ToothPicker: Apple Picking in the iOS Bluetooth Stack

Dennis Heinze
Secure Mobile Networking Lab
TU Darmstadt /
ERNW GmbH

Jiska Classen
Secure Mobile Networking Lab
TU Darmstadt

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt

Abstract

Bluetooth enables basic communication prior to pairing as well as low-energy information exchange with multiple devices. The *Apple* ecosystem is extensively using Bluetooth for coordination tasks that run in the background and enable seamless device handover. To this end, *Apple* established proprietary protocols. Since their implementation is closed-source and over-the-air fuzzers are very limited, these protocols are largely unexplored and not publicly tested for security. In this paper, we summarize the current state of *Apple*'s Bluetooth protocols. Based on this, we build the *iOS* in-process fuzzer *ToothPicker* and evaluate the implementation security of these protocols. We find a zero-click Remote Code Execution (RCE) that was fixed in *iOS 13.5* and simple crashes.

1 Introduction

In the past, Bluetooth was mainly used as an audio-only technology. Nowadays, it is used by integral operating system services that are permanently running in the background. Within the *Apple* ecosystem, most of these are part of the *Continuity* framework [5]. Examples are searching for devices before sharing files via *AirDrop* [31], the initial detection of an *Apple Watch* by *macOS* to start the *Auto Unlock* protocol and unlocking a *Mac*, or beginning to write an email on a mobile device and later continuing this on a desktop via *Handoff* [14, 23]. Further features outside of the *Continuity* framework include pairing *AirPods* once and seamlessly using them on all devices that are logged into the same *iCloud* account [15]. With *Exposure Notifications* introduced due to SARS-CoV-2, *Apple* expanded their rather closed ecosystem to work with *Google* devices [12]. Even users with just an *iPhone* who do not depend on the *Continuity* framework might enable Bluetooth for exposure notifications or audio streaming.

Testing these services over-the-air is cumbersome. *Apple*'s Bluetooth stacks almost immediately disconnect upon receiving invalid packets. A disconnect is hard to distinguish from a crash because the Bluetooth service restarts within seconds, thus, rendering feedback-based fuzzing difficult. Moreover, crafting arbitrary packets and transmitting them for Bluetooth Low Energy (BLE) as well as Classic Bluetooth has limited tool support. As of now, there is no full-stack open-source Software-Defined Radio (SDR) implementation. Thus, security researchers need to use commercial tools by *Ellisys* starting at US$ 10 k or extend *InternalBlue* [22], which is based on reverse-engineered firmware on off-the-shelf smartphones. During our research, *InternalBlue* proofed as powerful tool for bug verification but the overall nature of *Apple*'s Bluetooth stack requires a custom fuzzing solution.

We design and implement *ToothPicker*, an in-process Bluetooth daemon fuzzer running on *iOS 13*. To this end, we solve various harnessing challenges, such as creating virtual connections, partial chip abstraction, selecting protocol handlers, obtaining suitable corpora for undocumented protocols as well as getting crash and coverage feedback of this closed-source target. *ToothPicker* is based on FЯIDA and *radamsa* [16, 28] and harnesses the Bluetooth daemon while leaving the remaining interaction with *iOS* components intact. The main contributions of this paper are as follows:

- We implement *ToothPicker*, an *iOS* in-process Bluetooth fuzzer running on a physical *iPhone*.
- We provide an up-to-date Bluetooth protocol overview for *iOS*, *macOS*, and *RTKit*.
- We uncover various issues in *Apple*'s implementations that we can reproduce over-the-air. Based on our findings, *Apple* fixed the Bluetooth RCE *CVE-2020-9838* in *iOS 13.5* and the Denial of Service (DoS) *CVE-2020-9931* in *iOS 13.6*.

Our results indicate that *Apple* never systematically fuzzed their Bluetooth stack, with issues going back to *iOS 5* that still exist in *iOS 13*. However, fuzzing BLE/GATT, which is used by most Internet of Things (IoT) devices and supported by various *Nordic Semiconductor* based testing frameworks [9, 25], did not reveal any findings. Thus, *ToothPicker* indeed closes the gap between what is possible with over-the-air testing and fast in-process fuzzing for proprietary protocols.

This paper is structured as follows. We provide background information in Section 2. In Section 3, we start with a *ToothPicker* design overview, identify and prioritize various target protocols, and then harness the *iOS* Bluetooth daemon for fuzzing. We evaluate *ToothPicker* in Section 4. The fuzzing results and malicious payloads are provided in Section 5. Finally, we conclude our work in Section 6. Even though a detailed understanding of *Apple's* proprietary protocols is not required for fuzzing per se, they have not been documented before and we provide descriptions in the Appendix.

2 Background on Fuzzing Apple Bluetooth

In the following, we explain why we chose the *iOS* Bluetooth stack and provide a background on the fuzzing options.

