Nearby Threats: Reversing, Analyzing, and Attacking Google's 'Nearby Connections' on Android

***Abstract*—Google's Nearby Connections API enables any Android (and Android Things) application to provide proximity-based services to its users, regardless of their network connectivity. The API uses Bluetooth BR/EDR, Bluetooth LE and Wi-Fi to let "nearby" clients (discoverers) and servers (advertisers) connect and exchange different types of payloads. The implementation of the API is proprietary, closed-source and obfuscated. The updates of the API are automatically installed by Google across different versions of Android, without user interaction. Little is known publicly about the security guarantees offered by the API, even though it presents a significant attack surface.**

In this work we present the first security analysis of the Google's Nearby Connections API, based on reverse-engineering of its Android implementation. We discover and implement several attacks grouped into two families: connection manipulation (CMA) and range extension attacks (REA). CMA-attacks allow an attacker to insert himself as a man-in-the-middle and manipulate connections (even unrelated to nearby), and to tamper with the victim's interface and network configuration. REA-attacks allow an attacker to tunnel any nearby connection to remote locations, even between two honest devices. Our attacks are enabled by REArby, a toolkit we developed while reversing the API implementation. REArby includes a dynamic binary instrumenter, a packet dissector, and the implementations of custom Nearby Connections client and server. We plan to open-source REArby after a responsible disclosure period[1].

I. Introduction

Google's Nearby Connections API enables Android (and Android Things) developers to offer proximity-based services in their applications. A proximity-based service allows users of the same application to share (sensitive) data only if they are "nearby", e.g., within Bluetooth radio range. The API uses Bluetooth BR/EDR, Bluetooth LE and Wi-Fi and it claims to automatically use the best features of each depending on the type of communication required. For example, it uses Bluetooth for short-range low-latency communications and Wi-Fi for medium-range high-bandwidth ones. The API provides two different connection strategies (`P2P_STAR` and `P2P_CLUSTER`), that allows clients (discoverers) and servers (advertisers) to be connected using different topologies.

[1]In parallel to this submission, we will disclose our results to Google.

NDSS '19, 24-27 February 2019, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23xxx

The Nearby Connections API is implemented as part of Google Play Services. Google Play Services is a proprietary, closed-source and obfuscated library that allows Google to provide the same services to any Android and Android Things application, regardless of the version of the operating systems. The API is compatible with any Android device, version 4.0 or greater, and it is updated by Google without user interaction [31]. An attacker who can exploit this API can target any application using Nearby Connections in any Android mobile and IoT device. This implies a large attacker surface and represents a critical threat with severe consequences such as data loss, automatic spread of malware, and distributed denial of service.

The design specifications and implementation details of the Nearby Connections API are not publicly available. The only public source of information about the library is a few blog posts detailing sporadic security guarantees [17], [15]. According to these sources, the API uses encryption by default, but it does not mandate user authentication. The API automatically manages multiple physical layers and a device can simultaneously be a client and a server, and can connect to multiple applications at the same time. A Nearby Connections application is uniquely identified by a string named `serviceId` and clients and servers with different `serviceId` (or connection strategies) will not be able to connect.

Proximity-based services similar to the Nearby Connections API have been investigated in the context of wireless sensor networks for a long time, together with related security challenges [27], [32], [26]. In particular, eavesdropping and wormhole attacks are often discussed, which would allow the attacker to read or manipulate the traffic exchanged by nearby devices. Typically, such attacks are considered on link or network layer, i.e., on the routing or path finding protocols. As Nearby Connections is providing an application-layer service, the setting is different to most established work in the field, although similar security challenges does apply. In addition to attacks on routing, authentication of (mobile) users is known to be challenging, together with key exchange to establish a secure channel [8], [28].

In this work, we assess the security of the Nearby Connections API. We analyze the API by reverse-engineering its closed-source and obfuscated implementation. To perform this task we use advanced techniques such as dynamic binary instrumentation and manipulations of raw packets. We develop compatible implementations of Nearby Connections client and server. Based on the knowledge gained, we identify and implement a range of attacks. The impact of the attack ranges from man-in-the-middle interception and decryption of application-layer traffic, to triggering outgoing TCP connections to arbitrary targets. In addition, the attacker is able to introduce

a new (system wide) default route towards an access point under the his control. Hence, the attacker gains access to all the Wi-Fi traffic of the victim, including the traffic generated by applications that are not using the Nearby Connections API.

We summarize our main contributions as follows:

- We reverse engineer and perform the first security analysis of the closed-source and obfuscated Nearby Connections API.
- We identify and perform several attacks grouped into two families: connection manipulation and range extension attacks. The attacks can be performed by very weak adversaries and have severe consequences such as remote connection manipulation and data loss.
- We design and implement REArby, a toolkit that enables reverse engineering and attacking the Nearby Connections API. We plan to release it as open source after disclosure to Google.

Our work is organized as follows: in Section II, we introduce the Nearby Connections API. We present a security analysis of the API based on our reverse engineering in Section III. In Section IV, we present the connection manipulation and range extension attacks. The implementation details of REArby are discussed in Section V. We present the related work in Section VI. We conclude the paper in Section VII.

II. Background

A. Introduction to the Nearby Connections API

The Nearby Connections API is used to develop *proximity-based* applications. These applications provide services to users within radio range (approx $100\,\mathrm{m}$) [16] and consider these users as "nearby". Typical use cases of the API are: file sharing, gaming, and streaming of content. The API enables proximity-based services using a combination of three wireless technologies: Bluetooth BR/EDR, Bluetooth LE and Wi-Fi. Bluetooth BR/EDR stands for basic rate and extended data rate and it is typically used by high-end mobile devices. Bluetooth LE stands for low energy and it is typically used by low-end and high-end mobile devices [6]. In the rest of the paper we indicate Bluetooth BR/EDR with Bluetooth and Bluetooth LE with LE. For more information about the differences between the two refer to [6].

The Nearby Connections API is available on Android and Android Things. Android Things is a new operating system based on Android developed by Google that targets IoT devices. In this work, we focus on the Android implementation of the API. The latest major release of the API was introduced in June 2017 in Google Play Services (GPS) version 11.0 [17]. The GPS library is a core proprietary product of Google, and relies partly on the *security through obscurity* model. The main functional benefit and potential security weakness of the GPS library is that it allows its services (including Nearby Connections) to be usable by any application, across different Android versions from 4.0 onwards. The updates of the GPS library are pushed by Google and do not require user interaction to complete [9].

In a nearby connection there are two types of actor: the *discoverer* and the *advertiser*. The former acts as a client, while the latter acts as a server. Figure 1 describes the actions performed by these actors while using the Nearby Connections API. The client attempts to discover a service identified by a `serviceId`. The server announces the service (`serviceId`) along with a name (`ncname`), using one of two different strategies described in a moment. Two actors are allowed to connect if they use the same strategy and `serviceId`. A single device can discover and advertise different services at the same time. Each `serviceId` is meant to uniquely identify an application. Google suggests to set it equal to the package name of the application [18].

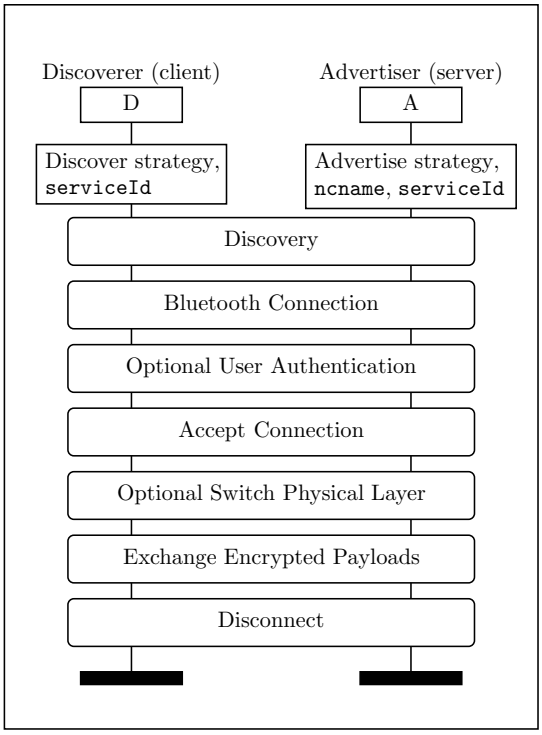

Fig. 1: The Nearby Connections API has two types of actors: the Discoverer (client) and the Advertiser (server). It uses application layer encryption and optional user authentication.

When the client discovers the server it requests a connection to it and the actors can optionally authenticate themselves. The actors mutually accept to connect and then the connection is established. The connection is *always* requested, initiated and established over Bluetooth. Once the connection is established, the actors optionally switch to a different physical layer, e.g., to Wi-Fi, and then they start exchanging encrypted payloads. The API provides three types of payloads: `BYTE`, `FILE`, and `STREAM`. The first type is used to transmit chunks of bytes, the second files and the third streams of data. Each payload type has a related proximity-based service associated, e.g., use `FILE` payloads in a file-sharing application. Each actor addresses a payload to a receiver with a unique four-digit string called `endpointId`. The nearby connection is closed whenever one of the two actor disconnects.

