ABSTRACT

Device pairing in large Internet of Things (IoT) deployments is a challenge for device manufacturers and users. Bluetooth offers a comparably smooth trust on first use pairing experience. Though, is well-known for security flaws in the pairing process. In this paper, we analyze how Apple improves the security of Bluetooth pairing while still maintaining its usability and specification compliance. The proprietary protocol that resides on top of Bluetooth is called MagicPairing. It enables the user to pair a device once with Apple’s ecosystem and then seamlessly use it with all their other Apple devices. We analyze both, the security properties provided by this protocol, as well as its implementations. In general, MagicPairing could be adapted by other IoT vendors to improve Bluetooth security. Even though the overall protocol is well-designed, we identified multiple vulnerabilities within Apple’s implementations with over-the-air and in-process fuzzing.

CCS CONCEPTS

- Security and privacy → Systems security; Software security engineering; Software reverse engineering;
- Networks → Application layer protocols.

KEYWORDS

Bluetooth, Pairing, Security

ACM Reference Format:

Dennis Heinze, Jiska Classen, and Felix Rohrbach. 2020. MagicPairing: Apple’s Take on Securing Bluetooth Peripherals. In 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec ’20), July 8–10, 2020, Linz (Virtual Event), Austria. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395351.3399343

1 INTRODUCTION

Bluetooth device pairing has a long history of security flaws [1, 2, 6, 15, 25, 26, 29]. While most issues were fixed in the Bluetooth 5.2 specification [7], it is reasonable to assume that even this version is not bullet-proof. Adding further layers of encryption within the applications using Bluetooth is one solution many IoT developers chose [10]—but this leads to their devices being incompatible in communicating with third-party applications and drains battery. Thus, encrypting data twice is no satisfying solution to this problem.

Looking back into the history of Bluetooth security issues, it is not the encryption itself that has been exploited this frequently. Most problems originated from the initial key negotiation and connection setup. In Bluetooth, trust is established on first use by generating a permanent key. This permanent key protects device authenticity, message integrity, and message confidentiality [7, p. 269]. It is established individually between each pair of devices and only changes when a user manually deletes and reestablishes a pairing. In Classic Bluetooth, the permanent key is called Link Key (LK), while it is called Long Term Key (LTK) in Bluetooth Low Energy (BLE)—however, they can be converted into each other [7, p. 280]. For the duration of each Bluetooth connection, a session key is derived from the permanent key. Thus, if a device is out of reach or switched off, this invalidates a session key.

In modern IoT deployments, Bluetooth device pairing has two major shortcomings: (1) It does not scale for pairing to many devices within an existing infrastructure, and (2) once the permanent key is leaked, all security assumptions break for past and future connections. The permanent key can either be attacked by an active Machine-in-the-Middle (MITM) during pairing [6, 15, 25] or by Remote Code Execution (RCE) vulnerabilities within the chip [9].

Apple solves both challenges by introducing a protocol called MagicPairing. It pairs AirPods once and then enables the user to instantly use them on all their Apple devices. Security is improved by generating fresh “permanent” keys based on the user-specific iCloud keys for each session. Seamless ecosystem integration and security are imperative, since AirPods are able to interact with the Siri assistant.

Despite being a proprietary extension, MagicPairing is specification-compliant to the Host Controller Interface (HCI), and thus, can use off-the-shelf Bluetooth chips. The general logic of MagicPairing could be integrated into any cloud-based IoT ecosystem, increasing relevance for the security community in general. Our contributions on research of MagicPairing are as follows:

- We reverse-engineer the MagicPairing protocol.
- We analyze the security aspects provided by this protocol and their applicability to other wireless ecosystems.
- We document the proprietary iOS, macOS, and RTKit Bluetooth stacks.
- We manually test MagicPairing for logical bugs and automatically fuzz its three implementations.
- We responsibly disclosed multiple vulnerabilities.

While the overall idea of MagicPairing is new and solves shortcomings of the Bluetooth specification, we found various issues in Apple’s implementations. As MagicPairing is available prior to pairing and encryption, it poses a large zero-click wireless attack surface. We found that all implementations have different issues, including a lockout attack and a Denial of Service (DoS) causing 100 % CPU load. We identified these issues performing both, generic over-the-air testing and iOS in-process fuzzing.
Our fuzzing techniques can also be used to test other Bluetooth stacks and protocols. The Proof of Concepts (PoCs) for the identified vulnerabilities as well as the over-the-air fuzzing additions are available within the InternalBlue project on GitHub. The ToothPicker in-process fuzzer part that integrates into InternalBlue will follow soon, but has to be slightly delayed due to further findings [13].

This paper is structured as follows. Section 2 gives an overview of the reverse-engineered MagicPairing protocol. Its security properties are explained in Section 3. Implementation internals regarding MagicPairing-specific details are provided in Section 4. Then, we explain our fuzzing setup in Section 5 used to identify the vulnerabilities explained in Section 6. We conclude our work in Section 7.

2 THE MAGICPAIRING PROTOCOL

MagicPairing is a proprietary protocol providing seamless pairing capabilities, for instance between a user’s AirPods and all their Apple devices. This is achieved by synchronizing keys over Apple’s cloud service iCloud. The ultimate goal of the MagicPairing protocol is to derive a Bluetooth Link Key (LK) that is used between a single device and the AirPods. A fresh LK is created for each connection, which significantly reduces the lifetime of this LK.

When a new or reset pair of AirPods is initially paired with an Apple device belonging to an iCloud account, Secure Simple Pairing (SSP) is used [7, p. 271ff]. All subsequent connections between the AirPods and devices connected to that iCloud account will use the MagicPairing protocol as pairing mechanism. MagicPairing involves multiple keys and derivation functions. It relies on Advanced Encryption Standard (AES) in Synthetic Initialization Vector (SIV) mode for authenticated encryption [11].

The protocol mainly consists of five phases. The protocol flow is visualized in Figure 1 and explained in the following. MagicPairing depends on a shared secret between the two participants. Therefore, the first phase establishes and exchanges a secret, followed by phases of the actual protocol. As the protocol is not publicly documented, our naming relies on debug output and strings found in the respective components, i.e., the Bluetooth daemon bluetoothd for iOS and macOS, as well as the AirPod firmware. Further implementation and Bluetooth stack details follow later in Section 4.