2.1 Selecting a Bluetooth Stack

Apple implements three different Bluetooth stacks. One is for *iOS* and its derivates. *macOS* uses another stack with duplicate protocol implementations that behave slightly different. Embedded devices, such as the *AirPods*, use the *RTKit* stack.

The embedded *RTKit* stack is the least accessible. In contrast, the *macOS* stack has already been tested recently with severe security issues uncovered [11]. We find that the *iOS* stack experienced only little research but implements the majority of *Apple*'s proprietary protocols. With the *checkra1n* jailbreak [18], it becomes fairly accessible on research *iPhones*.

2.2 *iOS* Fuzzing Options

Apple internally tests their software including fuzzing but it is unknown how exactly they do this. With neither the source code nor the fuzzing methods and targets being public, security of *Apple's* software should be tested by independent researchers. *Apple* keeps `bluetoothd` closed-source, meaning that only they are able to rebuild it with fuzzing instrumentation such as edge coverage and memory sanitization. As this binary is rather complex, statically patching it without source code is out of scope, as there is currently no such tool for *iOS* ARM64e binaries without symbols in Mach-O format.

In general, there are multiple tools for dynamic *iOS* analysis. The *iOS 12* kernel can be booted into an interactive shell using Quick Emulator (QEMU) but without any daemons running [1]. Moreover, KTRW enables *iOS 13* kernel debugging at runtime [7]. However, `bluetoothd` runs in user space. In contrast to the previous options, FЯIDA is a dynamic instrumentation framework that enables function hooking and coverage collection by rewriting instructions of user-space programs during runtime [28]. Thus, *ToothPicker* is based on FЯIDA and the fuzzing-specific *frizzer* extension [21].

The FЯIDA stalker follows program flow during runtime [27, 29]. To this end, FЯIDA copies code prior to execution, modifies that copy, and runs it. A significant speedup is achieved by adding a trust-threshold on code that is not modified during runtime, and, thus, has reusable blocks. The stalker observes basic block coverage, as required for feedback-based fuzzing. If exceptions occur while running the code copy, FЯIDA catches these and the original program does not crash.

3 *ToothPicker*

In the following, we design and implement *ToothPicker*. An overview of the *ToothPicker* architecture and its specifics is depicted in Figure 1. First, we provide an intuition about its design in Section 3.1. Then, we analyze various Bluetooth-based protocols and prioritize them as fuzzing targets in Section 3.2. In Section 3.3, we select corpora for these protocols and describe adaptions required for harnessing these. Finally, we describe the fuzzer operation in Section 3.4, including its virtual connection management in Section 3.5.

3.1 General Design and Concepts

The main challenge in fuzzing the *iOS* Bluetooth daemon is harnessing it while maintaining sufficient state to find realistic bugs. *ToothPicker* achieves this by attaching itself to the running daemon with FЯIDA [28]. However, fuzzing requires further extensions, as running on a physical *iPhone* has various side effects due to the remaining interactions with the Bluetooth chip, other daemons, and apps. To this end, *ToothPicker* creates virtual connections, partially abstracts between the physical Bluetooth chip and the changed behavior within `bluetoothd`, and gets coverage and crash feedback during fuzzing. Moreover, as fuzzing speed is limited, we have to understand *Apple's* proprietary protocols sufficiently to obtain meaningful corpora and select interesting protocol handlers.

Chip interaction can be harmful for the overall fuzzing process. For example, if a protocol that requires an active connection is fuzzed but the chip is not aware of such a connection, it reports the connection to be terminated. Yet, it is complicated to fully abstract from the chip, as `bluetoothd` requires it during startup and other complex operations. Thus, *ToothPicker* uses virtual connections and filters communication with the chip concerning these connections manually.

Another important factor are the daemons and apps that remain interacting with `bluetoothd` while it is being fuzzed. First of all, they might crash as well due to the fuzzing, with exceptions that cannot be caught by FЯIDA but are still captured within the *iOS* crash reports. Second, anything happening on the *iPhone* that also affects Bluetooth changes the fuzzing behavior. Simple instructions such as opening the Bluetooth device dialog already crash `bluetoothd` immediately. Moreover, the chip might still receive BLE advertisements or connection requests from other devices and forward them to `bluetoothd`. All this parallelism and statefulness often make the same fuzzing input result in different coverage.