B. Nearby Connections Strategies

The nearby connection strategy dictates the connection topology and the physical layer switch. At the time of

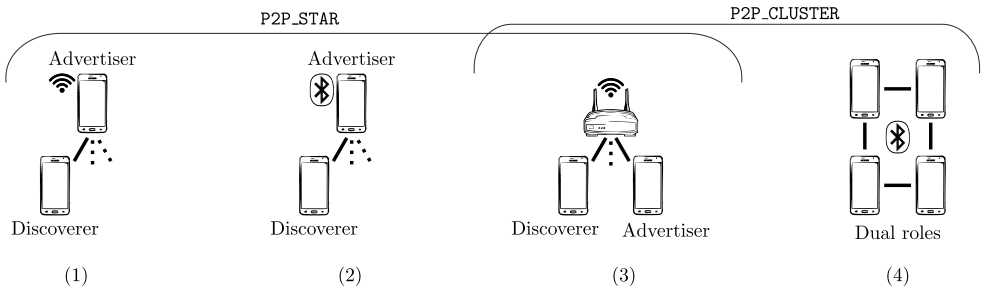

Fig. 2: Nearby Connections topology options for additional links after Bluetooth-based connection establishment. On the left, three `P2P_STAR` topologies; on the right, two `P2P_CLUSTER` topologies. In each case, all actors use the same `serviceId`.

writing, the Nearby Connections API provides two strategies: `P2P_STAR` and `P2P_CLUSTER`. A third one called `P2P_POINT_TO_POINT` can be found in the source code, but it is not public yet. In Figure 2 we show three examples of `P2P_STAR` links and two `P2P_CLUSTER` links. These links are established after the actors already completed all the phases from Figure 1, up to the optional physical layer switch. The `P2P_STAR` strategy has three types of links: (1) the advertiser acts as a soft access point and the discoverer connects to it; (2) the discoverer is the master and the advertiser is the slave of a Bluetooth network; and (3) both actors are connected to the same access point and they exchange payloads through it. There are only two options for the `P2P_CLUSTER` strategy: (3) is the same as for the `P2P_STAR` strategy and (4) several actors can connect to each other forming a mesh-like network. Hence, `P2P_CLUSTER` allows the connection of multiple discoverers and advertisers while `P2P_STAR` allows only one advertiser to be connected with multiple discoverers. Google recommends to use `P2P_STAR` for higher throughput and `P2P_CLUSTER` for more flexible network topologies. The actors do not have to communicate with a remote third party, and can exchange payloads without being connected to the Internet. Finally, two discoverers always communicate through an advertiser, even when using the `P2P_CLUSTER` strategy.

III. Reversing and Analyzing Nearby Connections

Our main goal is to perform a security assessment the Nearby Connections API because of its wide attack surface and usage of multiple wireless technologies. Unfortunately, the implementation of the API is proprietary, closed-source, and obfuscated. To overcome these obstacles we developed REArby a toolkit to reverse engineer the API. The implementation of REArby is presented Section V.

In this section we describe our understanding of the Nearby Connections API after reverse engineering its implementation on Android. For the remainder of this work, we refer to the target of our analysis as "the library". The analysis of the library allowed us to identify and perform several attacks that we present in Section IV. In particular, details on authentication and interactions between servers and clients (Section III-A) are used by us later to perform attacks in which we impersonate devices, and manipulate Nearby Connections traffic. Details of the keep-alive mechanism (Section III-E) are required for range extension attacks. The physical layer switch (Section III-F) is exploited later to manipulate system-wide routing tables of victims.

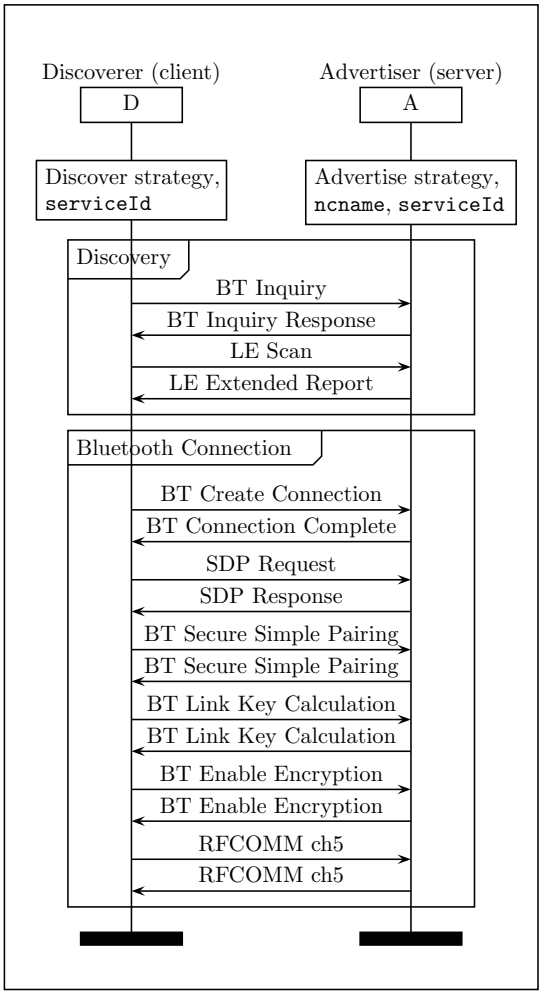

Fig. 3: Nearby Connections Request. BT is Bluetooth BR/EDR. Secure Simple Pairing provides link-layer encryption.

We note that, without access to the source code or specification of the library, we cannot claim that our findings are always complete.

A. Connection Request

A nearby connection is always requested using Bluetooth, regardless of the nearby connection parameters. Discovering and advertising are done in a deterministic and predictable way,

they are not bound to the Android application that is using the library, and they do not use encryption. In Section IV, we will show how to use this to impersonate Nearby Connections devices. The Bluetooth connection uses Secure Simple Pairing (SSP) with a link key that is not authenticated and not persistent.

An example of a nearby connection request is shown in Figure 3. The advertiser (server) advertises a strategy, a `serviceId` and a `ncname`. The discoverer (client) discovers a strategy and a `serviceId`. To find each other, both have to look for the same `serviceId`, and use the same strategy. The server to be discovered changes its Bluetooth name (btname) and sets custom LE extended report. The btname is changed to a string that depends on the strategy, the `endpointId`, the `serviceId`, and the `ncname`. btname is computed as follows:

btname = unpad(b64encode(strategy_code ‖ `endpointId` ‖ SHA256(`serviceId`)[:3] ‖ separator ‖ `ncname`))

The strategy_code is `0x21` for `P2P_STAR` and `0x22` for `P2P_CLUSTER`. The SHA256(`serviceId`)[:3] are the first three bytes of the SHA256 digest of the `serviceId`. The unpad function is used to remove the padding characters (=) from the base64 encoded string. For example, a server with strategy `P2P_CLUSTER`, `serviceId` = `sid`, `endpointId` = `aXCV` and `ncname` = `name` advertises with the following btname: `IjR1ZEE0s2QAAAAAAAAABG5hbWU`. The length of btname depends on `ncname`, in our experiments we discovered that the maximum length of the name is 131 bytes. The btname is easy to spot because it always starts with `I` and contains the `AAAAAAAAA` separator. On the LE side, the same parameters are used in a similar way to set the LE extended report. Some devices (such as the Nexus 5) do not support LE extended reports and only use Bluetooth while advertising.

The client (while discovering) sends Bluetooth inquiries and enables LE scanning. The server sends back Bluetooth inquiry responses containing btname and LE extended reports. The client discovers the server (endpoint) through these responses and establishes a Bluetooth connection with it.

After the Bluetooth connection is established, the client sends a service discovery protocol (SDP) request using a custom `uuid`. SDP is a protocol used to discover Bluetooth services. The information about each service is obtained by sending a SDP request containing its correspondent universally unique identifier (`uuid`) [6]. The nearby connection custom `uuid` is computed from the MD5 digest of the `serviceId` and some extra string manipulations. For example, if `serviceId` = `sid` then the `uuid` is `b8c1a306-9167-347e-b503-f0daba6c5723`. The client receives an SDP response containing the following fields: `name` = `serviceId`, `host` = Bluetooth address of the server and RFCOMM `port` = 5 (among others)[2].

The client uses Secure Simple Pairing (SSP), with optional Secure Connections, to share a secret with the server. The Secure Connections mode is enabled if both radio chips support it. The client and the server compute the same link key, and they agree on enabling link-layer encryption. Finally, the client establishes a link-layer encrypted RFCOMM connection with the server, always on port 5.

[2]RFCOMM is a Serial cable emulation protocol based on ETSI TS 07.10. It is a transport layer protocol for Bluetooth providing similar guarantees of TCP [6].