2.1 Phase 1: Key Creation and Distribution

MagicPairing relies on a shared secret between the AirPods and a user’s iCloud devices, the Accessory Key (also accKey). This key is created by the first device pairing AirPods for a specific iCloud account. After establishing an encrypted Bluetooth connection using SSP, the Accessory Key needs to be transmitted to the AirPods. Apple is using the AAP Protocol\(^1\) for the Accessory Key transfer. In addition to the Accessory Key, the host also creates an Accessory Hint, which uniquely identifies the connection between an iCloud

\(^1\)AAP is used for communication between a device and AirPods. Its services all revolve around configuring AirPods and obtaining information from them, such as firmware updates, getting and setting tapping actions, or exchanging key material.
account and the target device. The initiating device uses the iCloud account’s Master Key and Master Hint to create the Accessory Key and the Accessory Hint. In case these Master credentials do not exist yet, the device provisions them by creating random bytes. Another component that is needed to create the Accessory Key and Accessory Hint is the so-called Bluetooth Address Blob, which is a deterministic mutation of the Bluetooth address of the targeted device, as shown in Listing 1. The Bluetooth Address Blob is then encrypted with the Master Key using AES in ECB mode to create the Accessory Key. The Accessory Hint is created by encrypting the Bluetooth Address Blob with the Master Hint, respectively.

After the initial setup, both devices share the same Accessory Key. All devices logged into the iCloud account can generate the same Accessory Key. In the following example, the device connects to the AirPods, but all steps could also happen in the opposite direction.

\[
\begin{align*}
\text{blob}[1:5] &= \text{address}[5:0] \\
\text{blob}[6:9] &= \text{address}[1:4] \oplus \text{address}[8:3]
\end{align*}
\]

Listing 1: Creating a Bluetooth Address Blob.

### 2.2 Phase 2: Hint

The first repeating phase in the MagicPairing protocol is the Hint phase. It ensures that both sides will agree on the same fresh session key in the end that belongs to the correct device. The device initiates the pairing by sending a Hint message. The Hint message includes three entries, the h1rt, a random nonce generated by the initiating host, and a Ratchet. The Ratchet is a counter used in later steps of the pairing process to rotate keys.

The receiving end performs a local Accessory Key table lookup for the connecting device. The AirPods use the h1rt that is included in the Hint message as a reference. iOS and macOS devices use the connecting device’s Bluetooth address to look up the key. If no key is found, the protocol is aborted with a Status Message indicating that the initiating device is unknown.

### 2.3 Phase 3: Ratcheting

The Ratcheting phase is essentially a key rotation and derivation phase. The goal of Ratcheting is to renew and maintain short-lived session keys [21]. First, the Accessory Key is rotated and then a SIV Key is derived from the rotated key. The Accessory Key is rotated by encrypting a buffer of 16 null-bytes with the current Accessory Key using AES in Electronic Codebook (ECB) mode. After one rotation step, the current counter, or Ratchet, is incremented. This is done until the local Ratchet equals the Hint’s Ratchet. Then, the SIV Key is derived from the Accessory Key by encrypting the static 32 B string bt_aessivauthentbt_aessivencrypt with the Accessory Key using AES in ECB mode. Next, an AES-SIV value is created. For this, the device creates a local random value, concatenates it with the received nonce and its own Bluetooth address, and encrypts it with the SIV Key. This time, AES is used in SIV mode without a nonce or any additional data. At the end of this phase, a Ratchet AES-SIV message is sent back to the initiating device. It contains the local Ratchet value, as well as the AES-SIV value. The initiating device executes the same key derivation steps as mentioned above using the received Ratchet value. This leads to both devices having the same updated Accessory Key and SIV Key. Using the derived SIV Key, the initiating device can now decrypt the AES-SIV value to unpack the random value of the responding device.

### 2.4 Phase 4: AES-SIV

The initiating device will now create another AES-SIV value. However, this one is different from the one that the responding device created before. First, the device creates a new random value. Then it concatenates its nonce value, the new random value, the previously received AirPods random value, and the h1rt value. This 64 B value is then encrypted with the derived SIV Key using AES in SIV mode and sent to the responding device. If the AirPods can decrypt the received data, they send a MagicPairing Status success message.

### 2.5 Phase 5: Link Key Derivation

Finally, a Bluetooth-compliant connection LK is derived. First, two Session Pre Keys are created and XORed. The Session Pre Key 1 is created by encrypting the responding device’s random value with the initiating device’s random value as key using AES. The Session Pre Key 2 is created by encrypting a 16 B null-byte buffer with the responding device’s random value using AES.

It is important to note that even though MagicPairing is a custom key derivation protocol, the further usage of this key is still compliant to the Bluetooth specification and does not require any modifications to the Bluetooth chip. When establishing an encrypted connection, the chip sends an HCI command to ask the host for the stored LK [7, p. 1948]. In case of MagicPairing, the LK is not taken from the host’s storage but freshly created, which is completely transparent to the chip. In either case, the LKS are stored on the host. However, it is only short-lived within MagicPairing, while it is permanent for a normal Bluetooth pairing.

### 3 SECURITY PROPERTIES

The security goals of the MagicPairing protocol seem to be to provide authentication and a fresh shared key for each connection. It uses a symmetric ratcheting algorithm and authenticated encryption to achieve these goals.

The idea of ratcheting was introduced by Borisov, Goldberg, and Brewer [8]. They introduced a continuous Diffie-Hellman key exchange providing forward and post-compromise secrecy within a session. Marlinspike and Perrin [21] extended this notion in the Double Ratchet algorithm to include a second, symmetric ratchet that updates the key while one party is offline. A Double Ratchet only provides forward secrecy, but no post-compromise secrecy.

