LANShield: Analysing and Protecting Local Network Access on Mobile Devices

Angelos Beitis[*]
DistriNet, KU Leuven
angelos.beitis@kuleuven.be

Jeroen Robben[*]
DistriNet, KU Leuven
jeroen.robben@kuleuven.be

Alexander Matern
Technical University of Darmstadt
amatern@seemoo.de

Zhen Lei
Taiyuan University of Technology
leizhen0667@link.tyut.edu.cn

Yijia Li
Taiyuan University of Technology
2023520731@link.tyut.edu.cn

Nian Xue
Shandong University of Technology
xuenian@sdut.edu.cn

Yongle Chen
Taiyuan University of Technology
chenyongle@tyut.edu.cn

Vik Vanderlinden
DistriNet, KU Leuven
vik.vanderlinden@kuleuven.be

Mathy Vanhoef
DistriNet, KU Leuven
Mathy.Vanhoef@kuleuven.be

Abstract

Home and workplace networks typically safeguard against external threats but allow internal devices to communicate freely with each other. As a result, malicious code on an internal device can collect sensitive data about other devices or directly attack them.

In this paper, we study mobile apps as potential sources of local network attacks, analyse their behaviour, design new defences, and evaluate and bypass existing mitigations. We first focus on Android, where apps with only the Internet permission can access all devices in the Local Area Network (LAN), meaning malicious apps can extract private LAN data, manipulate discovery protocols to obtain a Machine-in-the-Middle (MitM) position, and directly attack devices. To defend against such mobile-based attacks, we define an access model to securely differentiate between LAN and global Internet access. We implement this model on Android by creating LANShield: an app that refines Android's permission model, and can monitor and block LAN access of apps using a virtual network interface. We use LANShield to manually perform tests of 399 Android apps and find, among other observations, that 89 apps unexpectedly access the LAN, and 93 apps scan the network. In contrast to Android, iOS already separates the local and global Internet, but does so based on a proprietary LAN access model. We compare this access model to ours, and present multiple bypasses for an app to circumvent Apple's local network permission. Finally, we reported all our findings to affected vendors, and hope our work will motivate the adoption of stronger permission models on mobile devices.

Keywords

LANShield, mobile app permissions, app firewall, LAN security

[*]The first two authors contributed equally.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–19
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Local home networks are generally assumed to be secure, enclosed networks that are safe from malicious actors without physical presence. As a result, devices in local networks are often unprotected against access from other devices in the same network. For example, anyone in the same Wi-Fi network can control Sonos and other Universal Plug and Play (UPnP)-based smart speakers and raise/lower the volume levels. Furthermore, Internet of Things (IoT) devices are rarely updated [35], making them especially vulnerable and susceptible to LAN attacks.

The rise of untrusted apps on a user's mobile device challenges the assumption that connections from within the LAN are inherently trustworthy: currently, malicious or compromised apps enable attackers to infiltrate a victim's LAN without the need for physical access. For instance, the Switcher and `Roba.o` malware were contained in malicious mobile apps and attacked Wi-Fi routers by brute-forcing the admin password, altering Domain Name System (DNS) server settings, and thereby redirecting all DNS queries to an attacker-controlled server [23, 60].

Unfortunately, there is no open or standardised defence to prevent mobile apps from attacking the LAN. For instance, on Android, there is no separation between LAN and global Internet access, allowing malicious apps with only the common install-time Internet permission to attack LAN devices. As motivation, and to establish a baseline for evaluating defences, we first implement four LAN attacks in an Android app. While doing so, we become the first to show that an Android app with only the Internet permission can obtain a MitM position between LAN devices. Previously, this was only possible using rooted phones [21, 22, 29, 36, 37, 40, 67, 82]. Most notably, we abuse this MitM position to disrupt an SSH service and to brute-force the password of a smart power socket.

As a defence, we define a formal model that distinguishes local from global Internet access. We realise this model in LANShield, an Android app allowing users to control LAN access of other apps. LANShield notifies users when an app attempts to access the LAN, allowing the user to exclude apps from accessing a specific address range. All combined, LANShield refines Android's Internet permission to differentiate between LAN and global Internet access.

To understand the impact of our LAN access model as the basis for a local network permission, and to more generally investigate the conditions under which mobile apps access the LAN, we test apps and analyse their LAN behaviour using LANShield. In contrast to previous work, which relied on automated setups to assess app LAN behaviour [79], we *manually* test apps to gain deeper insight into behavioural patterns and ensure broad app and code coverage. In particular, we manually interact with apps and examine the effect on LAN access when, (1) the app runs in the background/-foreground, (2) the user (dis)agrees with the privacy policy, and (3) the user does (not) grant specific runtime permissions. In total, we conducted 854 tests, covering 399 apps from five different categories. We identify 89 apps that unexpectedly access the LAN. We further identified 93 apps that perform network scans using service discovery protocols and 24 that unexpectedly try to access every single IP address in the LAN. Most interestingly, a notable fraction of these LAN accesses is caused by a monetisation Software Development Kit (SDK) and can occur even when disagreeing with the app's privacy policy. We compare these results against automated tests to highlight the strengths and limitations of each approach.

Unlike Android, iOS has a permission to prevent unauthorised apps from accessing the LAN without the user's consent. Although this permission is a step in the right direction, its design is not open, meaning its precise security guarantees are unclear. We systematically analyse its security and discover three novel methods to bypass this permission. Additionally, and in contrast to previously-discovered iOS bypasses [79], we implement the bypasses in a Proof-of-Concept (PoC) app using standard developer tools and demonstrate the practical impact of such bypasses. For instance, we abuse our novel iOS bypasses to make a router use a malicious DNS server. We responsibly disclosed our findings to Apple and demonstrated mitigations for each bypass. In addition, we reverse-engineered the permission's implementation and documented its inner workings.

To summarise, our main contributions are:

- We demonstrate through four example attacks how malicious apps on non-rooted phones with only the Internet permission can directly attack devices in the LAN or obtain a MitM position (Section 3).
- We define what is considered LAN access in the context of mobile apps and develop LANShield, an app that refines Android's permission model and intercepts LAN traffic (Section 4). We release the app and its source code.
- We conduct 854 manual app tests using LANShield and compare the results to automated approaches (Section 5).
- We analyse and reverse-engineer the iOS local network permission and find several bypasses (Section 6).

Lastly, we cover related work in Section 7.2 and conclude in Section 8.

2 Background

This section introduces our primary threat model, the security models of Android and iOS, and common service discovery protocols.

2.1 Threat model: Malicious App

Threat Model. The user unintentionally installs a malicious app controlled by the attacker, e.g., by disguising it as a benign app. Likewise, the adversary may also develop a Trojan SDK, which a benign app developer unknowingly includes. The malicious app can perform LAN I/O operations typically limited by the mobile platform to the transport layer (TCP/UDP) and ping (ICMP echo) requests. Furthermore, the app can send/receive multicast and broadcast packets. The attacker does not have access to the link layer since the user's phone is not rooted (e.g., Address Resolution Protocol (ARP) spoofing is not possible). On iOS, this app is also subject to Apple's permission model, requiring the user to give consent to access the LAN.

Threat Types. We consider attackers with three possible goals and consider three possible threats: *T.1* Directly attacking devices in the local network to manipulate their behaviour; *T.2* Scanning the network to profile the user and extract personal data; *T.3* Achieving a MitM position between devices in the network using multicast and broadcast-based protocols.

Examples of threat *T.1* are the Switcher [23] and `Roba.o` malware [60] that attack the router. Examples of threat *T.2* are malicious apps using discovery protocols such as multicast DNS (mDNS) and Simple Service Discovery Protocol (SSDP) to fingerprint local networks. For instance, Girish et al. passively monitored 2335 mobile apps and discovered that 9% scan home networks [41]. Finally, examples of threat *T.3* include mDNS service takeovers, as described by Farrah et al. [38] and other multicast and broadcast discovery protocols.

2.2 Android Security Model

Android permits untrusted code to run on a device, provided the end user explicitly consents to it through downloading and installing an app. The platform follows a multi-party consent model, meaning no action can be performed unless the user, the platform, and the app's developer approve it [66]. Security decisions are made and enforced at the process boundary. Android does not have in-process compartmentalisation. External code included by the app's developer, such as a monetisation SDK, is executed with the same rights and permissions as the rest of the app. There are generally three permission types [45]:

Install-time permissions. This type of permission is presented to the user before installing an app and is granted upon installation. A sub-type of install-time permissions is the *normal* permission, which defines permissions with a low privacy risk. An example of a normal permission is the `INTERNET` permission.

Runtime Permissions. *Runtime* permissions, also called *dangerous* permissions, require explicit user consent at runtime before an app can access certain features or data, e.g., the location set of permissions.

Special permissions. Permissions labelled as *special* prompt the user to explicitly activate them on a separate UI page, usually through a toggle. These permissions are set to protect especially sensitive resources and can be viewed in the *Special app access* page on the user's phone.

2.3 iOS Security Model

Android and iOS have similar security models. No external third-party code is executed before the user installs an app from an app store and opens it. Like Android, all code in an iOS app runs with

Table 1: Local network communication actions and whether they require Apple's local network permission, as stated by an Apple representative [73].

Proto.	Action	Perm. req.
TCP	create outgoing connection	✓
	accept incoming connection	✗
UDP	send unicast, multicast, broadcast	✓
	bind (i.e., connect) socket to a destination	✓
	receive incoming unicast	✗
	receive incoming multicast, broadcast	✓

the same permission level. Therefore, a malicious SDK linked by a benign developer has the same capabilities as the app. Apple offers a similar permission system to Android: many permissions require user consent, while some less sensitive permissions are granted upon installation.

In contrast to Google, Apple introduced the "local network privacy controls" in iOS and iPadOS 14 in 2020 [8]. The system automatically requests permission when an app attempts to access the LAN for the first time. Apple justifies this permission by arguing that apps can collect sensitive information on devices in their users' LANs and use it to profile them [8]. They define a *local network* as an "IP network associated with a broadcast-capable network interface" [72]. Every IP address on such a network is considered a local network address [72]. Table 1 lists all network communication subject to the local network permission model. All outgoing communication to the LAN will require explicit permission from the user. Incoming unicast connections with UDP and TCP are unrestricted and not considered harmful. Furthermore, Apple declares that the permission not only affects low-level networking but extends to high-level APIs such as `NSURLSession` and `WKWebView` [73].

2.4 Service Discovery Protocols

In the LAN, devices can advertise services and allow users and other devices to access them. In contrast to the Internet's DNS, local hosts do not have fixed names that describe their services and, therefore, need to advertise their services differently.

mDNS & DNS-SD. The mDNS protocol was proposed in a draft IETF document [90], implemented by Apple, and later defined in RFC 6762 [26]. It uses the multicast addresses `224.0.0.251` and `ff02::fb` on UDP port 5353 and can be combined with DNS-based Service Discovery (DNS-SD) to, in a decentralised manner, discover and advertise services in the local network [25]. The DNS-SD protocol typically queries domains such as `a._tcp.local` or `a._udp.local`, where `a` is the name of the requested service. Since 2021, Android also resolves `.local` domains using mDNS [44]. When a host joins the multicast group, they will first query for all resource records they wish to allocate, e.g., hostname, service name, to ensure that no other host in the LAN has already reserved them [26]. This is known as the *probing phase*. After receiving no reply, the host will send a multicast packet with their new registered resource records. This is known as the *announcing phase*. Acquiring a hostname allows a host to advertise services other LAN clients can access, such as SSH, HTTP, etc. When a host in the LAN wants to discover a service, they can send an mDNS query for the specific service type, e.g., `_http._tcp.local` [25]. When a conflicted record is identified, the host providing the service will go back to the *probing* and *announcing* phases, where the winner will get to keep the conflicting record, and the loser must change the conflicting record. This will prompt hosts offering such services to reply, and devices in the multicast group will update their cache to facilitate the newly discovered services. Linux implements mDNS/DNS-SD using Avahi [17], while Apple calls their implementation Bonjour [7].

