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

Daniele Antonioli
Singapore University of Technology and Design
daniele_antonioli@mymail.sutd.edu.sg

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security
tippenhauer@cispa.saarland

Kasper Rasmussen
Department of Computer Science
University of Oxford
kasper.rasmussen@cs.ox.ac.uk

***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 the API), and to tamper with the victim's network interface and configuration. REA-attacks allow an attacker to tunnel any nearby connection to remote (non-nearby) locations, even between two honest devices. Our attacks are enabled by REarby, a toolkit we developed while reversing the implementation of the API. REarby includes a dynamic binary instrumenter, a packet dissector, and the implementations of custom Nearby Connections client and server.

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.

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 [1]. An attacker who can exploit this API can target (at least) 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 [18], [16]. According to these sources, the API uses encryption by default, but it does not mandate user authentication. The API automatically manages and uses multiple physical layers and this does not sound trivial. The API uses a custom application layer connection mechanism. A device can simultaneously be a client and a server, and can connect to different 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 [28], [32], [27]. 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 [10], [29].

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

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23367
www.ndss-symposium.org

and implement several attacks. The impact of these attacks ranges from intercepting and decrypting application-layer data (using man-in-the-middle or impersonation), to forcing the establishment of TCP connections between the victim and arbitrary (non-nearby) devices. In addition, the attacker is able to introduce system wide default network routes on the victims' devices. As a result, the attacker is able to redirect a victim to an access point under his control and 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 released parts of the toolkit as open source in our proof of concept code[1].

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 describe 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.

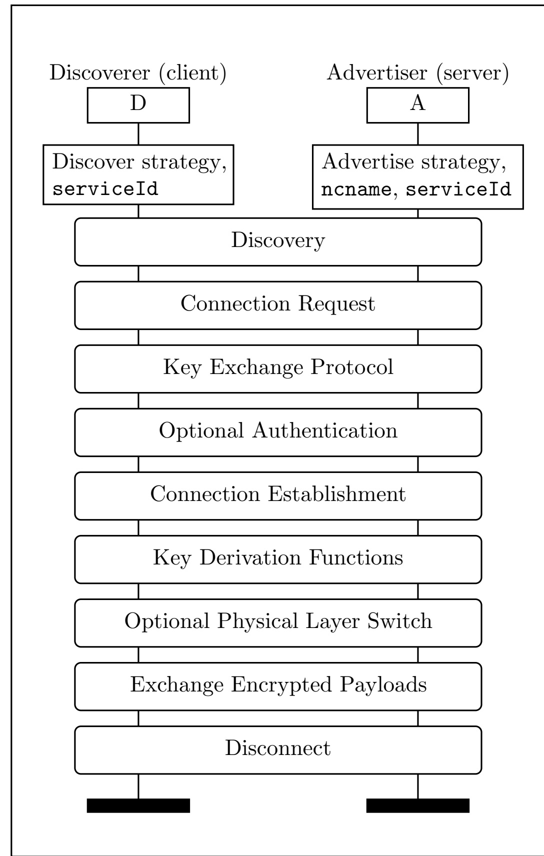

Fig. 1: The Nearby Connections API has two types of actors: the Discoverer (client) and the Advertiser (server). It uses application layer encryption and optional user authentication.

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 $100\,\mathrm{m}$) [17] 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 [8]. 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 [8].

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 [18]. 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 [1].

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 [19].

If the client discovers the server then he sends a connection request to the server 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

[1]Repository at https://github.com/francozappa/rearby.

layer, e.g., to Wi-Fi, and then they start exchanging (encrypted) data 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 writing, the Nearby Connections API provides two strategies: `P2P_STAR` and `P2P_CLUSTER`. There is also a third one called `P2P_POINT_TO_POINT` that it is still not public. 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 using Bluetooth in 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. In any case, the actors can exchange payloads without being connected to the Internet and two discoverers always communicate through an advertiser (even when using the `P2P_CLUSTER` strategy).

III. Reversing and Analyzing Nearby Connections

In this section we describe our understanding of the Nearby Connections API after reverse engineering its implementation on Android. Our main goal is to perform a security assessment of the API because of its wide attack surface and complex interactions between 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, its implementation is presented Section V. For the remainder of this work, we refer to the target of our analysis as "the library". We note that, without access to the source code or specification of the library, we cannot claim that our findings are always complete.

