Understanding Android VoIP Security: A System-level Vulnerability Assessment

En He
OPPO ZIWU Cyber Security Lab
OPPO
Shenzhen, China
he_en@qq.com

Daoyuan Wu*
Department of Information Engineering
The Chinese University of Hong Kong
Hong Kong SAR, China
dywu@ie.cuhk.edu.hk

Robert H. Deng
School of Information Systems
Singapore Management University
Singapore
robertdeng@smu.edu.sg

***Abstract*—VoIP is a class of new technologies that deliver voice calls over the packet-switched networks, which surpasses the legacy circuit-switched telecom telephony. Android provides the native support of VoIP, including the recent VoLTE and VoWiFi standards. While prior works have analyzed the weaknesses of VoIP network infrastructure and the privacy concerns of third-party VoIP apps, no efforts were attempted to investigate the (in)security of Android's VoIP integration at the system level. In this paper, we first demystify Android VoIP's protocol stack and all its four attack surfaces. We then propose a novel vulnerability assessment approach that assembles on-device Intent/API fuzzing, network-side packet fuzzing, and targeted code auditing. By testing Android from version 7.0 to the recent 9.0, we have discovered 8 zero-day Android VoIP vulnerabilities, all of which were confirmed by Google with bug bounty awards. The security consequences are serious, including denying voice calls, caller ID spoofing, unauthorized call operations, and remote code execution. To mitigate these vulnerabilities and further improve Android VoIP security, we uncover a new root cause that requires developers' attention during their design and implementation.**

I. Introduction

VoIP is a class of new technologies that deliver voice calls over the packet-switched networks, instead of the legacy circuit-switched telecom networks, i.e., the so-called Public Switched Telephone Network (PSTN). By transmitting the voice data over the Internet, VoIP offers clear benefits over the PSTN calling service, including improved quality of service, high-fidelity codecs, and lower monetary costs. As a result, network operators are actively promoting VoIP to modern Android smartphones [1], [4], [12], with the latest VoLTE (Voice over LTE) and VoWiFi (or Wi-Fi Calling) schemes being deployed.

Existing works on Android VoIP security, however, are far from comprehensive. They focused either on the weaknesses of VoIP network infrastructure, e.g., the insecure deployment of VoIP protocols at the network service providers' side, or on the privacy concerns of third-party VoIP apps. Notably, Li et al. [29] discovered several vulnerabilities in VoLTE's control- and data-plane functions, and Xie et al. [43] uncovered four vulnerabilities in operational Wi-Fi calling services. Regarding Android VoIP's client-side security, only the privacy risks of some VoIP apps were tested [17], [23], e.g., whether their traffic are encrypted with SSL/TLS. It is thus unclear whether Android's VoIP integration at the operating system level are secure or not.

In this paper, we conduct the first study to systematically analyze Android VoIP's (in)security at the system level. Our study begins with a demystification of Android VoIP's protocol stack and its attack surfaces. Specifically, we study VoIP-related Android system code to identify VoIP components and their implementations, including SIP (Session Initiation Protocol) via the `nist-sip` library, SDP (Session Description Protocol) via `gov.nist.javax.sdp`, RTP (Real-time Transport Protocol) via `librtp-jni.so`, codecs via `libstagefright`, and SIP user agent via the system phone and dialer apps. Furthermore, we identify all the four potential attack surfaces that allow physical, local, remote, and nearby attacks against Android VoIP.

With these components and their attack surfaces in mind, we propose a novel vulnerability assessment approach that assembles on-device Intent/API fuzzing, network-side packet fuzzing, and targeted code auditing. First, we perform both Android Intent fuzzing and system API fuzzing to comprehensively fuzz the local surface. After that, we set up a novel VoIP testbed to perform three protocol fuzzing methods that mutate different fields in SIP, SDP, and RTP protocols either directly from a user agent or through a Man-In-The-Middle proxy. Lastly, we combine automatic fuzzing tests with targeted code auditing, including log-driven and protocol specification based auditing, to eventually determine vulnerabilities.

By fuzzing VoIP components on recent Android OS from version 7.0 to 9.0, we have discovered a total of nine zero-day vulnerabilities, eight of which are system vulnerabilities and have been confirmed by Google with bug bounty awards. Two-thirds of these vulnerabilities can be exploited by a network-side adversary, which suggests that Android VoIP's major risks come from the remote and nearby attack surfaces. Moreover, seven of nine vulnerabilities' severity levels were rated by Google Android security team as high or critical (the most two serious levels), which implies that most of Android VoIP vulnerabilities are serious. The incurred security consequences include denying voice calls, caller ID spoofing, unauthorized call operations, and remote code execution. Furthermore, we

† This is a technical report from The Chinese University of Hong Kong, available online on 26 August 2019.

* Daoyuan Wu is the corresponding author.

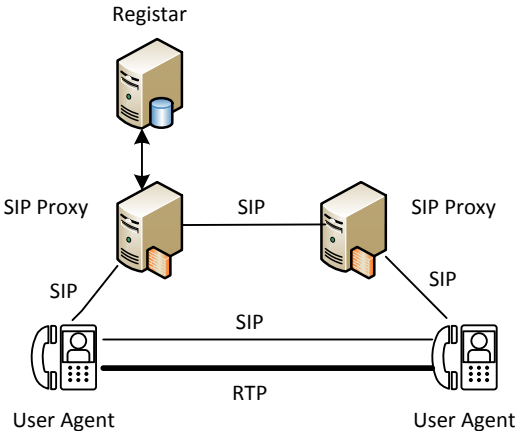


Fig. 1: A typical network infrastructure of SIP.

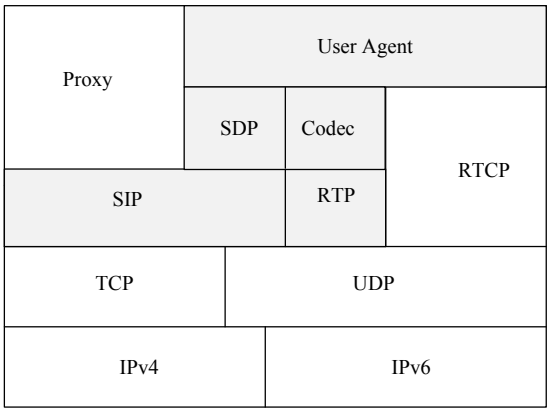


Fig. 2: Android's integration of VoIP protocol stack.

uncover a new root cause, incompatible processing between VoIP and PSTN calls, that leads to six VoIP vulnerabilities and requires developers' extra attention in their future design and implementation.