SSDP/UPnP. The SSDP uses the multicast addresses `239.255.255.250`, `ff02::c` (link-local IPv6), and `ff05::c` (site-local IPv6) on port 1900 to discover local devices. It is based on HTTP messages, notably M-SEARCH requests and is used in UPnP [2, 5].

3 Motivation: Attacks against LAN Devices

Local networks typically consist of privacy-sensitive devices such as IP cameras, which co-exist in the same network as phones with several different installed apps. Unbeknownst to users, apps with the normal install-time Internet permission can passively monitor the network, leverage MitM positions, or even attack other devices. To highlight the need for more refined Internet permissions, this section demonstrates how malicious apps *without* root privileges can compromise the LAN. More specifically, we demonstrate four example attacks.

Firstly, we demonstrate two direct attacks against devices in the LAN. The first attack alters the router's DNS settings, while the second attack gains access to an IP camera. Secondly, we present two MitM attacks. The first attack targets mDNS services and allows a malicious app to obtain a MitM position in the LAN, while the second attack obtains a MitM position between a client and a smart power plug which utilises a custom UDP-based broadcast protocol.

3.1 Direct LAN Device Attacks

We demonstrate how a malicious app can directly attack devices in the LAN, matching threat type *T.1*.

3.1.1 Attacking the router. For our first attack, we developed a PoC Android app that can reconfigure the DNS settings of a popular home router. The targeted router, a TP-Link WR841N, is configurable using a local web interface and uses factory-default credentials. To carry out the attack, our app uses three HTTP requests: one to log in, one to update the DNS settings, and one to reboot the device. Other possible attacks include: 1) setting up a VPN interface to route all Internet traffic through an attacker-controlled server, 2) disabling network access, and 3) uploading a hijacked firmware image to execute arbitrary code or gain persistent control over the router. Note that similar attacks were also observed in practice [23, 60]. An iOS variant of this router attack was also implemented, enabled by an iOS bypass later discussed in Section 6.2.2.

3.1.2 Attacking an IP camera. In our second attack, we target an IP video camera in the LAN. This attack is based on the reverse engineering work of [89], adapted to Android. The malicious app first scans the local IP subnet to discover the device using a UDP-based custom protocol. Then, the malicious app can follow the IP camera's unencrypted and unauthenticated custom session establishment

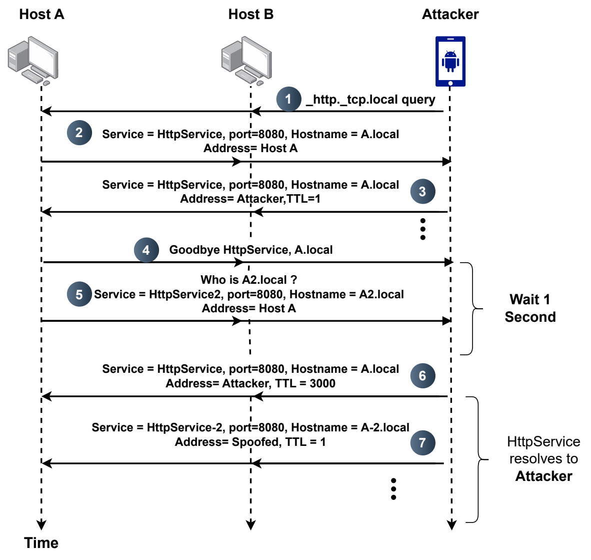

Host A
Host B
Attacker
1 _http._tcp.local query
2 Service = HttpService, port=8080, Hostname = A.local Address= Host A
Service = HttpService, port=8080, Hostname = A.local Address= Attacker,TTL=1 3
4 Goodbye HttpService, A.local
Who is A2.local ?
5 Service = HttpService2, port=8080, Hostname = A2.local Address= Host A
Wait 1 Second
Service = HttpService, port=8080, Hostname = A.local Address= Attacker, TTL = 3000 6
Service = HttpService-2, port=8080, Hostname = A-2.local Address= Spoofed, TTL = 1 7
HttpService resolves to Attacker
Time

Figure 1: mDNS attack against service names.

protocol. Upon establishing a session, the attacker can receive all data traffic from the IP camera.

3.2 Obtaining a MitM with Mobile Apps

We next demonstrate how a malicious Android app with merely the normal install-time Internet permission can abuse multicast and broadcast-based protocols to obtain a MitM position. Through a MitM position, an attacker can disrupt, fingerprint and exploit devices and services in the LAN. These attacks match threat type *T.3*, but also threat type *T.2*, since broadcast and multicast protocols can be used to passively fingerprint the LAN (e.g., listen for broadcast and multicast packets and retrieve information such as services offered in the network).

3.2.1 mDNS Attacks. We show that a malicious app can send mDNS response messages, announcing a service name currently allocated by another host in the network. This ultimately forces the service-providing host to abandon its service and adopt a new one, allowing the attacker to impersonate the service-providing host. Here we use the terms *service-providing host* and *client host* to refer to hosts in a LAN that provide an mDNS service and who want to access an mDNS service, respectively.

In our experiments, we identified that some implementations, such as Avahi Linux, might not return to the probing phase upon receiving conflicting mDNS responses. Instead, they will send mDNS "goodbye" messages for all their services under that hostname. These "goodbye" messages, which are mDNS reply messages where the resource records in the answer section have a TTL of 0, will make clients in the group delete the hostname and service from their caches 1 second after the message has been sent [26]. If an advertisement is received for that hostname before the 1 second has passed, that hostname may be "rescued", meaning that all resource records that were about to expire will be recovered. As a result, a device attempting to resolve the service will not obtain the attacker's address, since the service has been removed from every device's cache. After sending the "goodbye" messages, the service-providing host will acquire a new hostname and send a service advertisement.

To circumvent this, we refined Farrah's et al. [38] service name attack, resulting in an attack with two differences: 1) this attack not only forces the service-providing host to change to another service, but also, due to an Avahi implementation, disallows them from establishing any permanent service in the network, and 2) this attack works against service-providing hosts which send "goodbye" messages upon identifying conflicting service records. The mDNS attack is depicted in Figure 1. In step ①, the attacker will query for a specific type of service, e.g., `_http._tcp.local` services, acquiring the service name and hostname sent by the service-providing host in step ②. In step ③, the attacker will send query responses identical to those of the service-providing host, only changing the IP address of the A record to match that of the attacker and setting the TTL of all resource records to 1. After the service-providing host identifies the conflict, it will send a "goodbye" message and attempt to acquire a new service name and hostname, as shown in steps ④ and ⑤. The attacker will then wait 1 second to ensure that the service-providing host's address is removed from the group's cache [26]. In step ⑥, the attacker can re-acquire the original service and hostname with a longer TTL, e.g., TTL equal to 3000, to be kept in a MitM position for a longer time (e.g., 3000 seconds).

The attacker sets the TTL to 1 in step ③ because if the multicast group receives one of these packets after the "goodbye" message in step ④ , resource records are only "rescued" for 1 second. This minimises the attacker's waiting period to 1 second before all resource records are deleted. Otherwise, the attacker would wait a longer period.

When the service-providing host acquires the new hostname and service name, the attacker repeats the same process. For the new services and hostnames the service-providing host attempts to acquire, the attacker advertises them under a spoofed, potentially random address, as shown in step ⑦. This will cause the service-providing host to identify a conflict, but potential clients will not map the new hostnames and service names to the attacker, allowing the attacker to remain in a MitM position using the original hostname and service name.

Finally, during experimentation, we identified that the Avahi mDNS implementation on Linux will set a long rate limit after 8 consecutive mDNS conflicts. This entails that the malicious app will remain in an *exclusive* MitM position for a specific service for an extended period. We provide an unlisted YouTube video showcasing the *service attack* against an Avahi host providing an SSH service.

3.2.2 Smart Power Socket. As a second example, we demonstrate that a malicious Android app can control an insecure Kankun smart power socket. Our PoC is a modified version of a Python script [4] adapted for Android. Our PoC replies to the Kankun's custom broadcast UDP protocol to establish a MitM position, which allows it to subsequently intercept control packets. Since the smart power socket uses a fixed AES key to encrypt all control packets, our PoC app can then decrypt the power socket's password, and, using the recovered password, remotely control the smart power socket.

3.2.3 Required Permissions. According to Android documentation, on some devices, the multicast lock must be set to reliably receive multicast traffic [48]. Otherwise, the Wi-Fi stack of these devices drops incoming multicast and broadcast packets. To set the multicast lock, the app requires the `CHANGE_WIFI_MULTICAST_STATE` permission. However, when testing the attacks on our Android device, they were always successful without requiring the multicast lock or the permission. To validate this, we forcefully closed all other apps and entered Ultra power-saving mode to ensure the attacks succeeded even when no other app held a multicast lock. Therefore, we conjecture that on most devices, the Wi-Fi stack delivers multicast/broadcast frames to the Android kernel by default. All in all, a malicious app only requires the widely-used `INTERNET` permission to carry multicast and broadcast-based attacks.

3.3 LAN Security Still Unsolved

As demonstrated in this section, attackers can exploit the lack of distinction between LAN and global Internet access permissions on Android phones to obtain a MitM position between devices or even directly attack them. Separating LAN access from global Internet connectivity would allow users to block untrusted apps from accessing privacy-sensitive information within the LAN while still allowing them access to the Internet.

Unfortunately, no open model exists for mobile apps that securely separates LAN from global Internet access. Furthermore, existing firewall apps like NetGuard do not have functionality to differentiate between LAN or Internet access [19]. Although iOS has a local network access permission, its implementation and design are closed, making its security guarantees unclear. In this paper, we address this gap by defining a model for mobile apps that securely separates LAN from global Internet accesses, similar to Private Network Access (PNA) for browsers [78]. We implement this model on Android in an app called LANShield, study the LAN behaviour of Android apps using LANShield, and analyse the security of iOS's local network permission in the context of our LAN access model.

4 A Model and Android Implementation for Secure LAN Access

In this section, we first define a model on how to securely separate LAN from global Internet access. We then implement this model on Android by creating LANShield, an app that monitors and controls other apps' access to the local network and is designed to be usable even by non-technical users. We verify its security by confirming it prevents the attacks of Section 3. In the next section, we use LANShield to study the LAN behaviour of Android apps.

4.1 LAN Access Model

To separate LAN from global Internet access, we need a precise definition of LAN access; specifically, we need a LAN access model. A naive yet insecure definition would be to define the local network as *only* the subnet associated with a network interface, e.g., the Wi-Fi or Ethernet interface. This definition would allow an app in a network with the subnet `192.168.1.0/24` to access `192.168.2.1` without being considered LAN access, because the latter address is not part of a subnet associated with a network interface. While the latter address is not part of a directly connected local network, a network configuration with multiple local networks that can access each other is possible. For instance, if a user attaches a second router to their ISP-provided router, then the user's LAN consists of multiple local networks and subnets, and access to devices within any of these subnets should be considered local network access.

In our mobile LAN model, we define local network access as any subnet associated with a network interface *and* as a selection of private or local subnet ranges. In general, we consider *local*, *private*, and (non-global) *broadcast and multicast* addresses as LAN access. The term *local* refers to all private-use and link-local addresses typically used to interact with the LAN, and the term *private* refers to reserved non-globally routable addresses as defined by IANA [20]. All combined, the following ranges are always considered LAN access in our model (see also Table 4 in the Appendix):

Local addresses. For IPv4, a range of private-use addresses is defined in RFC1918 [69], which, for instance, contains the subnet `192.168.0.0/16`. For IPv6, the site-local range `fec0::/10` has a similar purpose, and these addresses are valid only within an organisation, i.e., routers must not forward such packets outside their organisation [31]. Additionally, unique local addresses in the subnet `fc00::/7`, which are the successors of the (now deprecated) site-local IPv6 addresses [52, 58]. Link-local addresses are also considered LAN access for both IPv4 and IPv6 with the subnets of `169.254.0.0/16` and `fe80::/10`, respectively [24, 32].