The analysis of the library allowed us to identify and perform several attacks that we present in Section IV. In particular, we use details of the authentication and interactions between advertisers and discoverers (Section III-A) to perform attacks in which we impersonate them and we manipulate Nearby Connections traffic. Details of the Nearby Connections keep-alive mechanism (Section III-E) are required for range extension attacks. The physical layer switch (Section III-F) is exploited to manipulate system-wide routing tables of victims.

A. Discovery and 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 without using encryption. An attacker who posses a clone of the library can pretend to be any advertiser and discoverer of any application, and he can use any Bluetooth compatible device to request a connection. The Bluetooth connection uses Secure Simple Pairing (SSP) with a link key that is not authenticated and not persistent. Indeed, an attacker could insert himself as a man in the middle in the Bluetooth link and he can force the re-establishment of the link key at any time.

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:

$$\text{btname} = \text{unpad}(\text{b64encode}(\text{strategy_code} \parallel \texttt{endpointId} \parallel \text{SHA256}(\texttt{serviceId})[:3] \parallel \text{separator} \parallel \texttt{ncname}))$$

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 (by an attacker) 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`) [8]. 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. RFCOMM is a serial cable emulation

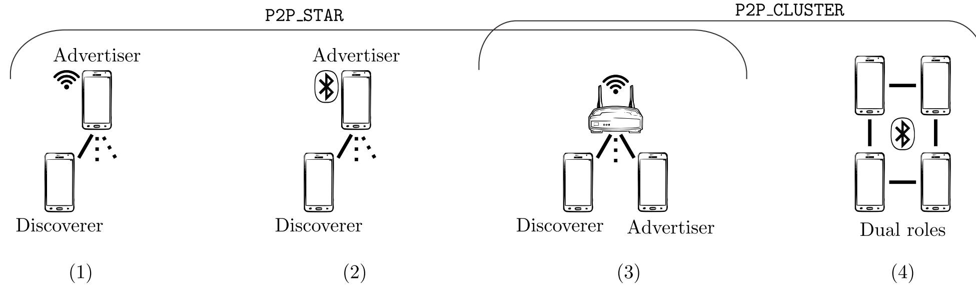

Fig. 2: Nearby Connections connection strategies using Bluetooth and Wi-Fi. On the left, three `P2P_STAR` topologies; on the right, two `P2P_CLUSTER` topologies. In each case, all actors use the same `serviceId`.

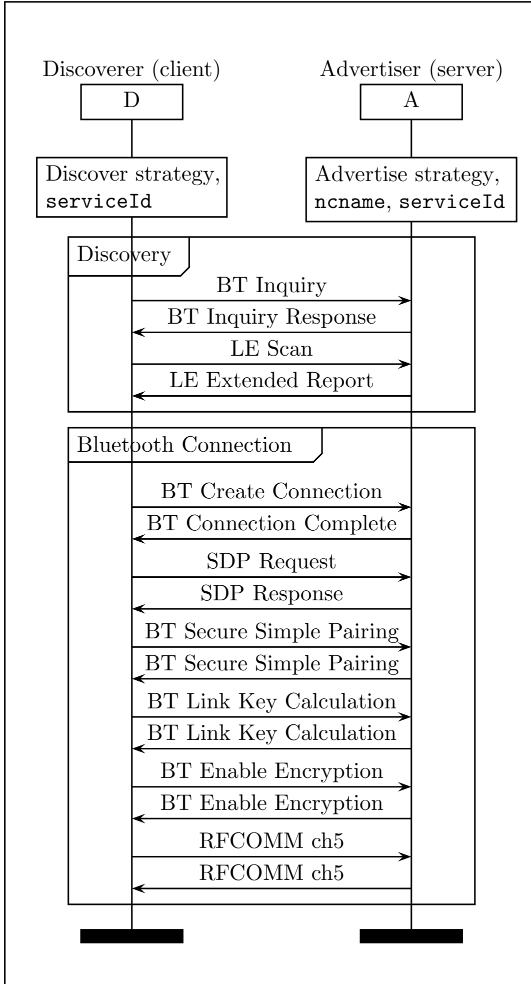

Fig. 3: Nearby Connections Request. BT is Bluetooth BR/EDR. Secure Simple Pairing provides link-layer encryption.

transport protocol based on ETSI TS 07.10 providing similar guarantees of TCP [8].

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.