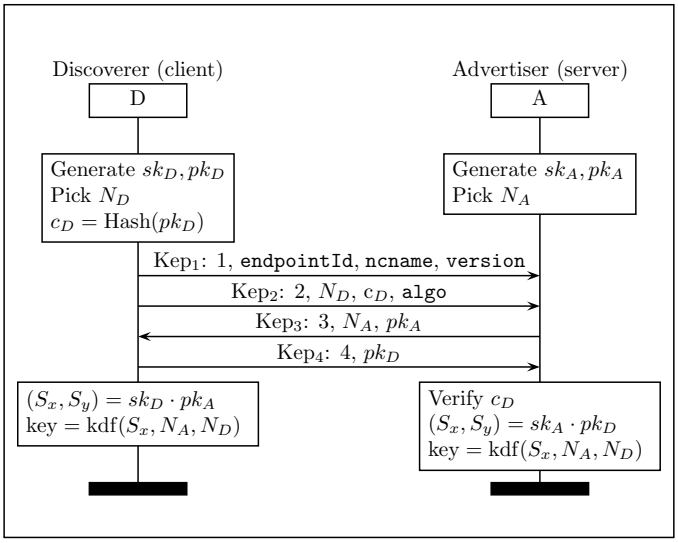

Fig. 4: Nearby Connections key exchange protocol based on ECDH (secp256r1) and Connection Initiation. `algo` is always `AES_256_CBC-HMAC_SHA256`.

B. Key Exchange Protocol (KEP) and Connection Initiation

We found that the library uses a custom key exchange protocol (KEP) based on elliptic-curve Diffie Hellman (ECDH) on the secp256r1 (NIST P-256) curve. ECDH is a good choice for mobile embedded system because it is faster and requires shorter keys than finite field Diffie-Hellman. The secp256r1 curve is recommended by NIST [3], however some crypto experts have questioned the security of this curve [4].

The key exchange protocol consists of four packets that we refer to as: Kep_1, Kep_2, Kep_3, and Kep_4. Table I lists their most relevant fields. This protocol provides several security guarantees, e.g., fresh shared secrets, negotiation of strong crypto primitives for confidentiality and integrity and usage of sequence numbers and sanity checks of the elliptic curve points. However, we identified a number of issues, e.g., lack of a key derivation function, possibility to use the numbers 1 and $n-1$ as private keys, weird usage of nonces and commitments, and transfer of useless key material (y coordinate of the public keys).

The key exchange protocol is shown in Figure 4. The client generates a key pair sk_D, pk_D and the server generates a key pair sk_A, pk_A. sk is the private (secret) key, and pk is the public key. Each public key is a point on the secp256r1 curve. The client and the server generate two 32 byte random nonces N_D and N_A. The client builds Kep_1 and Kep_4. The relevant fields of Kep_1 are: sequence number, `ncname`, `endpointId` and what we believe is the Nearby Connections version number (`version`). In our experiments we observed protocol version 0x2 and 0x4. The relevant field of Kep_4 are: sequence number and pk_D (the client's public key).

The client computes the SHA512 of pk_D to act as a kind of

commitment. We define it as c_D. Then, the client builds Kep_2 that has the following relevant fields: sequence number, N_D, c_D and a string that we define as `algo`. The value of `algo` is fixed to `AES_256_CBC-HMAC_SHA256`. This means that the nearby connection application layer uses encryption and message authentication codes, and that strong crypto primitives are used: AES256 in CBC mode, and HMAC with SHA256. It could also indicate future plans of introducing a negotiation feature, otherwise there seems little point in exchanging this information. The server builds Kep_3 that has the following relevant fields: sequence number, N_A, and pk_A (the server's public key). Note that Kep_4 is build before Kep_2 because the latter contains c_D (commitment) that is computed over the former.

After that, the network traffic takes place (over link layer encrypted RFCOMM). The client sends Kep_1 and Kep_2 to the server, the server answers with Kep_3, and the client sends Kep_4. In our experiments these packets are always exchanged in this order. Sometimes, Kep_1 is split and transmitted using two sequential RFCOMM packets. The server verifies the client's commitment using c_D and Kep_4. Afterwards, both nodes compute the (same) secret point in the curve, defined as (S_x, S_y), by multiplying their own private key with the public key of the other. The x coordinate of the secret point is used as key (shared secret). We refer to this as S_x. Hence, the library does not use any (recommended) ECDH key derivation functions such as HKDF or NIST-800-56-Concatenation-KDF [3]. The details about how we managed to discover this are presented in Section V. After the client and the server completed the KEP, the nearby connection is initiated (but not yet established).

C. Optional Authentication and Connection Establishment

Before establishing a nearby connection, the client and the server can *optionally* authenticate each other. The Nearby Connections authentication is based on a five-digit token, containing letters from the base64 alphabet excluding lowercase ones. The token is generated by the library, it depends on S_x (the shared key), and it is computed by the client and the server even if it is not used. It is up to the Nearby Connections application developer to decide whether and how to utilize

TABLE I: Main Fields of the Key Exchange Protocol Packets. (G_x, G_y) is the generator point for the ECDH curve.

Packet	Field	Description	Default value(s)
Kep_1	sn	Sequence number	1
	`endpointId`	Discoverer id	None
	`ncname`	Discoverer name	None
	`version`	Protocol version	0x02, 0x04
Kep_2	sn	Sequence number	2
	N_D	Nonce	Random
	c_D	Commitment	SHA512(Kep_4[4:])
	`algo`	Negotiated ciphers	`AES_256_CBC-HMAC_SHA256`
Kep_3	sn	Sequence number	3
	N_A	Nonce	Random
	x_A	x-coord of pk_A	x-coord from $(G_x, G_y) \cdot sk_A$
	y_A	y-coord of pk_A	y-coord from $(G_x, G_y) \cdot sk_A$
Kep_4	sn	Sequence number	4
	x_D	x-coord of pk_D	x-coord from $(G_x, G_y) \cdot sk_D$
	y_D	y-coord of pk_D	y-coord from $(G_x, G_y) \cdot sk_D$

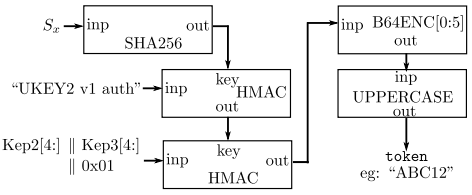

Fig. 5: Computation of the authentication token.

it. In the nearby connection sample code from Google, the authentication token is ignored by the applications [22]. In a proper application, the users would have to visually authenticate each other, e.g., they must confirm that they are seeing the same token on both screens.

The reversed procedure to generate the authentication token, defined as `token`, consists of five sequential steps. These steps are described with the help of Figure 5. First, a SHA256 hash of S_x is computed. Second, this hash is used as the key for a SHA256 HMAC, from now on HMAC, of the `UKEY2 v1 auth` string. This string reveals that the generation procedure of the token is versioned (`v1`), and it is labeled as `auth`. Third, the output of the first HMAC is used as a key in a second HMAC. The input of the second HMAC is the concatenation ($\|$) of a subset of Kep_2, a subset of Kep_3 and the integer 0x01. This means that the entropy used in the computation of `token` is fresh and comes both from the client (Kep_2) and the server (Kep_3). This choice provides security guarantees such as protection against replay attacks. In the fourth step, the output of the second HMAC is base64 encoded and truncated to its first five characters (bytes). Finally, `token` is generated by converting these five characters to uppercase. An example of `token` is `ABC12`.

The connection establishment process is agreed mutually and asynchronously: both devices have to accept the connection, and it does not matter who accepts it first. In our experiments, we observed that this process uses only link layer encryption (over RFCOMM) and constant payloads. To accept the connection, a node sends `0x0000000a0801120608021a020800` and to reject a connection it sends `0x0000000b0801120708021a0308c43e`. Pre-connection keep alive packets are exchanged between the connection initiation and establishment periods. These packets always contain `0x00000008080112040805320`0. As for the KEP and the connection initiation, Wi-Fi and Bluetooth LE are not used to establish a nearby connection. We discovered that devices with the same `ncname` are allowed to connect, this means that only the `endpointId` uniquely identifies a node.

While testing the connection establishment phase we discovered interesting things about the `serviceId`. Any advertiser leaks its `serviceId` if queried with a generic SDP request, e.g., using `sdptool browse adv_btaddress`. Moreover, two devices from different applications can use the same `serviceId` to establish a connection. This means that it is possible to predict the `serviceId` of any application. We also discovered that the library is still using an undocumented `serviceId` named `__LEGACY_SERVICE_ID__`.

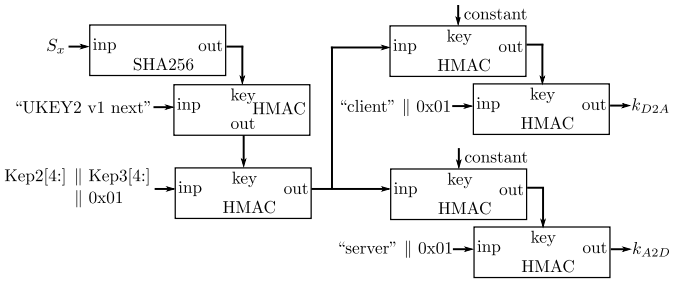

Fig. 6: Computation of k_{D2A} and k_{A2D}.