The MagicPairing protocol uses only the symmetric ratcheting and therefore does not provide post-compromise secrecy. However, the usage of no expensive public-key cryptography makes this protocol feasible for usage with IoT devices like the AirPods. Further, note that MagicPairing uses ratcheting in a slightly different way than the previous work: Instead of creating a new key per message, the protocol creates a new key per Bluetooth connection. What is defined in the Double Ratchet algorithm as message key is therefore a connection key in this protocol.

In the Double Ratchet algorithm, the symmetric ratchet consists of a Key Derivation Function (KDF) that, given a chain key, produces a new chain key and an independent message key. This is done for each new message, so each new message gets encrypted with
a new key. As the KDF cannot be inverted, the knowledge of the chain key at some point only allows to calculate future chain and message keys, but no previous chain and message keys. Further, the knowledge of a message key does not enable an adversary to calculate any of the chain keys. MagicPairing uses two separate KDFs to accomplish the same goal: The first KDF,

$$KDF_1(k) = \text{enc}_{\text{ECB}}(k, 0^{16}),$$

is used to update the chain key. By using plain AES keyed with the old chain key to encrypt a constant (here: the bit string consisting of only zeros), it uses the Pseudo-Random Function (PRF) property of AES, which guarantees that without knowledge of the old chain key $k$, the new chain key is indistinguishable from a random key. The second KDF,

$$KDF_2(k) = (\text{enc}_{\text{ECB}}(k, c_1), \text{enc}_{\text{ECB}}(k, c_2)),$$

where $c_1$ is the string bt_aessi_authent and $c_2$ bt_aessi_encrypt, produces a connection key, which itself consists of two different key parts, an authentication and an encryption part. By the same argument as for $KDF_1$, the chain key cannot be calculated from the produced key and both key parts are independent.

The ratchet is initialized with the account key and the position in the ratchet is synchronized by the values ratchet_host and ratchet_airpod.

For the encryption of the messages between the host and the AirPod, AES is used in the SIV mode of operation. SIV, an authenticated encryption mode, was introduced by Rogaway and Shrimpton [24] and standardized in the combination with AES in RFC5161 [11]. It is used without any headers in MagicPairing, which is secure as long as the entropy of each message is high enough. As all messages encrypted with AES-SIV contain a new random number, the entropy is sufficient.

Finally, MagicPairing uses a third KDF to generate the key used for the Bluetooth connection, based on two random values, one generated by the host and one generated by the AirPod:

$$KDF_3(r_h, r_a) = \text{enc}_{\text{ECB}}(r_h, r_a) \oplus \text{enc}_{\text{ECB}}(r_a, 0^{16})$$

Again, this uses the PRF property of plain AES to generate a key that is indistinguishable from random as long as not both random values are known.

Knowledge of the final key implies knowledge of the SIV key, which in turn implies the knowledge of the account key, which identifies the party as being connected to the iCloud account. Further, for each connection a new key is used in a forward-secret manner. Therefore, the protocol meets the security goals of authentication and forward secrecy.

4 IMPLEMENTATION DETAILS

In the following, we discuss Apple-specific implementation details, which impact our security analysis. Section 4.1 compares the three Bluetooth stacks. As all of them differ significantly, the attack surface as well as bugs in their implementations vary. Section 4.2 lists the MagicPairing message formats, which are relevant for fuzzing the protocol, as well as understanding the fuzzing results and attacks. Section 4.3 explains the advertisements sent by AirPods, and based on these, connections are initiated. Finally, we spot many spelling mistakes, as shown in Section 4.4, which outline the MagicPairing code quality.

4.1 Apple’s Bluetooth Stacks

Apple uses three fundamentally different Bluetooth stacks in their recent devices. Each stack is for an individual device type and supports a subset of features. Thus, the protocols they support have duplicate implementations. While this circumstance helps us to reverse engineer these protocols, it raises maintenance overhead for Apple. From a security perspective, this results in different issues in these stacks, as shown later in Section 6.

RTKit is a separate framework for resource-constraint embedded devices. While this separation to reduce features makes sense, also iOS and macOS have individual Bluetooth stacks. As they are closed-source and there is only little public documentation, we provide an overview in the following. Figure 2 compares all stacks.

4.1.1 macOS. The most recent version of the macOS Bluetooth stack was investigated and documented previously to integrate InternalBlue [28]. The macOS kernel exposes a user-space 10Kit device-interface for Bluetooth [4]. 10Kit communicates using a Mach port with the IOBluetoothFamily driver, which supports connectivity to USB, Universal Asynchronous Receiver-Transmitter

Figure 2: Apple’s Bluetooth stacks: macOS, iOS, and RTKit.
(UART), and PCIe chips. User-space applications connect to Bluetooth devices using the 10Bluetoot private Application Programming Interface (API), which exposes methods to access the chip via HCI and send Asynchronous Connection-Less (ACL) data. The macOS blueoothd manages all Bluetooth logic and connects to other daemons such as blueoothauditd for music streaming. The public API to access Bluetooth on macOS is CoreBluetooth, which communicates with blueoothd via Cross-Process Communication (XPC) and further abstracts the methods exposed by 10Bluetoot.

4.1.2 iOS. Apple’s mobile operating system is iOS and has derivatives called iPadOS, tvOS, and watchOS. On iOS, the Bluetooth chip is exposed as serial character device to the user-space. On initialization, blueoothd directly connects to the exposed Bluetooth socket of this character device. Then, blueoothd offers Bluetooth-related functionality as an XPC service. Similar to macOS, this XPC service is accessed by the public CoreBluetooth API. However, iOS CoreBluetooth does not allow apps to create and use Classic Bluetooth connections, which is slightly different from macOS. Instead, it offers a higher-level application protocol called External Accessory that can be used in combination with Made for iPhone/iPad/iPod (MFi) certified Bluetooth devices [5].

Even though HCI is not openly accessible, it is needed by system components. blueoothd exposes a Mach port for features like HCI, which is only accessible by system components. This private framework is called MobileBlueTooth.