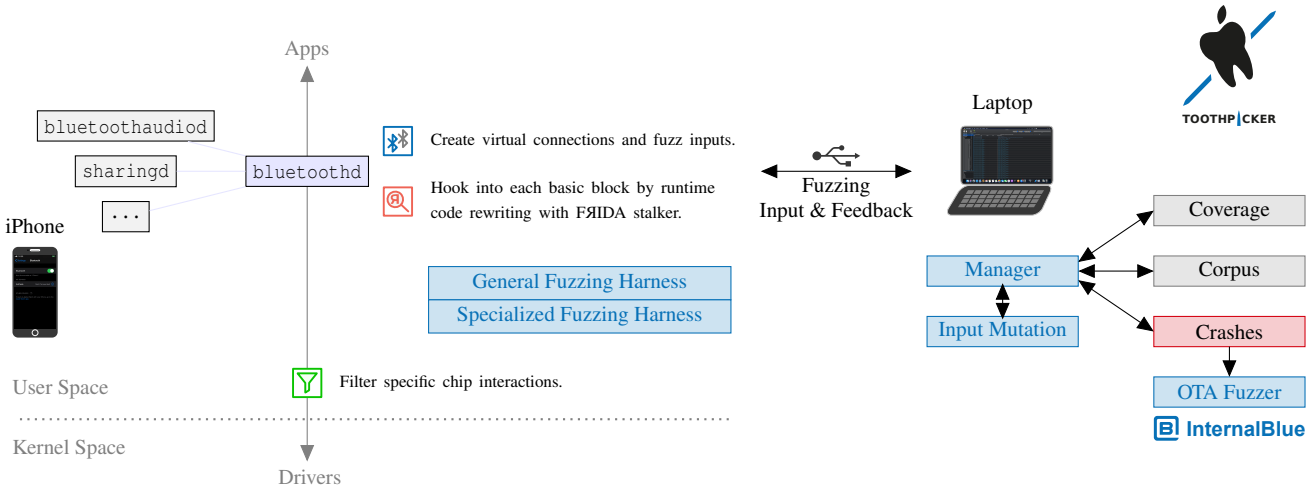


Figure 1: *ToothPicker* architectural overview and fuzzing setup.

The total `ToothPicker` fuzzing speed is limited by running on a physical *iPhone 7*. However, it is still significantly faster than over-the-air fuzzing. While over-the-air fuzzing based on *InternalBlue* achieves only 1–2 packet/s [22], *ToothPicker* speeds this up to 25 packet/s. The actual speedup compared to over-the-air fuzzing is even higher, as *ToothPicker* overwrites functions within `bluetoothd` that disconnect on invalid packets and optimizes its packets based on the coverage feedback. Nonetheless, running on an *iPhone* is a limitation and parallelization requires multiple *iPhones*.

While the overall FʁIDA-based setup is quite complex, the FʁIDA workflow also supports reverse-engineering proprietary protocols. Stalking and hooking allow observing protocols and intercepting valid payloads to build a corpus.

Fuzzed payloads are recorded during the fuzzing process. *ToothPicker* provides an over-the-air replay script for *InternalBlue*. This step is required to confirm that fuzzing results do not originate from any FʁIDA hooking side effects. Additionally, it provides Proofs of Concept (PoCs) that can be tested against arbitrary devices, including non-jailbroken *iPhones*, other *Apple* devices, and even non-*Apple* devices.

3.2 Target Protocol Selection for Fuzzing

Prior to fuzzing, we need to identify protocols that we can fuzz and prioritize them.

3.2.1 Attack Surface Considerations

The zero-click RCE surface within *iOS* Bluetooth only affects protocol parts that are available prior to pairing. Such attacks do not require any user interaction, an attacker within wireless range could take control over a device. Protocols that become available after pairing, such as tethering, hold significantly more state but also require user interaction.

Protocols without response channel, such as BLE advertisements including exposure notifications [12], miss the feedback required to bypass Address Space Layout Randomization (ASLR). However, if further Bluetooth services on the target stop sending packets and these can be assigned to a specific target device, this indicates `bluetoothd` crashes, which might be a sufficient ASLR bypass primitive as this daemon restarts automatically [13]. We explicitly skip BLE advertisements, because there has been exhaustive study of *Continuity* and *Handoff* already [10, 14, 23], and getting a meaningful response channel is rather complex if possible at all.

3.2.2 Initial Protocol Analysis

Protocols are either documented within the Bluetooth specification [8] or proprietary and *Apple*-specific. The latter are usually undocumented and often not publicly mentioned at all, yet they are available on hundreds of millions of *Apple* devices. The Bluetooth *PacketLogger*, which is included in the *Additional Tools for Xcode* [3], decodes specification-compliant protocols as well as many proprietary *Apple* protocols on *macOS* and *iOS*. Some parsers are only available within one specific version, such as chip memory pool statistics, and were likely forgotten to remove from non-internal builds. We use *PacketLogger* to initially observe and select protocols.

As these protocols are undocumented, we provide an overview in the Appendix. Note that fuzzing focuses on implementation bugs and RCEs. Understanding these protocols in detail is not required to this end. However, such details can reveal further issues. For example, custom Logical Link Control and Adaptation Protocol (L2CAP) echo replies leak the operating system type (see Section A.1.1) and LE Audio (LEA) leaks the *iOS* version (see Section A.1.3).

Table 1: List of *Apple*'s Bluetooth protocols and potential targets.