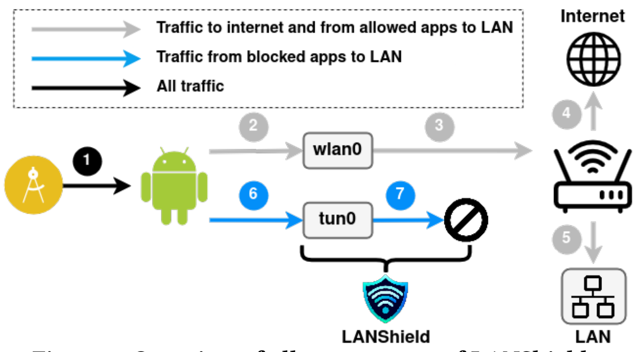
Private addresses. Traffic to non-globally routable addresses, i.e., *private* addresses, is also considered. Although these are not typically used for LAN access, misconfigured or non-standard networks may still use them. This includes ranges for benchmarking [1, 30], documentation [15, 34, 59], discard-only IPv6 addresses [55], the 6in4 relay anycast prefix [57, 87], and the IPv6 segment routing prefix [61]. For IPv4, we additionally consider the IETF subnet `192.0.0.0/24` [28] to be LAN access since almost all its addresses are not globally routable.

Broadcast and multicast addresses. All IPv4 broadcast and multicast traffic, and all IPv6 multicast traffic, including traffic towards global-scope multicast destinations, are considered LAN access [32].

Comparison to PNA. Compared to PNA for browsers [78], which attempts to prevent websites from attacking devices in a client's internal network through the browser, we do not consider the Carrier-Grade NAT address range, i.e., 100.64.0.0/10, as LAN access. This is because we observed several apps in certain countries accessing these IP ranges during normal usage. A second difference is that PNA only includes one private address range, namely the benchmarking range 198.18.0.0/15 [1], while we include 15 private ranges (see Table 4). A third essential difference is that the loopback ranges 127.0.0.0/8 and ::1/128 are considered private access in PNA while they are not considered LAN access in our model. Although loopback traffic can also be a security concern [64, 91], we consider the protection of on-device inter-app communication an orthogonal problem, and here focus on access control to LAN devices. Fourth, our model includes broadcast and multicast ranges for both IPv4 and IPv6. Lastly, we also include the (now deprecated) site-local address range fec0::/10 since older devices may still use it.

4.2 Implementation on Android: LANShield

We implemented our LAN access model on Android by creating LANShield. This app separates LAN and global Internet traffic,


Traffic to internet and from allowed apps to LAN
Traffic from blocked apps to LAN
All traffic
Internet
wlan0
tun0
LANShield
LAN
1
2
3
4
5
6
7
Figure 2: Overview of all components of LANShield.

effectively refining Android's Internet permission. LANShield accomplishes this by intercepting all traffic destined for the LAN from apps that are not permitted to access it and redirecting it to a sink. The user can choose for which apps LAN access is blocked.

4.2.1 Design. Our app creates a virtual network interface (TUN device) that acts as a sink for blocked traffic. We update the device's routing table to direct traffic destined for specific addresses or subnets based on the LAN access model discussed in Section 3 to this sink. Additionally, we add routing rules to exclude traffic from apps that are allowed LAN access from being routed through the virtual interface. Adding and configuring the virtual network device is implemented using the `VpnService` API and does not require root access nor a modified version of Android. Previous works used a similar mechanism to intercept and monitor Internet traffic on Android [62, 65, 76, 81].

An overview of how LANShield works is shown in Figure 2. In the first example (top, grey arrows), ① an app generates traffic towards a public Internet address. ② Android routes this traffic via the Wi-Fi interface (`wlan0`), bypassing LANShield. ③ The traffic is transmitted to the access point and ④ forwarded to the Internet.

In the second example, ① an app sends traffic to a local address. ⑥ If the app is blocked from LAN access, Android routes the traffic to the TUN interface (`tun0`). ⑦ LANShield intercepts this traffic, logs metadata, optionally notifies the user, and drops the packet.

Local traffic from apps that are explicitly allowed is not routed through the TUN interface and instead follows the same steps as public Internet traffic. It is sent to the access point ③ and forwarded onto the local network ⑤. This ensures no overhead is introduced for traffic the user does not wish to block.

4.2.2 App Access Policy. LANShield lets users control which apps can access the LAN. A global default policy blocks or allows LAN traffic, with distinct settings for system and non-system apps. Users can override this setting per app. For example, a user can configure LANShield to block all non-system LAN access by default. When a non-system app then attempts LAN access for the first time, LANShield shows a notification prompting the user to set a policy for that app, similar to Android's runtime permission dialogues, see Figure 6 in the Appendix.

4.2.3 Intercepted Address Ranges. We intercept traffic according to our LAN model defined in Section 4.1. In other words, we iterate over all interfaces and intercept traffic to the subnet(s) associated with each interface, which is especially essential for IPv6 since local networks may use globally routable addresses. Additionally, we intercept traffic to all ranges listed in Table 4 of the Appendix.

4.2.4 Evaluation. As a security evaluation of LANShield, we assess its ability to block malicious traffic targeting the LAN. Specifically, we test whether LANShield can detect and prevent malicious LAN traffic generated by the malicious apps developed in Section 3. In all four attacks, LANShield successfully notifies the user of the malicious app's attempt to access the LAN. Moreover, if the user chooses to block LAN traffic, the attacks become infeasible.

4.2.5 Open access. We will release LANShield as open-source software allowing researchers to study the LAN behaviour of Android apps. Furthermore, we publish the app on the Google Play and F-Droid app stores, offering users control over which apps are allowed to access their LAN. Figure 7 contains screenshots of LANShield's user interface.

4.3 Data Collected by LANShield

LANShield logs metadata for each blocked packet. Specifically, it records the destination IP address and port number, transport layer protocol (e.g., TCP/UDP/ICMP), the UID of the sending app, and a timestamp. Users can view this metadata in the LANShield interface, where blocked traffic is grouped per app. There is also an option to export the recorded metadata to a JSON file.

4.4 Performance Evaluation

We evaluate the overhead of LANShield in terms of battery, memory, and network performance using a Pixel 9 Pro device.

4.4.1 Energy Consumption. We measure LANShield's energy consumption across different use cases to identify potential battery usage overheads (see Figure 9 in the Appendix). For each case, we collect ten 10-minute measurements using *Power Profiler* [49], capturing CPU, WLAN, Disk, and Memory power usage. We apply t-tests to assess statistical significance between scenarios. In the first experiment, we compare (1) an app sending max-MTU UDP packets at 1k packets/sec with LANShield allowing traffic, and (2) the same app with LANShield disabled. No significant overhead is observed (p-values > 0.65 for all components). In our second experiment, we benchmark power consumption in two scenarios: (1) an app scanning a /24 subnet every minute with LANShield blocking traffic, and (2) the same app with LANShield disabled. Here, we observe a statistically significant difference in CPU, memory and WLAN ($p < 0.0001$ for all three). More specifically, CPU and memory power consumption see an increase of approximately 26% and 13% when LANShield is blocking LAN traffic, while WLAN usage decreases by 18% . This suggests LANShield reduces WLAN energy use by blocking traffic, but increases the CPU's and memory's battery consumption when processing it.

4.4.2 Memory Usage. To estimate LANShield's memory usage, we track LANShield's memory consumption for 5 minutes in three scenarios and choose a sample value every 30 seconds, resulting in 10 values for each scenario. We compare: (1) an app sending 1k max-MTU UDP packets every second to a host on the LAN while LANShield is blocking the traffic, (2) two apps sending 1k packets each while LANShield is blocking the traffic and (3) LANShield is enabled, but no traffic is sent. In scenarios 1, 2 and 3, the app's average memory consumption is 29.1 MB, 28.66 MB, and 25.91 MB with standard deviations 1.11, 1.18 and 0.03, respectively. Furthermore, we identify a statistically significant difference when LANShield is blocking 1k and 2k packets vs. when no traffic is sent ($p < 0.0001$ for both), but no statistically significant difference when LANShield is blocking 1k vs. 2k packets. This indicates that LANShield causes additional memory consumption by logging metadata of blocked traffic, which does not necessarily increase with the amount of traffic being processed.

4.4.3 Network Throughput. We benchmark the transfer rate and bitrate of the device using *iPerf3* [3] by sending TCP traffic to a server in our LAN and collecting 10 measurements for when LANShield is disabled vs. enabled. We conducted a two-tailed t-test and found no statistically significant differences. This is expected; permitted network traffic is routed as usual, leaving transfer rate and bitrate unaffected. To identify any potential impact on global traffic, we benchmark transfer rate and bitrate in scenarios where two processes send global and local traffic simultaneously when LANShield is: (1) allowing LAN traffic, (2) disabled, and (3) blocking LAN traffic. We identify no statistically significant difference for global traffic regarding transfer or bitrates in (1) vs. (2) , but identify a statistically significant difference between (3) vs. (1) and (2) ($p <= 0.0001$ for both transfer rate and bitrate in all cases). This statistically significant difference may indicate that LANShield blocking LAN traffic improves Internet transfer rate and bitrates when sending LAN and global traffic simultaneously, as blocking LAN traffic frees up Wi-Fi bandwidth.

5 Testing the LAN Behaviour of Android Apps Using LANShield

In this section, we study the LAN behaviour of apps by manually interacting with them and compare our results to an automated approach. We highlight the impact of granting permissions, (dis)agreeing with privacy policies, and we discuss suspicious scanning behaviour of advertisement SDKs.

5.1 Testing Methodology

We manually interacted with apps while LANShield was enabled and configured to detect LAN accesses. To ensure a varied testing pool, we selected at most 100 apps from each of the following five categories, resulting in a total of 399 usable apps:

Random. To have a representative baseline of tested apps, we randomly selected 100 apps from Androzoo [63]. The selected apps had to have the `INTERNET` permission in their manifest, be available on the Play Store, have a minimum of one thousand downloads, and be last updated in 2024 (i.e., the app's last version was less than a year old).

Most Downloaded. To study whether widely-used apps that have access to network status information might access the local network, we selected the top 100 most downloaded apps that have one or more of the following permissions in their manifest: `ACCESS_NETWORK_STATE`, `READ_PHONE_STATE` or the `BACKGROUND`, `COARSE` or `FINE` location permissions. We selected apps with these permissions to study the impact of granting privacy-related permissions that may lead to fingerprinting. More specifically, these permissions may allow an app to gain network information. For example, the `getScanResults()` method, which returns a list of all Access Points (APs) from the latest scan, requires both `ACCESS_NETWORK_STATE` and `FINE` location permissions [51].

Multicast. To study apps that are more likely to access devices on the LAN, we selected apps that contained strings related to multicast discovery protocols. Namely, we selected apps that contained the strings `_tcp.local` or `_udp.local` relating to mDNS, or the strings `239.255.255.250` or `M-SEARCH * HTTP/1.1` used by SSDP/UPnP (recall Section 2.4). We collaborated with Google to scan all APKs tested by Play Protect in a single day for these strings. For each of the four strings, we randomly selected 25 apps that contained this string resulting in a total of 100 apps.

IPv6. To increase the probability of detecting local IPv6 traffic, we collaborated with Google to scan for apps containing the strings `ff02::1`, `ff02::2`, and `ff02::fb`, corresponding to the multicast IPv6 address to reach all nodes, all routers, and mDNS, respectively. This resulted in only 12 apps. While creating LANShield, we found that the app "Kuaishou" used IPv6 and added it as the 13th app in the IPv6 category.

Umlaut SDK. While creating LANShield, we noticed that some apps scan the entire network. By reverse-engineering these apps we found that the `com.umlaut.crowd` SDK was responsible for this behaviour in the observed cases. To study the behaviour of this SDK in more detail, we collaborated with Google to query for all Play Store apps that used this SDK. We selected the top 100 most downloaded apps containing this SDK, as these are more likely to be functional.

Two people tested each app by manually navigating through its User Interface (UI). Several aspects were evaluated such as whether: (1) the app accessed the local network; (2) the access was vital for the app's functionality (3) the LAN access occurred immediately after granting permissions or was delayed until the user agreed to a privacy policy, and (5) the access was considered malicious (e.g., sweep scanning). Installing, testing, and recording results for one app took approximately 10 minutes, amounting to roughly 18 days of work hours for all experiments.