4.1.3 RTKit and Marconi. For embedded devices, Apple is using a real-time operating system based on the RTKit framework. RTKit is used on multiple embedded controllers and in all recent Bluetooth peripherals, such as the AirPods 1, 2, and Pro, Siri Remote 2, Apple Pencil 2, and Smart Keyboard Folio. While RTKit is not well-known, it has an incredibly high market share. For example, AirPods are accounted for 60% of the global wireless earbud market [23]. Moreover, RTKit powers a number of other devices and chips in the Apple ecosystem, such as the Always-On Processor (AOP) firmware included in most of Apple’s mobile devices like the iPhone and AppleWatch.

The RTKit framework lacks public documentation by Apple but has been briefly mentioned by other researchers [17]. The newest AirPod Pro firmware strings reveal version information, such as RTKitAudioFramework@2, RTKitOSPlatform=628.60.2616, and RTKitDefaultSDK@2. The latter lets us conclude that Apple has an internal Software Development Kit (SDK) used to develop RTKit applications.

We consolidate all these peripherals into a single Bluetooth stack, however, their firmware is very different due to their technologies and use cases. The Siri Remote, Apple Pencil, and Smart Keyboard only use BLE, while the AirPods rely on both BLE and Classic Bluetooth. Nonetheless, the basic RTKit code is the same.

On the AirPods, the communication to the Bluetooth chip is provided via the Apple Controller Interface (ACI) instead of the specification-compliant Host Controller Interface (HCI). This is because the AirPods use Apple’s new Bluetooth chip Marconi. An older version of the PacketLogger contains a file ACI_HCILib.xm, which names and partially describes all ACI commands. Some of these are AirPod-specific, such as synchronization of a pair of AirPods and primary to secondary switching. The Marconi Bluetooth chip firmware itself is also based on the RTKit framework, as it is just another peripheral.

Note that it is very complex to debug root causes for crashes on the AirPods. As they are an embedded device, they reboot within approximately 2 s. Thus, a Bluetooth connection reset is indistinguishable from a device reboot when performed wireless tests.

4.2 MagicPairing Messages

The general layout of a MagicPairing message is shown in Figure 3a. It starts with a 2 B header, which is followed by data, depending on the type of the message. In general there are two different types of messages with a slightly different structure. The first type, a Key Message, contains key material (such as the AES SIV, Ratchet, or Hint data). The second type, a Short Message, contains just one byte of data after the header. The data in the Key Message is in a Type Length Value (TLV) structure, as shown in Figure 3b. The number of keys is encoded after the header. The Short Message contains a fixed amount of data after the header.

The MagicPairing Ping message can initiate the protocol. When a device receives a Ping message, it replies with a Hint message. While the Ping message does not necessarily need any additional data, it is still 3 B with the data set to 0x00. The Status message indicates success or, in case of an error, the reason for failing. The Ratchet type message seems to be currently unused, as its reception handler implementation is empty on iOS and macOS blueoothd.

4.3 MagicPairing AirPods Advertisements

In addition to the pairing mechanism provided by MagicPairing, it also offers the capability to decrypt BLE advertisements sent by the AirPods. These advertisements have been shown to be linkable to AirPods in general [20]. Advertisements notify other Apple devices of the presence of the AirPods and encode battery state information. When an iOS device receives advertisements for a pair of AirPods that belong to the same Apple ID as the iOS device, a pop-up shows an AirPod image, the name of the AirPods, and the current battery state. The encrypted part constitutes the MagicPairing data. A new key is introduced, which is called MagicPairing EncryptionKey.

4.4 Code Quality

The MagicPairing implementations on iOS and macOS contain various spelling mistakes in logging messages, and in case of the macOS Blueoothd also in function names. For example, the words Ratchet and Upload were spelled differently various times. As these mistakes vary with the stack, each stack was probably implemented by a different developer. While spelling mistakes are not directly related to flaws in an implementation, they leave the impression the code was not extensively reviewed, and development probably outsourced.

2 A character device is exposed for all Broadcom UART chips, which are at least present in the iPhone 6, SE, 7, 8, X, XR, and various iPads. iOS also supports Marconi (newer AppleWatches) and Broadcom PCIe (iPhone XS and 11) Bluetooth chips.
5 FUZZING WITH TOOTHPICKER

The wireless attack surface of MagicPairing is rather large. First of all, it is available prior pairing—it provides a connection via the Logical Link Control and Adaptation Protocol (L2CAP), which is used for all kinds of data transfer within Bluetooth [7, p. 252]. Second, the MagicPairing attack surface is further enlarged by the different implementations for iOS, macOS, RTKit. Instead of using a common library, the macOS implementation is written in Objective C, the iOS implementation is based on C/C++, and the RTKit firmware on the AirPods is a slightly feature-restricted variant written in C [13]. Last, MagicPairing is always available on all Apple devices with Bluetooth enabled, no matter if the user owns AirPods.

Based on our knowledge about MagicPairing and its implementations, we perform further tests. We implement both, a generic over-the-air fuzzer (Section 5.1) and an iOS in-process fuzzer (Section 5.2). While the over-the-air fuzzer is platform-independent and required to confirm vulnerabilities, it is limited in speed and does not provide coverage. In contrast, the iOS in-process fuzzer is faster and not limited by connection resets, but needs a lot of platform-specific tuning. Our overall setup is explained in Section 5.3. As we apply a rather specific tooling to enable iOS in-process fuzzing with FIIDA, we further describe it in Section 5.4

5.1 Over-the-Air Fuzzing

An over-the-air fuzzer runs independently of the target system. Still, the protocol needs to be re-implemented to fuzz inputs. Our fuzzer extends InternalBlue, which already provides a generic interface to add custom protocols on top of existing Bluetooth stacks, including iOS and macOS [19]. This approach has two main advantages that cannot be reached with in-process fuzzing.

(+) Few False Positives The fuzzer behaves just as any other Bluetooth peripheral. Anything found can be used comparably easy for a PoC.

However, wireless Bluetooth fuzzing has various limitations that motivate us to also perform in-process fuzzing.

(-) Connection Termination The connection is terminated once a few invalid packets are received. Thus, a lot of time is spent on reconnecting to the target. Moreover, it is difficult to nearly impossible to distinguish between a terminated connection and a crashed Bluetooth daemon.

(-) Speed The fuzzer’s speed is limited by the physical connection to the target.

