *IEEE Trans. Mobile Comput.*, vol. 22, no. 10, pp. 5769–5786, Oct. 2023.

[39] A. Acar et al., "Peek-a-boo: I see your smart home activities, even encrypted!," in *Proc. 13th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, Jul. 2020, pp. 207–218.

[40] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi, "Information exposure from consumer IoT devices: A multidimensional, network-informed measurement approach," in *Proc. Internet Meas. Conf.*, Oct. 2019, pp. 267–279.

[41] S. Dong, Z. Li, D. Tang, J. Chen, M. Sun, and K. Zhang, "Your smart home can't keep a secret: Towards automated fingerprinting of IoT traffic," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 47–59.

[42] X. Liu, Y. Han, and Y. Du, "IoT device identification using directional packet length sequences and 1D-CNN," *Sensors*, vol. 22, no. 21, p. 8337, Oct. 2022.

[43] (2026). *NEPacketTunnelProvider*. [Online]. Available: https://developer.apple.com/documentation/networkextension/nepackettunnelprovider

[44] (2026). *WinDivert*. [Online]. Available: https://windivert.com/

[45] T.-D. Pham, T.-L. Ho, T. Truong-Huu, T.-D. Cao, and H.-L. Truong, "MAppGraph: Mobile-app classification on encrypted network traffic using deep graph convolution neural networks," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2021, pp. 1025–1038.

[46] H. Xu, S. Li, Z. Cheng, R. Qin, J. Xie, and P. Sun, "TrafficGCN: Mobile application encrypted traffic classification based on GCN," in *Proc. IEEE Global Commun. Conf.*, Dec. 2022, pp. 891–896.

[47] K. Shi, Y. Zeng, B. Ma, Z. Liu, and J. Ma, "MT-CNN: A classification method of encrypted traffic based on semi-supervised learning," in *Proc. IEEE Global Commun. Conf.*, Dec. 2023, pp. 7538–7543.

[48] R. Babaria, S. C. Madanapalli, H. Kumar, and V. Sivaraman, "FlowFormers: transformer-based models for real-time network flow classification," in *Proc. 17th Int. Conf. Mobility, Sens. Netw. (MSN)*, Dec. 2021, pp. 231–238.

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.

[49] M. Shen, J. Zhang, L. Zhu, K. Xu, and X. Du, "Accurate decentralized application identification via encrypted traffic analysis using graph neural networks," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 2367–2380, 2021.
[50] S. Rezaei, B. Kroencke, and X. Liu, "Large-scale mobile app identification using deep learning," *IEEE Access*, vol. 8, pp. 348–362, 2020.
[51] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescape, "Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 2, pp. 445–458, Jun. 2019.
[52] A. Bremler-Barr, H. Levy, and Z. Yakhini, "IoT or NoT: Identifying IoT devices in a short time scale," in *Proc. NOMS - IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2020, pp. 1–9.
[53] A. Hameed and A. Leivadeas, "IoT traffic multi-classification using network and statistical features in a smart environment," in *Proc. CAMAD*, Sep. 2020, pp. 1–7.
[54] R. Kolcun et al., "Revisiting IoT device identification," in *Proc. IFIP TMA*, 2021, pp. 1–9.
[55] N. Moustafa, B. Turnbull, and K.-K.-R. Choo, "An ensemble intrusion detection technique based on proposed statistical flow features for protecting network traffic of Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4815–4830, Jun. 2019.
[56] A. J. Pinheiro, J. M. Bezerra, and D. R. Campelo, "Packet padding for improving privacy in consumer IoT," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2018, pp. 00925–00929.
[57] A. J. Pinheiro, P. Freitas de Araujo-Filho, J. de M. Bezerra, and D. R. Campelo, "Adaptive packet padding approach for smart home networks: A tradeoff between privacy and performance," *IEEE Internet Things J.*, vol. 8, no. 5, pp. 3930–3938, Mar. 2021.
[58] S. Xiong, A. D. Sarwate, and N. B. Mandayam, "Network traffic shaping for enhancing privacy in IoT systems," *IEEE/ACM Trans. Netw.*, vol. 30, no. 3, pp. 1162–1177, Jun. 2022.
[59] B. Li, Y. Chen, L. Zhang, L. Wang, and Y. Cheng, "HomeSentinel: Intelligent anti-fingerprinting for IoT traffic in smart homes," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 4780–4793, 2024.
[60] A. Shenoi, P. K. Vairam, K. Sabharwal, J. Li, and D. M. Divakaran, "IPET: Privacy enhancing traffic perturbations for secure IoT communications," *Proc. Privacy Enhancing Technol.*, vol. 2023, no. 2, pp. 206–220, Apr. 2023.
[61] Y. Perera, N. Ahmed, S. Kanhere, W. Hu, and S. Jha, "IoT traffic obfuscation: Will it guarantee the privacy of your smart home?," in *Proc. IEEE Int. Conf. Commun.*, May 2022, pp. 2954–2959.