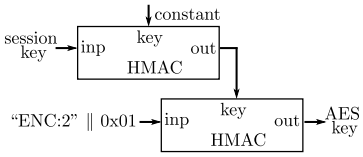

Fig. 7: Computation of the AES (symmetric) key.

D. Key Derification Functions (KDFs)

When a nearby connection is established then the client and server compute two symmetric session keys: k_{D2A} and k_{A2D}. The former is used to secure communications from the client to the server, and the latter from the server to the client. The reversed computation of these keys is shown in Figure 6. The description of the steps is similar to the one presented in Section III-C: it starts with SHA256 hashing and then it uses a chain of HMACs. We omit the detailed description of the steps, as it is similar to the one of Figure 5.

However, it is important to note four points from Figure 6. First, from the `UKEY2 v1 next` string we deduce that the library uses the same version number of the `auth` phase (see Figure 5), and it labels this phase as `next`. Second, the only thing that differentiates k_{D2A} from k_{A2D} is their last HMAC step: the former uses `client` as part of the input and the latter uses `server`. Third, k_{D2A} and k_{A2D} depend on Kep_2, Kep_3 and S_x, indeed they enjoy the same security benefits of `token`. Finally, the library uses one session key for each direction of communication, this is a good practice in protocol design, e.g., it prevents reflection attacks.

The session keys are used to derive the encryption and the message authentication code (MAC) keys. In other words, k_{D2A} and k_{A2D} generate respectively the keys to encrypt, decrypt, sign and verify packets from the client to the server and from the server to the client. Encryption is performed using AES256 in CBC mode. The generation of the AES key from a session key is a two step process (involving HMAC), which is shown in Figure 7. The string `ENC:2` indicates that this process is computing an encryption key, but it is not clear yet what does `2` refer to. The computation of the MAC key is similar to the AES one and it is shown in Figure 8. In this case, we find the `SIG:1` string. Again, the string indicates that a signing key is computed, but it is not clear what does the `1` refers to. Figure 8 contains the last step the computation of the MAC. Nearby Connections uses an *encrypt-then-mac* scheme, as the MAC is computed over a subset of the packet containing the ciphertext (ct) and the initialization vector (iv).

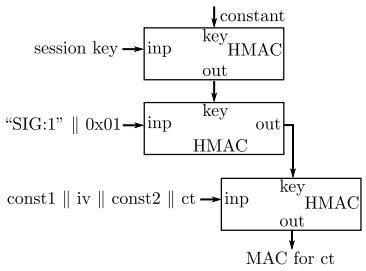

Fig. 8: Computation of the MAC key and the MAC.

Using dynamic binary instrumentation (discussed in Section V), we see that the library *always re-derives the same keys from the session keys*. Indeed, every time a node wants to transmit a packet, the library performs the following: it (re)computes the AES key, encrypts the payload, (re)computes the MAC key, signs the payload and then builds the packet. Similarly, when it is time to receive a packet, the library (re)computes the MAC key, verifies the MAC, (re)computes the AES key and decrypts the packet's payload. In our opinion, these repeated computations are not very efficient. It is possible that the library's developers intend to use a key evolution mechanism where the AES and MAC keys could change over time according to some logic. However, in our experiments the library recomputes always the same keys. Another reason to use these procedures may relate to key storage in memory. It is true that—if you recompute a key and discard it when you do not need it—the keys it stays in memory for shorter time. But the library needs the session keys k_{D2A} and k_{A2D} to be able to compute the AES and the MAC keys and these are stored in memory in any case.

E. Encrypted Keep-Alive (EKA) and Exchange of Payloads

The library uses an encrypted keep alive protocol (EKA) with a generous timeout of 30 seconds. In Section IV, we discuss how that can be exploited to connect remote victims. We now describe the details of the EKA protocol and exchange of payloads.

Once a nearby connection is established and the session keys are derived, the protocol can be considered symmetric, e.g., it does not matter who is the discoverer (client) and the advertiser (server). To emphasize this, we rename the discoverer to Dennis and the advertiser to Alice. Dennis uses k_{D2A} and Alice uses k_{A2D} and their communications are encrypted (AES256 in CBC) and authenticated (HMAC with SHA256) at the application layer, and encrypted at the link-layer (SSP).

Dennis and Alice keep the connection alive by using the encrypted keep-alive protocol shown in Figure 9. This protocol involves a specific type of packet that we define as `Eka`. These packets have a constant header and contains a (directional) counter. Dennis initializes a directional counter, that we define as c_{D2A}, to 1. c_{D2A} counts the number of packets sent from Dennis to Alice. Dennis builds an `Eka` packet and send it to Alice. Alice maintains her local c_{D2A} counter and checks that her local values match with the ones that she gets in the packets. Dennis sends an `Eka` packet every 5 seconds incrementing each time c_{D2A}. Alice may answer with either an `Eka` packet (that counts the packets in the other direction) or a packet

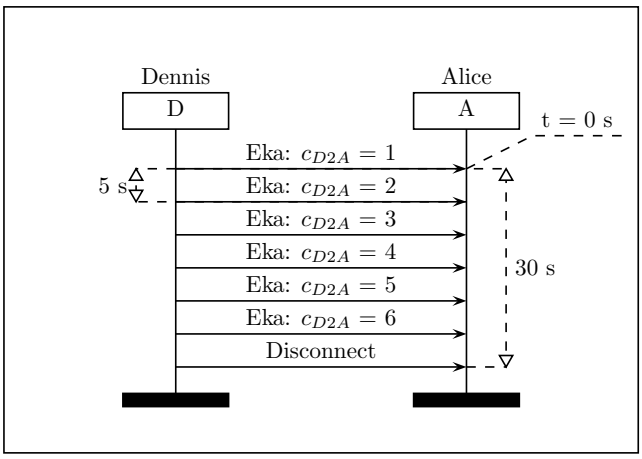

Fig. 9: Nearby Connections Encrypted Keep-Alive. The period is 5 seconds, the timeout is 30 seconds.

Algorithm 1 Nearby Connections Physical Layer Switch.

```
Require: D = discoverer, A = advertiser
Ensure: RFCOMM uses port 5 and Wi-Fi uses TCP
  if A is connected to an hotspot then
    A tells D how to switch to shared WLAN
    D contacts A over TCP
  else if strategy is P2P_STAR then
    if D and A support Wi-Fi Direct then
      A tells D how to switch to Wi-Fi Direct
    else
      A tells D how to switch to hostapd
    end if
    D connects to A's soft AP
    D contacts A over TCP
  else
    A and D continue to use Bluetooth
  end if
```

containing a nearby connection payload. Dennis closes the nearby connection after sending six sequential unanswered `Eka` packets. This means that the EKA timeout is 30 seconds. The same encrypted keep alive protocol happens asynchronously from Alice to Dennis using c_{A2D} to count the packets sent from Alice to Dennis and Dennis maintains its local c_{A2D} counter.

While the nearby connection is alive, Dennis and Alice are able to exchange payloads. There are three types of payloads: `BYTE`, `FILE`, `STREAM` and all the type are transmitted over the same transport layer. Each payload generates at least two packets and each packet contains the appropriate directional counter value (either c_{D2A} or c_{A2D}). The packets are sent sequentially without acknowledgment. A node can send and receive payloads asynchronously. Again, a payload packet contributes to keep the connection alive in the EKA protocol.

F. Connected Session and Optional Physical Layer Switch

In this section, we discuss the physical layer switch in the Nearby Connections API. This switch which will be exploited by us for connection manipulation attacks (CMA), with details on the implementation provided in Section IV.

Once a nearby connection is established, the client and the server might optionally use Wi-Fi after performing a physical layer switch. From our experiments, we see that the physical layer switch can be *predicted and manipulated*. The switch always happens from Bluetooth to one of the three Wi-Fi mode supported by the library. We define these modes as: shared WLAN, hostapd, and Wi-Fi Direct. In the shared WLAN case the devices use a common access point: see (3) and (4) in Figure 2. In the hostapd and Wi-Fi Direct case the advertiser acts as a soft AP: see (1) in Figure 2. The nearby connection documentation tells us that the library uses an heuristic in real-time to determine when and how to switch from Bluetooth to Wi-Fi.

Algorithm 1 describes the reversed logic of the physical layer switch. The advertiser is always in charge of the switch and there is no negotiation with the discoverer. In order to bind the Wi-Fi and the Bluetooth secure connections we would expect the library sharing secret material between the two physical layers. If this material exists it should be exchanged over the Bluetooth link before the switch to Wi-Fi. However, in our experiments we observed that when two connected devices switch to Wi-Fi, *they only use application layer encryption over TCP*. Hence, the shared WLAN link is cryptographically weaker than the Bluetooth link because it uses only one layer of encryption. We believe that the automatic Wi-Fi switch is enforced by `autoUpgradeBandwidth=true` (a private parameter of the library that we reversed).