For special cases, such as apps that were restricted in specific countries, a third person performed an additional test. Finally, the metadata of all local network flows (recall Section 4.3) were exported to a JSON file using LANShield, which enabled cross-checking of the information provided by the testers and allowed further analysis at a later stage.

5.2 Results

A total of 854 tests were conducted using LANShield on 413 unique apps (see Table 5 in the Appendix for raw results). Out of the 413 apps, 14 (3.39%) could not be tested because they were either not available on the Play Store, not accessible through third-party websites, only compatible with old Android devices, or crashed on startup. This resulted in a total of 399 available apps to test.

5.2.1 General LAN Behaviour. When considering a random selection of apps or looking at the most downloaded apps, respectively

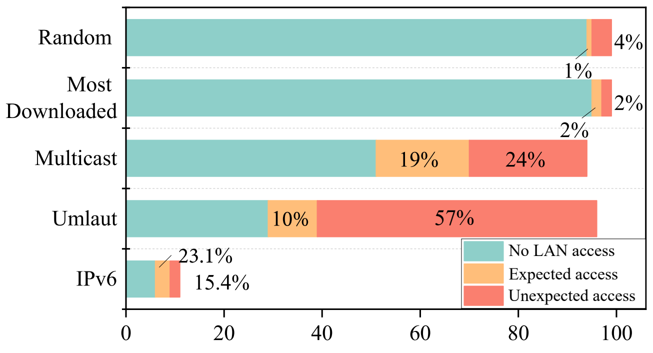


Figure 3: App categories and their LAN behaviour. More detailed results are provided in Table 5.

5% and 4% of them accessed the local network (see Figure 3). In the Multicast category, 43% of the apps accessed the LAN, which confirms that our string-based search is more effective in returning apps that send local network traffic than a random selection of apps. Interestingly, this means that the other 57% of Multicast apps contain multicast-related strings without LAN access being detected. This may be because apps only access the LAN under specific conditions that we did not explore or, in some cases, might have unreachable code.

In the Umlaut category, 67% of the apps accessed the LAN, a strong indication that the `com.umlaut.crowd` SDK contained in these apps accesses the LAN. In the IPv6 category, although only 13 apps were tested, 5 (38%) exhibited LAN access behaviour. However, only the “Kuaishou” app used IPv6 to access the LAN: it contacted an address starting with `fd00:6868:6868:0`. The purpose of this is unclear, although it appears to be a hardcoded address.

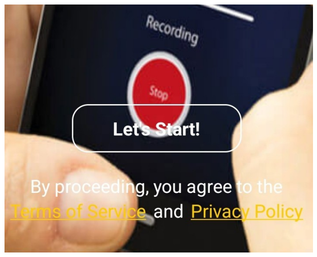
Broadcast/Multicast Network Scanning. We detected 93 apps that use SSDP, 3 that use NetBIOS, and 2 that use mDNS to discover local devices. Additionally, 6 apps broadcast packets to `255.255.255.255` and 12 only access unicast IPs in the LAN.

Background Access. Among the 124 apps that access the LAN, 18 also did so when running in the background, with only 2 being expected to access the LAN. This suggests that it would be beneficial to have a permission that only allows LAN access when the app is being used, similar to Android’s location or microphone permissions.

5.2.2 *Categorising (Un)expected LAN Access.*

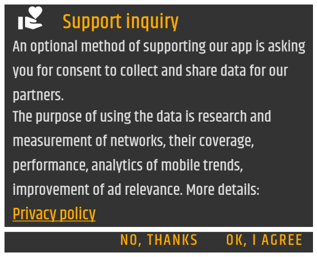
The detected LAN accesses of the 399 tested apps were classified as expected and unexpected. Unexpected access implies that LAN access did not appear to be required for the app’s functionality, i.e., the tester would have blocked the app from accessing the LAN, and the functionality would not have been impaired. Expected access implies that LAN access was required for the app to function normally.

In the Random and Most Downloaded categories, 80% (4 out of 5) and 50% (2 out of 4) of LAN accesses were considered unexpected, respectively. In the IPv6 category, 40% (2 out of 5) of LAN accesses were unexpected. Overall, this indicates that any Android user might reasonably likely have at least one app installed that accesses the local network without their knowledge.



By proceeding, you agree to the Terms of Service and Privacy Policy

(a) Auto Call Recorder.



Support inquiry

An optional method of supporting our app is asking you for consent to collect and share data for our partners.

The purpose of using the data is research and measurement of networks, their coverage, performance, analytics of mobile trends, improvement of ad relevance. More details: Privacy policy

(b) Simple Weather App.

Figure 4: An implicit and explicit privacy policy of two apps that scan the local network for all hosts in the current subnet.

In the Multicast category, 55.8% (24 out of 43) out of the LAN accesses were considered unexpected. This suggests that, out of the apps that contain strings or code to access the local network, a slight majority will access the local network without the user’s knowledge, highlighting the need for developers and platforms to provide more transparency on an app’s local network activities.

For the Umlaut category, 85% (57 out of 67) of the LAN accesses were unexpected. This is significantly higher than in other categories, likely because the Umlaut SDK is being used in these apps.

5.2.3 *Potentially Malicious behaviour.*

Most apps are not expected to sweep scan the entire local network, i.e., accessing every IP address in the LAN. However, we found that 33 apps ran a sweep scan on the LAN, and 73% (24 out of 33) of those apps were not expected to access the LAN at all. Since network scans may be used to fingerprint the LAN and extract data [41], we consider this behaviour potentially malicious. We considered an app to be sweep scanning when it accesses more than 50 unique local IP addresses.

5.2.4 *Impact of Privacy Policies on LAN Behaviour.*

We define a privacy policy in the context of an app as a UI element that discloses information related to data collection, sharing, storing, etc., by the app. An example of a privacy policy that was initially not detected by both testers is shown in Figure 4a: pressing “Let’s Start!” entails you implicitly agree with the privacy policy. This implies that the app is not usable without agreeing to the privacy policy, in contrast with other apps such as the one in Figure 4b. While testing apps, we observed that some access the local network immediately after a user agrees to a privacy policy. This was noticeable because LAN-Shield shows a notification for each new LAN access. To investigate this further, we measured how many apps access the LAN before showing a privacy policy: among the 124 apps that accessed the LAN, 80.6% (100 out of 124) initiated network access without the user’s consent to any privacy policy. Notably, this LAN behaviour was most prevalent in the Multicast and Umlaut categories, with 32% (32 out of 100) and 61% (61 out of 100) respectively. In contrast, other categories exhibited significantly lower rates, all below 4%. This implies that apps frequently access the network without user consent in the Multicast and Umlaut categories, thereby potentially posing a risk to user privacy.

Among the 54 apps that accessed the LAN and had a privacy policy, 22.2% (12 out of 54) altered their LAN behaviour when the privacy policy was declined, either ceasing or reducing LAN access.

This indicates that some apps adjust their network activity when users reject the privacy policy to avoid unauthorised actions, at least when the app is usable without agreeing to the privacy policy.

5.2.5 Impact of Runtime Permissions on LAN Behaviour.

Out of 79 apps that requested runtime permissions, 8.9% (7) accessed the LAN only after obtaining these permissions. We found this surprising since accessing the LAN does not require any runtime permissions. Additionally, 79.0% (98 out of 124) of the apps that accessed the LAN did so without requesting or being granted runtime permissions. When denying runtime permissions, 45.6% (36 out of 79) of the apps changed their LAN access behaviour, either ceasing or reducing their access. We conjecture that these apps only execute certain code after obtaining a given set of permissions, highlighting that it is necessary to grant runtime permissions to fully explore an app's LAN access behaviour.

The *unexpected* network scans were mainly performed by Umlaut apps: 38% (22 out of 57) of Umlaut apps that were not expected to access the LAN perform a network scan. Reverse engineering of these apps indicated that they used the `isReachable` API on every IP address in the current subnet.

5.2.6 The Monedata/Umlaut SDK

As previously indicated, many network scans, which indicate potentially malicious behaviour, were caused by the `com.umlaut.crowd` SDK. Based on online information, this company sources crowd data by integrating their SDK into thousands of Android and iOS apps to collect mobile network performance and usage data [88]. The Umlaut SDK is, in turn, part of the Monedata SDK, which combines several tracking SDKs into a single solution [83]. It is noteworthy that the documentation of Monedata mentions *"Collecting the user consent is MUST to be able to monetise your app"* [68]. However, 71.87% (69 out of 96) of apps, such as `smsr.com.cw`, performed a network scan or ran an SSDP discovery *before* agreeing to a privacy policy.

5.3 Case Studies

Simple Weather App. An interesting case of an app with *unexpected* and *potentially malicious* LAN access is the weather app `net.difer.weather`. After installation, it accesses the LAN without opening the app. This app uses the Monedata SDK; however, when it is first initialised, it only asks for the location and notification permissions. After briefly using the app or reopening it, it also asks for the background location permission. Once full location access is granted, the user is asked to agree to a support inquiry and the privacy policy. If the user agrees, it will start scanning the entire local network. This LAN scan is also performed if other runtime permissions are not granted, highlighting the need for a new LAN-specific runtime permission.

CallSafe: Caller ID & Contacts. This app detected whether a VPN was enabled and, if so, disallowed the user from using the app. We tested this app by connecting to a Wi-Fi hotspot and using Wireshark to monitor network traffic. The app used SSDP/UPnP to discover devices and scanned for all hosts in the current subnet. This is suspicious behaviour: the app mentions the risk of a VPN stealing your data while unexpectedly scanning the local network.

5.4 Comparison to automated testing

To compare the effectiveness of triggering LAN access automatically vs. via human-guided app stimulation, we conduct automated tests using the Android Exerciser Monkey [42]. These tests are done on the same set of apps, using the same setup with LANShield to monitor LAN traffic as used in the manual tests.

Test setup. For our automated tests, we use a physical Android 15 device. Before testing, the device is factory reset, and all non-system apps are removed. For each app, the test procedure includes: (1) start LANShield, configured to only capture traffic from the app being tested, (2) install the target APK, (3) grant all runtime permissions listed in the app's manifest, (4) simulate user interaction using the Exerciser Monkey for five minutes, (5) stop and uninstall the app, and (6) stop LANShield and collect the captured traffic.

Results. Table 6 in the Appendix contains the results. Automated testing triggered LAN access in 114 apps, slightly fewer than the 124 apps detected manually. Automated testing missed LAN access in 31 cases identified manually (*Only H*), but detected LAN access in 21 apps not detected manually (*Only A*). Similarly, automated testing triggered a sweep scan in 37 apps, of which 23 were not detected manually, but missed the sweep scan in 19 apps that were detected manually.

Discussion. Both human-guided and automated testing were effective in triggering LAN accesses, where each detected some LAN-accessing apps that the other missed. We conjecture that manual tests missed some LAN accesses due to differences in the testing procedure, human error, and time constraints faced by manual testers. For example, automated testing granted all runtime permissions at the start of each test, which may have influenced app or SDK behaviour. In particular, the Umlaut SDK was often observed to initiate LAN scanning only when location access was granted (see 5.2.6), potentially explaining the higher detection rate of LAN scanning in the Umlaut category for automated testing. In contrast, manual testers may have accidentally declined permission requests or failed to trigger them, resulting in missed behaviours. While having some limitations, the advantage of manual testing remains that it allows determining the impact of granting permissions, the impact of agreeing with privacy policies, and determining whether LAN access was expected or not. We recommend combining both approaches in future testing workflows to maximise coverage and detection of LAN interaction or other app behaviours.

5.5 Implications

Our manual tests revealed that many Android apps access the local network without the user's awareness or consent. While many of the tested apps had some legitimate use-case, e.g., interacting with an IoT device, we also identified that a monetisation SDK automatically performs network scans without prior consent. Only a few apps requested the user consent for LAN access; many apps accessed the LAN before obtaining user consent. Considering the possible attacks against devices in the LAN, these issues highlight the need for a new Android permission so users can control whether an app is allowed to access the LAN.

We argue for implementing an Android runtime permission that allows users to control if and when an app can access the local

Table 2: Summary of bypasses allowing a malicious app to circumvent the local network permission on iOS, the *threats* (2nd column), and if user *interaction* is required (3rd column).