To summarize, we have made the following contributions in this paper:

- The first demystification of Android VoIP's protocol stack and all its four attack surfaces (Sec. III);
- A novel approach that assembles on-device Intent/API fuzzing, network-side packet fuzzing, and targeted code auditing (Sec. IV);
- New and comprehensive vulnerability assessment results, with nine zero-day vulnerabilities analyzed and their root causes uncovered (Sec. V and Sec. VI).

II. Background

Before presenting our work, we first introduce the necessary background on VoIP and Android in this section.

A. VoIP Background

Android VoIP mainly uses the SIP (Session Initiation Protocol) protocol, which was drafted by IETF in RFC 3261 [8]. As a VoIP signaling protocol, SIP provides a mechanism for one or more participants to create, modify, and terminate sessions. Fig. 1 presents a typical network infrastructure of SIP, which consists of the following components:

- User Agent (UA): A SIP user agent is a logical network node of SIP, which is responsible for creating, sending, and receiving SIP messages and maintains a SIP session.
- Proxy Server: A SIP proxy server helps deliver SIP messages between different user agents. It can also perform routing control and check the integrity of SIP messages.
- Registrar Server: A SIP registrar server is used for accepting SIP REGISTER requests from user agents, and places the location information it receives in those requests.

Similar to HTTP, SIP is a text-based protocol. It employs SDP (Session Description Protocol) to describe session contents. A typical SIP message can be an INVITE, REGISTER, OPTIONS, BYE, or CANCEL request. One important field in the SIP header is the SIP URI (Unified Resource Identifier), which represents the sender or receiver address. A SIP URI is in this format: `sip:user_name@server_ip_address`, e.g., `sip:anonymous@192.168.8.151`.

A SIP call involves three phases: the initial signaling phase, the conversation phase, and the end signaling phase. The INVITE and BYE requests are used in the two signaling phases. During the conversation phase, two calling parties exchange audio/video streams using the codecs that are negotiated via RTP (Real Transmission Protocol) [9].

B. Android Background

On Android, each application, no matter a system or a third-party app, runs in its own app sandbox [40]. Different apps communicate with each other through a new IPC (Inter-Process Communication) channel called Binder-based *Intent*. Each app has its own private data and requires permissions to access system's resources. For example, systems VoIP apps have the `RECORD_AUDIO` and `CALL_PRIVILEGES` permissions.

III. Demystifying Android VoIP

In this section, we demystify Android VoIP's implementation and all its four attack surfaces. To the best of our knowledge, we are the first to give this demystification.

A. Android VoIP's Protocol Stack

By studying Android's source code, we are able to depict its implementation of VoIP protocol at different layers. Fig. 2 highlights Android VoIP's protocol stack in the gray color. Starting from the bottom layer, the stack consists of the following components:

- *SIP (Session Initiation Protocol)*: Android's SIP implementation directly uses the `nist-sip` library, which was developed by National Institute of Science of Technology (NIST). It is a purely Java based SIP implementation, and provides API classes (e.g., `SipSession` and `SipProfile`) via the `android.net.sip` package.
- *SDP (Session Description Protocol)*: Similar to SIP, Android's SDP also uses the NIST implementation

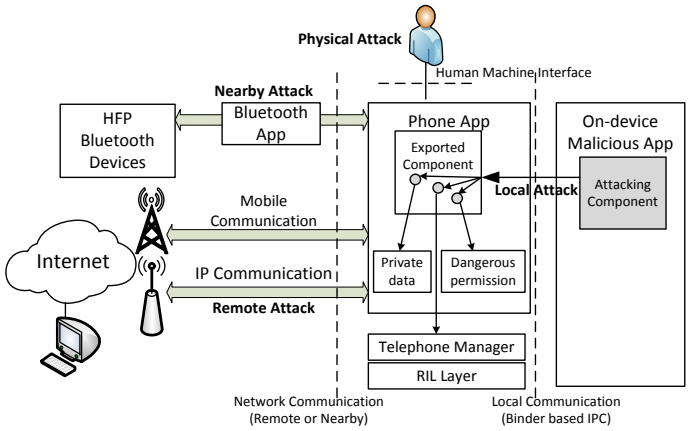


Fig. 3: Android VoIP's four attack surfaces: physical, local, remote, and nearby attack surfaces.

(`gov.nist.javax.sdp`), and provides a hidden API class called `SdpSessionDescription`.

- *RTP (Real-time Transport Protocol)*: Android implements RTP in a C/C++ dynamic link library called `librtp-jni.so`. It also provides a few API classes via the `android.net.rtp` package.
- *Audio or Video Codec*: Android VoIP supports only a handful of codecs, including PCM (Pulse-Code Modulation) type A and type U codec, AMR (Adaptive Multi-Rate) codec, and GSM EFR (Enhanced Full Rate) codec. Supporting these codecs relies on `libstagefright`.
- *SIP UA (User Agent)*: Android VoIP implements its UA into the system phone app (`com.android.phone`). It is a high-privilege app under the Linux user group of `radio`. Hence, it can not only access typical phone-related permissions (e.g., accessing user contacts and making a phone call) but also low-level resources in the Telephone Manager and Radio Interface Layer (RIL). Additionally, displaying VoIP caller numbers is handled by the system dialer app (`com.android.dialer`).

It is worth noting that these VoIP components are not isolated in Android. Indeed, a VoIP session on Android always initiates from the SIP UA and goes through all those protocol and codec components. As a result, by targeting at the system phone and dialer apps, we can trigger Android VoIP's code flows and test the entire Android VoIP components.

B. Android VoIP's Attack Surfaces

Fig. 3 shows all the potential surfaces that Android VoIP could be attacked. We illustrate them as follows:

- *Physical Attack Surface*: If an adversary could physically access a victim user's phone, he is able to set the phone's VoIP configuration without the authorization, causing a security breach. Although such attack is rare, it still needs to be considered, as we will demonstrate in Sec. V.
- *Local Attack Surface*: Since the system phone app is a privileged app, it can access not only permission-protected resources but also system interfaces in Telephone Manager and Radio Interface Layer (RIL). An on-device malicious app thus can attack the phone app via the IPC communication to obtain VoIP-related privileges.
- *Remote Attack Surface*: Since the phone needs to communicate with outside via IP and mobile communication, it brings another attack surface. Specifically, a network-side adversary can send crafted payloads in SIP/SDP/RTP packets to exploit Android VoIP components remotely, causing remote denial of service and code execution.
- *Nearby Attack Surface*: With the popularity of HFP (Hand-Free Profile) devices, a user may use a Bluetooth earphone or a Bluetooth car kit during her VoIP call. These nearby Bluetooth devices bring a new attack surface. On one hand, the malicious payload in VoIP traffic may reach to the system Bluetooth components. On the other hand, the malicious traffic from Bluetooth devices may also attack VoIP components.