The shared WLAN mode has the highest priority regardless of the nearby connection strategy. If shared WLAN is used we expect to see the exchange of network parameters over the Bluetooth link before the Wi-Fi switch. In our experiments we observed such exchange (and show how to leverage this as an attacker in Section IV). We observed that only the discoverer has to be connected to a WLAN when the devices are using the shared WLAN mode.

Wi-Fi Direct and hostapd are used only if the strategy is `P2P_STAR`. Both modes allow the advertiser to act as a soft access point (soft AP) with an `essid` but without an Internet connection. The discoverer should be able to find and connect to it. When any of these modes is in use we expect to see an exchange of information about the soft AP over Bluetooth before the Wi-Fi switch. In our experiments we observed this information, and also leverage that mechanism in our attacks. Wi-Fi Direct uses constant `essid` (22 Bytes, always starts with `DIRECT-`) and WPA2 password (8 Bytes). hostapd uses randomized base64-encoded `essid` (28 Bytes) and WPA password (12 Bytes). In both cases, the advertiser sends its `essid` and password to the discoverer. When the connection is terminated, the advertiser does not restore the original hotspot configuration of the device.

The capability of the Nearby Connections library to switch from Bluetooth to Wi-Fi has side effects that are valuable for an attacker. In Section IV, we show how to abuse the mechanism to change the state of the Wi-Fi antenna of a target device, without user intervention. For example, we were able to switch on the Bluetooth, Wi-Fi (hotspot) functionalities of a target device. This means that the library can be misused to interrupt any active Wi-Fi connection of any node by forcing a physical

layer switch to either hostapd or Wi-Fi Direct. In both cases the victim loses Internet connectivity.

IV. Attacking Nearby Connections

Based on our reverse engineering and analysis of the Nearby Connections library presented Section III, we were able to develop our custom discoverer (client) and advertiser (server). We now summarize which attacks we were able to conduct experimentally, using our tools. The implementation details of our tools are provided in Section V. We classify our attacks as two families: connection manipulation (CMA), and range extension attacks (REA). The attack families are orthogonal, and can be combined. Among other attacks, we describe an attack that allows us to inject (or replace) a system-wide IP default route of the victim, changing it to a malicious AP. As result, the attacker is able to install himself as MitM for all IP-based connections of the victim. Before discussing the attacks, we describe our threat model.

A. Threat Model

Our threat model is composed of the victims, e.g., discoverer and advertiser, and the attacker. The legitimate discoverer and the advertiser establish nearby connections as described in Section II and Section III. If an attack is effective regardless who is the discoverer and who is the advertiser, we indicate the victim and the attacker as *nodes*. Applications and libraries used by the victim are not compromised, i.e., the attacker cannot modify them directly, only interact with the victims via network connections.

The attacker has the same knowledge of the library that we describe in Section III, and access to a tool such as ours. As result, the attacker is capable of using custom advertiser and discoverer. If required, the attacker can forward traffic to the Internet, create his own Wi-Fi access point, and jam wireless links. The attacker does not require remote access to the devices of the victim(s), root privileges in his smartphones, and we do not assume that he installs any malware on the victim's device. The attacker does not require advanced instrumentations such as software-defined radio, directional antennas and commercial wireless sniffers.

The attacker has two main goals. He wants to tamper with nearby connections nodes from remote locations, e.g., establish a nearby connection between two countries. This violates the basic assumption that only devices within radio range can establish nearby connections. In addition, he wants to manipulate these connections in arbitrary ways, i.e., install himself as man-in-the-middle, take over existing connection, manipulate the state of radio hardware at victims, or avoid or weaken the security mechanisms of the involved protocols.

B. Connection Manipulation Attacks

We now discuss several connection manipulation attacks we implemented: impersonation of legitimate nodes, establishment of a wormhole-like man-in-the-middle attack on Bluetooth and Wi-Fi, forcing of a physical-layer switch, and most importantly the injection of a new OS-wide default route for the victim, pointing towards the attacker. In addition, we show how we were able to manipulate the state of radios of the victim (leading to loss of connectivity, and resource exhaustion).

A node should establish nearby connections only with trusted and appropriate nodes. However, the library presents several authentication issues. It does do not perform any of the following: authenticate the Bluetooth link key, bind the Bluetooth and the Wi-Fi physical layers, mandate user authentication, and authenticate the application that is requesting the nearby connection service. The documentation suggests that "encryption without authentication is essentially meaningless" [19] and we argue that this is the case here.

Impersonation. The library uses only the strategy and the `serviceId` to uniquely identify a nearby connection, and both are predicable. We were able to learn any `serviceId` using an SDP request, allowing us to impersonate any node requesting a service from any application just by changing our Nearby Connections strategy and `serviceId`. Based on this insight, we were able to launch such man-in-the-middle attack on any node on the Bluetooth and Wi-Fi links, regardless of the strategy.

MitM on Bluetooth. The library does not authenticate the Bluetooth link key. This allows us (as attacker) to man-in-the-middle the link by using a malicious advertiser and a malicious discoverer at the same time. The malicious advertiser gets discovered by a victim discoverer, and the malicious discoverer connected to a victim advertiser. Then, we can complete the key exchange protocol with two victims, compute the shared secrets and two pairs of session keys over Bluetooth. As result, we are able to establish two parallel sessions with the victim nodes, forward traffic between them, and observe all exchanged traffic in clear text. We note that it was even possible to connect to different benign Nearby Connections applications through such a setup.

MitM on Wi-Fi. If the advertiser requests a physical layer switch it is possible to also launch a man-in-the-middle attack on the Wi-Fi link. Assuming the attacker is connected to the same Wifi network as the two victims, we were able to use a simple ARP spoofing attack to redirect the Wifi traffic to the attacker. As the Wi-Fi link layer is not encrypted, we were then able to decrypt and tamper with all the application layer traffic using the session keys. We show later that we can achieve the same goal even if the attacker is not associated to the legitimate AP, by making the victim connect to an AP under control of the attacker (strategy has to be `P2P_STAR`).

Attacker-induced physical layer switch. Interestingly, the attacker can also manipulate the physical layer switch algorithm, regardless the nearby connection strategy. As shown in Algorithm 1, the advertiser dictates when and how to switch, and the discoverer "blindly" follows him. For example, we were able to inject packets into the Nearby Connections between the victims that caused the discoverer to establish a Wi-Fi-based (TCP/IP) connection to a destination IP and port of the choice of the attacker. Details about how we crafted the packet are presented in Section V-F. As result of our attack, the victim activates her Wi-Fi interface (if necessary), associates to a legitimate AP and establishes a TCP session with a target determined by the attacker. Note that, the target's IP can be outside of the local area network of the victim.

Injection of default route via attacker AP. If the victim's application uses the `P2P_STAR` strategy, a more serious attack

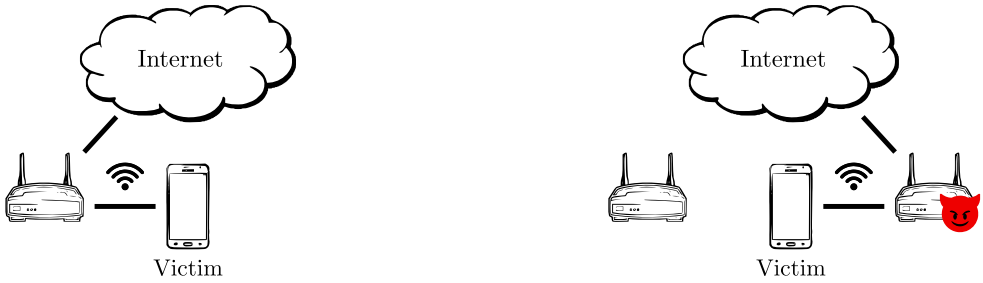

Fig. 10: Connection Manipulation Attack. On the left, the state before the attack; the victim is connected to the Internet through a benign AP. On the right, the state after the attack; the adversary has forced the victim to silently connect to a malicious access point, and inserted a new default route. The adversary will now be able intercept and manipulate any Internet-bound traffic sent by *any* application on the victim's phone. Note that this includes all applications, even if unrelated to the nearby connection application/library.

is possible. In this variant, we are able to redirect a victim to an AP under our control, rather than making the victim connect to the known (legitimate) AP. The attack works by manipulating the hostapd and Wi-Fi Direct modes of Algorithm 1.

Figure 10 shows a victim before and after the connection manipulation attack. Initially, the victim is connected to a benign AP via Wi-Fi (or not connected via Wi-Fi). The victim uses a Nearby Connections client app with `P2P_STAR` strategy. The nearby attacker impersonates an advertiser (or performs a MitM attack). Then, the attacker induces a physical-layer switch to Wi-Fi, asking the victim to connect to an AP of the attacker with provided essid and password (normally, this would be a hotspot created dynamically by the advertiser). As result, the victim will associate to the malicious AP, and configure her Wi-Fi interface using DHCP, with data supplied by the attacker. That enables the attacker to supply a new default route (along with suitable IP configurations), which will in turn redirect all traffic by the victim device to the AP of the attacker. As a result, the attacker is now able to monitor and tamper with *all* traffic coming from the victim. The traffic includes the packets generated by *other* applications that are not using the library, yet they require Internet access such as email clients, web browsers, cloud services. We understand that most of such traffic is are secured with TLS, but we believe that this attack is still serious, e.g., the attacker can learn information about the victim from encrypted traffic [36].