Issue	Threats	Inter.
WKWebView (Section 6.2.1)	*T.1*	✗
Safari Extension(Section 6.2.2)	*T.1*	✓
Device Discovery Extension (Section 6.2.3)	*T.1, T.2*	✓
watchOS Relay (Section 6.2.4)	*T.1, T.2*	✗
Mis-definition of LAN (Section 6.2.5)	*T.1*	✗

network. For example, a user should be able to restrict an app from having LAN access when it is running in the background. Our app LANShield is a first step in this direction, showing how fine-grained LAN access permissions can be implemented.

6 Analysis of iOS' Local Network Permission

The previous sections demonstrated the necessity for a separate LAN permission, different from a general Internet one. Apple knew about these threats and implemented the local networking permission in iOS 14, released in September 2020. As with all permissions on iOS, the user must explicitly grant it by confirming a system dialogue. A malicious app that could bypass this permission and potentially gain unrestricted access to the local network could attack devices in the network, profile and localise the user, or acquire a MitM position between local devices. Unfortunately, bypassing Apple's permission system is not uncommon. For example, over 100 bypasses of the macOS permission system were found by two researchers in the last four years [39]. In this section, we analyse the implementation of the local networking permission on iOS and identify five possible bypasses.

Schmidt et al. [79] analysed the iOS local networking permission during a similar time frame as we did. Their article [79], to be published at IEEE S&P 2025, lists two issues that we also independently found and extended with example attacks (Section 6.2.5 & Section 6.2.1). Although they reported these issues in June 2022, Apple has still not mitigated them at the time of writing. Furthermore, we reported the same issues to Apple in June 2024, and Apple did not mark these issues as duplicates, potentially indicating miscommunication within Apple's product security team.

Beyond the bypasses described in [79], we found three additional ways to circumvent the local network permission. Each bypass was implemented as part of a PoC in a malicious app; the bypasses only use public iOS APIs, and we demonstrate how each could exploit the LAN. We reported the security issues to Apple; they reproduced each report and confirmed the issues. Apple mitigated the issue described in Section 6.2.3 with the iOS 18 update, and they assigned `CVE-2024-44147`.

6.1 Methodology

We describe our methodology for analysing the iOS local network permission. We use a combination of testing based on Apple's developer documentation and reverse engineering the implementation to find potential issues. First, we define a set of test cases identified while reading Apple's documentation. We examined official APIs for network access and app extensions, which run as separate processes and could therefore have different permissions than the host app. We added a complete list of the test cases in the Appendix (Table 3). We developed a malicious app that includes all our test cases. A test case succeeds if the app can access a host on the local network or find hosts through service discovery without showing the permission dialogue to the user. All tests are performed on iOS 17.5 and confirmed on iOS 18 using non-rooted iPhones.

Second, we reverse-engineer the Apple local network permission by combining static and dynamic reverse engineering. We follow the proven reverse engineering strategies of [27, 54, 84]. We identify binaries and iOS system frameworks handling permissions using system logs. Then, we use *Ghidra* to analyse these binaries statically and *Frida* to dynamically hook into functions, to print arguments or modify behaviour. We use an Apple security research device, i.e., an iPhone with root-level access provided by Apple to security researchers. We primarily use reverse engineering to understand how the local network permission is implemented and to identify the root causes of the bypasses found. We present the results of our reverse engineering effort in Section 6.3.

Example Attacks. To verify our bypasses, we implement an attack for two threats described in Section 2.1 (*T.1, T.2*). *T.1* has been implemented as an attack on the Wi-Fi router that changes the router's DNS settings. We focus on routers with default passwords and have implemented two variants of the attack: OpenWRT and a TP-Link router. *T.2* has been implemented as mDNS service discovery using Apple's `NWBrowser` API. The attack returns the service names and hostnames in the local network. Our YouTube Playlist demonstrates the bypasses in Section 6.2.1 to Section 6.2.5 using our PoC app.

6.2 Local Network Permission Bypasses

We found several issues that would allow a malicious app to access the local network. In this section, we describe the issues, example exploits, their limitations, and mitigations.

6.2.1 WebView Bypass. The `WKWebView` allows an app to browse and interact with web content without the user leaving the app. Most third-party browsers on iOS must also use `WKWebView` because Apple does not allow other web-rendering engines outside of the European Union [85]. Although `WKWebView` is hosted in a third-party app, web requests in the `WKWebView` are not bound to the local network permission. The root cause of this issue lies in the details of the `WKWebView` implementation. Requests made through `WKWebView` are not performed by the app but by the *WebPageProxy*. This proxy creates a separate *WebContent* process for each visited website. As these processes are system processes, websites on Safari are handled in the same manner; no restrictions apply. WebKit is open-source, allowing one to verify this implementation flaw by reading the source code [86].

Exploiting this behaviour. The host app can set `WKWebView`'s opacity to 0.0, effectively hiding the attack from a victim. As the host app manages the web view, it is entirely under its control. The `WKWebView` allows the injection of custom JavaScript code, which runs in the context of the visited website [14]. We exploit this with an attack on threat *T.1*. The app opens the router's management website in the web view and executes JavaScript code, which logs in with a default password and manipulates the settings. Finally, the app reboots the router. The JavaScript code is not hindered by the Same-Origin Policy (SOP) because the web view already points to the router's website.

Limitations. As the adversary may only access the local network via a browser-like web view, it is limited to the capabilities of a browser and custom JavaScript code. Therefore, an adversary cannot create low-level socket connections or send mDNS service discovery requests [70].

Mitigation. A `WKWebView` must have the same permissions as other networking-related code in an app. The current bypass is clearly unintended by Apple because they explicitly mention the *WKWebView* as part of the local network permission model in their FAQs [73].

6.2.2 *Safari Web Extension Bypass.*

Apple introduced app extensions in iOS 8, which allow apps to run code outside their host app's process. At the time of writing, iOS supports 37 app extensions [9]. By definition, extensions should share their permissions with the host app and also not allow access to the LAN if the user did not grant permission in the host app [74].

However, we find that browser extensions, i.e., Safari web extensions, do not follow this approach. Safari web extensions can extend and personalise browsing functionality by modifying web content or blocking ads. They are programmed using the same API as Firefox, Chrome, and Edge extensions. Developers must program the extension in JavaScript, and they can define when it executes, e.g., when loading a website. On iOS, browser extensions need to be bundled with a host app. In addition to the JavaScript code, Safari extensions also contain a native part that can communicate with the host app and the extension. We find that the JavaScript code of extensions is executed in the context of the browser and is, therefore, not bound to the permissions of the host app.

Exploiting this behaviour. A malicious app can bundle a genuine-looking Safari browser extension, e.g., an ad blocker, with the app. App extensions, like web views, are also bound to the SOP by default. However, using the extension's configuration file (`manifest.json`), the adversary can define exceptions, allowing the extension to access any host (see [43]). The user has to give consent to this configuration, but this consent does not list further privileges, such as LAN access. In our PoC app, we demonstrate the exploit for threat *T.1* and modify the DNS of an OpenWRT-based router that uses the default login.

Limitations. Similar to the web view bypass, the extension is limited to the JavaScript APIs in a browser. Furthermore, web extensions have stronger restrictions and may not access cookies returned in the `Set-Cookie` header or set all headers in a web request, e.g., the `Referrer` header is omitted. Consequently, the web extension may not be able to exploit any IoT device in the network, nor can it use sockets to perform service discoveries with mDNS and SSDP. However, our PoC demonstrates that even the widely used OpenWRT routers can be vulnerable to such attacks.

6.2.3 *Device Discovery Extension Bypass.*

Another extension that does not need to request permission to access the LAN is the DeviceDiscoveryExtension (DDE). This extension enables developers to find third-party media receivers, e.g., Google Chromecast, to which an app may then stream content. Developers can execute custom code in the extension to scan for media receivers using mDNS, SSDP, or Bluetooth. By design, the extension is not affected

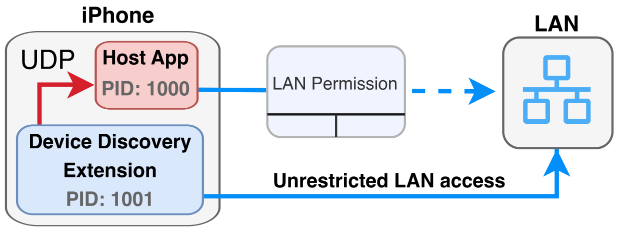


Figure 5: The DeviceDiscoveryExtension (DDE) has full access to the local network and can share all information over a UDP socket to the host app, which is restricted by the permission.

by the iOS permission model and can access the local network and perform Bluetooth scanning without restrictions. This behaviour is expected and documented: "Because DDE runs in a system sandbox, the extension doesn't need to ask the user for local network or Bluetooth permissions." [10] The mentioned *system sandbox* should limit the code running in it and prevent it from communicating back to the host app. The sandbox prohibits the extension from writing to disk. However, we find that an oversight enables the extension to use an open UDP socket in the host app as a side-channel. Opening UDP sockets and receiving data is not prohibited by the local networking permission (see Table 1).

Exploiting this behaviour. Figure 5 presents the basis of an attack in which the unrestricted extension may access the LAN while the permission prohibits the host app. Since the host app continues to run while the extension is executed, the DDE can send any data retrieved from the local network back to the host app using an open UDP socket. In our PoC app, we implement two exploits: as network access is unrestricted, we exploit threat *T.1* and modify the DNS of a TP-Link router that uses a default password. Additionally, since mDNS service discovery is not restricted, we scan for services in the network and send all information to the host app.

Limitations. To launch the extension, the user has to interact with the app to initiate media streaming by tapping a system-defined button. The button cannot be invoked in the background (see Fig. 8a in the Appendix). When launched, the system also displays a media receiver picker, which is identical to the AirPlay receiver picker in the system UI. The extension's process is killed when the user closes the media receiver picker. Since our attack completes within 5 seconds, we are not hindered by this time limit.

Mitigation. Apple introduced the DDE to give privileged access to the local network and Bluetooth for a limited time without granting the app full access. Apple has already limited the DDE by disallowing it from writing files to disk. Also, the media receiver picker presented is rendered by the Springboard process, prohibiting the host app from accessing it directly or creating screenshots. Only the selected media receiver will be provided to the app. We propose that Apple adds a temporary permission for the DDE. When the user taps the button, the system displays a dialogue asking the user if they permit access to the local network and Bluetooth to find media receivers; this way, the user can make an informed decision. Apple mitigated this issue after our report by blocking outgoing UDP packets to localhost, effectively removing our side-channel.

6.2.4 *watchOS Relay.*

The Apple Watch Operating System (OS) shares most of its permissions with iOS but lacks the local network permission. Therefore, we find that a watchOS app may access any host in the LAN. Furthermore, if the watchOS app starts an audio session, it will gain temporary access to low-level networking APIs, allowing the app to perform mDNS service discovery.

Exploiting this behaviour. Most watchOS apps are extensions or companion apps to an iOS app. Therefore, both apps running on different devices can communicate in the background using the `WatchConnectivity` framework. Our adversary model uses an iOS app accompanied by a watchOS app. By default, watchOS apps are installed automatically on every Apple Watch paired with an iPhone. In our PoC app, we include a watchOS companion app that the iOS app can instruct to run an attack exploiting threats *T.1* and *T.2*. We can use the full network APIs while the watchOS app performs audio playback, allowing us to exploit the router and scan for services with mDNS.

Limitations. One limitation is that the watch app requires starting an audio playback session to gain access to the low-level networking APIs needed for mDNS [11]. However, only the Apple Watch Series 10 has an integrated speaker, allowing audio playback. If the attacker starts an audio playback session on another Apple Watch model, the watch will connect to Bluetooth headphones, potentially alerting the user. This limitation does not impact the attack against the Wi-Fi router.

Mitigation. As the Apple Watch can perform networking similar to iOS, it should also include the same local network permission. We suggest porting the permission from iOS to the Apple Watch, as Apple has already done with other permissions, such as Bluetooth.