Category	Protocol	iOS	macOS	RTKit	Exposure	Specification	Knowledge	Target
Fixed L2CAP Channels	*BLE Security Manager*	✓	✓	?	↑	[8, p. 1666ff]	↑	
	BLE Signal Channel	✓	✓	?	↑	[8, p. 1046ff]	↑	
	Classic Security Manager	✓	✓	?	↑	[8, p. 1666ff]	↑	
	Classic Signal Channel	✓	✓	✓	↑	[8, p. 1046ff]	↑	✓
	Connectionless Channel	✓	✓	?	↑	[8, p. 1035]	•	
	DoAP	✓	✓	✓	?	—	•	
	FastConnect Discovery	✓	✓	✓	↑	—	↑	✓
	GATT	✓	✓	(✓)	↑	[8, p. 1531ff]	↑	✓
	LEAP	✓		✓	•	—	•	✓
	LEAS	✓		✓	•	—	↓	
	MagicPairing	✓	✓	✓	↑	—	↑	✓
	Magnet	✓	✓		•	—	•	✓
Dynamic L2CAP Channels	*AAP*	✓	✓	✓	↓	—	↑	
	Apple Pencil GATT	✓		✓	↓	—	↑	
	External Accessory (iAP2)	✓	✓	✓	↓	—	•	
	FastConnect	✓	✓	✓	↓	—	•	
	Magnet Channels	✓	✓		↓	—	↓	
	SDP	✓	✓	✓	↑	[8, p. 1206ff]	↑	✓
Other	*ACL (Classic+BLE)*	✓	✓	✓	↑	[8, p. 477ff]	↑	✓
	BLE Exposure Notification	✓			↑	[12]	↑	
	BRO/UTP			✓	?	—	↓	
	USB OOB Pairing		✓	(✓)	•	—	•	

Exposed describes how accessible the protocol is to an attacker. This includes protocol setup or authentication requirements of the protocol. There are three possible options *high* ↑, *medium* •, and *low* ↓. The *Specification* column indicates whether the protocol is a proprietary protocol by *Apple* or specified otherwise. *Knowledge* determines how much information of this protocol is either openly available or can be extracted by existing debug tools. *Targets* indicates that the protocol is targeted for further analysis. The brackets around the checkmarks on *RTKit* indicate that this protocol is not available on all *RTKit* devices.

3.2.3 Target Prioritization

ToothPicker can hook arbitrary functions within `bluetoothd` for fuzzing. On *iOS 13.5*, `bluetoothd` has 24 625 functions consisting of 153 620 basic blocks. However, only a few of those directly handle incoming data prior to pairing. This allows us to focus on selected protocols once identified.

In the following, we prioritize the protocols to derive which ones are interesting for further analysis. This prioritization shown in Table 1 is based on operating system, exposure prior to pairing, publicly available specifications, and the knowledge we were able to obtain about each protocol. The target column determines whether a protocol was chosen for further analysis. While there are other protocols from the Bluetooth specification that have a high accessibility and distribution within the operating systems, such as the *BLE Signal Channel*, or the *Security Manager* protocols, we mainly focus on Asynchronous Connection-Less (ACL), Generic Attribute (GATT), the *Classic Signal Channel*, and Service Discovery Protocol (SDP). Due to their low exposure we skip most dynamic L2CAP channel protocols.

3.3 Harnessing and Corpus Collection

For each target in Table 1, corpora need to be collected or generated. Moreover, to properly harness some of the protocols, they need additional optimizations. An overview of the corpora and their optimizations is listed in Table 2.

3.3.1 Initial Protocol Corpus

The second column in Table 2 describes how the corpus for the particular protocol was generated. The corpora are mostly generated by intercepting (i.e., *recording*) both the ACL reception and transmission functions while the specific data is sent over a real physical connection. In practice, this is implemented using a FЯIDA hook in the ACL reception handler and the Universal Asynchronous Receiver Transmitter (UART) write function that is used to send data to the Bluetooth chip. The data received by the ACL reception handler as first argument can be stored as a corpus file, as it is already of the format that *ToothPicker* expects. The data arriving at the UART write function still needs to be filtered, as other data, such as Host Controller Interface (HCI) commands, is also received by this function. By filtering for data starting with `0x02` [8, p. 1727], ACL data sent by `bluetoothd` can be captured.

In cases where this is not possible, the corpus is generated manually, for example for the LE Audio Protocol (LEAP) protocol. As we do not have access to any device using LEAP, the corpus was created by reverse-engineering the protocol and manually generating valid messages. In case of the *MagicPairing* protocol, we augment the recorded corpus with a manually created message type, that would otherwise not be included as it does not occur in regular connections. Figure 2 shows the corpus for *MagicPairing*.

Table 2: *ToothPicker* corpora and optimizations for fuzzing.