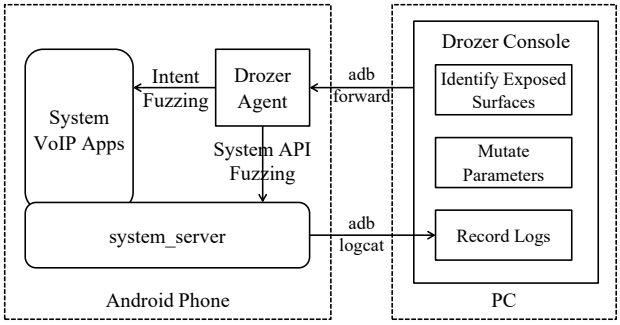


Fig. 4: The on-device fuzzing framework.

IV. Methodology

After understanding Android's VoIP integration and its attack surfaces, we propose a novel approach to systematically assess Android VoIP's vulnerabilities. In this approach, we first automatically test Android VoIP components via both on-device and network-side fuzzing, and further combine them with targeted code auditing to eventually determine vulnerabilities. In this section, we present these three modules, among which network-side packet fuzzing is the most novel one.

A. On-device Intent/API Fuzzing

To comprehensively fuzz the local surface of Android VoIP components, we perform both Android Intent fuzzing and system API fuzzing. Specifically, Intent fuzzing aims to test exported components in VoIP system apps, while system API fuzzing tries to discover unprotected VoIP system service interfaces. In this subsection, we first introduce the fuzzing framework before present its two detailed fuzzing methods.

On-device fuzzing framework. As shown in Fig. 4, we develop an on-device fuzzing framework based on Drozer [2]. We use a drozer console on PC to control the fuzzing process on a test phone via its drozer agent. We deliver fuzzing commands through Android's `adb forward` command and receive fuzzing logs through the `adb logcat` command. For both Intent and system API fuzzing, we perform these three

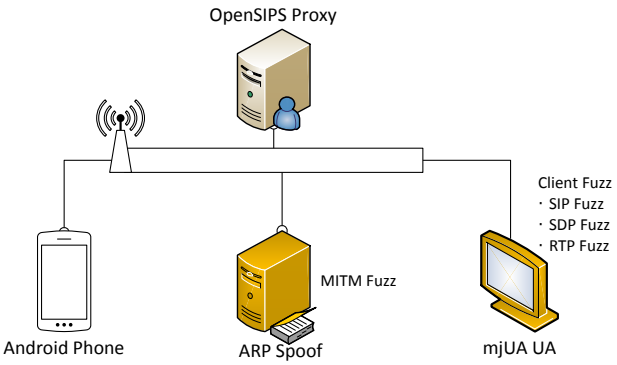


Fig. 5: Our testbed for network-side fuzzing.

steps: identifying exposed surfaces, mutating parameters, and recording logs.

On-device Intent fuzzing. In the Intent fuzzing, exposed surfaces are VoIP apps' `exported` components that can be accessed by any other third-party apps on the same phone. We identify these exported components by analyzing component information in the app's `AndroidManifest.xml` file. To mutate Intent parameters, we try both empty (i.e., `null`) parameters and the parameters that satisfy a component's `data` schemes (e.g., `content://` and `vk.voip`).

On-device system API fuzzing. In the system API fuzzing, exposed surfaces are those unprotected system service interfaces. We identify them by using Java reflection to invoke Android `ServiceManager`'s `listServices` function, which can list not only all the available system service interfaces but also their accepted parameter types. We then launch targeted fuzzing against these exposed service interfaces according to their parameter types.

B. Network-side Packet Fuzzing

To test Android VoIP's network components, we need to launch network-side packet fuzzing. In this subsection, we first introduce our testbed for network-side fuzzing, and then present three protocol fuzzing and two fuzzing modes.

Setting up the testbed. Fig. 5 shows the architecture of our testbed for network-side fuzzing, where an Android phone acts as the victim user and a mjSIP-based User Agent mimics the adversary. Note that mjSIP [6] is a command-line based SIP UA implementation with flexible options. Additionally, we use OpenSIPS [7] to establish a SIP proxy server, and connect all these three parties in the same Wi-Fi network.

Fuzzing different protocols. We leverage mjSIP (`uac.sh`) to fuzz all the three protocols in the Android VoIP stack (see Sec. III), namely SIP, SDP, and RTP fuzzing. Listing 1 shows the mjUA commands used in our three fuzzing methods. Additionally, we install an AutoAnswer app in the Android phone to automate the entire fuzzing process.

- *SIP Fuzzing*: In this fuzzing, we mutate the user name and server name in a SIP URI name. For example, we can use a long SIP name to launch the fuzzing: `$./uac.sh --user <long_SIP_name>`. Additionally, we can also change the display SIP name using the `display-name` option, as shown in Listing 1.
- *SDP Fuzzing*: In this fuzzing, we mutate different fields in the SDP's media description. We launch the SDP fuzzing by preparing variants of a mjSIP configuration file: `$./uac.sh -f configFile.cfg`. The media format of this configuration file is listed in Listing 2. Specifically, we can change the "media" and "media_spec" parameters in multiple ways. For example, we can use different media type, port, and protocol/codec for the "media" parameter and specify different media attributes for the "media_spec" parameter.
- *RTP Fuzzing*: To fuzz RTP codecs, we generate PCMU/PCMA/AMR/GSM-EFR codec corpuses and send them to the Android phone one by one via mjUA's `send-file` option. The detailed fuzzing code is shown in Fig. 6. Specifically, we first prepare a seed file called `sample-gsm-8000.gsm`, and use this seed file to randomly generate different audio files (`fuzz_$i.tone`).

```
$ ./uac.sh -h
  -f <file>: specifies a configuration file (sdp fuzzing)
  -c <call_to>: config the victim SIP URI
  -y <secs>: could be used as fuzz interval time
  --display-name <str>: display name (sip fuzzing)
  --user <user> : user name (sip fuzzing)
  --send-file <file> the specified audio file (rtp fuzzing)
```