(-) Coverage Without collecting information from the target, the input cannot be adapted to trigger missing code paths.

As the MagicPairing protocol has a low complexity, we implement a generation-based fuzzer, which generates all possible message types. It randomly generates valid and invalid messages based on the reverse-engineered protocol definition. Apart from connecting to the target device, no further setup is required for fixed L2CAP channels. The fuzzer keeps sending the generated L2CAP payloads until it receives an HCI_Disconnection_Complete event (see [7, p. 2296]). This indicates that the target device either disconnected due to multiple invalid received messages or due to a crash. The fuzzer then tries to reconnect to the device.

The target device is additionally monitored using the PacketLogger, which is available for macOS and also on mobile devices since iOS 13 with a Bluetooth Profile [3]. This enables us to determine if the device crashed and when the connection was terminated. A crash can be detected by searching for the message “Connection to the iOS device has been lost.”

In practice, the HCI_Disconnection_Complete event is quite unreliable. In multiple occasions the connection was terminated, but the fuzzer did not receive the event. This lowers the efficiency of
the fuzzer as it needs to estimate when a connection is terminated in case it did not receive the disconnection event. Moreover, to reliably send packets and confirm events, we restricted the fuzzer speed to 1–2 packets/s. Despite the mentioned issues, the fuzzer ended up finding multiple bugs in the protocol implementations.

5.2 In-Process Fuzzing

An in-process, coverage-guided fuzzer improves the efficiency when fuzzing the reception handlers of interesting L2CAP-based protocols. While the throughput of messages and the stability of the payload delivery is much higher than in the over-the-air implementation, the in-process fuzzer comes with a different set of drawbacks.

(-) Many False Positives The usual operation of the Bluetooth daemon is altered, which can lead to unexpected behavior or crashes that are related to the fuzzing operation itself.

(-) Platform Dependence Injecting and preparing the fuzzer inside the target process differs significantly for different operating systems. Even within the same Bluetooth stack, function addresses and implementation details change with updated versions and need to be adapted.

We reduce the false positives by minimizing the amount of crashes related to the injected fuzzing code. This is done by observing any side-effects during fuzzing and patching the affected functions.

RTKit currently cannot be altered, thus, only the macOS and jailbroken iOS Bluetooth stack remain for in-process fuzzing. As iOS jailbreaks were comparably rare in the past but became available with checkm8 and checkra1n recently [16], we implement an in-process fuzzer for iOS.

In practice, the iOS in-process fuzzer’s speed varies between 5–30 packets/s. Though, as connections are not dropped with in-process fuzzing, the overall speedup is much higher.

5.3 Setup Overview

Figure 4 shows the fuzzing setup, with a main focus on the specialized in-process fuzzer. The in-process fuzzer is divided into two components: (1) The manager running on a computer, and (2) the fuzzer itself running on the target device.

The manager starts and maintains the fuzzing process. It injects fuzzing harness into the target process and handles the communication with it. Additionally, it maintains a set of crashes occurred during fuzzing, a corpus to derive inputs, and coverage information collected during fuzzing. The manager generates new inputs by sending entries to the corpus of the input mutation component, which randomly mutates the input based on a seed.

The fuzzing harness is divided into two sub-components. The first component is a general fuzzing harness, which is responsible for the overall fuzzing of bluetoothd. It creates virtual connections and applies patches ensuring a stable fuzzing process. Moreover, it collects code coverage and receives fuzzing input from the manager. The second component, the specialized fuzzing harness, is specific for the target function and protocol to be fuzzed, such as MagicPairing. It is responsible for preparing the received input and calling the function handler, as well as any other preparation needed to fuzz the protocol-specific reception handler function.

The fuzzer is initialized with an initial corpus of valid protocol messages, i.e., function arguments. It then collects the initial coverage by sending the initial corpus to the fuzzing harness. The specialized harness executes the payloads. The collected coverage is returned to the manager.

Once the initial coverage is collected, the actual fuzzing begins. The manager picks one of the entries in the corpus and a seed value. These are passed to the input mutator, which mutates the input and sends it back to the manager. The manager sends the mutated input to the specialized fuzzing harness. If desired, the specialized fuzzing harness further mutates the input—which is required for fields that require deterministic values or length fields.

In this case, the specialized fuzzing harness first reports the modified input back to the manager before calling the function under test. This ensures that the additional mutation is saved, even when the injected harness crashes together with the target. While the function is called, the harness collects basic block coverage. There are three possible results of the function call:

Ordinary Return The function was executed successfully and returns. The collected coverage is reported to the manager.

Exception The function results in an exception, which is returned to the manager. The manager stores the input and the exception as a crash.

Uncontrolled Crash The target, i.e., bluetoothd, crashes in a thread not controlled by the fuzzing harness. It crashes and generates a crash report. In this case, the exception cannot be sent to the manager. However, the manager detects this crash and stores the generated input as a crash. The corresponding crash report is manually gathered from the operating system.

These results may contain false positives, even in the case of an exception. Therefore, we verify identified crashes with the over-the-air fuzzer.

![Figure 4: ToothPicker fuzzing setup.](image-url)
5.4 Attaching the In-Process Fuzzer

Our in-process fuzzer is based on frizzer [18], which provides a basic fuzzing architecture including coverage collection, corpus handling, and input mutation. These are already a large part of the manager component. Our fuzzer, like frizzer, is built on FRIDA, which is a dynamic instrumentation toolkit [22]. FRIDA can inject code into a target process using JavaScript. Thus, our fuzzing harness is implemented in JavaScript and injected into bluetoothd.

The manager is implemented in Python, as FRIDA also provides Python bindings. We use the test case generator raddamsa as input generator component [14].

On iOS, bluetoothd is missing symbols. Nonetheless, we can identify various functions by static reverse engineering. These include creating a BLE handle, or creating an ACL handle, which is needed to receive L2CAP data. Due to the lack of symbols, we need to resolve function pointers via their static offsets to make them callable with FRIDA. In the following example, these offsets are valid for an iPhone 7 on iOS 13.3. In Listing 2, we call the function that creates an ACL handle. The input arguments are a Bluetooth address, and another state value set to 0 as found by dynamic analysis. Similar to calling the ACL connection creation function, we also call the specialized MagicPairing handler.