Protocol	Corpus	Optimizations	Technology
ACL (BLE)	• Manually created BLE Signal Channel messages	—	BLE
ACL (Classic)	• Record of L2CAP Echo Requests and Responses	—	Classic
Classic Signal Channel + FastConnect Discovery	• Record traffic by connecting and disconnecting *AirPods* • Record *FastConnect* Discovery messages	• Correct ACL length	Classic
GATT	• Record of interaction with GATT exploration app	• Correct ACL and L2CAP length • No fragmentation	BLE
LEAP	• Manually created by reverse engineering	• Correct ACL and L2CAP length • No fragmentation	BLE
MagicPairing	• Record of *AirPods* pairing • Manually created *Ping* message	• Correct ACL and L2CAP length • No fragmentation	Classic
Magnet	• Connect and disconnect *Apple Watch*	• Correct ACL, L2CAP, and *Magnet* length • Keep track of protocol version • No fragmentation	BLE
SDP	• Record traffic by connecting to *macOS* and query device with *Bluetooth Explorer*	• Correct ACL, L2CAP, and SDP length • No fragmentation	Classic

3.3.2 Specialized Protocol Harness

The third column in Table 2 describes the optimization that the specialized harness applies to the mutated fuzzing input. In most cases, there are three optimizations: correcting the L2CAP length field, the ACL length field, and the flags in the handle that determine the L2CAP fragmentation. While correcting the length field removes the fuzzing input's randomness, it ensures that the generated input packets do not unnecessarily fail at length checks and rather reach deeper into the actual parsing code.

We wrote an additional script that intercepts various logging functions and prints the messages to monitor the fuzzing process. While *iOS*'s standard logging (e.g., via the `Console.app` on *macOS*) could be used to monitor the logs, our approach has the advantage that even internal debug logging becomes visible. Before some of `bluetoothd`'s logging calls, there are checks that determine whether it is an internal build. We patched `bluetoothd` in such a way that it assumes it is an internal build to increase the verbosity even more. Monitoring the logs turns out to be useful to observe certain behavior.

During the initial fuzzing rounds of the *Magnet* protocol, we observed that the majority of these internal logging messages were concerning an invalid length field. This leads us to two additional optimizations for the *Magnet* protocol. First, *Magnet*'s own length field is correctly calculated, and second, the fuzzer keeps track of which version is currently negotiated. This is important for correcting the length field, as the offset and the size of the *Magnet* length field changes. Keeping track of the current version is as simple as monitoring the generated packets. As soon as a *Version* packet is sent, the specialized harness adapts the following version fields accordingly.

The last column in Table 2 indicates the Bluetooth technology (Classic Bluetooth or BLE). Depending on the technology the harness has to adapt the creation of the virtual connection.

Ping Message

```
0b20 0600 0200 3000 f001
```

Hint Message

```
0b00 3900 3500 3000 0101 0310 0010 00f3 2871 7bb7 6ed6
f2b6 07d3 0d1c 5c47 fc20 0010 00f3 9028 c260 502e 08d4
2e62 69c6 aa2a 1900 0104 0001 0000 00e8 d4
```

Ratchet Message

```
0b20 4900 4500 3000 0201 0280 0036 00a4 fac9 831d c027
32d8 ddab 70e9 ff9c b63e 37f6 7c4f e47d 956d 394e 4f40
ed91 bd66 5a12 1ea0 356f 05f3 8d70 dca4 23d3 7136 a767
50bf 5f00 0104 0001 0000 00
```

Ratchet AES SIV Message

```
0b00 5b00 5700 3000 0301 0180 0050 0002 aad4 1dca d96e
96e6 6877 9e9b 7b1d 142d f683 a623 d287 4175 b0bb deba
271a 275c a669 e849 2a36 c1fb 9ee3 e0d6 1373 0f9c 41b5
4390 ec69 baf4 115a 3a45 6586 dcca b94e c0f8 8742 8c10
e043 9bee 205d 52
```

Status Message

```
0b20 0700 0300 3000 ff01 00
```

Status Message (Error)

```
0b00 0700 0300 3000 ff01 08
```

Figure 2: *MagicPairing* corpus.

3.4 Fuzzer Internals

In the following, we provide technical details about the fuzzer. Figure 1 also contains the fuzzer's components.

3.4.1 Underlying Technology

We implement *ToothPicker* based on *frizzer* [21]. *frizzer* provides the basic fuzzing architecture, like coverage collection, corpus handling, input mutation, and thus a large part of the manager component. *ToothPicker*, like *frizzer*, is built on FЯIDA, which is a dynamic instrumentation toolkit [28].

With FЯIDA, custom code can be injected into a target process in the form of *JavaScript* code. Thus, the fuzzing harness is implemented in *JavaScript* and injected into `bluetoothd`. The manager is implemented in *Python* by using FЯIDA's *Python* bindings. The test case generator *radamsa* acts as the input generation component [16].