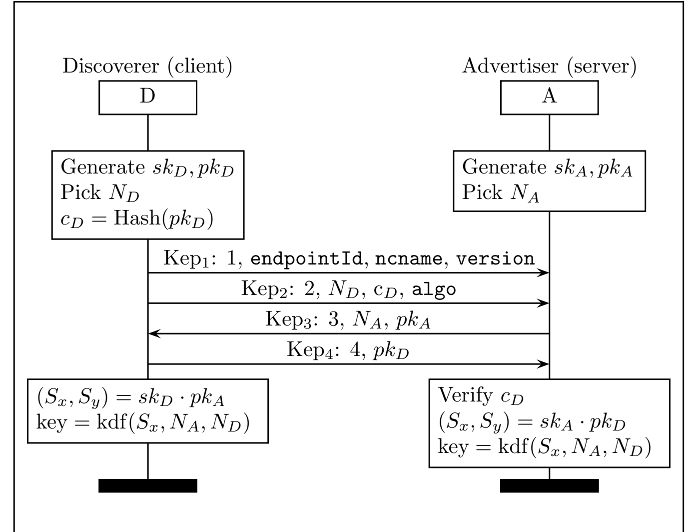

Fig. 4: Nearby Connections Key Exchange Protocol (KEP) based on ECDH (secp256r1). `algo` is always `AES_256_CBC-HMAC_SHA256`.

B. Key Exchange Protocol (KEP)

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 [5], however some crypto experts have questioned the security of this curve [6]. 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 standard key derivation function (such as HKDF), weird usage of nonces and commitments, and transferring 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: 1 (sequence number), `endpointId`, `ncname` 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: 4 (sequence number) and pk_D (the client's public key).

The client computes an hash of pk_D (based on SHA512) that should be a commitment on his public key. We define the commitment as c_D. Then, the client builds Kep_2 that has the following relevant fields: 2 (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. The presence of `algo` could indicate that the developers are planning to introduce cipher negotiation as a feature, otherwise there seems little point in exchanging this information. The server builds Kep_3 that has the following relevant fields: 3 (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. The library does not use any (recommended) ECDH key derivation functions such as HKDF or NIST-800-56-Concatenation-KDF [5]. The details about how we managed to discover it are presented in Section V. After the client and the server completed the KEP, the nearby connection is initiated (but not yet established).

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 $(G_x, G_y) \cdot sk_D$

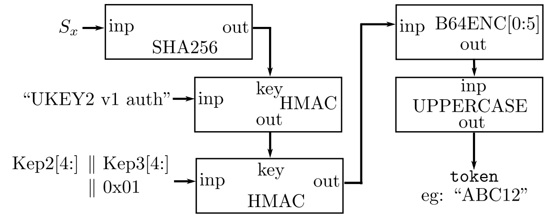

Fig. 5: Computation of the authentication token.

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 uppercase alphabetic characters, numbers, and special characters from the base64 alphabet (search space of size 38^5). 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 it. At the time of writing, the nearby connection sample code from Google *ignores* the authentication tokens [23]. 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` 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.

In the connection establishment both devices have to accept the connection, and it does not matter who accepts it first. In our experiments, we observed that the devices always use *the same* payloads:

- `0x000000080801120408053200` to keep the pre-connection alive
- `0x0000000a0801120608021a020800` to accept a connection
- `0x0000000b0801120708021a0308c43e` to reject a connection

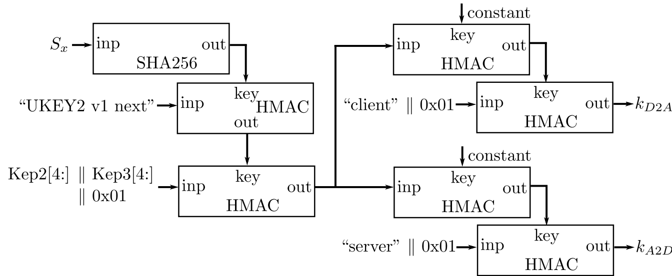

Fig. 6: Computation of k_{D2A} and k_{A2D}.

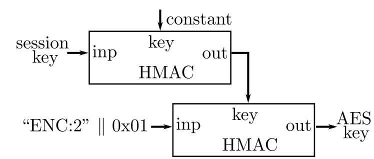

Fig. 7: Computation of the AES (symmetric) key.

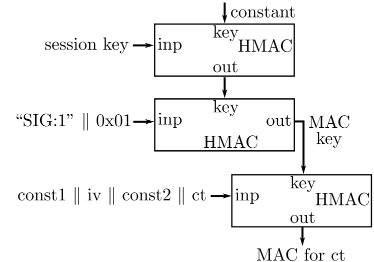

Fig. 8: Computation of the MAC key and the MAC.