Listing 1: A list of the mjUA commands used in our fuzzing.

```
# Media descriptors:
# One or more 'media' (or 'media_desc') parameters specify
    for each media: media type, port, and protocol/codec.
# Zero or more 'media_spec' params can be used to specify
    attributes: codec name, sample rate, and frame size.
# Examples:
#   media=audio 4000 rtp/avp
#   media_spec=audio 0 PCMU 8000 160
#   media_spec=audio 8 PCMA 8000 160
#   media_spec=audio 101 G726-32 8000 80
#   media_spec=audio 102 G726-24 8000 60
#   media=video 3002 rtp/avp
#   media_spec=video 101
```

Listing 2: The media description we leverage for SDP fuzzing.

```
#!/bin/bash

ITER=$1
SEED=fuzztone/sample-gsm-8000.gsm

for i in $(seq $ITER)
do
   # cat $SEED | radamsa -m bf,br,sr -p bu > fuzztone/fuzz_$i.tone
    echo $i
    ./uac.sh --send-file fuzztone/fuzz_$i.tone -f fuzz_config/amr.cfg --send-only
   # ./uac.sh --send-file blankfile -f fuzz_config/amr.cfg --send-only
    adb shell log -p e -t fuzzrtp fuzz_$i
    adb logcat -c
    declare -i i=i+1
done

```

Fig. 6: A code illustration of our RTP/Codec fuzzing.

Direct fuzzing and MITM fuzzing. As shown in Fig. 5, we provide two fuzzing modes: direct fuzzing from the UA and MITM (Man-In-The-Middle) fuzzing. To enable the MITM fuzzing, we leverage the following Ethercap [3] command to perform an ARP spoof for constructing a transparent proxy.

`sudo ettercap -T -V hex -F rtpfuzz.ef -M arp /192.168.8.152// /192.168.8.191//`.

With such a MITM proxy, it is convenient for us to leverage existing VoIP traffic for mutation. For example, we can mutate RTP headers by setting an Ethercap filter, which can specify which packet to filter and how to manipulate. The mutated new packets will be then forwarded to the Android phone.

C. Targeted Code Auditing

To eventually determine vulnerabilities, it is necessary to launch manual code auditing after the automatic fuzzing. In this subsection, we propose two *targeted* code auditing methods that leverage fuzzing logs and protocol specification to reduce manual efforts.

Log-driven auditing. Both on-device and network-side generate a number of fuzzing logs. We thus leverage them for a log-driven code auditing. Specifically, for a process crash produced by our fuzzing, we can collect either a Java exception for Java components (e.g., `IllegalStateException: Reject SDP: no suitable codecs`) or a fault status for native code (e.g., `pid: 8112, tid: 8161, name, XXX, signal 11 (SIGSEGV), fault addr: YYY`). Moreover, we can obtain the detailed location where the code encounters an error, e.g., `createAnswer(SipAudioCall.java:805)` and `libbluetooth_jni.so(clccResponseNative+30)`. We then use these code locations to driven our auditing.

Protocol specification based auditing. PSTN and VoIP protocols have some specifications that we can leverage for a targeted auditing. For example, special attributes, e.g., the call transfer splitting character "&" and the phone number prefix "phone-context", in PSTN may have different behaviors in VoIP, which we will illustrate later. We then leverage this kind of protocol specification differences for an efficient auditing.

V. Evaluation

In this section, we present the results of fuzzing VoIP components on recent Android OS from version 7.0 to 9.0 over a period of around two years. As shown in Table I, we have discovered a total of nine zero-day vulnerabilities, eight of which are system vulnerabilities and have been confirmed by Google with bug bounty awards. Table I lists the meta information of these vulnerabilities, including the entry components where vulnerabilities can be triggered from, the severity level rated by Google Android Security team, and the corresponding security consequence.

A. Discovered Vulnerabilities via On-device Fuzzing

By performing on-device fuzzing, we find that Android VoIP generally protects its local attack surface, with only one vulnerability discovered by the system API fuzzing and no vulnerable component identified by the Intent fuzzing. To also demonstrate the effectiveness of our Intent fuzzing, we test and identify a VoIP vulnerability in a very popular app called VK[1], which has cumulatively over 100 million installs on Google Play.

V1: Maliciously triggering a VoIP call in the VK app. The VK app (version 5.13) was identified by us to contain an exported component, `LinkRedirActivity`, which accepts an Intent with the `content://` scheme and with the `vk.voip` data type. Surprisingly, `LinkRedirActivity` would directly make a VoIP call to a VK user account specified by the `vk.voip` data. As a result, an on-device malicious app can send a crafted Intent to trigger a VoIP call without user's consent and even when the phone screen is turned off. More seriously, the victim user could be eavesdropped if the callee VK account was set to an account under the attacker's control, the idea of which is similar to the login CSRF (Cross-Site Request Forgery) [5] attack in web security. To patch this vulnerability, VK added a user confirmation dialog before `LinkRedirActivity` can make any VoIP call.

V2: Unauthorized call transfer in the IMS interface. Android has a system service called QtilMS, which is for IMS (IP Multimedia Subsystem) related functionality and implemented by Qualcomm. However, our system API fuzzing found that QtilMS exposed two VoIP APIs, `SendCallTransferRequest` and `SendCallForwardUncondTimer`, to any third-party app. Normally, these two system APIs are only accessible to those with the `CALL_PRIVILEGES` permission. However, our fuzzing shows that any app without the permission can also invoke the APIs, because no checking is enforced by QtilMS. As a result, an on-device malicious app can misuse those two privileged APIs to set unauthorized call transfer. To mitigate this, Qualcomm added the permission check for the access of those two QtilMS APIs.

B. Discovered Vulnerabilities via Network-side Fuzzing

Compared to the on-device fuzzing, our network-side fuzzing discovered more VoIP vulnerabilities, as shown in Table I. This suggests that Android VoIP's major risks come from the remote and nearby attack surfaces. In this subsection, we first introduce two vulnerabilities that can be exploited remotely, and then present another two vulnerabilities that involve the nearby Bluetooth-based HFP (Hands-Free Profile) devices.