3.4.2 Fuzzer Components

The fuzzer consists of two components, the manager, running on a computer, and the fuzzing harness running on an *iOS* device. These components work as follows:

Manager The manager is responsible for starting and maintaining the fuzzing process. It injects the fuzzing harness into the target process and handles the communication with it. In addition to that, it maintains a set of crashes that occurred during fuzzing, a corpus to derive inputs from, and the coverage that was achieved during fuzzing. New inputs are created by the input mutation component. Based on a seed it randomly mutates data from the corpus to derive new input data. As of now, the only supported input mutation is provided by *radamsa* [16], and we correct length, fragmentation and version fields as listed in Table 2.

Harness The fuzzing harness consists of two parts: a general fuzzing harness and a specialized fuzzing harness. The general fuzzing harness is responsible for all general operations required for fuzzing `bluetoothd`. It can create virtual connections and applies the necessary patches that are required for a stable fuzzing process. It also provides the means to collect code coverage and receives the fuzzing input from the manager. The specialized harness is specific to the target function and protocol that is to be fuzzed. It is responsible for preparing the received input and calling a protocol's function handler, as well as any other handler-specific preparations.

3.4.3 Fuzzer Operation

ToothPicker is initialized with a corpus of valid protocol messages. More specifically, this corpus consists of function arguments for the fuzzed protocol handlers. The fuzzer then starts to collect the initial coverage by sending the initial corpus to the fuzzing harness, which, by using the specialized harness, executes the payloads. The collected coverage is then returned to the manager, which stores it for later use.

Once the initial coverage is collected, the actual fuzzing starts. Each iteration works as follows. The manager picks one of the entries in the corpus and sends it along with a seed value to the input mutator. The mutator then mutates the input and sends it back to the manager. Afterward, the manager proceeds by sending the input to the specialized fuzzing harness. If desired, it can mutate the input further. This is useful in cases with input fields that require deterministic values or length fields that, for certain cases, should be correct. The specialized fuzzing harness can modify the input and send it back to the manager. Sending back the modified input before actually calling the function under test is required for cases where the target crashes as a result of the input. As the injected harness crashes together with the target, the modified input would be lost. Once the manager receives the modified input, the target function can be called with this input.

Usually, the protocol reception handlers within `bluetoothd` run in a separate reception thread—in our case, this thread is called `RxLoop`. Since `bluetoothd` keeps operating normally except from hooked functions, the `RxLoop` continues calling functions within `bluetoothd` in case it receives data such as BLE advertisements. Any function call within the `RxLoop` could interfere with our fuzzing. On *iOS*, many actions could trigger sending or receiving Bluetooth packets. Thus, during fuzzing, the *iPhone* should be in *Do not disturb* mode, locked, and with Wi-Fi switched off to ensure stable fuzzing results. Also isolating the *iPhone* from other sources of interference by wrapping it in tinfoil can help.

The fuzzing harness runs in its own thread, which is a FЯIDA-specific behavior. This thread then calls the target function. This has the advantage that a custom exception handler can be implemented for this thread. If an exception occurs while fuzzing a function that would normally result in a crash, this handler can catch the exception and terminate gracefully without crashing `bluetoothd`. While the function is called, the harness is collecting basic block coverage. After calling the function, there are three possible outcomes:

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

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

3. Uncontrolled Crash In case the target crashes in a thread or external component not controlled by the fuzzing harness via the FЯIDA exception handler, it will crash and generate a crash report. In this case, the exception cannot be sent to the manager. However, the manager detects a crash and can store the generated input as a crash. The corresponding *iOS* crash report can be manually gathered from the *iPhone*.

Even in the case of an exception, this might be a false positive. Therefore, it is important to verify the identified crashes. This is done by using an over-the-air fuzzer based on *InternalBlue*, which opens a connection and then replays crashes stored by the manager. This validation must be done while the in-process fuzzer is not running. As with the general operation

of the over-the-air fuzzer, the payload should be sent while monitoring the device with *PacketLogger*.

In case of an uncontrolled crash, the *iOS* crash logs can be examined to determine the cause. These crashes are sometimes within FЯIDA-related threads and most of the time caused by non-fuzzed input to `bluetoothd`, for example in its `RxLoop` or `StackLoop`, with the latter being responsible for communication with the Bluetooth chip. When opening the scan dialog within the *iOS* settings, the external `SpringBoard` component tends to crash and sometimes even the `sharingd` daemon. These crashes typically do not reproduce over-the-air, as they originate from inconsistent states introduced by *ToothPicker* itself.

3.5 Connection Management

A protocol handler only accepts payloads if it is convinced that an active connection for this protocol exists. In Bluetooth, most data is transferred based on ACL. Thus, one of the tasks the specialized harness has to handle is creating a forged ACL connection. In the first *ToothPicker* version, we tried copying physical connections, however, creating virtual connections turned out to be the better solution.