Huafeng Bian received the B.S. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2020, where he is currently pursuing the Ph.D. degree with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering. His research interests focus on wireless networks, the IoT security, and traffic fingerprinting defense.



Jianfeng Li received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University in 2018. He was a Post-Doctoral Fellow with The Hong Kong Polytechnic University from 2019 to 2022. He is an Assistant Professor with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University. His research interests have centered on traffic analysis, privacy of mobile platform, and large-scale cyber security.



Haodan Luo received the bachelor's degree in information security from Zhongnan University of Economics and Law in 2025. She is currently pursuing the master's degree with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China. Her research interests lie in cybersecurity and machine learning.



Yixuan Liu received the bachelor's degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2024, where she is currently pursuing the master's degree with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering. Her research interests lie in cybersecurity and machine learning.



Xiaobo Ma (Member, IEEE) received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2014. He was a Post-Doctoral Research Fellow with The Hong Kong Polytechnic University in 2015. He is currently a Professor with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Shaanxi Province Key Laboratory of Computer Network, Xi'an Jiaotong University. His research interests include internet measurement and cyber security. He is an Editorial Board Member of the IEEE Transactions on Dependable and Secure Computing and a Technical Program Committee (TPC) Member of the ACM Conference on Computer and Communications Security (CCS).



Zhenhua Li (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science and technology from Nanjing University in 2005 and 2008, respectively, and the Ph.D. degree in computer science and technology from Peking University in 2013. He is currently a Tenured Associate Professor with the School of Software, Tsinghua University. His research interests include network measurement, mobile networking/virtualization, and cloud computing/gaming. He is a Senior Member of ACM.



Jialong Zhang received the Ph.D. degree in computer engineering from Texas A&M University in 2016. He was a Research Scientist with IBM Research, ByteDance, and Tencent, from 2016 to 2024. He is currently a Professor with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Shaanxi Province Key Laboratory of Computer Network, Xi'an Jiaotong University. His research interests have centered on cyber security and AI security.



Jigang Wang received the Ph.D. degree in computer application technology from Harbin Engineering University in 2006. He is the Vice President of ZTE Corporation, the General Manager of the Network Security Product Line, and the Chair of the Operating System and Network Security Technology Committee. His research interests have centered on cyber security, operating systems, and artificial intelligence.



Wei Wang (Senior Member, IEEE) received the Ph.D. degree from Xi'an Jiaotong University, China, in 2006. He is currently a Full Professor with the Ministry of Education Key Laboratory for Intelligent Networks and Network Security, aka MOE KLINNS Laboratory, Xi'an Jiaotong University. He has authored or co-authored more than 100 peer-reviewed papers in various journals and international conferences. His recent research interests include social networks and data security. He is an Editorial Board Member of *Computers and Security* and a Young AE of *Frontiers of Computer Science*. He is a Highly Cited Chinese Researcher in Elsevier.

Authorized licensed use limited to: Tsinghua University. Downloaded on November 02,2026 at 17:59:31 UTC from IEEE Xplore. Restrictions apply.