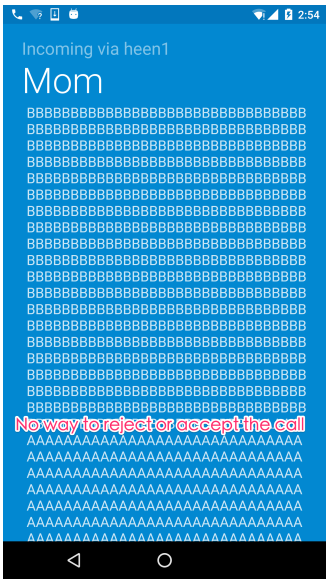
V3: Undeniable VoIP call spam due to long SIP name. We discovered this vulnerability through a SIP fuzzing test using the long SIP name: `$./uac.sh --user <long_SIP_name> <victim's sip account>`. As shown in Fig. 7, the callee user's VoIP phone interface could be filled up by the very long SIP name, e.g., 1,043 characters in our test case. In this scenario, the victim user cannot answer or reject a call, because no button is shown up. If the adversary frequently launches this undeniable VoIP call spam, the victim has to disable the network connection or shutdown her phone. We call this kind of denial of service attack "VoIP call bomb", as similar to SMS bomb [13]. To defend against this attack, Google restricts the length of SIP user name.

[1]https://play.google.com/store/apps/details?id=com.vkontakte.android

TABLE I: Zero-day Android VoIP vulnerabilities discovered in our work.

Discovery Method	ID	CVE/AID	Attack Vector	Vulnerable Entry Component	Affected Android	Severity Level	Security Consequence
On-device Fuzzing	V1	H1-#386144	Local	`com.vkontakte.android`	All	Low	Triggering a call without user's consent
	V2	CVE-2017-11042	Local	`org.codeaurora.ims`	$\leq$ 7.1.2	Moderate	Unauthorized setting of call transfer
Network-side Fuzzing	V3	A-31823540-1	Remote	`com.android.dialer`	$\leq$ 7.1.1	High	Undeniable VoIP call spam
	V4	CVE-2017-0394	Remote	`com.android.phone`	$\leq$ 7.1.1	High	Remote DoS once accepting a call
	V5	CVE-2018-9475	Remote*	`com.android.bluetooth`	$\leq$ 9.0	Critical	Remote code execution due to overflow
	V6	A-79431031	Remote*	`com.android.bluetooth`	$\leq$ 9.0	High	Remote DoS once receiving a call
Code Auditing	V7	CVE-2016-6763	Physical	`com.android.phone`	$\leq$ 7.0	High	Sensitive data leak; Permanent DoS
	V8	A-31823540-2	Remote	`com.android.dialer`	$\leq$ 7.1.1	High	Caller ID spoofing
	V9	A-32623587	Remote	`com.android.dialer`	$\leq$ 7.1.1	High	Caller ID spoofing

* These two remote vulnerabilities could be triggered only when the phone is connected with a *nearby* Bluetooth-based HFP (Hands-Free Profile) device.



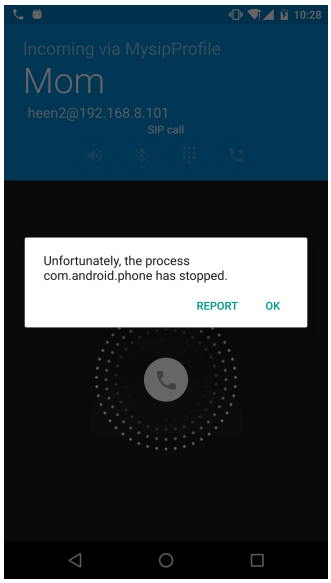


Fig. 7: A demo of exploiting V3. Fig. 8: A demo of exploiting V4.

V4: Remote DoS in Telephony once accepting a call. We discovered this vulnerability through the SDP fuzzing using a malformed configuration file: `$./uac.sh -f malformed.cfg`. As shown in Fig. 8, it can crash the victim's phone process once she accepts the call, causing a remote DoS (denial of service). Our fuzzing identified two weaknesses in the affected Telephony module, either of which could be exploited for the attack. One way is to use a codec that is not in the supported codec list (see Sec. III-A). For example, if we add "media_spec=audio 102 G726-24 8000 60" into the malformed.cfg file, the phone process crashes with an illegal state exception "Reject SDP: no suitable codecs". The other way is to use the invalid SDP description. For example, if we add "media=AAAA 4000" into the malformed.cfg file, the phone process crashes with an illegal SDP argument exception. To patch these weaknesses, Google added exception catch statements for those two unhandled exceptions.

The model of Bluetooth-involved VoIP vulnerabilities. As shown in Table I, the V5 and V6 vulnerabilities could be triggered only when the phone is connected with a nearby Bluetooth device. We thus first explain the model of these Bluetooth-involved VoIP vulnerabilities before presenting their

```
bt_status_t HeadsetInterface::ClccResponse(...) {
  ...
  if (number) {
    size_t rem_bytes = sizeof(ag_res.str) - res_strlen;
    char dialnum[sizeof(ag_res.str)]; //length is 513 bytes
    size_t newidx = 0;
    if (type == ADDRTYPE_INTERNATIONAL && *number != '+')
      dialnum[newidx++] = '+';
  }

  for (size_t i = 0; number[i] != 0; i++) {
    if (utl_isdialchar(number[i]))
      dialnum[newidx++] = number[i]; //Overflow when > 513
  }
  ...
}
```

Listing 3: The vulnerable code of stack buffer overflow in V5.

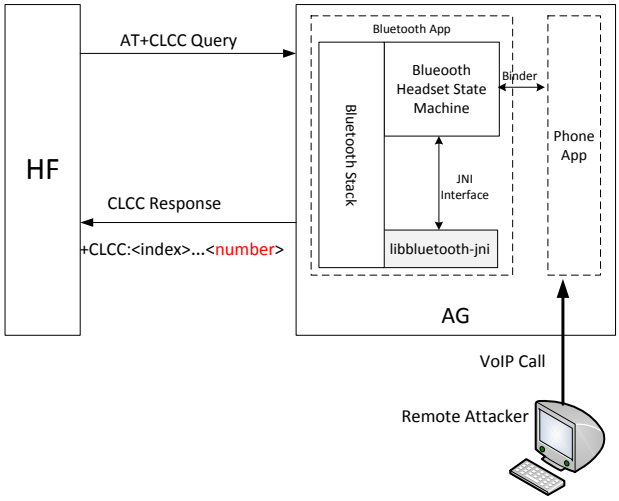


Fig. 9: A model of Bluetooth-involved VoIP vulnerabilities.