3.5.1 Copying Physical Connections

The entry point for most application data arriving from a Bluetooth connection is the ACL reception handler. All data received by a connected peer is handled by this function. Therefore, the ACL reception handler can be used to potentially fuzz any application-layer Bluetooth protocol. The ACL reception handler accepts three arguments. A Bluetooth connection handle, the length of the ACL data, and a pointer to the received data.

```
void acl_reception_handler(uint16_t handle, size_t
     length, void* data)
```

In the first *ToothPicker* version, we tried the following fuzzing strategy. First, we hooked the reception handler function. Then, we created a physical over-the-air Bluetooth connection. Afterward, we sent over-the-air ACL data to the target device. We copied the connection handle value of the physical connection structure and called the reception handler with this copy. The fuzzing harness would start calling the ACL reception handler with the stored connection and arbitrary ACL data. While this simple approach ensures that the proper data structures for a connection are in place, it has similar drawbacks as fuzzing over-the-air. First, a physical connection is required, and second, when one of the peers decides to terminate the connection, the allocated connection structure is destroyed and a new connection has to be created. This also implies hooking the function again and storing a new connection structure.

3.5.2 Creating Virtual Connections

Instead of copying physical connections, we virtualize connections. This can be done by calling the function that allocates the connection structure from our fuzzing harness. While reverse-engineering `bluetoothd` on *iOS*, we identified a function that is used to allocate exactly this connection. Due to the lack of symbols we call this function `allocateACLConnection`. This function can now be called using the specialized harness to create a forged ACL connection and corresponding handle. It accepts two arguments, the Bluetooth address of the peer and another value that is stored directly in a field of the ACL connection structure. While reverse-engineering and dynamically analyzing this structure, we found that this seems to be a field indicating the status of the handle. This status is mostly set to the value `0` when the ACL reception handler is called. Listing 1 shows how to create such a virtual ACL connection in FЯIDA.

The ACL reception handler requires the handle value as a first parameter. Therefore, we need to associate a known handle to our newly created connection structure. We chose `0x11` as it is within the range of handles that would usually be created on a physical connection. Based on this, FЯIDA scripts can call protocol-specific ACL reception handlers. For example, the fixed channel L2CAP protocols in `bluetoothd` can be fuzzed. Similar to creating an ACL handle, BLE handles and dynamic L2CAP channels can be created.

3.5.3 Stabilizing and Using Virtual Connections

A virtual connection can still be disconnected. We identified two functions that disconnect or destroy an ACL or BLE connection. We overwrite these functions to prevent our forged connections from being disconnected, such as the function `OI_HCI_ReleaseConnection`.

While hooking and replacing these functions prevents the connection structures from being destroyed, this disconnection prevention technique cannot be used for the over-the-air fuzzer. During an over-the-air fuzzing session, there exist four distinct representations of the connection, two for each of the peers. The first representation is used by the data structures

```
// Create a buffer for the Bluetooth address
var bd_addr = Memory.alloc(6);
// Resolve function address
var base = Module.getBaseAddress("bluetoothd");
var fn_addr = base.add(symbols.allocateACLConnection);
// Create function reference to call it from JavaScript
var allocateACLConnection = new NativeFunction(fn_addr, "
     pointer", ["pointer","char"]);
// Write the Bluetooth address to memory
bd_addr.writeByteArray([0xca,0xfe,0xba,0xbe,0x13,0x37]);
// Call the function and create a forged ACL connection
// If handle is != 0 then the call was successful
var handle = allocateACLConnection(bd_addr, 0);
// Set the connection's handle value to 0x11
Memory.writeShort(handle, 0x11);
```

Listing 1: Creating a virtual ACL handle using FЯIDA.

that the Bluetooth stack creates, such as an ACL connection. These usually store information about the controller's HCI handle and other application-layer information, such as open L2CAP channels. The second representation of the connection resides within the chip. The chip allocates an HCI handle the stack can use to reference the connection. The chip also holds additional state to keep the connection alive. Even if we manage to prevent `bluetoothd` from destroying the connection, we cannot easily control the other involved components. Thus, the in-process fuzzing variant with the virtual connection can be controlled the best.

4 Evaluation

The overall performance of *ToothPicker* is strongly limited by the computation power of the *iPhone* it is running on as well as potential concurrency issues within `bluetoothd`. Moreover, the dynamic FЯIDA-based instrumentation has significant performance drawbacks. In the following, we provide a coverage and performance analysis, options for performance optimization as well as details on compatible *iPhones* and *iOS* versions.

4.1 Coverage

Figure 3 shows how the basic block coverage increases over time for a combination of all corpora, which are 29 different inputs based on Table 2. Initially, the coverage increases very fast as the non-mutated corpora cover 1295 basic blocks. We use the same corpora within each run and reuse the same seed a couple of times. Even for the same seed the coverage differs due to concurrent operation within `bluetoothd`. However, the coverage still shows different tendencies depending on the initial seed. In case of an uncontrolled crash not caught by FЯIDA, we abort the run. Uncontrolled crashes can be blacklisted by analyzing the *iOS* crash logs replacing the according functions with a return. While this significantly reduces uncontrolled crashes and might reach different parts of the `bluetoothd` binary, it also reduces the code coverage for the given inputs. As indicated by no longer getting any new basic block coverage feedback, `bluetoothd` can also hang without crashing as a result of fuzzing.