DoS on all victim traffic. We noted that the library does not care about the connection status of the devices before using hostapd and Wi-Fi Direct, and does not restore the original soft AP configuration of the devices. For cases where the strategy was `P2P_STAR`, we were able to leverage this to launch a denial of service attack on connectivity of a victim discoverer (and alter her list of remembered connections). We used a custom advertiser to connect to a discoverer and told the victim to use either hostapd or Wi-Fi Direct. Then, the victim connected to the our AP, enabling us to drop all traffic. This issue indirectly affects all the applications running on the victim's device that need an Internet connection to work properly [20]. Moreover, after connecting to the attacker's soft AP, the victim adds the attacker's essid and password to her list of remembered connections. This allows an attacker to overwrite legitimate entries in this list with malicious ones.

Radio state manipulation. The library is able to switch on the Bluetooth and Wi-Fi antenna of any node, and it does not switch if off after a disconnection. We were able to use this capability to switch on the antennas of a victim's device, regardless of the nearby connection strategy. For example, we were able to switch on the Bluetooth antenna of the victim by establishing a nearby connection with her, and then disconnecting. We were also able to switch on the Wi-Fi antenna of any discoverer by instructing our custom advertiser to perform a physical layer switch and then disconnect.

C. Range Extension Attacks (REA)

The Nearby Connections API is supposed to be used by devices that are "nearby". The documentation suggests that they have to be "within radio range (approx $100\,\mathrm{m}$)" [16]. However, the library lacks strict time and position requirements hence, an attacker can violate the "nearby" constraint. In particular, the library uses an encrypted keep alive protocol with a generous timeout of 30 seconds, which is more than enough to forward traffic from across continents without being noticed [12], [29]. In addition, the library does not validate locations being nearby, e.g., by comparing the locations reported by the Global Positioning System. We were able to leverage this in several ways, regardless of the strategy and whether the attacker is a custom discoverer or a custom advertiser.

In particular, the custom advertiser and custom discoverer in the MitM on Bluetooth attack described earlier do not need to be at the same location. The attacker can forward traffic between both over large distances, i.e., the Internet. This can effectively create a wormhole with two victims at distant locations. Once this wormhole connection is established, the attacker has 30 seconds to forward the packets in each direction (to keep the connection alive). The attacker could even answer to the keep alive requests himself, effectively allowing arbitrary delays. As result, this establishes a connection to devices "not nearby", without being detectable by the victims (without additional timing checks by the victims not part of the library). We argue that this attack *extends the range of a nearby connection outside of the intended bound*. This is not advisable because a victim can perceive a false sense of security given by the fact that the connection is (supposed to be) within the radio range. A victim might better trust a proximity-based secure service than a secure cloud service. This attack takes inspiration from [24],

[38], [12].

V. REArby Toolkit Implementation

To implement the attacks presented in Section IV, we require several capabilities based on our analysis of the Nearby Connections library presented in Section III. These capabilities includes: establishment of wireless connections using Bluetooth and Wi-Fi, ad-hoc usage of cryptographic primitives, manipulation of raw network packets, and usage of custom (security) protocols. For these purposes, we develop a set of tools and we group them in a project that we call REArby. REArby includes our custom discoverer and advertiser capable to perform all the nearby connection phases from Figure 1. REArby also includes a dynamic binary instrumenter, a packet dissector and a custom Android application. The project mainly uses three programming languages: Python, Java, and JavaScript and contains approximately 2000 lines of code. It requires a minimal setup, e.g., a laptop running Linux. In the rest of this section we explain how we implement all the phases of a nearby connection.

A. Connection Request

We manage all the Bluetooth operations with the `bluetooth` Python module. The discovery phase begins using the `discover_bt` function. This function returns the Bluetooth's name (btname) of all the discoverable devices that are in range. The custom discoverer detects the presence of any advertiser by looking at btnames. It extracts the strategy, `endpointId` and `ncname` using in reverse the btname formula of Section III-A. The custom advertiser starts an RFCOMM server on port 5 using `BluetoothSocket(RFCOMM)`. Then it computes the custom `uuid` based on the `serviceId` and it starts an SDP server advertising the `uuid` using `serviceId` as the name of the SDP service. The custom advertiser waits for the discoverers and manage each of them with a separate socket. All the Bluetooth connections are encrypted at the link layer using secure simple pairing (SSP).

B. Key Exchange Protocol (KEP) and Connection Initiation

To initiate a connection the custom discoverer computes the custom `uuid` (based on the `serviceId`) and performs an SDP request using that `uuid`. The response contains the `serviceId` and the Bluetooth address of the advertiser. The custom discoverer uses an RFCOMM socket to connects the advertiser on port 5. The `bluetooth`'s Python module manages the low level detail of the RFCOMM socket.

Once connected, the custom advertiser and the custom discoverer respectively complete the Nearby Connections key exchange protocol as in Figure 4. To be able to send meaningful KEP packets we perform several steps. We develop a custom Android application based on [21] and we use it to generate the KEP packets. We capture the unencrypted packets using the HCI snoop log functionality of Android. HCI is the host controller interface protocol spoken by the Bluetooth host (the OS) and the Bluetooth controller (the radio chip) [6]. We analyze the packets using scapy [5], a network analysis tool. We discover that the plaintext is serialized using a *custom* format, i.e., the library uses the `Serializable` Java class.

Listing 1 `Kep3` scapy dissection class for Kep_3.

```
class Kep3(Packet):
    name = "Kep3: adv -> dsc"
    fields_desc = [
        IntField("len1", None),
        XByteField("sep1", 0x08),
        ByteField("sn", 3),
        XByteField("NC_SEP", NC_SEP),
        ByteField("len2", None),
        StrFixedLenField("NC_HEAD2", NC_HEAD2,
         ↪ length=3),
        BitFieldLenField("nA_len", None, size=8,
         ↪ length_of="nA"),
        StrLenField("nA", "", length_from=lambda
         ↪ pkt:pkt.nA_len),
        StrFixedLenField("NC_KEP3_HEAD",
         ↪ NC_KEP3_HEAD, length=3),
        ByteField("len3", None),
        StrFixedLenField("NC_HEAD2", NC_HEAD2,
         ↪ length=3),
        ByteField("len4", None),
        XByteField("NEWLINE", NEWLINE),
        BitFieldLenField("xA_len", None, size=8,
         ↪ length_of="xA"),
        StrLenField("xA", "", length_from=lambda
         ↪ pkt:pkt.xA_len),
        XByteField("NC_SEP", NC_SEP),
        BitFieldLenField("yA_len", None, size=8,
         ↪ length_of="yA"),
        StrLenField("yA", "", length_from=lambda
         ↪ pkt:pkt.yA_len),
```

Listing 1 shows the `Kep3` Scapy dissection class that we use to decode the serialized data in the Kep_3 packets. Kep_3 is sent from the advertiser to the discoverer and it contains four relevant fields: the sequence number (`sn`), a nonce (`nA`), the coordinates of the public key of the advertiser (`xA`, `yA`). These values are serialized using variable length fields. A variable length field has a leading Byte containing the length of the field in Bytes concatenated with the actual Bytes containing the value of the field. For example, `nA_len` indicates that `nA` is a 32 Byte nonce and its value is referenced by `nA`. The same holds for `xA` and `yA`. We use similar classes to decode Kep_1 Kep_2, and Kep_4.

From the decoded KEP packet we realize that the library uses ECDH on secp256r1, by testing the public keys contained in Kep_3 and Kep_4 on standard ECDH curves. The next task is to correctly compute the shared secret (S_x). To understand how to compute it we use *dynamic binary instrumentation (DBI)* DBI allows to monitor a target application (process) in real-time by attaching a monitoring process to it. To implement our DBI we use Frida [33] a reverse-engineering toolkit that has native compatibility with Android. After observing our Android application generating ECDH shared secrets we find out that the cryptographic operations are managed by a separate Android process called `com.google.android.gms.nearby.connection` that we indicate with `ncproc`.