specific weaknesses. Fig. 9 depicts such a vulnerability model. Specifically, mobile phone acts as an AG (Audio Gateway) in the HFP (Hands-Free Profile) communication, and Bluetooth earphone or Bluetooth car kit is the HF (Hand Free) device. When a remote attacker makes a VoIP call to a phone currently connected with a HF device, the HF device will query all the call information (e.g., caller number) from the phone via HFP's AT+CLCC command. As a result, the VoIP call input will be delivered to `libbluetooth-jni` for processing. A vulnerability could happen if it cannot process an unexpected VoIP call input (e.g., a long user name), because Bluetooth may consider only the traditional, instead of VoIP, phone call.

```
case BTHF_CALL_STATE_INCOMING:
  if (num_active || num_held)
    res = BTA_AG_CALL_WAIT_RES;
  else
    res = BTA_AG_IN_CALL_RES;

  if (number) {
    int xx = 0;
    // number (xx) might be longer than sizeof(ag_res.str)
    xx = snprintf(ag_res.str, sizeof(ag_res.str), "\"+%s\""
         ,number);
    ag_res.num = type;
    if (res == BTA_AG_CALL_WAIT_RES)
      snprintf(&ag_res.str[xx], sizeof(ag_res.str)-xx,",%d"
         ,type); //a negative value becomes a large integer
  }
  break;
```

Listing 4: The vulnerable code of integer underflow in V6.

V5: Remote code execution due to stack buffer overflow. Both V5 and V6 suffer from the unexpected long user name (or caller number) in a VoIP call. For V5, the vulnerable code locates in the function of preparing CLCC response, as shown in Listing 3. It tries to return the caller number in the CLCC response, but uses only a 513-byte array (`dialnum`) to store it. A stack buffer overflow thus happens when a caller number with more than 513 bytes is inputted. This vulnerability allows an adversary to overwrite the return address of the `ClccResponse` function, causing remote code execution. For example, the adversary can launch the exploit using this command: `$./uac.sh --user $(python -c 'print ``8''*1055')`.

V6: Remote DoS in Bluetooth once receiving a call. This vulnerability is similar to V5, but it is triggered when the call state changes, i.e., `BTHF_CALL_INCOMING` in Listing 4. In this example, developers also did not expect the long caller number in a VoIP call. Specifically, the return value of the first `snprintf` statement can be greater than `sizeof(ag_res.str)`'s 513 bytes. Since the `sizeof(ag_res.str)-xx` variable now is an unsigned negative number, it becomes a very large positive integer, which eventually triggers the `abort` checking statement and causes remote DoS. Compared to the DoS in V4, triggering DoS in V6 just needs to receive, rather than answer, a call.

To patch V5 and V6, Google restricted the length of caller number inputed in the Bluetooth module.

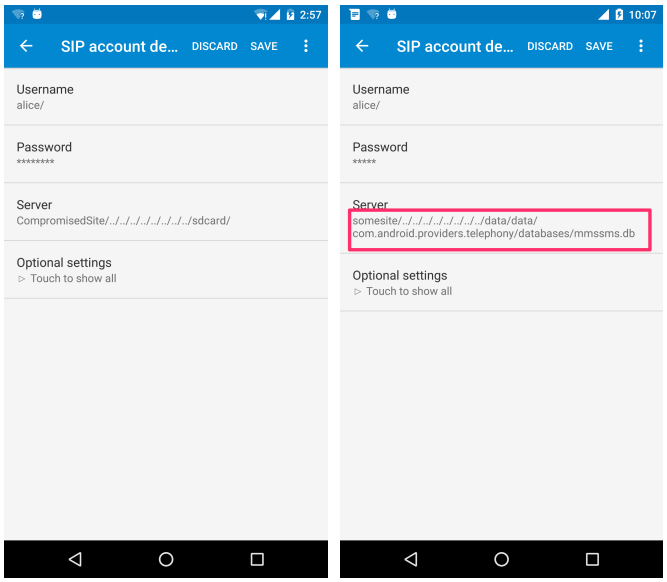
C. Discovered Vulnerabilities via Code Auditing

In this subsection, we present the vulnerabilities that are dedicatedly discovered by our targeted code auditing. Specifically, we are able to use protocol specification based auditing to discover these vulnerabilities, since their root causes are the inconsistency between VoIP's specification and Android's traditional phone call processing.

V7: Data leak and permanent DoS due to path traversal. In this vulnerability, we exploit the inconsistency between

```
File f = new File(mProfileDirectory + p.getProfileName())
File f = new File(new File(root, name), ".pobj")
```

Listing 5: Simplified vulnerable code of path traversal in V7.



(a) Leaking data to SD card. (b) Causing permanent DoS.

Fig. 10: Demo screenshots of exploiting the vulnerability V7.

SIP URI and Android/Linux file directory. Specifically, SIP URI treats ".." and "/" as normal characters, whereas they are special characters in the Android's file name convention. As a result, a path traversal vulnerability appears in the code shown in Listing 5. The directory that contains the serialized ".pobj" SIP profile file is named in this format: "sip_user@server_ip", e.g., "alice@171.11.160.202". An attacker thus can misuse these two names to manipulate the path of `mProfileDirectory`. For example, by physically setting "sip_user" and "server_ip" in the format of Fig. 10(a), `mProfileDirectory` becomes "/data/data/-com.android.phone/files/alice/@SomeSite/../../../../../../sdcard/" and leaks the sensitive SIP profile file to the public SD card. A permanent DoS could also happen if "server_ip" is set to overwrite another system app's file, e.g., mmssms.db shown in Fig. 10(b). Due to this fake mmssms.db file, the real one cannot be created and thus deny any SMS functionality. Only a factory reset can recover the phone.

V8: Caller ID spoofing due to mis-parsing "&". The last two vulnerabilities, V8 and V9, are due to the inconsistency between SIP URI and PSTN (Public Switched Telephone Network) number format. In vulnerability V8, it is related to a special character "&" in the caller number. For a caller number with "&", the system dialer app treats the number before "&" as the actual calling number and the number after "&" as the call transfer number, according to PSTN's convention. However, the dialer does not consider an incoming VoIP call and performs the same for a VoIP call number. As a result, an adversary can mimic any phone number by simply adding a "&" character in the end, causing a caller ID spoofing attack. For example, the attacker can mimic the emergency number by setting the SIP name as "911&", as shown in Fig. 11(a). He

(a) Spoofing as an emergent number. (b) Spoofing as a contact number.