6.2.5 Mis-definition of Local Network Access. Apple defines the local network as any subnet associated with a broadcast-capable network interface, e.g., the Wi-Fi or Ethernet interface [72]. This definition includes networks using a subnet with public IP addresses, e.g., non-RFC1918 IPv4 subnets such as `94.100.43.1/24` or non-local IPv6 subnets such as `2001:41b8:83c:f900::/64`.

Unlike our own LAN access model of Section 4.1, Apple allows an app in a network with, e.g., subnet `192.168.1.0/24`, to access devices in other local subnets such as `192.168.2.0/24` without requiring local network permission. We reported this to Apple, and they replied that this is expected behaviour. However, we and previous work [79] consider this behaviour insecure and misleading since malicious apps can still access private devices within a user's home or organisation. This observation also contrasts with the PNA standard for browsers, where the local network is defined by a range of IP addresses instead of only the current subnet [78]. Furthermore, Schmidt et al. [79] found that the local network permission does not apply to the address space of a VPN configuration.

Exploiting this behaviour. A malicious app can access any local IP address outside the device's current subnet, e.g., if a user has both their ISP's router and a custom router, malicious apps connected to one router could attack devices on the other router's network. We implement this attack in our PoC app: the iPhone is connected to the subnet at `192.168.2.0/24`; our app connects to a router outside its subnet with the local IP address `192.168.1.1`. We exploit threat *T.1* and change the DNS of the router.

Limitations. This attack would only work for a limited number of network configurations. Service discovery using mDNS or SSDP is not possible, as these protocols use fixed multicast addresses.

Mitigation. We propose that all local addresses defined by RFC1918 should be subject to the permission [69]. For IPv6, Apple should extend its definition of LAN addresses to include link-local and site-local addresses, as defined in RFC4193 [52].

6.3 Internals of the Local Network Permission

We reverse-engineered the implementation of iOS' local network permission and explain the details in this section. The local network permission is deeply integrated into the system and follows a VPN-like configuration, allowing connection- and packet-based decisions about whether an app can access the local network.

Checking Permission. The `Network` framework handles all networking operations of an app. As an example, consider a simple HTTP request to a LAN address. When the app sends this request to a LAN address, the app uses the `Network` framework to set up the TCP connection. Then, the Network Extension Control Protocol (NECP) subsystem decides whether the access is granted. NECP is integrated into the kernel and manages the access of processes to specific interfaces. The implementation of NECP is part of Apple's open-source macOS kernel [13, 75].

Granting Permission. If the user has not yet decided whether LAN access should be granted to the app in question, the `UserEventAgent` process receives an Inter Process Communication (IPC) message instructing it to ask for the user's permission. This process then instructs the `SpringBoard` (the iOS home screen) to display a permission alert for requesting local network access. If the user accepts, the `nehelper` process updates the configuration file to allow local network access for this app. Once the configuration is updated, the `nesessionmanager` restarts the local process with the updated configuration, enabling the app to access the local network.

Interestingly, the logs also refer to a *VPN* configuration. While NECP does not run a full VPN setup, it uses the same routing tables to decide which IP address ranges can be accessed by which app.

7 Discussion & Related Work

This section outlines limitations of our work and compares it to related work.

7.1 Limitations

LANShield. LANShield uses Android's `getConnectionOwnerUid` API to identify the app responsible for generated traffic. However, for UDP packets, this API only reliably returns the sending app's UID while the corresponding socket remains open. If the socket is closed immediately after sending, the UID may not be resolved, and the sender app remains unknown. In such cases, LANShield still notifies the user, who can infer the app based on contextual information (e.g., foreground app). Furthermore, Android does not support intercepting incoming connections without root access, a limitation shared by tools like NetGuard [19].

Experiments. Around 32% (127 out of 399) apps required a valid login to use their full functionality. Such apps were tested by exploring only the functionality accessible without logging in. Third, the features of some apps only function in specific countries, e.g., local streaming or TV apps such as `net.jawwy.tv`.

As with all behavioural testing, our approach may yield false negatives. Apps may access the LAN only under specific conditions, e.g., after account creation, not triggered during testing. However, false positives are not possible in our study, i.e., LANShield will never misclassify non-LAN traffic as LAN traffic.

7.2 Related Work

LAN Traffic Analysis. Reardon et al. created a framework to detect covert channels in Android apps [77]. They found 42 Unity-based apps using ioctl calls to get the device's MAC address, without holding the ACCESS_NETWORK_STATE permission. Moreover, they found 5 apps that retrieved the MAC addresses of local devices from the ARP cache. Accessing the ARP cache is no longer possible in Android 10 and above [6, 46]. Girish et al. studied how local protocols are used in smart homes [41]. They tested apps, comprising a mix of IoT-specific and randomly selected apps, while being stimulated with programmatically generated random user input on a test device connected to their custom IoT test network with companion IoT devices. The traffic generated by these apps was captured and analysed along with the crowdsourced IoT Inspector dataset [56]. They also estimate the available entropy of *passively observed* local network traffic. Hagar studied 2300 apps to determine whether they access the LAN [53]. They found 8 apps doing ARP scans and 29 apps sending SSDP requests. Schmidt et al. studied the security of the iOS LAN permission, analysed how users understand this permission and automatically tested 10862 apps for LAN access.

While prior work previously examined apps accessing the LAN, we demonstrate attack vectors for MitM and direct attacks from Android apps on non-rooted phones. Furthermore, we implement LANShield offering a fine-grained LAN permission on Android. Using LANShield, we manually test apps for LAN access, replicating the behaviour of a regular user. Finally, evaluate the LAN permission on iOS and find three novel bypasses.

mDNS Attacks. Although prior work explores how attackers can establish a MitM using mDNS [16, 18, 38], our work is the first to showcase how an attacker can establish a MitM position through a malicious Android app. Furthermore, to the best of our knowledge, our attack is the first which considers clients who terminate conflicting resource records by sending "goodbye" messages.

Android. In 2011, Android 4.0 added the VPNService API to add and configure a virtual network interface (TUN) [47]. Later, in 2015, the PrivacyGuard project used this to intercept network traffic, similarly to LANShield, to detect information leaks [81]. Razaghpanah et al. created Haystack that also uses a virtual interface to intercept and analyse traffic [76]. To the best of our knowledge, the PrivacyGuard and Haystack code is not public. Le et al. created AntMonitor, which also relies on VpnService to monitor traffic on Android [62], but lacks IPv6 and ICMP support, which we considered essential for LANShield. Wu et al. found that roughly 15% of Android apps have open ports [91]. This was discovered using a crowdsourcing app that periodically reads the proc filesystem to get a list of open sockets. This approach no longer works on newer phones since Android blocks access to the proc filesystem.

Defenses. Apple extended the local network permission to macOS starting from macOS Sequoia [12]. For browsers, there is a draft standard to protect cross-origin requests to local network resources [78]. Several researchers proposed solutions that aim to protect the LAN and IoT devices from internal and external attackers through a new network device, which monitors connections, identifies malicious accesses, and blocks them based on policies [71, 80].

App-based Solutions for LAN Access Control. Soteris et al. created HanGuard [33] in 2017, an app that coordinates with the home router to protect against attacks from malicious apps. HanGuard works both for Android and iOS, before iOS implemented a LAN permission. On Android, the solution regularly checks the procfs file system to detect ongoing TCP and UDP connections. Their monitor app reports allowed connections to the home router, which will forward or drop the packets. HanGuard provided a battery-efficient solution for Android versions below 10, but is unfortunately no longer compatible with newer Android releases as these block access to the procfs file system [50]. The router must also collaborate with the HanGuard app to allow or block connections. This requires an updated router firmware, initial setup, and a secure communication channel between the router and the app.

NetGuard also uses a virtual interface to intercept traffic, but cannot distinguish between LAN and Internet traffic on a per-app basis [19]. Instead, NetGuard can only block domains globally, and only has an option to allow LAN access globally.

LANShield provides an up-to-date implementation running on Android 9 to 15; the latest version at the time of writing. It does not require a specific home router, protecting any network from malicious apps. While virtual interfaces can impact performance, LANShield routes only blocked apps' LAN traffic through its sink interface, leaving other traffic unaffected.

8 Conclusion

We found that Android apps may access and even scan the local network without the user's knowledge, with some monetisation SDKs even scanning all hosts in the network. To motivate the need for a separate LAN permission, we demonstrate four different attacks a malicious Android app can perform with only the Internet permission. Despite iOS's local network permission, we found bypasses that still grant apps LAN access.

We hope our work encourages mobile OSs, such as Android, to add a new runtime permission which allows users to control apps' LAN access. Additionally, as a more fine-grained defence, mobile operating systems should provide methods to only allow an app to communicate with user-selected devices in the local network. Finally, we disclosed our findings to Google during our collaboration and are hopeful that our results will inspire them to deploy a new local network permission to Android.

Ethics Considerations

LAN Attacks. The malicious apps in Section 3 were only installed on our own device in a lab environment, where no other users could be affected, i.e., all devices in the LAN were under our control.

iOS Security Analysis. All the iOS vulnerabilities were reported to Apple. Apple has reproduced them and will address them in upcoming updates to their operating systems.

Artifacts

All artifacts, including LANShield and PoC attack apps, can be found on https://zenodo.org/uploads/15660194

Acknowledgments

This work is supported by the NLnet Foundation, Google's Android Security and PrIvacy REsearch (ASPIRE) Award, the Cybersecurity Research Programme Flanders, the Research Fund KU Leuven, and the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research, and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

The researchers used GPT-4o and Grammarly to correct grammatical errors and typographical errors across the entire paper.

References

[1] 1999. Benchmarking Methodology for Network Interconnect Devices. RFC 2544. https://doi.org/10.17487/RFC2544

[2] 2008. UPnP Device Architecture 1.0. Retrieved 1 July 2024 from https://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf.

[3] 2025. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. Retrieved 12 May 2025 from https://iperf.fr/.

[4] 0x00string. 2015. kankuncontroller. Retrieved 24 February 2025 from https://github.com/0x00string/kankuncontroller/tree/master.

[5] Shivaun Albright, Paul J. Leach, Ye Gu, Yaron Y. Goland, and Ting Cai. 1999. *Simple Service Discovery Protocol/1.0*. Internet-Draft draft-cai-ssdp-v1-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-cai-ssdp-v1/03/ Work in Progress.

[6] Network Analyzer. 2023. Privacy restrictions in Android 10. Retrieved 30 June 2024 from https://support.netanalyzer-an.techet.net/article/153-privacy-restrictions-in-android-q.

[7] Apple. 2002. Bonjour. Retrieved 5 January 2025 from https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Introduction.html.

[8] Apple. 2023. If an App Would like to Connect to Devices on Your Local Network. Retrieved 20 August 2024 from https://support.apple.com/en-us/102229.

[9] Apple. 2024. App Extensions. Retrieved 13 August 2024 from https://developer.apple.com/app-extensions/.

[10] Apple. 2024. DeviceDiscoveryExtension: Stream media to a third-party device that a user selects in a system menu. Retrieved 30 June 2024 from https://developer.apple.com/documentation/devicediscoveryextension.

[11] Apple. 2024. TN3135: Low-level Networking on watchOS. Retrieved 5 November 2024 from https://developer.apple.com/documentation/technotes/tn3135-low-level-networking-on-watchos.

[12] Apple. 2024. What's new in privacy. Retrieved 30 June 2024 from https://developer.apple.com/videos/play/wwdc2022/10096/.

[13] Apple. 2024. XNU Kernel. https://github.com/apple-oss-distributions/xnu

[14] Apple. 2025. addUserScript(). Retrieved 9 January 2025 from https://developer.apple.com/documentation/webkit/wkusercontentcontroller/adduserscript(_:).

[15] Jari Arkko, Michelle Cotton, and Leo Vegoda. 2010. IPv4 Address Blocks Reserved for Documentation. RFC 5737. https://doi.org/10.17487/RFC5737

[16] Antonios Atlasis. 2017. An Attack-in-Depth Analysis of multicast DNS and DNS Service Discovery. Retrieved 26 December 2024 from https://github.com/aatlasis/Pholus.

[17] Avahi. 2004. Avahi. Retrieved 5 January 2025 from https://avahi.org.