Using Frida we list all the Java classes and shared libraries of `ncproc` and isolate all the security related classes, methods and functions By monitoring the `OpenSSLECDHKeyAgreement` class we discover that the library is only using the x coordinate of S_x as the key, i.e., it is not using a key derivation function. This information

Listing 2 Frida (JavaScript API) function to overload `android.util.Base64.encodeToString`.

```
1  // input is a byte[], return value is a String
2  function Base64_encodeToString() {
3    Java.perform(function () {
4      var target = Java.use("android.util.Base64");
5          target.encodeToString.overload('[B',
           ↪ 'int').implementation = function(inp,
           ↪ flags) {
6        B64ENC_COUNT += 1
7
8        print_backtrace();
9
10       var inp_str = JSON.stringify(inp)
11       console.warn("B64ENC " + B64ENC_COUNT + "
          ↪ inp: " + inp_str)
12
13       var retval = this.encodeToString(inp,
          ↪ flags);
14       console.warn("B64ENC " + B64ENC_COUNT + "
          ↪ out: " + retval)
15
16       return retval
17     };
18   });
19 }
```

Listing 3 Backtrace of the `ncproc` crypto stack. Long lines are truncated with three dots (...). The library is using javax.crypto that calls Conscrypt that calls BoringSSL.

```
1  at com.google.android.gms.org.conscrypt...
2  at com.google.android.gms.org.conscrypt...
3  at com.google.android.gms.org.conscrypt...
4  at com.google.android.gms.org.conscrypt...
5  at com.google.android.gms.org.conscrypt...
6  at javax.crypto.Cipher.doFinal(Cipher.java:1502)
7  at blah.a(:com.google.android.gms...
8  at blam.a(:com.google.android.gms...
9  at blam.a(:com.google.android.gms...
10 at bkyj.a(:com.google.android.gms...
11 at bkyg.b(:com.google.android.gms...
12 at acxt.c(:com.google.android.gms...
13 at acyg.a(:com.google.android.gms...
14 at acyd.run(:com.google.android.gms...
15 at pmz.run(:com.google.android.gms...
16 at java.util.concurrent.ThreadPoolExecutor...
17 at java.util.concurrent.ThreadPoolExecutor...
18 at ptb.run(:com.google.android.gms...
19 at java.lang.Thread.run(Thread.java:818)
```

enables us to initiate a Nearby Connections by implementing the Nearby Connections KEP protocol from Figure 4. We use the python's `cryptography` module to implement all the cryptographic protocols. Note that, our setup allows us to modify and fuzz each field of every KEP packet, e.g., public keys, `algo`, `version`, `endpointId` and the nonces.

C. Optional Authentication and Connection Establishment

After the nearby connection is initiated we compute `token` to perform the optional user authentication phase. Listing 2 shows how we monitor the `encodeToString` method of the `android.util.Base64` class [10]. This method is called in the last step of the `token` computation from Figure 5. `encodeToString` takes an array of bytes as input and returns its base64 encoding representation as a string. The important lines of code are from 6 to 16. In line 6, we use `B64ENC_COUNT` to count how many times this method is called at runtime. In line 8 we use `print_backtrace` to (recursively) see who called the method using which parameters. In line 10-11 we save the original input of the method as a string in `inp_str`, and we print it on our console. The original method is called in line 13 with its original inputs (`inp`, `flags`) and its return value is saved into `retval`. In line 14-16, we print the return value and return the control to the `ncproc` process.

We use functions similar to Listing 2 to monitor reconstruct the computation of `token` from Figure 5. We implement its computation in our custom advertiser and discoverer using standard python's modules. To establish the connection we implement the pre-connection keep alive, the connection acceptance and rejection reusing the constant payloads mentioned in Section III-C.

D. Key Derivation Functions

Our dynamic binary instrumentation setup is quite powerful. It allows us to observe, tamper with, and reimplement every method call used by *any* Android process, such as `ncproc`. All of this without having access to the implementation of the Nearby Connections library. Using our DBI we reconstruct all the key derivation functions presented in Section III-D, and we implement them using standard python modules. This enables our custom discoverer and advertiser to establish a nearby connection and compute the correct session (Figure 6), encryption (Figure 7), and MAC keys (Figure 8).

E. Connection Keep-Alive and Exchange of Payloads

An established nearby connection is kept alive using the encrypted keep-alive (EKA) protocol from Figure 9. The EKA protocol requires the knowledge of the cryptographic stack used to encrypt-then-mac the application layer packets and the capability to build meaningful application layer packets (with directional counters) as explained in Section III-E. Implementing the EKA protocol allows our custom discoverer and advertiser to perform the range extension attacks presented in Section IV.

We reverse the cryptographic stack of the library by looking at the backtrace of its lowest level cryptographic methods such as `NativeCrypto_EVP_CipherFinal_ex()`. Listing 3 shows the backtrace with its *nineteen (19)* stack frames (truncated lines are terminated with three dots). Starting from the top we see that the low level crypto functions are managed by conscrypt (line 1-5). Conscrypt is an open-source Java security providers developed by Google [14]. Conscrypt in turns uses BoringSSL [13], Google's open-source fork of OpenSSL. Going down the backtrace we see the use of javax.crypto module (line 6). This module provides high level interfaces for Java cryptographic operations. The next 9 stack frames (line 7-15) are created by the Nearby Connections library and the names of the classes and the methods are obfuscated, most probably using ProGuard [23]. The lowest part of the backtrace contains calls to thread methods. Overall, the cryptographic stack of the library uses standard implementation that we are able to replicate using python's `cryptography` module.

Using our crypto stack we to create valid ciphertext using the symmetric key computed in Figure 7 and valid message

TABLE II: scapy dissection classes used to reverse the Nearby Connections library. The arrows indicate the direction in which the correspondent packet is sent.

ClassName	Relevant Fields	Usage	Direction
`Kep1`	sn, eid, ncname, version	ECDH	D $\longrightarrow$ A
`Kep2`	sn, N_D, c_D, algo	ECDH	D $\longrightarrow$ A
`Kep3`	sn, N_A, xA, yA	ECDH	D $\longleftarrow$ A
`Kep4`	sn, xD, yD	ECDH	D $\longrightarrow$ A
`Eka`	iv, ct, mac	App Layer	D $\longleftrightarrow$ A
`KA`	count	App Layer	D $\longleftrightarrow$ A
`Pay`	iv, ct, mac	App Layer	D $\longleftrightarrow$ A
`Pt`	pid, ptype, pay_len, pay, count	App Layer	D $\longleftrightarrow$ A
`Pt2`	pid, ptype, pay_len, pt_len, count	App Layer	D $\longleftrightarrow$ A
`WL`	ip, tcp_port, count	Wi-Fi	D $\longleftarrow$ A
`HA`	essid, password, count	Wi-Fi	D $\longleftarrow$ A
`Error`	emsg	Misc	D $\longleftrightarrow$ A

authentication code using the key in Figure 8. We use scapy to build properly formatted application layer packets that include the ciphertext, the mac and the appropriate directional counter (either c_{A2D} or c_{D2A}).

F. Optional Physical Layer Switch

The implementation of the physical layer switch is key to perform the attack presented in Section IV. There are two packets of interests that the advertiser send to the discoverer to tell him when and how to switch from Bluetooth to Wi-Fi. We use two scapy dissection classes to manage these packets: `WL` and `HA`. Their relevant fields are shown in Table II. The former is used with the shared WLAN mode. Our custom advertiser sends an `WL` packet to the discoverer containing an arbitrary `ip` and `tcp_port`. This packet is usually sent just after the start of the EKA protocol. The latter is used with the Wi-Fi Direct and the hostapd. In this case the attacker can redirect the discoverer to an arbitrary AP by sending him an `HA` packet with spoofed `essid` and `password`. Usually, this packet is sent by the advertiser after three `Eka` packets.

G. Summary

Our REArby toolkit allows to analyze and attack the Nearby Connections library. It includes several components such a custom discoverer and advertiser dynamic binary instrumentation based on Frida, and packet dissector based on scapy. Our custom discoverer and advertiser are able to discover, advertise, request, initiate, authenticate, accept/reject, establish a connection and tell when and how to switch to a different physical layer. They maintain the connection alive by speaking the EKA protocol. They send `BYTE` and `FILE` payloads. They allow the attacker to specify the strategy, `serviceId`, `ncname` and `endpointId` and to modify and fuzz any dissected packets.

Table II summarizes the most important scapy dissection classes that are in use. Table III lists the Java classes and methods that we monitored while reversing `ncproc`. Table IV lists the device that we use for our analysis and attacks.

VI. Related Work

We are not aware of related work on the Nearby Connections API (for Android devices, or others). In general, a large amount of academic work has investigated security issues

TABLE III: List of the security related classes and methods used by `ncproc`.

ClassName	MethodName	Usage
OpenSSLCipher	engineDoFinal()	AES256 in CBC
	engineInit()	HMAC with SHA256
	engineUpdate()	HMAC with SHA256
	engineDoFinal()	HMAC with SHA256
OpenSSLMessageDigestJDK	engineUpdate()	SHA1, SHA2
	engineDigest()	SHA1, SHA2
OpenSSLECDHKeyAgreement	engineInit()	ECDH
	engineDoPhase()	ECDH
	engineGenerateSecret()	ECDH
NativeCrypto	RAND_bytes()	RNG
Base64	encodeToString()	Encode base64
	decode()	Decode base64

TABLE IV: Devices used in our Nearby Connections experiments and attacks. GPS is Google Play Services. SSP stands for Secure Simple Pairing and SC for Secure Connections.