As in the connection request phase, 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 in a nearby connection 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__`.

D. Key Derivation Functions (KDFs)

We were able to reverse the key derivation functions responsible for the creation of the session, encryption, and message authentication code keys. When a nearby connection is established then the client and server compute two symmetric session keys that we define as 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. 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 key computation 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 describes also how the library does compute a MAC. Nearby Connections uses *encrypt-then-mac with HMAC using SHA256*. The MAC is computed using the MAC key and as input a subset of a payload containing the ciphertext (ct), an initialization vector (iv) and some constant fields.

Using dynamic binary instrumentation (discussed in Section V), we see that the library *for each application layer packet re-derives the same AES and MAC keys from the session keys*. In particular, 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 key stays in memory for a 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. Exchange Encrypted 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}

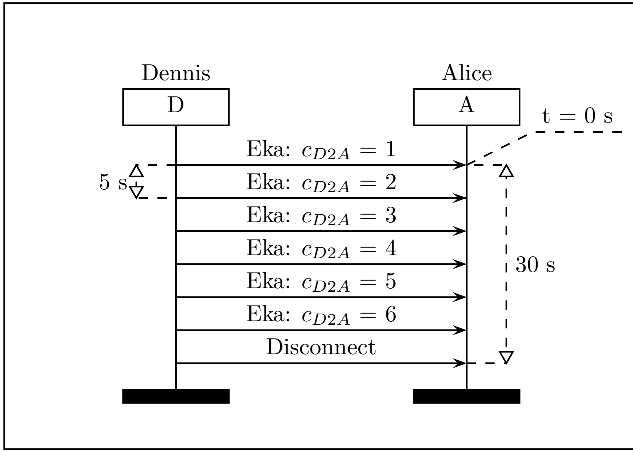

Fig. 9: Nearby Connections Encrypted Keep-Alive. The period is 5 seconds, the timeout is 30 seconds.

Algorithm 1 Nearby Connections Physical Layer Switch.

Require: D = discoverer, A = advertiser
Result: Bluetooth uses RFCOMM, Wi-Fi uses TCP, no secrets shared between Wi-Fi and Bluetooth
  if A is connected to an hotspot **then**
    A tells D how to switch to a 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

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 (EKA) shown in Figure 9. From our experiments we discovered that *a nearby connection has a connection timeout of 30 seconds*. In Section IV, we discuss how that can be exploited to extend the range of our attacks. The EKA protocol uses a custom type of packet that we define as `Eka`. These packets have a constant header and contains a directional counter.

As we can see from Figure 9, 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 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`, and `STREAM`. The payloads are sent either using Bluetooth (over RFCOMM) or Wi-Fi (over TCP) and they are encoded using custom application layer packets. Each payload generates at least two packets and each one contains the appropriate directional counter value (either c_{D2A} or c_{A2D}). The packets are sent sequentially without application layer acknowledgments. A node can send and receive payloads asynchronously. A payload packet contributes to keep the connection alive in the EKA protocol.

F. Optional Physical Layer Switch

Once a nearby connection is established, the client and the server might switch from Bluetooth to Wi-Fi. From our experiments, we see that the physical layer switch can be *predicted and manipulated*. We misuse this mechanism to perform a connection manipulation attacks (CMA) presented in Section IV. The switch always happens from Bluetooth to one of the three Wi-Fi modes 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 cases the advertiser acts as a soft AP, see (1) in Figure 2. The nearby connection documentation tells us that the library uses a real-time heuristic to determine when and how to switch.

Algorithm 1 describes the result of our reversing 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 connections we would expect the library sharing secret material between Bluetooth (that is encrypted at the link and application layers) and Wi-Fi. If this material exists it should be exchanged before switching to Wi-Fi. However, in our experiments we observed that this is not the case because *the devices after the switch 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).