Even fake connections created as in Listing 2 can be disconnected. We keep the connection alive by overwriting the function `HCI_ReleaseConnection` as found by dynamic analysis. Similar to calling the ACL connection creation function, we also call the specialized MagicPairing handler.

Note that this in-process fuzzing disconnection prevention does not work for over-the-air fuzzing. When a connection is initiated by the Bluetooth chip itself, it holds an HCI handle, which the stack uses to reference the connection. Moreover, the chip holds additional state to keep the connection alive. While we can control bluetoothd with FRIDA hooks, we cannot overwrite chip-internal behavior.

```javascript
// Resolve function address
var fn_addr = base.add(0xc81a0); // iOS 13.3, iPhone 7
// Create JavaScript-callable function reference
var allocateACLConnection = new NativeFunction(fn_addr, "pointer", 
    ["pointer", "char"]);
// Write a (random) Bluetooth address to memory
bd_addr.writeByteArray([0xca, 0xfe, 0xba, 0xbe, 0x13, 0x37]);
// Call the function and create a forged ACL connection
var handle = allocateACLConnection(bd_addr, 0);
```

Listing 2: Creating a forged ACL handle using FRIDA.

6 VULNERABILITIES IN THE MAGICPAIRING IMPLEMENTATIONS

In the following, the identified vulnerabilities in the MagicPairing protocol are described. All vulnerabilities are summarized in Table 1.

6.1 Null Pointer Dereferences

Testing the MagicPairing protocol resulted in multiple NULL pointer dereferences or dereferencing addresses in the NULL page. The NULL page is not mapped on 64 bit iOS and macOS. This results in a bluetoothd crash. launchd immediately restarts bluetoothd after crashing. Thus, these bugs are merely a bluetoothd DoS. An attacker does not have any control over the dereferenced value, and we assume that these dereferences are not exploitable.

6.2 MP1: iOS Ratcheting

When sending a MagicPairing Ping message to an iOS device from a Bluetooth device that is not a known pair of AirPods, it responds that it does not have a hint for this sending device. If a Ratcheting message is then sent to the device, bluetoothd will crash while trying to dereference a pointer in the NULL page. Listing 3 shows an excerpt of the crash log that is generated by the operating system.

The invalid access to address 0xa8 is caused by a missing check for the return value of a lookup function shown in Listing 4. The function looks up an entry in bluetoothd’s table of known MagicPairing devices by the sender’s Bluetooth address and returns NULL. The issue is that this return value is never checked and assumed to be a pointer to a valid MagicPairing-related structure. Then, to respond to the Ratcheting message, the structure is accessed at offset 0xa8, which leads to the crash.

```c
void recv_mp_ratchet_aes_siv(char *bd_addr, char *data) {
    [...]
    // Returns NULL for unknown Bluetooth addresses
    mp_entry = lookup_mp_entry_by_bd_addr(bd_addr);
    [...]
    // The NULL entry is dereferenced with an offset
    memmove(mp_entry->remoteAESSIV, data + aessiv_offset, 0x36);
}
```

Listing 4: Pointer dereference MP1.

6.3 MP2–5: macOS/iOS Hint and Ratcheting

MP2–5 have a similar cause as the previous dereference in MP1. The return value of the lookup function is not properly verified. On iOS and macOS, this affects the Ratcheting (MP1, MP3, MP5) and the Hint (MP2, MP4) messages. As before, they lead to a dereference of an invalid address, which is a fixed offset into a MagicPairing structure at address 0x8. Thus, all vulnerabilities are equally unlikely exploitable other than crashing bluetoothd. The reason why the Ratcheting messages lead to different crashes on iOS is that the order of keys in the message determines which fields in the mp_entry are accessed.
### Table 1: List of identified MagicPairing and L2CAP vulnerabilities, status April 28 2020.

| ID   | Attack              | Effect       | Detection Method          | OS         | Disclosure       | Status |
|------|---------------------|--------------|---------------------------|------------|-----------------|--------|
| MP1  | Ratcheting          | Crash        | Over-the-Air, In-Process  | iOS        | Oct 30 2019     | Not fixed |
| MP2  | Hint                | Crash        | Over-the-Air, In-Process  | iOS        | Dec 4 2019      | Not fixed |
| MP3  | Ratcheting          | Crash        | Over-the-Air              | macOS      | Oct 30 2019     | Not fixed |
| MP4  | Hint                | Crash        | Over-the-Air              | macOS      | Oct 30 2019     | Not fixed |
| MP5  | Ratcheting          | Crash        | In-Process                | macOS      | Mar 13 2020     | Not fixed |
| MP6  | Ratcheting Abort    | Crash        | In-Process                | macOS      | Mar 13 2020     | Not fixed |
| MP7  | Ratcheting Loop     | 100% CPU Load| Over-the-Air              | macOS      | Oct 30 2019     | Not fixed |
| MP8  | Pairing Lockout     | Disassociation| Manual                   | iOS & macOS| Feb 16 2020     | Not fixed |
| L2CAP1 | L2CAP Zero-Length  | Crash        | Over-the-Air              | RTKit      | Dec 4 2019      | Not fixed |
| L2CAP2 | L2CAP Groups       | Crash        | In-Process                | iOS 5–13   | Mar 13 2020     | Not fixed |

#### 6.4 MP6: Ratcheting Abort

This crash is caused by an assertion failure that leads to an abort. The code that parses the `Ratcheting` message attempts reading from the message buffer. An assertion ensures that it does not read beyond this buffer. However, if the assertion fails, the parser does not return gracefully and instead calls `abt`, which leads to the termination of `bluetoothd`.

#### 6.5 MP7: Ratcheting Loop

The `macOS` `bluetoothd` can be forced to enter a ratcheting loop with a very large iteration count. Unlike the previous vulnerabilities, this issue is not solely caused by implementation mistakes, but originates from an inherent problem in the protocol’s design. The receiver trusts the values sent in the `Hint` message, without verifying that it was actually sent by a known MagicPairing peer. An attacker can forge the `Ratcheting` value in the `Hint` message. The `Hint` message also includes a nonce, but this is random. The `Hint` value itself, which is encrypted and could be used to verify the sender’s Bluetooth address, is ignored. Instead, `macOS` trusts the connection’s Bluetooth address.