Fig. 11: Demo screenshots of exploiting the vulnerability V8.

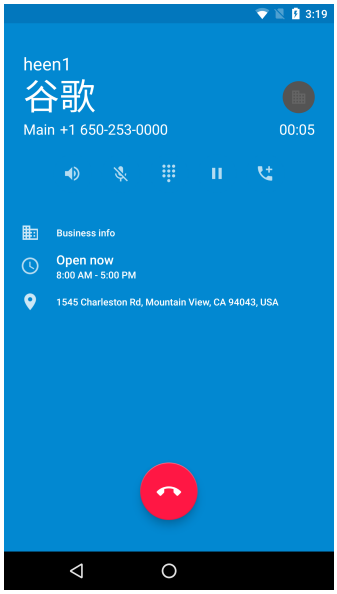


Fig. 12: A demo screenshot of exploiting the vulnerability V9.

can also spoof as a contact number of the victim if the attacker knows the number, and the dialer will display the name and profile photo of the spoofed contact, as shown in Fig. 11(b).

V9: Caller ID spoofing due to "phone-context". Another inconsistency between SIP URI and PSTN number format is the "phone-context" parameter [10], which can be used to specify the prefix of a phone number. For example, in PSTN's convention, the number "650253000;phone-context=+1" is equivalent to "+1650253000", where the value of "phone-context" becomes the prefix of the number. However, such convention should not apply to VoIP calls, which is unfortunately ignored by the dialer app. As a result, an adversary can intentionally set the caller number as "650253000;phone-context=+1", and the dialer app will interpret it as "+1650253000" and display it as Google's call, which is clearly presented in Fig. 12. Note that such mapping from "+1650253000" to Google is automatically performed by Android's CallerID mechanism [11], which tries to correlate well-known phone numbers or mark spam numbers in the normal scenario. But here it worsens the severity instead.

VI. A New Root Cause

Besides the vulnerability-level cause analysis in Sec. V, we try to uncover the root causes underneath those vulnerabilities. Among the nine vulnerabilities we discovered, three of them have previously known root causes, i.e., no protection of exported components in V1 [26], [30], no checking of system APIs in V2 [15], [32], and missed error handling in V4 [42]. For the rest of six vulnerabilities, we identify a new root cause that is dedicated to Android VoIP and not known before.

We call this root cause "incompatible processing between VoIP and PSTN calls". Specifically, since both VoIP calls and traditional PSTN calls are handled by the Android telephony system, there exist some incompatible processing behaviors between VoIP and PSTN calls. Such incompatibility is the root cause of six VoIP vulnerabilities we identified, as summarized in Table II. Understanding these incompatible behaviors and other potential incompatibility between VoIP and PSTN calls can help us further improve Android VoIP security. We thus call for VoIP developers' extra attention in their future design and implementation.

TABLE II: Incompatible behaviors between VoIP and PSTN calls.

ID	Attribute	Incompatible behaviors	
V3	Number length	513+ bytes in SIP URI	$\leq$ 513 in PSTN no.
V5	Number length	513+ bytes in SIP URI	$\leq$ 513 in PSTN no.
V6	Number length	513+ bytes in SIP URI	$\leq$ 513 in PSTN no.
V7	"../" character	Part of SIP URI	Parent dir in Linux
V8	"&" character	Part of SIP URI	Call transfer in PSTN
V9	"phone-context"	Part of SIP URI	Prefix for PSTN no.

VII. Related Work

In this section, we present the closely related research on VoIP security, protocol fuzzing, and Android dynamic testing.

VoIP security. There were some research [19], [20], [27], [28], [31] to explore the general security issues of VoIP, e.g., denial of service, eavesdropping, and call hijacking, since over ten years ago. In particular, the VOIPSA organization gave a clear taxonomy [14] of VoIP's threats. Recently, with the high popularity of Android phones and mobile networks, researchers started to investigate the security of VoIP apps and network infrastructure in the real world. They have identified the privacy risks in some VoIP apps [17], [23] and infrastructure vulnerabilities in several mobile carriers [29], [43]. Compared with these works, we are the first to systematically study the security of system-level VoIP implementation on Android, with 8 zero-day vulnerabilities identified and confirmed by Google.

Protocol fuzzing. Our network-side fuzzing in Sec. IV-B belongs to the category of network protocol fuzzing. In the classic book of *Fuzzing: Brute Force Vulnerability Discovery* [35], the authors explained the network protocol fuzzing on both Windows and Unix. SNOOZE [18] and Prospex [22] are two pioneer systems for stateful network protocol fuzzing. AutoFuzz [25] is an open-source network protocol fuzzing framework. There are also some fuzzers specific to certain protocols, such as for OPC protocol [37] and TLS libraries [24], [33]. Moreover, KiF [16] is a dedicated SIP fuzzer that was released in 2007, but unfortunately, it does not apply to Android phones. In this paper, our network-side fuzzing tool is the first Android VoIP fuzzer for SIP, SDP, and RTP fuzzing.

Android dynamic testing. Our on-device fuzzing in Sec. IV-A is related to the general Android dynamic testing [34], [36], [38], [39], [41]. Specifically, SMV-Hunter [34] and FileCross [38] are two representative systems, both of which leveraged Android `adb` commands to dynamically test Android apps' security vulnerabilities. The closest work to our Intent fuzzing is IntentFuzzer [44], which also leveraged Drozer for Intent fuzzing. The difference is that our fuzzing targets at VoIP components, instead of the permission-protected components in IntentFuzzer [44]. Additionally, buzzer (Binder Fuzzer) [21] analyzed input validation vulnerabilities associated with Android system services, which is similar to our System API fuzzing except that we use Java reflection to effectively identify service interfaces and their parameters. Furthermore, our on-device fuzzing is an unified framework that performs both Intent and System API fuzzing.

VIII. Conclusion

In this paper, we conducted the first study to systematically investigate the (in)security of Android's VoIP integration at the system level. We began with a demystification of Android VoIP's protocol stack and all its four attack surfaces. We then proposed a novel vulnerability assessment approach that first employs on-device Intent/API fuzzing and network-side packet fuzzing to automatically test Android VoIP components, and further combines them with targeted code auditing to eventually determine vulnerabilities. By fuzzing VoIP components on recent Android OS from version 7.0 to 9.0, we discovered a total of nine zero-day vulnerabilities, two-thirds of which can be exploited by a network-side adversary. These vulnerabilities caused serious security consequences, including denying voice calls, caller ID spoofing, unauthorized call operations, and remote code execution. Finally, we uncovered a new root cause, incompatible processing between VoIP and PSTN calls, that leads to six VoIP vulnerabilities and requires developers' extra attention in their future design and implementation.