From Algorithm 1 we note that 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). 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 without an Internet connection. The discoverer should be able to find its `essid` and connect to the 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 we leverage this mechanism in our attacks. Wi-Fi Direct uses a constant `essid` (22 Bytes, always starts with `DIRECT-`) and a WPA2 password (8 Bytes). hostapd uses a randomized base64-encoded `essid` (28 Bytes) and a WPA password (12 Bytes). In both cases, the advertiser sends the `essid` and the password to the discoverer. When the connection is terminated, the library 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 this capability to to switch on the Bluetoothand Wi-Fi (hotspot) antenna of the victims without user interaction. Another side effect of the switch is 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

In this section we present the attacks that we on the Nearby Connections library, based on our reverse engineering presented in Section III. To perform the attacks we developed several tools that are presented in Section V. We classify our attacks in two families: connection manipulation attack (CMA), and range extension attack (REA). The attack families are orthogonal, and can be combined. *We remark that our attacks manipulate application layer packets to reach some goal and indeed are effective regardless the specific Android application that is using Nearby Connections as a service.* In general, if an attack is effective regardless whether the attacker is the discoverer or the advertiser, we indicate the victim and the attacker as *nodes*.

A. Threat Model

Our attack scenario includes the victims (discoverer and advertiser) and an attacker who posses at least one device in range with the victims. The legitimate discoverer and the advertiser establish nearby connections as described in Section II and Section III. 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*, e.g., install himself as man-in-the-middle, take over existing connection, manipulate the Bluetooth and Wi-Fi radio of the victims, and break or weaken the security mechanisms of the nearby connection library.

The attacker has the same knowledge of the library that we describe in Section III. He is capable of using custom advertiser and discoverer and to craft raw packets conforming to the nearby connection protocol using tools similar to the ones developed (REarby). An extensive discussion of the tools is presented in Section V. The attacker can create his own Wi-Fi access point, and jam the wireless links. He does not have access to the victims devices e.g., he is not able to install rootkits or malicious applications. The applications and libraries used by the victims are considered safe. The attacker does not require advanced and potentially expensive instrumentation such as software-defined radio, directional antennas, rooted devices and commercial wireless sniffers.

B. Connection Manipulation Attacks

In principle, a node should establish nearby connections only with trusted nodes. However, the library presents several authentication issues that allows an attacker to manipulate a connection. In particular, the library 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” [20] and we argue that this is the case here.

Advertiser and discoverer impersonation. The library uses only the strategy and the `serviceId` to uniquely identify a nearby connection, and both are predicable. We can learn the `serviceId` of an application by using a Bluetooth SDP request and we can guess the strategy (only 2 options). As a result, we can impersonate both an advertiser advertising a proper `serviceId` and a discoverer trying to connect to a legitimate advertiser. This capability is a stepping stone to perform more elaborate attacks that we present in this section.

Application layer MitMs. The lack of proper authentication of the library allows us (the attacker) to man-in-the-middle two victims at the application layer. We accomplish this attack using a malicious advertiser and a malicious discoverer at the same time. The malicious advertiser gets discovered by the first victim discoverer, and the malicious discoverer connects to the second victim advertiser (the two malicious devices forward traffic between each other). Then, we can complete two parallel Bluetooth pairings with the victims, we perform the application layer phases of Figure 1, and we compute the shared secrets and the keys (session, encryption and mac) for each victim. As a result we are able to decrypt, observe, manipulate and encrypt the application layer packets.

If the advertiser requests a physical layer switch we launch a parallel man-in-the-middle attack on the Wi-Fi link. We know the credentials of the Wi-Fi network because they are transmitted by the victim advertiser on the Bluetooth link from which we are eavesdropping. The Wi-Fi MitM is accomplished using a simple ARP spoofing attack. Wi-Fiis not encrypted at the ink layer, thus we can continue to observe and manipulate the application layer traffic also in the Wi-Fi link. We note that the MitM works even between a victim discoverer and a victim advertiser using different Android applications (different `serviceId`).

Shared WLAN manipulation. The attacker can also manipulate the physical layer switch phase, regardless the nearby connection strategy. As shown in Algorithm 1, the advertiser dictates when and how to switch (from Bluetooth to Wi-Fi) and the discoverer “blindly” follows him. For example, if the advertiser is connected to an hotspot it will tell the discoverer its IP address and TCP port (shared WLAN Wi-Fi mode). However, we are able to redirect a discoverer to an arbitrary IP and TCP port by intercepting and crafting the legitimate physical layer switch packet sent by the advertiser before the switch. The details about how we craft this and other types of packets are