Name	Library	Bluetooth	Soft AP
Android 6.0.1			
Motorola G3 (x2)	GPS 12.8.74	4.1 SSP with SC	Wi-Fi Direct, hostapd
Nexus 5 (x2)	GPS 12.8.74	4.1 SSP	hostapd
Linux 4.4			
Thinkpad x1	Bluez 5.49-4	4.2 SSP	hostapd, airbase-ng

in wireless standards, such as cryptographic weaknesses in early versions of Bluetooth [25], and practical sniffing attacks on Bluetooth [35]. Similarly, cryptographic attacks have been found on the confidentiality of Bluetooth LE [34] and early versions of IEEE 802.11/WiFi [7], and later improvements such as WPA [37]. Lately, additional key reinstallation attacks have been discovered for WPA2 [39], which compromise key freshness.

In addition to analysis of standards, libraries that implement the standards have also been investigated. For example, a number of vulnerabilities in Apple and Android devices' Bluetooth stack were identified in 2017, dubbed "BlueBorne" [1].

In this work, our focus is not on attacking any design or implementation aspect of Bluetooth and Wi-Fi standards. Instead, we point out that introduction of libraries such as Nearby Connections can lead to unexpected side effects for traffic of all applications on the victim, e.g., by disconnecting hosts, redirecting traffic via and attacker, and can lead to effects such as resource exhaustion (due to attackers manipulating radio states). Usage of third-party libraries has already been studied for the Android ecosystem, e.g., in [2]. However, our paper investigates a library developed directly by Google through its Google Play Services service, pushed to end users. The misuse of cryptographic primitives on Android is also well-known [11]. While in the case of Nearby Connections, the developer (or user) is not responsible for choosing the cryptographic primitives, he (or she) has to trust the library to be securely designed.

In [30], the authors investigated security issues arising from interactions of malicious apps with paired mobile devices, e.g., via Bluetooth, essentially related to lack of access controls. In some sense, some of our attacks are related as Apps influence other Apps (due to routing reconfiguration). We argue

that Nearby Connections poses an even bigger threat, as it is ubiquitously supported and optimized to require no user interactions. If the attacker managed to insert a default route to his AP, then all communication of the victim is affected.

A rich set of literature on attacks on routing schemes in the context of wireless sensor networks is available [24]. We argue that in this work, the system attacked is not a (multi-hop) routing scheme, as it is intended for direct communication between nearby hosts. Practical wormhole attacks that extend communication range from nearby hosts to remote hosts have been demonstrated in the context of car keys [12] and NFC communication [29].

VII. Conclusion

In this work, we present the first security analysis of the proprietary, closed-source and obfuscated Nearby Connections API by Google. The API is installed and available to applications on any Android device from version 4.0 onwards. It is also available on Android Things, a new OS by Google for IoT devices. The API connects nearby devices using multiple physical layers. In order to perform the analysis, we studied its public API, and reverse engineered its implementation on Android. We found and implemented several attacks (classified as connection manipulation and range extension attacks).

For example, we were able to impersonate an advertiser, allowing us to trick the victim discoverer to disconnect from its currently associated access point, and connecting to an access point controlled by the attacker. The attacker was then able to push a default route (via DHCP) to the victim, effectively redirecting all victim traffic (not only from the Nearby Connections application) to the attacker. This is a novel attack, in which a vulnerability in the Nearby Connections API, is impacting all the network communication on that Android device.

Our implementation of the attacks is based on REArby, a toolkit we developed to reverse engineer and analyze the implementation of the Nearby Connections API. REArby includes our custom discoverer and advertiser capable of performing all the nearby connection phases. REArby also includes a dynamic binary instrumenter, a packet dissector and a custom Android application. This toolkit will be made available after a responsible disclosure to Google.

Our findings show that in the current state, Google's Nearby Connections API is not only open to attack, but actively posing a threat to the many devices on which it is installed and enabled by default, even to applications that do not directly use it. Nearby attackers can establish themselves as man-in-the-middle for all Wi-Fi connections of the victims (even if the victims are associated with a secured access point).

References

[1] Armis, "The attack vector blueborne exposes almost every connected device," http://go.armis.com/hubfs/BlueBorne%20Technical%20White%20Paper-1.pdf?t=1517293112971, Accessed: 2018-01-26.

[2] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 356–367.

[3] E. Barker, L. Chen, and e. a. Roginsky A, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf, 2018, Recommendations of the NIST.

[4] D. J. Bernstein and T. Lange, "Safecurves: choosing safe curves for elliptic-curve cryptography," https://safecurves.cr.yp.to, Accessed: 2018-07-16.

[5] P. Biondi, "Scapy: Packet crafting for python2 and python3," https://scapy.net/, Accessed: 2018-01-26.

[6] S. Bluetooth, "Bluetooth core specification v5.0," https://www.bluetooth.org, Accessed: 2018-01-26, 2016.

[7] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting mobile communications: the insecurity of 802.11," in *Proceedings of the Annual international Conference on Mobile computing and networking (MobiCom)*. ACM, 2001, pp. 180–189.

[8] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 197–213.

[9] A. Developers, "Overview of google play services," https://developers.google.com/android/guides/overview, Accessed: 2018-01-26.

[10] ——, "Utilities for encoding and decoding the base64 representation of binary data," https://developer.android.com/reference/android/util/Base64, Accessed: 2018-01-26.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. New York, NY, USA: ACM, 2013, pp. 73–84.

[12] A. Francillon, B. Danev, and S. Capkun, "Relay attacks on passive keyless entry and start systems in modern cars," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2011.

[13] Google, "Boringssl is a fork of openssl that is designed to meet google's needs." https://boringssl.googlesource.com/boringssl/, Accessed: 2018-01-26.

[14] ——, "Conscrypt is a java security provider," https://www.conscrypt.org/, Accessed: 2018-01-26.

[15] ——, "Nearby connections: Get started," https://developers.google.com/nearby/connections/android/get-started, Accessed: 2018-07-17, 2017.

[16] ——, "Nearby connections: Strategies," https://developers.google.com/nearby/connections/strategies, Accessed: 2018-07-17, 2017.

[17] ——, "Nearby connections: v11 update," https://developers.google.com/nearby/connections/v11-update, Accessed: 2018-07-17, 2017.

[18] ——, "Nearby connections: Advertise and discover," https://developers.google.com/nearby/connections/android/discover-devices, Accessed: 2018-07-17, 2018.

[19] ——, "Nearby connections: Manage connections," https://developers.google.com/nearby/connections/android/manage-connections, Accessed: 2018-07-17, 2018.

[20] ——, "Nearby connections: Wi-fi issues," ttps://stackoverflow.com/questions/49401461/google-nearby-blocks-android-application-accessing-to-internet-it-switches-to, Accessed: 2018-07-17, 2018.

[21] ——, "Rockpaperscissors sample app for nearby apis on android," https://github.com/googlesamples/android-nearby/tree/master/connections/rockpaperscissors, Accessed: 2018-07-17, 2018.

[22] ——, "Samples for nearby apis on android," https://github.com/googlesamples/android-nearby/tree/master/connections, Accessed: 2018-07-17, 2018.

[23] Guardsquare, "ProGuard: The open source optimizer for Java bytecode," https://www.guardsquare.com/en/products/proguard, Accessed: 2018-07-17, 2018.

[24] Y.-C. Hu, A. Perrig, and D. B. Johnson, "Wormhole attacks in wireless networks," *IEEE journal on selected areas in communications*, vol. 24, no. 2, pp. 370–380, 2006.

[25] M. Jakobsson and S. Wetzel, "Security weaknesses in bluetooth," in *Cryptographers Track at the RSA Conference*. Springer, 2001, pp. 176–191.

[26] B. Kannhavong, H. Nakayama, Y. Nemoto, N. Kato, and A. Jamalipour, "A survey of routing attacks in mobile ad hoc networks," *IEEE Wireless communications*, vol. 14, no. 5, 2007.

[27] C. Karlof and D. Wagner, "Secure routing in wireless sensor networks: Attacks and countermeasures," in *Proceedings of the Workshop on Sensor Network Protocols and Applications*. IEEE, 2003, pp. 113–127.

[28] D. Liu, P. Ning, and R. Li, "Establishing pairwise keys in distributed sensor networks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 41–77, 2005.

[29] K. Markantonakis, L. Francis, G. Hancke, and K. Mayes, "Practical relay attack on contactless transactions by using nfc mobile phones," *Radio Frequency Identification System Security: RFIDsec*, vol. 12, p. 21, 2012.

[30] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device mis-binding on android." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[31] O. of Google Play Services, "Nearby connections: v11 update," https://developers.google.com/android/guides/overview, Accessed: 2018-07-17, 2018.

[32] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[33] O. A. V. Ravns, "Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," https://www.frida.re/, Accessed: 2018-01-26.

[34] M. Ryan, "Bluetooth: With low energy comes low security," in *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, vol. 13, 2013, pp. 4–4.

[35] D. Spill and A. Bittau, "Bluesniff: Eve meets alice and bluetooth." in *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, vol. 7, 2007, pp. 1–10.

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

[37] E. Tews and M. Beck, "Practical attacks against WEP and WPA," in *Proceedings of the second ACM conference on Wireless network security*. ACM, 2009, pp. 79–86.

[38] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the requirements for successful GPS spoofing attacks," in *Proceedings of the ACM conference on Computer and communications security (CCS)*. ACM, 2011, pp. 75–86.

[39] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1313–1328.