References

[1] "Android VoLTE settings by T-Mobile Support," https://support.t-mobile.com/docs/DOC-22754.

[2] "Drozer: Comprehensive security and attack framework for Android," https://labs.mwrinfosecurity.com/tools/drozer/.

[3] "Ethercap," https://www.ettercap-project.org/.

[4] "How to Enable Wi-Fi Calling on an Android Phone," https://www.howtogeek.com/234608/how-to-enable-wi-fi-calling-on-an-android-phone/.

[5] "Login CSRF - Detectify Knowledge Base," https://support.detectify.com/customer/portal/articles/1969819.

[6] "mjSIP," http://www.mjsip.org/mjua.html.

[7] "OpenSIPS," https://opensips.org/.

[8] "RFC 3261 - SIP: Session Initiation Protocol," https://tools.ietf.org/html/rfc3261.

[9] "RFC 3550 - RTP: A Transport Protocol for Real-Time Applications," https://tools.ietf.org/html/rfc3550.

[10] "RFC 3966 - The tel URI for Telephone Numbers," https://tools.ietf.org/html/rfc3966.

[11] "Use caller ID & spam protection," https://support.google.com/phoneapp/answer/3459196?hl=en.

[12] "VoLTE support in UK's networks," https://www.4g.co.uk/what-is-volte/.

[13] "What Is an SMS Bomber?" https://www.techwalla.com/articles/what-is-an-sms-bomber.

[14] "VoIP Security and Privacy Threat Taxonomy," https://www.voipsa.org/Activities/VOIPSA_Threat_Taxonomy_0.1.pdf, 2005.

[15] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "AceDroid: Normalizing diverse Android access control checks for inconsistency detection," in *Proc. ISOC NDSS*, 2018.

[16] H. J. Abdelnur, R. State, and O. Festor, "Kif: A stateful SIP fuzzer," in *Proc. ISOC IPTComm*, 2007.

[17] A. Azfar, K.-K. R. Choo, and L. Liu, "Android mobile VoIP apps: a survey and examination of their security and privacy," *Springer Electronic Commerce Research*, vol. 16, no. 1, 2016.

[18] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *Proc. Springer Information Security Conference (ISC)*, 2006.

[19] R. Birke, M. Mellia, and M. Petracca, "Understanding VoIP from backbone measurements," in *Proc. IEEE INFOCOM*, 2007.

[20] D. Butcher, X. Li, and J. Guo, "Security challenge and defense in VoIP infrastructures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 37, no. 6, 2007.

[21] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with Android system services," in *Proc. ACM ACSAC*, 2015.

[22] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proc. IEEE Symposium on Security and Privacy*, 2009.

[23] T. Dargahi, A. Dehghantanha, and M. Conti, "Forensics analysis of Android mobile VoIP apps," *Contemporary Digital Forensic Investigations of Cloud and Mobile Applications*, vol. Chapter 2, 2017.

[24] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proc. USENIX Security*, 2015.

[25] S. Gorbunov and A. Rosenbloom, "AutoFuzz: Automated network protocol fuzzing framework," *International Journal of Computer Science and Network Security*, vol. 10, no. 8, 2010.

[26] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. ISOC NDSS*, 2012.

[27] A. D. Keromytis, "Voice-over-IP security: Research and practice," *IEEE Security & Privacy*, vol. 8, no. 2, 2010.

[28] ——, "A comprehensive survey of voice over IP security research," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 2, 2012.

[29] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang, "Insecurity of voice solution VoLTE in LTE mobile networks," in *Proc. ACM CCS*, 2015.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM CCS*, 2012.

[31] S. McGann and D. C. Sicker, "An analysis of security threats and tools in SIP-based VoIP systems," in *Second VoIP Security Workshop*, 2005.

[32] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao, "Kratos: Discovering inconsistent security policy enforcement in the Android framework," in *Proc. ISOC NDSS*, 2016.

[33] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proc. ACM CCS*, 2016.

[34] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *Proc. ISOC NDSS*, 2014.

[35] M. Sutton, A. Greene, and P. Amini, "Fuzzing: Brute force vulnerability discovery," 2007.

[36] X. Tang, Y. Lin, D. Wu, and D. Gao, "Towards dynamically monitoring Android applications on non-rooted devices in the wild," in *Proc. ACM WiSec*, 2018.

[37] T. Wang, Q. Xiong, H. Gao, Y. Peng, Z. Dai, and S. Yi, "Design and implementation of fuzzing technology for OPC protocol," in *Proc. International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 2013.

[38] D. Wu and R. K. C. Chang, "Analyzing Android Browser Apps for file:// Vulnerabilities," in *Proc. Springer Information Security Conference (ISC)*, 2014.

[39] ——, "Indirect file leaks in mobile applications," in *Proc. IEEE Mobile Security Technologies (MoST)*, 2015.

[40] D. Wu, Y. Cheng, D. Gao, Y. Li, and R. H. Deng, "SCLib: A practical and lightweight defense against component hijacking in Android applications," in *Proc. ACM CODASPY*, 2018.

[41] D. Wu, D. Gao, R. K. C. Chang, E. He, E. K. T. Cheng, and R. H. Deng, "Understanding open ports in Android applications: Discovery, diagnosis, and security assessment," in *Proc. ISOC NDSS*, 2019.

[42] D. Wu, D. Gao, E. K. T. Cheng, Y. Cao, J. Jiang, and R. H. Deng, "Towards understanding Android system vulnerabilities: Techniques and insights," in *Proc. ACM AsiaCCS*, 2019.

[43] T. Xie, G.-H. Tu, C.-Y. Li, C. Peng, J. Li, and M. Zhang, "The dark side of operational Wi-Fi calling services," in *Proc. IEEE Conference on Communications and Network Security (CNS)*, 2018.

[44] K. Yang, L. Zhou, Y. Wang, J. Zhuge, and H. Duan, "IntentFuzzer: Detecting capability leaks of Android applications," in *Proc. ACM AsiaCCS*, 2014.