Setting the `Ratchet` to a very high value will cause `bluetoothd` to enter a long ratcheting loop. The `Ratchet` field holds a 4 B value, thus the maximum value of a `Ratchet` can be `0xffffffff`. During normal usage however, the `Ratchet` is only incremented for every pairing process. Therefore, it is rather small in practice. The attack was tested on a `MacBook Pro Early 2015`, 13-inch, 2.9 GHz Dual Core i5 on `macOS Catalina 10.15` with an initial `Ratchet` value of 2. Sending a `Hint` message with a `Ratchet` value of `0xffffffff` caused `bluetoothd` to enter a ratcheting loop, with the local `Ratchet` value increasing at a rate of approximately 7000/s—causing a ratcheting loop running multiple days.

During the ratcheting loop attack, the `bluetoothd` reception thread is blocked. This disables further Bluetooth-based communication, for example, the device under attack can no longer receive files via `AirDrop`.

#### 6.6 MP8: Pairing Lockout

It is possible to corrupt the established pairing between an `iOS` or `macOS` device and a pair of `AirPods`. For this, an attacker needs to know the victim’s Bluetooth address, as well as the target `AirPods` Bluetooth address. The attacker can manipulate the local ratchet value of a host device by sending one or more `Ratcheting` messages with a ratchet value higher than the device’s current one. The current ratchet value can be obtained by sending a `Ping` message to the host. It responds with a `Hint` message, which contains its current local ratchet value. This value can then be incremented and sent in a `Ratcheting` message. The keys for encrypting the `AES-SIV` value are not required, as the ratchet value is sent in plaintext. Therefore, an attacker can set a bogus value for the `AES-SIV` part of the message and set the incremented ratchet value. Then, the receiving host starts a ratcheting loop. As `bluetoothd` on `iOS` has a timeout functionality, the forged ratchet value should not be chosen too high. Once the ratcheting loop is finished, the host’s local ratchet value is successfully increased, even if the decryption of the `AES-SIV` entry of the message fails. This corrupts an active pairing because the `AirPods` have a threshold value for the discrepancy between their local ratchet value and the value received by the paired host.

This causes the `AirPods` to decline the continuation of the MagicPairing protocol and thus the whole pairing process. The user does not have any options to reset the MagicPairing data and does not get any feedback about the error. The only solution is to reset the `AirPods` and freshly pair them with the user’s `iCloud` account.

As shown in Figure 5, the attack can be conducted as follows:

---

**Figure 5: Lockout attack.**
(1) The attacker changes the Bluetooth address to that of the target’s AirPods.

(2) The attacker connects to the victim and sends a Ping message\textsuperscript{3} to initiate a MagicPairing process.

(3) The victim responds with a Hint message which contains its current local ratchet value.

(4) The attacker increases this value by 10 and sends a Ratcheting message with the incremented ratchet value and a random AES-SIV value.

(5) The victim will start the ratcheting loop with the received ratchet value and derive the SIV key for decrypting the AES-SIV value. As the AES-SIV value is random, the victim will not be able to decrypt it and sends a Status message indicating an internal error. However, its local ratchet value stays incremented and is not reset to its previous value.

The issue originates from using an untrusted ratchet to increment an internal value and execute a key rotation. As the ratchet value is neither encrypted nor authenticated, an attacker can easily forge the ratchet.

A solution to this problem is to only store the incremented ratchet value and the rotated key when the AES-SIV part of the message was successfully decrypted. Otherwise, the whole MagicPairing message should be considered untrusted and the ratchet value should stay as it was before.

6.7 L2CAP1: L2CAP Zero-Length

While fuzzing MagicPairing over-the-air, we identified a crash in the RTKit Bluetooth stack, more specifically, the AirPods 1 and 2. When sending an L2CAP message with the length field set to zero and no payload, the AirPods crash. As there are no publicly documented debugging capabilities for the AirPods, it is not possible to tell whether the Bluetooth thread or the whole operating system crashes. We observe that the music stops playing, the connected iPhone reports the AirPods as disconnected, and after a few seconds, the AirPods play a sound indicating a successful connection.

6.8 L2CAP2: L2CAP Groups

This crash is another NULL pointer dereference, albeit more severe than the previous ones. It is accessible via both BLE and Classic Bluetooth and is part of L2CAP Group feature. This is indicated by logging messages in the crashing function that mention the file corestack/12cap/group.c. However, the L2CAP Group feature is no longer supported since Bluetooth 1.1. We assume the group reception function has been accidentally left in the code. In the newest Bluetooth specification, the channel ID 0x0002 is reserved for connectionless traffic instead of group traffic [7, p. 1035].

Depending on the data that is received, the L2CAP Group handler tries to find a matching entry in a function table allocated on the heap. However, this table has only been allocated, not initialized. Thus, all its entries are zero. When the payload starts with a NULL byte, the first entry is identified as matching entry. The code then tries to jump to the function pointer stored in that table entry, which also is a NULL pointer. However, any control over this table would immediately result in control over the instruction pointer.

In addition to an iPhone 7 on iOS 13.3, we were able to reproduce the crash on an iPad 2 with iOS 9.3.5 (released on August 25 2016), and an iPhone 4 with iOS 5.0.1 (released on November 10 2011). While the crash is not critical per se, it shows how long the iOS Bluetooth stack has not been tested. As iOS 5 and 9 still had another Bluetooth stack architecture, the crash is within BTServer instead of bluetoothd.

7 CONCLUSION

In this paper, we showed how Apple deals with seamless pairing of Bluetooth peripherals in their large connected ecosystem. While MagicPairing is proprietary, its general ideas and techniques can be integrated into other IoT ecosystems. Furthermore, other Bluetooth peripheral vendors could benefit from the MagicPairing protocol and infrastructure. All Apple needs to do is to provide an API that lets developers generate and receive an Accessory Key that is stored in the user’s iCloud account. Vendors could then implement MagicPairing in their products and benefit from the same security properties and seamless pairing experience as the AirPods.