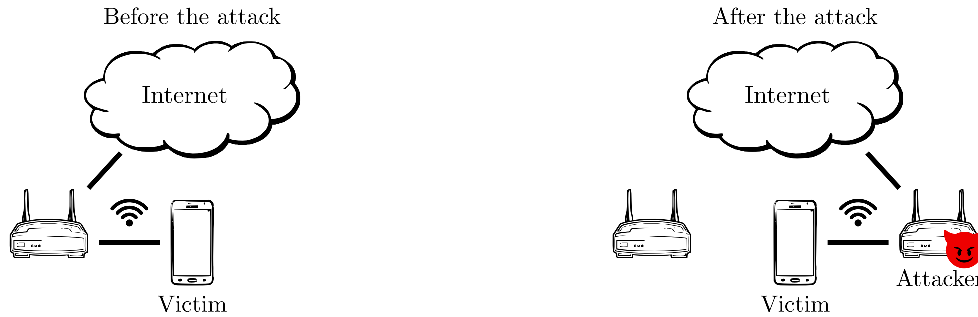

Fig. 10: Soft AP manipulation attack. On the left, before the attack the victim is connected to the Internet through a benign AP. On the right, after the attack the adversary has forced the victim to connect to a malicious access point, and inserted a new default route in the victim's routing table. The adversary is able to intercept and manipulate any Internet-bound traffic sent by *any* application on the victim's phone that is using Internet.

presented in Section V-F. As result of this attack, the victim activates her Wi-Fi interface, associates to a legitimate AP and establishes a TCP session with a target determined by the attacker. Note that, this attack work regardless the connection strategy because the shared WLAN mode is picked as first choice in both, and the IP spoofed by the attacker can be outside of the local area network of the victim.

Soft AP manipulation. If the target nearby connection uses the `P2P_STAR` strategy, the advertiser might act as a soft access point without an Internet connection, and provide the AP credentials to the discoverers. We take advantage of this feature *to redirect discoverers to an access point controlled by the attacker that is connected to the Internet.* Figure 10 shows the victim connection status before (left side) and after the attack. Initially, the victim (discoverer) is connected to a legitimate AP. The victim connects to an advertiser using the `P2P_STAR` strategy. The attacker either is the advertiser or performs the application layer MitM attack presented earlier. The attacker also controls a rogue access point.

Once the nearby connection is established the attacker manipulates either the hostapd or Wi-Fi Direct switch packets to redirect the victim to the rogue access point i.e., he provides the victim valid essid and password. Then, the victim associates to the malicious AP (see Figure 10), and configures her Wi-Fi interface with values supplied by the attacker over DHCP. That enables the attacker to install a new default route (along with suitable IP configurations), which redirects all the victim's traffic to the rogue AP. Indeed, the attacker is able to monitor and tamper with *all the Wi-Fi traffic* coming from the victim. The traffic includes the packets generated by other applications (not using the library) requiring Internet access such as email clients, web browsers, and cloud services. We understand that most of such traffic is secured with TLS, but we still believe that this attack is novel and it has serious consequences. In particular, the attacker can target a single application using Nearby Connections to get access to all Wi-Fi network traffic of the victim. The attacker then can perform traffic analysis even on encrypted packets [36].

DoS Internet connections. The library does not care about the connection status of the devices before using hostapd and Wi-Fi Direct. We leverage this fact to launch a denial of service (DoS) attack on several discoverers at the same time. The attack assumes that one ore multiple discoverers are connected to a legitimate Wi-Fi network and they want to use Nearby Connections. The attacker uses a malicious advertiser to connect to the victims and tell them to use either hostapd or Wi-Fi Directand to connect to a non-existent AP. The victims try to connect to the AP, and as a result they lose their Internet connectivity. This issue indirectly affects all the applications running on the victim's device that need an Internet connection [21].

Alter network configurations and radios. The library does not backup and restore the original wireless network configuration. In particular, the original soft AP (Wi-Fi hotspot) configuration is overwritten by the library and the newly created network is appended to the list of known ones. This allows an attacker to append and overwrite arbitrary essid-password pairs in the network configuration files of the victim. The attack technique is the same presented in the DoS attack paragraph.

The library is able to switch on the Bluetooth and Wi-Fi antenna of device that is using it, and it does not switch them off after a disconnection. We use this feature manipulate the antennas of a victim device, regardless of the nearby connection strategy. We are able to switch on the Bluetooth antenna of the victim by establishing a nearby connectionand then disconnecting. We can switch on the Wi-Fi antenna of any discoverer by using our custom advertiser to connect with the victim and by telling the him to switch to Wi-Fi and to disconnect.