[18] Xiaolong Bai, Luyi Xing, Nan Zhang, XiaoFeng Wang, Xiaojing Liao, Tongxin Li, and Shi-Min Hu. 2016. Staying Secure and Unprepared: Understanding and Mitigating the Security Risks of Apple ZeroConf . In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 655–674. https://doi.org/10.1109/SP.2016.45

[19] Marcel Bokhorst. 2024. NetGuard: Frequently Asked Questions (FAQ). Retrieved 20 July 2024 from https://github.com/M66B/NetGuard/blob/master/FAQ.md.

[20] Ron Bonica, Michelle Cotton, Brian Haberman, and Leo Vegoda. 2017. Updates to the Special-Purpose IP Address Registries. RFC 8190. https://doi.org/10.17487/RFC8190

[21] bponury. 2011. DroidSheep : ARP-Spoofing App for Android. Retrieved 10 December 2024 from https://xdaforums.com/t/app-wifikill-disable-internet-for-network-hoggers-android-4-x.1282900/.

[22] bponury. 2011. WifiKill - disable internet for network hoggers (Android 4.x). Retrieved 10 December 2024 from https://xdaforums.com/t/app-wifikill-disable-internet-for-network-hoggers-android-4-x.1282900/.

[23] Nikita Buchka. 2016. Switcher: Android joins the 'attack-the-router' club. Retrieved 25 August 2023 from https://securelist.com/switcher-android-joins-the-attack-the-router-club/76969/.

[24] Stuart Cheshire, Dr. Bernard D. Aboba, and Erik Guttman. 2005. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927. https://doi.org/10.17487/RFC3927

[25] Stuart Cheshire and Marc Krochmal. 2013. DNS-Based Service Discovery. RFC 6763. https://doi.org/10.17487/RFC6763

[26] Stuart Cheshire and Marc Krochmal. 2013. Multicast DNS. RFC 6762. https://doi.org/10.17487/RFC6762

[27] Jiska Classen, Alexander Heinrich, Robert Reith, and Matthias Hollick. 2022. Evil Never Sleeps: When Wireless Malware Stays On after Turning Off iPhones. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (San Antonio, TX, USA) *(WiSec '22)*. Association for Computing Machinery, New York, NY, USA, 146–156. https://doi.org/10.1145/3507657.3528547

[28] Michelle Cotton, Leo Vegoda, Ron Bonica, and Brian Haberman. 2013. Special-Purpose IP Address Registries. RFC 6890. https://doi.org/10.17487/RFC6890

[29] Decimation. 2015. [Tutorial] How to use netKillUI (WiFiKill for iOS). Retrieved 10 December 2024 from https://www.reddit.com/r/jailbreak/comments/35caeg/tutorial_how_to_use_netkillui_wifikill_for_ios/.

[30] Dr. Steve E. Deering, Robert L. Fink, Tony L. Hain, and Bob Hinden. 2000. Initial IPv6 Sub-TLA ID Assignments. RFC 2928. https://doi.org/10.17487/RFC2928

[31] Dr. Steve E. Deering and Bob Hinden. 2003. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513. https://doi.org/10.17487/RFC3513

[32] Dr. Steve E. Deering and Bob Hinden. 2006. IP Version 6 Addressing Architecture. RFC 4291. https://doi.org/10.17487/RFC4291

[33] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A. Gunter, Xiaoyong Zhou, and Michael Grace. 2017. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Boston, Massachusetts) *(WiSec '17)*. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/3098243.3098251

[34] Diego Dugatkin, Ahmed Hamza, Gunter Van de Velde, and Chip Popoviciu. 2008. IPv6 Benchmarking Methodology for Network Interconnect Devices. RFC 5180. https://doi.org/10.17487/RFC5180

[35] Frank Ebbers. 2023. A Large-Scale Analysis of IoT Firmware Version Distribution in the Wild . *IEEE Transactions on Software Engineering* 49, 02 (Feb. 2023), 816–830. https://doi.org/10.1109/TSE.2022.3163969

[36] EdgaBimbam. 2012. Network Spoofer *trick your friends*. Retrieved 10 December 2024 from https://xdaforums.com/t/app-network-spoofer-trick-your-friends.1692921/.

[37] emeykey. 2021. dSploit ArpSpoof forked module for Android 6-11. Retrieved 10 December 2024 from https://github.com/emeykey/arpspoof-android.

[38] Dhia Farrah and Marc Dacier. 2021. Zero Conf Protocols and their numerous Man in the Middle (MITM) Attacks . In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE Computer Society, Los Alamitos, CA, USA, 410–421. https://doi.org/10.1109/SPW53761.2021.00060

[39] Csaba Fitzl and Wojciech Reguła. 2024. The Final Chapter: Unlimited Ways to Bypass Your macOS Privacy Mechanisms. Retrieved 5 November 2024 from https://www.youtube.com/watch?v=Px3uzzQxdag.

[40] Hosein Ghanbari. 2014. networkKill-IOS. Retrieved 10 December 2024 from https://github.com/hosein-gh/networkKill-IOS.

[41] Aniketh Girish, Tianrui Hu, Vijay Prakash, Daniel J. Dubois, Srdjan Matic, Danny Yuxing Huang, Serge Egelman, Joel Reardon, Juan Tapiador, David Choffnes, and Narseo Vallina-Rodriguez. 2023. In the Room Where It Happens: Characterizing Local Communication and Threats in Smart Homes. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (Montreal QC, Canada) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 437–456. https://doi.org/10.1145/3618257.3624830

[42] Google. 2023. UI/Application Exerciser Monkey. Retrieved 3 August 2023 from https://developer.android.com/studio/test/other-testing-tools/monkey.

[43] Google. 2024. Declare Permissions | Chrome Extensions. Retrieved 13 August 2024 from https://developer.chrome.com/docs/extensions/develop/concepts/declare-permissions.

[44] Google. 2024. DNS Resolver. Retrieved 3 January 2025 from https://source.android.com/docs/core/ota/modular-system/dns-resolver.

[45] Google. 2024. Permissions on Android. Retrieved 9 January 2025 from https://developer.android.com/guide/topics/permissions/overview.

[46] Google. 2024. Privacy changes in Android 10. Retrieved 30 June 2024 from https://developer.android.com/about/versions/10/privacy/changes#proc-net-filesystem.

[47] Google. 2024. VPN | Connectivity. Retrieved 2 July 2024 from https://developer.android.com/develop/connectivity/vpn.

[48] Google. 2024. WifiManager.MulticastLock. Retrieved 23 January 2024 from https://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.html.

[49] Google. 2025. Power Profiler. Retrieved 12 May 2025 from https://developer.android.com/studio/profile/power-profiler.

[50] Google. 2025. Privacy Changes in Android 10. Retrieved 6 May 2025 from https://developer.android.com/about/versions/10/privacy/changes.

[51] Google. 2025. WifiManager. Retrieved 18 February 2025 from https://developer.android.com/reference/android/net/wifi/WifiManager#getScanResults().

[52] Brian Haberman and Bob Hinden. 2005. Unique Local IPv6 Unicast Addresses. RFC 4193. https://doi.org/10.17487/RFC4193

[53] Paul Theodor Hager. 2022. *Curious Apps: Large-scale Detection of Apps Scanning Your Local Network*. Bachelor's Thesis. Technische Universität Wien.

[54] Alexander Heinrich, Milan Stute, Tim Kornhuber, and Matthias Hollick. 2021. Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System. In *Proceedings on Privacy Enhancing Technologies*, Vol. 2021 (3). 227–245. https://doi.org/10.2478/popets-2021-0045

[55] Nick Hilliard and David Freedman. 2012. A Discard Prefix for IPv6. RFC 6666. https://doi.org/10.17487/RFC6666

[56] Danny Yuxing Huang, Noah Apthorpe, Frank Li, Gunes Acar, and Nick Feamster. 2020. IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale. 4, 2, Article 46 (June 2020), 21 pages. https://doi.org/10.1145/3397333

[57] Christian Huitema. 2001. An Anycast Prefix for 6to4 Relay Routers. RFC 3068. https://doi.org/10.17487/RFC3068

[58] Christian Huitema and Brian E. Carpenter. 2004. Deprecating Site Local Addresses. RFC 3879. https://doi.org/10.17487/RFC3879

[59] Geoff Huston, Anne Lord, and Dr. Philip F. Smith. 2004. IPv6 Address Prefix Reserved for Documentation. RFC 3849. https://doi.org/10.17487/RFC3849

[60] Kaspersky. 2023. Roaming Mantis implements new DNS changer in its malicious mobile app in 2022. Retrieved 29 June 2023 from https://securelist.com/roaming-mantis-dns-changer-in-malicious-mobile-app/108464/.

[61] Suresh Krishnan. 2024. *SRv6 Segment Identifiers in the IPv6 Addressing Architecture*. Internet-Draft draft-ietf-6man-sids-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-6man-sids/06/ Work in Progress.

[62] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data* (London, United Kingdom) *(C2B(1)D '15)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/2787394.2787396

[63] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices *(C2B(1)D '15)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/2787394.2787396

[64] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets. In *Network and Distributed Systems Security (NDSS) Symposium*. https://doi.org/10.14722/ndss.2014.23049

[65] Wouter Louman, Mitchell Vernee, Danique de Bruijn, Babette van't Riet, and Hani Alers. 2021. Mobile Firewall applications: An analysis of usability and effectiveness. In *2021 IEEE 5th International Conference on Cryptography, Security and Privacy (CSP)*. IEEE Computer Society, Los Alamitos, CA, USA, 148–152. https://doi.org/10.1109/CSP51677.2021.9357491

[66] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.* 24, 3, Article 19 (Aug. 2021), 35 pages. https://doi.org/10.1145/3448609

[67] midnightchips. 2019. [Release] Harpy: the best way to control your network. Retrieved 10 December 2024 from https://www.reddit.com/r/jailbreak/comments/aep7xo/release_harpy_the_best_way_to_control_your_network/.

[68] Monedata. 2024. Setup SDK. Retrieved 26 August 2024 from https://web.archive.org/web/20240826014841/https://jei6f891.docs.monedata.io/android/setup-monedata-sdk.

[69] Robert Moskowitz, Daniel Karrenberg, Yakov Rekhter, Eliot Lear, and Geert Jan de Groot. 1996. Address Allocation for Private Internets. RFC 1918. https://doi.org/10.17487/RFC1918

[70] Mozilla. 2023. Web APIs | MDN. Retrieved 26 August 2024 from https://developer.mozilla.org/en-US/docs/Web/API.

[71] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. 2014. An experimental study of security and privacy risks with emerging household appliances. In *2014 IEEE Conference on Communications and Network Security*. 79–84. https://doi.org/10.1109/CNS.2014.6997469

[72] Quinn "The Eskimo!". 2020. Local Network Privacy FAQ-1 | Apple Developer Forums. Retrieved 20 August 2024 from https://developer.apple.com/forums/thread/663848.

[73] Quinn "The Eskimo!". 2020. Local Network Privacy FAQ-2 | Apple Developer Forums. Retrieved 13 August 2024 from https://developer.apple.com/forums/thread/663874.

[74] Quinn "The Eskimo!". 2020. Local Network Privacy FAQ-7 | Apple Developer Forums. Retrieved 13 August 2024 from https://developer.apple.com/forums/thread/663813.

[75] Quinn "The Eskimo!". 2023. A Peek Behind the NECP Curtain | Apple Developer Forums. Retrieved 20 August 2024 from https://developer.apple.com/forums/thread/725715.

[76] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. 2015. Haystack: In situ mobile traffic analysis in user space. *arXiv* (2015), 1–13. https://doi.org/10.48550/arXiv.1510.01419

[77] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 603–620. https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[78] Titouan Rigoudy and Mike West. 2024. Private Network Access - Draft Community Group Repor. Retrieved 7 July 2024 from https://wicg.github.io/private-network-access/.