Apple’s three different Bluetooth stacks for iOS, macOS, and RTKit also reflect the variety of Bluetooth implementations outside of their ecosystem. Many vendors choose to implement their own stacks and protocols. This makes efficient testing of Bluetooth devices challenging, but our over-the-air fuzzing setup based on InternalBlue can also be useful to test further Bluetooth stacks. As MagicPairing is a rather simple protocol, over-the-air fuzzing was sufficient to identify multiple vulnerabilities, despite the lack of speed and coverage information. However, our iOS-based in-process fuzzer had a better performance in practice.

Overall, Apple keeps their Bluetooth ecosystem rather closed to third-party vendors. Already using Classic Bluetooth requires them to apply for MFi. However, this enables an overall smooth user experience. Bluetooth runs silently in the background most of the time and manages tasks like AirDrop and Handoff [12, 27]. Since iOS 13, the Bluetooth icon has been removed from the status bar, even during audio streaming. Any incentive for disabling Bluetooth in the Apple ecosystem is missing.

While all of this is great for user experience, we were surprised by the vulnerabilities uncovered within MagicPairing. We assume that this protocol never had an extensive code review and was never fuzzed before integrating it as always-active Bluetooth background service. We are looking forward to Apple integrating patches for the vulnerabilities we identified, but also hope that they will elaborate their other wireless protocols better in the future.

ACKNOWLEDGMENTS

We thank Bianca Mix, Oliver Pöllny, and Alexander Heinrich for proofreading this paper. Moreover, we thank Matthias Hollick for his feedback and Anna Stichling for the ToothPicker logo.

This work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, as well as by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297.
REFERENCES

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. 2020. BIAS: Bluetooth Impersonation Attacks. In Proceedings of the IEEE Symposium on Security and Privacy (S&P).

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. 2019. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli. In 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, 1047–1061.

[3] Apple. 2020. Bug Reporting—Profiles and Logs. https://developer.apple.com/bug-reporting/profiles-and-logs/.

[4] Apple. 2020. Developer Documentation – IOKit. https://developer.apple.com/documentation/okkit.

[5] Apple. 2020. MFi Program. https://developer.apple.com/programs/mfi/.

[6] Eli Biham and Lior Neumann. 2018. Breaking the Bluetooth Pairing: Fixed Coordinate Invalid Curve Attack. http://www.cs.technion.ac.il/~biham/ BT/bt-fixed-coordinate-invalid-curve-attack.pdf.

[7] Bluetooth SIG. 2020. Bluetooth Core Specification 5.2. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[8] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. 2004. Off-the-record communication, or, why not to use PGP. In Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004, Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati (Eds.). ACM, 77–84. https://doi.org/10.1145/1029179.1029200

[9] Jiska Classen. 2019. All Wireless Communication Stacks are Equally Broken.

[10] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. 2018. Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware. In PAM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT).

[11] Dan Harkins. 2008. Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). RFC 5297. RFC Editor. https://tools.ietf.org/html/rfc5297

[12] Alexander Heinrich. 2019. Analyzing Apple’s Private Wireless Communication Protocols with a Focus on Security and Privacy.

[13] Dennis Heinze. 2020. ToothPicker: Enabling Over-the-Air and In-Process Fuzzing Within Apple’s Bluetooth Ecosystem.

[14] Aki Helin. 2020. radamsa - a general-purpose fuzzer. https://github.com/akihe/radamsa.

[15] Konstantin Hypponen and Keijo MJ Haataja. 2007. “Nino” Man-in-the-Middle Attack on Bluetooth Secure Simple Pairing. In 3rd IEEE/IFIP International Conference in Central Asia on Internet. IEEE.

[16] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. 2020. BIAS: Bluetooth Impersonation Attacks. In Proceedings of the IEEE Symposium on Security and Privacy (S&P).

[17] Jonathan Levin. 2019. New OS X Book, Volume II, “iOS Internals:Kernel Mode. 20–22 pages. http://newosxbook.com

[18] Dennis Mantz. 2019. Frida-based general purpose fuzzer. https://github.com/demantz/frizzer.

[19] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. 2019. Inter-nallBlue - Bluetooth Binary Patching and Experimentation Framework. In The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19). https://doi.org/10.1145/3307334.3326089

[20] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Poppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. 2019. Handoff All Your Privacy–A Review of Apple’s Bluetooth Low Energy Continuity Protocol. Proceedings on Privacy Enhancing Technologies 2019, 4 (2019), 34–53.

[21] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf

[22] Ole AndråF V. RavnÆs. 2020. Frida - A world-class dynamic instrumentation framework. https://frida.re/.

[23] Don Reisinger. 2019. Apple’s AirPods Business Is Bigger Than You Think. https://fortune.com/2019/08/06/apple-airpods-business/.

[24] Phillip Rogaway and Thomas Shrimpton. 2006. A Provably-Secure Treatment of the Key-Wrap Problem. In Advances in Cryptology – EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings (Lecture Notes in Computer Science), Serge Vaudenay (Ed.), Vol. 4004. Springer, 373–390. https://doi.org/10.1007/11761679_23

[25] Mike Ryan. 2013. Bluetooth: With Low Energy Comes Low Security. In Presented as part of the 7th USENIX Workshop on Offensive Technologies. https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf

[26] Shaked, Yaniv and Wool, Avishai. 2005. Cracking the Bluetooth PIN. In Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services. ACM.

[27] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Gisvarya Noulus, and Matthias Hollick. 2019. A Billion Open Interfaces for Eve and Mallory: MiM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. https://www.usenix.org/conference/usenixsecurity19/presentation/stute. In 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, 37–54.

[28] Davide Toldo. 2019. Analyzing the macOS Bluetooth Stack.

[29] Maximilian von Tschirschnitz, Ludwig Peuckert, Fabian Franzen, and Jens Grossklags. 2020. Method Confusion Attack on Bluetooth Pairing. In Under submission.