C. Range Extension Attacks (REA)

The Nearby Connections API is supposed to be used by devices that are effectively nearby. The documentation suggests that they have to be within radio range (approx 100 m) [17]. However, at the time of writing, *the library does not enforce strict time requirements between connected nodes and does not check the geo-location of the nodes.* The library uses an encrypted keep alive protocol with a generous timeout of 30 seconds, which is more than enough to forward traffic across continents without aborting the nearby connection [13], [30]. This allows an attacker to violate the fundamental assumption that nearby-connected devices are in proximity by extending the range of any nearby connection.

The attacker can extend the range of any attack presented in Section IV-B. In our experiment we implement a wormhole-attack to extend the range of the application layer MitM attack. We are able to let two non-nearby victims talk between each other, i.e., the two victims might be advertising and discovering nearby connection in different continents. The attacker uses two devices, each one in range with a victim, to perform the MitM attack and then forwards the traffic over the Internet creating a wormhole. The attacker has 30 seconds to forward the packets in each direction (to keep the connection alive) and he could even answer to the keep alive requests himself, effectively allowing arbitrary delays. This attack technique takes inspiration from [25], [38], [13]. Range extension is not advisable because a victim perceives a false sense of security given by the fact that a nearby connection is supposed to be within radio range and it is expected not to use Internet. In other words, a victim might better trust a proximity-based secure service than a secure cloud service.

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, creation and manipulation 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 custom discoverer and advertiser capable to perform all the nearby connection phases from Figure 1. It includes a dynamic binary instrumenter to analyze the library at runtime and a packet dissector usable to decode and manipulate application layer packets. REarbyalso contains a custom Android application that we developed for testing. Out toolkit makes use of 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. Discovery and 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 manages each of them with separate sockets. All the Bluetooth connections are encrypted at the link layer using the shared secret computed from the secure simple pairing (SSP).

Listing 1 `Kep3` scapy dissection class for Kep_3.

```
1  class Kep3(Packet):
2      name = "Kep3: adv -> dsc"
3      fields_desc = [
4          IntField("len1", None),
5          XByteField("sep1", 0x08),
6          ByteField("sn", 3),
7          XByteField("NC_SEP", NC_SEP),
8          ByteField("len2", None),
9          StrFixedLenField("NC_HEAD2", NC_HEAD2,
            ↪  length=3),
10         BitFieldLenField("nA_len", None, size=8,
            ↪  length_of="nA"),
11         StrLenField("nA", "", length_from=lambda
            ↪  pkt:pkt.nA_len),
12         StrFixedLenField("NC_KEP3_HEAD",
            ↪  NC_KEP3_HEAD, length=3),
13         ByteField("len3", None),
14         StrFixedLenField("NC_HEAD2", NC_HEAD2,
            ↪  length=3),
15         ByteField("len4", None),
16         XByteField("NEWLINE", NEWLINE),
17         BitFieldLenField("xA_len", None, size=8,
            ↪  length_of="xA"),
18         StrLenField("xA", "", length_from=lambda
            ↪  pkt:pkt.xA_len),
19         XByteField("NC_SEP", NC_SEP),
20         BitFieldLenField("yA_len", None, size=8,
            ↪  length_of="yA"),
21         StrLenField("yA", "", length_from=lambda
            ↪  pkt:pkt.yA_len),
```

B. Key Exchange Protocol (KEP)

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 connect the advertiser on port 5. The `bluetooth`'s Python module manages the low level details 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 [22] and we use it to generate samples of 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) [8]. We analyze the packets using scapy [7], a network analysis tool. We discover that the plaintext is serialized using custom serialization mixing variable and fixed length fields. This is confirmed by the fact that the library uses the `Serializable` Java class.

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

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 }
```

holds for `xA` and `yA`. We use similar classes to decode Kep_1 Kep_2, and Kep_4. From the decoded KEP packets we realize that the library uses ECDH on the secp256r1 curve, by testing the public keys contained in Kep_3 and Kep_4 against standard elliptic 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), a technique that 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 standard key derivation function. These information enables us to initiate a nearby connection by implementing the Nearby Connections KEP protocol from Figure 4. We use the Python `cryptography` module to implement all the cryptographic procedures. Note that, our setup allows us to tamper with and fuzz every field of the KEP packets such as the 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 and overload the `encodeToString` method of the `android.util.Base64` class [11] using Frida. 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 most important lines of code are from line 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 reconstruct the computation of `token` from Figure 5. We implement its computation in our custom advertiser and discoverer using standard Python 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. Encrypted Keep-Alive and 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. For example, we have to maintain the directional counters (c_{A2D}, c_{D2A}) as explained in Section III-E. Implementing the EKA protocol is a key requirement to perform the 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 [15]. Conscrypt in turns uses BoringSSL [14], Google's open-source fork of OpenSSL. Note that BoringSSL is native code. 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 [24]. The lowest part of the backtrace contains calls to thread methods.

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)
```