[79] David Schmidt, Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, and Martina Lindorfer. 2025. Analyzing the iOS Local Network Permission from a Technical and User Perspective . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 4229–4247. https://doi.org/10.1109/SP61157.2025.00045

[80] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. 2015. Network-level security and privacy control for smart-home IoT devices . In *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE Computer Society, Los Alamitos, CA, USA, 163–167. https://doi.org/10.1109/WiMOB.2015.7347956

[81] Yihang Song and Urs Hengartner. 2015. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices *(SPSM '15)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2808117.2808120

[82] George Stagg. 2017. Packages. Retrieved 10 December 2024 from https://github.com/georgestagg/repo/blob/8110d42d11badcb57925a5554e7bb62429a51a2f/Packages.

[83] Startup Bubble News. 2024. Is Data Monetisation the Future of Mobile App Industry Revenue Streams? Retrieved 1 July 2024 from https://startupbubble.news/is-data-monetisation-the-future-of-mobile-app-industry-revenue-streams/.

[84] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. 2021. Disrupting Continuity of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3917–3934. https://www.usenix.org/conference/usenixsecurity21/presentation/stute

[85] The European Parliament and the Council of the European Union. 2022. Regulation (EU) 2022/1925 of the European Parliament and of the Council of 14 September 2022 on Contestable and Fair Markets in the Digital Sector and Amending Directives (EU) 2019/1937 and (EU) 2020/1828 (Digital Markets Act) (Text with EEA Relevance). Retrieved 3 July 2024 from http://data.europa.eu/eli/reg/2022/1925/oj/eng.

[86] The WebKit Open Source Project. 2024. WebKit/WebKit. Retrieved 5 November 2024 from https://github.com/WebKit/WebKit.

[87] Ole Trøan and Brian E. Carpenter. 2015. Deprecating the Anycast Prefix for 6to4 Relay Routers. RFC 7526. https://doi.org/10.17487/RFC7526

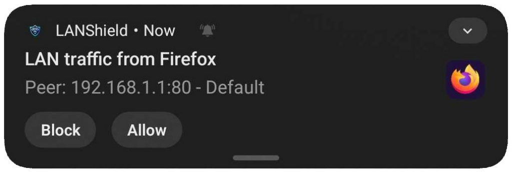
[88] Umlaut. 2023. Optimize your customer experience with network insights from umlaut. Retrieved 1 July 2024 from https://www.snowflake.com/wp-content/uploads/2023/02/optimize-your-customer-experience-with-network-insights-from-umlaut.pdf.

[89] David Ventura. 2024. cam-reverse. Retrieved 27 February 2025 from https://github.com/DavidVentura/cam-reverse.

[90] Bill Woodcock and Bill Manning. 2000. *Multicast Domain Name Service*. Internet-Draft draft-manning-dnsext-mdns-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-manning-dnsext-mdns/00/ Work in Progress.

[91] Daoyuan Wu, Debin Gao, Rocky KC Chang, En He, Eric KT Cheng, and Robert H Deng. 2019. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *Network and Distributed Systems Security (NDSS) Symposium*. https://doi.org/10.14722/ndss.2019.23171

A Appendix



LAN traffic from Firefox

Peer: 192.168.1.1:80 - Default

Figure 6: Notification shown to the user when an app tries to access the LAN for the first time. By pressing *block* or *allow*, the user can configure whether LAN traffic should be allowed or blocked for that specific app.

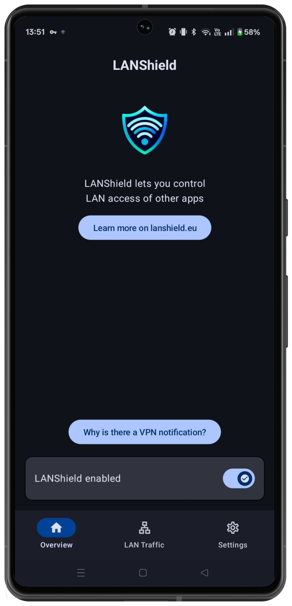
Table 3: Evaluated Test-Cases for LAN access on iOS without user granted permission.

Test-Case	Exploitable	OS version
Developer APIs		
`NSURLSession` (HTTP Request)	✗	17.5.1
`NWBrowser` (mDNS service discovery)	✗	17.5.1
`WKWebView` (WebView for web browsing)	✓	17.5.1
Low-level socket connection	✗	17.5.1
SSDP	✗	17.5.1
Apple Watch		
`NSURLSession` (HTTP Request)	✓	10.5
`NWBrowser` (mDNS service discovery)	✓	10.0
Low-level socket connection	∼[1]	10.5
SSDP	∼[1]	10.5
App Extensions		
Action Extension	✗	17.5.1
App Intents Extension	✗	17.5.1
Device Discovery Extension	✓	17.5.1
Keyboard Extension	✗	17.5.1
Matter	✗	17.5.1
Safari Web Extension	✓	17.5.1

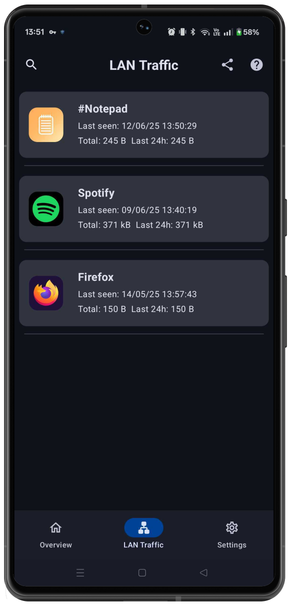
[1]Depends on if the app is performing audio streaming. Low-level networking is only allowed for audio streaming.

Table 4: Non-global routable subnets monitored by LANShield.

Address Range	Description	Type
10.0.0.0/8	Private-Use	Local
172.16.0.0/12	Private-Use	Local
192.168.0.0/16	Private-Use	Local
0.0.0.0/8	“This network”	Private
169.254.0.0/16	Link-Local	Local
192.0.0.0/24	IETF Protocol Assignments	Private
192.0.2.0/24	Documentation	Private
192.88.99.0/24	6to4 Relay Anycast (Deprecated)	Private
198.18.0.0/15	Benchmarking	Private
198.51.100.0/24	Documentation	Private
203.0.113.0/24	Documentation	Private
224.0.0.0/4	Multicast	Private
255.255.255.255/32	Broadcast	Private
240.0.0.0/4	Multicast	Private
100::/64	Discard-Only Address Block	Private
2001:2::/32	Benchmarking	Private
2001:db8::/32	Documentation	Private
5f00::/16	Segment Routing SIDs	Private
fc00::/7	Unique-Local	Local
fe80::/10	Link-Local Unicast	Local
fec0::/10	Site-local addresses	Local
ff00::/8	Multicast	Private



(a) The user can activate LANShield by enabling the slider.



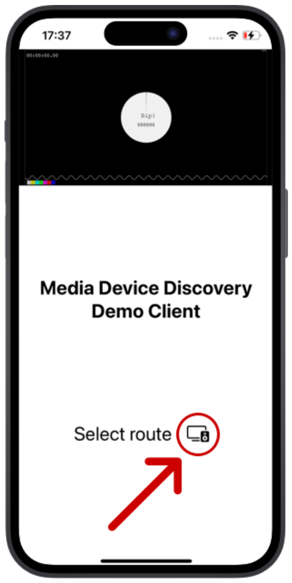
#Notepad
Last seen: 12/06/25 13:50:29
Total: 245 B Last 24h: 245 B

Spotify
Last seen: 09/06/25 13:40:19
Total: 371 kB Last 24h: 371 kB

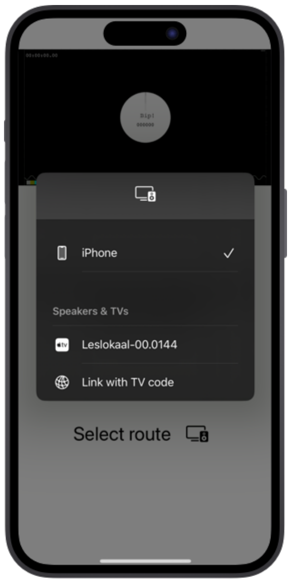
Firefox
Last seen: 14/05/25 13:57:43
Total: 150 B Last 24h: 150 B

(b) Overview of LAN traffic detected by LANShield. By clicking on an entry, the user can view details for a specific app.

Figure 7: Screenshots of the LANShield app.

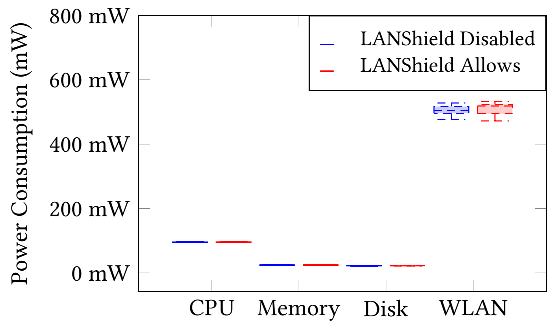


(a) The user must tap the circled button to initiate the discovery, which will execute the malicious extension.

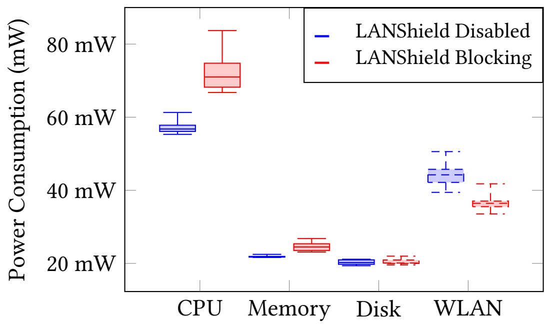


(b) Device picker shown to the user. The extension executes as long as the picker is visible.

Figure 8: Screenshots of the sample app for a DeviceDiscoveryExtension (DDE).



(a) Device battery consumption (mW) per aspect (LANShield allowing traffic vs. disabled)



(b) Device battery consumption (mW) per aspect (LANShield blocking scans vs. disabled)

Figure 9: Battery Consumption comparison

Table 5: A summary of the results of the manual analyses and the impact of privacy policies on LAN behaviour as discussed in 5.2. Where two values are shown separated by a slash (/), the former value represents the number of applications that unexpectedly accessed the LAN and the latter the number of applications that were expected to access the LAN.

	Random	**Most Downloaded**	**Multicast**	**Umlaut**	**IPv6**	**Totals**
Total number of apps	100	100	100	100	13	413
Available apps	99	99	94	96	11	399
General LAN access patterns						
LAN access detected	**5**	**4**	**43**	**67**	**5**	**124**
Unexpected LAN access	4	2	24	57	2	89
Full network scan	0 / 0	1 / 0	0 / 2	22 / 4	1 / 3	24 / 9
Background LAN access	1 / 0	1 / 0	4 / 1	10 / 1	0 / 0	16 / 2
Impact of Privacy Policies on LAN Behaviour						
LAN access before accepting any privacy policy	4 / 0	1 / 0	22 / 10	53 / 8	2 / 0	82 / 18
LAN access detected, and privacy policy prompt shown	0 / 1	0 / 0	5 / 4	36 / 6	1 / 1	42 / 12
LAN access only after accepting privacy policy	0 / 0	0 / 0	0 / 3	1 / 0	0 / 0	1 / 3
LAN access behaviour changed upon declining privacy policy	0 / 0	0 / 0	2 / 1	9 / 0	0 / 0	11 / 1
App unusable without accepting privacy policy	0 / 1	0 / 0	2 / 2	17 / 4	0 / 0	19 / 7
Impact of Runtime Permission on LAN Behaviour						
Runtime permissions requested	2 / 0	1 / 2	11 / 7	45 / 7	1 / 3	60 / 19
LAN access only after granting permissions	0 / 0	0 / 1	0 / 4	2 / 0	0 / 0	2 / 5
LAN access behaviour changed upon denying runtime permissions	2 / 0	0 / 0	12 / 4	16 / 0	0 / 2	30 / 6

Table 6: Comparison of LAN access and sweep scanning detection using automated and human-guided testing.

Category	**LAN Access**				**LAN Scanning**			
	Automated	**Human**	**Only A**	**Only H**	**Automated**	**Human**	**Only A**	**Only H**
Random	1	5	0	4	1	0	1	0
Most downloaded	4	4	3	3	1	1	0	0
Multicast	38	43	6	11	5	2	4	1
Umlaut	70	67	12	9	30	26	18	14
IPv6	1	5	0	4	0	4	0	4
Total	114	124	21	31	37	33	23	19