Overall, the cryptographic stack of the library uses standard implementations that we are able to replicate using the Python `cryptography` module. Using our crypto stack we create valid ciphertext using the symmetric key computed as in Figure 7 and valid message authentication code using the key as in Figure 8. We use scapy to build properly formatted application layer packets that include the length fields, the ciphertext, the mac and the appropriate directional counter (either c_{A2D} or c_{D2A}).

F. Optional Physical Layer Switch

The correct implementation of the physical layer switch is paramount to perform the attack presented in Section IV. There are two packets of interests that the advertiser has to send to the discoverer to tell him when and how to switch from Bluetooth to Wi-Fi. We use two scapy dissection classes, that we call `WL` and `HA`, to manage these packets. Their relevant fields are shown in Table II. `WL` is used with the shared WLAN mode, our custom advertiser (the attacker) sends it to the discoverer to redirect him to arbitrary `ip` and `tcp_port`. This packet is usually sent just after the start of the EKA protocol. `HA` is used with the Wi-Fi Direct and the hostapd. By spoofing the `essid` and `password` fields in of this packet our custom advertiser redirects the victim (discoverer) to an arbitrary soft AP. 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: scapy dissection classes used to reverse the Nearby Connections library. The count field contains the application layer directional counter.

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

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 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 in wireless standards, such as cryptographic weaknesses in early versions of Bluetooth [26], 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/Wi-Fi [9], and later improvements such as WPA [37]. Lately, additional key reinstallation attacks were discovered for WPA2 [39], which compromise key freshness. The impact of physical layer features (such as MIMO and beamforming in recent standard amendments) on practical eavesdropping on 802.11 was discussed in [2]. 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" [3].

In this paper 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

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

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 [4]. 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 [12]. 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 [31], 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. We argue that Nearby Connections poses an even bigger threat, as it is ubiquitously supported and optimized to require no user interactions.

A rich set of literature about attacks on routing schemes in the context of wireless sensor networks is available [25]. 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 [13] and NFC communication [30].

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, an OS developed by Google for IoT devices. The API connects nearby devices using multiple physical layers. In order to perform our 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, in the Soft AP manipulation attack, we trick the victim (discoverer) to disconnect from its currently associated access point, and connect to an access point controlled by the attacker. Consequently, the attacker is able to push a default route in the victim's network configuration (via DHCP) and to redirect all his Wi-Fi traffic (not only from the Nearby Connections application) to him. This is a novel attack, in which a vulnerability in the Nearby Connections API, is impacting all the network communication of the victim.

Our implementation of the attacks is based on REarby, a toolkit we developed to reverse engineer and analyze the Nearby Connections API. REarby includes custom discoverer and advertiser capable of performing the nearby connection phases. It also includes a dynamic binary instrumenter, a packet dissector and a custom Android application. Our findings were acknowledged by Google in a responsible disclosure process, and we released a proof of concept of the Soft AP attack as open source at https://github.com/francozappa/rearby. 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 all Android applications using it (the library is automatically installed and updated without user interaction) and even to Android applications that do not use it.

References

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

[2] D. Antonioli, S. Siby, and N. O. Tippenhauer, "Practical evaluation of passive COTS eavesdropping in 802.11b/n/ac WLAN," in *Proceedings of Conference on Cryptology And Network Security (CANS)*, November 2017.

[3] Armis, "The Attack Vector BlueBorne Exposes Almost Every Connected Device," https://armis.com/blueborne/, Accessed: 2018-01-26.

[4] 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.

[5] 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.

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

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

[8] Bluetooth SIG, "Bluetooth Core Specification v5.0," https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043, Accessed: 2018-10-28, 2016.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

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

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

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

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

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

[21] ——, "Nearby connections: Wi-fi issues," ttps://stackoverflow.com/questions/49401461, Accessed: 2018-07-17, 2018.

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

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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] K. Markantonakis, L. Francis, G. Hancke, and K. Mayes, "Practical relay attack on contactless transactions by using NFC mobile phones," *Proceedings of Workshop on Radio Frequency Identification System Security (RFIDsec)*, vol. 12, p. 21, 2012.

[31] 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.

[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. Ravnås, "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.