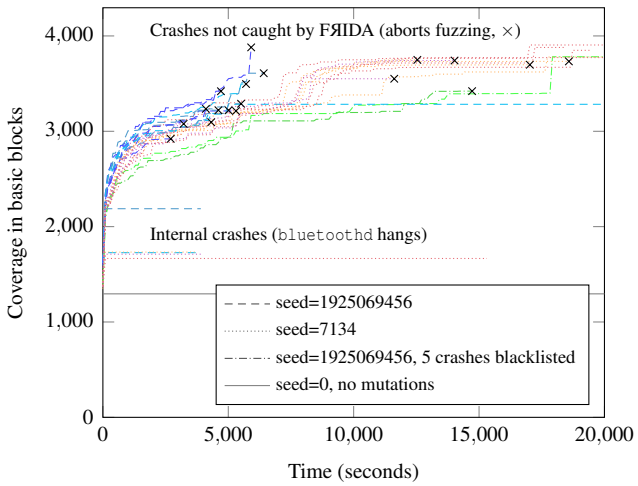


Figure 3: Coverage for a combination of all corpora, *iOS 13.3.1* for an *iPhone 7*, manager on *Intel Core i7-6600U*.

While we stopped execution upon a crash in Figure 3, *ToothPicker* can be restarted within the same configuration and continue based on the meanwhile increased corpus. Thus, running *ToothPicker* in a loop and killing `bluetoothd` from time to time automates the fuzzing process.

The overall coverage reached is small compared to the total number of basic blocks. The better runs in Figure 3 cover around 4 k basic blocks, which corresponds to 3 % of the `bluetoothd` binary. Though, *ToothPicker* only fuzzes protocols prior to pairing, as pairing would require user interaction. Complex protocols like tethering and music streaming with a lot of data transfer and state handling are not considered. When checking the coverage information with *Lighthouse*, the protocol handlers under test are well-covered. Moreover, *ToothPicker* also discovers new handlers—*CVE-2020-9838*, discussed later in Section 5, is in a protocol not contained in any corpus of Table 2.

4.2 Speed and Bottlenecks

The major bottleneck originates from the FЯIDA instrumentation. Moreover, the *radamsa*-based input mutation is slow.

When running *ToothPicker* with *radamsa* mutations on the corpus, it reaches 25 inputs/s on average on an *iPhone 7*. However, without the mutations, it reaches 65 inputs/s on average. FЯIDA applies a trust-threshold on blocks it is dynamically instrumenting, which significantly increases performance [27]. Not mutating the input means that FЯIDA is always executing the same functions with the same inputs and, thus, executing the same basic blocks.

One *radamsa* input mutation, measured within the manager component of *ToothPicker*, takes about 8 ms on an *Intel Core i7-4980HQ* with 2.8 GHz running on *macOS* and 14 ms running on an *Intel Core i7-6600U* with 2.6 GHz on *Linux*. We compared two variants of this. The first opens *radamsa* as a subprocess to mutate the input (*frizzer* default implementation) and the second uses *libradamsa* with *ctypes*. Surprisingly, we found that the variant calling *radamsa* as subprocess is slightly faster. *libradamsa*'s lack of speed has been documented before [17]. When fuzzing with 25 inputs/s on average, this means that 8 ms *radamsa* input generation make up 20 % of the *ToothPicker* runtime. For comparison, directly reading inputs from a file within the *ToothPicker* manager takes 0.1 ms on the same machine. Using a different fuzzing engine, such as American Fuzzy Lop (AFL), will be considered for the next version of *ToothPicker*. An AFL ex-

Table 3: List of vulnerabilities identified with *ToothPicker* and their status as of July 2020.

ID	Description	Effect	Detection	OS	Disclosure	Status
MP1	Ratchet AES SIV	Crash	ToothPicker	iOS 13.3–13.6, 14 Beta 2	Oct 30 2019	Not fixed
MP2	Hint	Crash	ToothPicker	iOS 13.3–13.6, 14 Beta 2	Dec 4 2019	Not fixed
MP7	Ratchet AES SIV	Crash	ToothPicker	iOS 13.3–13.6, 14 Beta 2	Mar 13 2020	Not fixed
MP8	Ratchet AES SIV	Crash	ToothPicker	iOS 13.3–13.6, 14 Beta 2	Mar 13 2020	Not fixed
L2CAP2	Group Message	Crash	ToothPicker	iOS 5–13.6, 14 Beta 2	Mar 13 2020	Not fixed
LEAP1	Version Leak	Information Disclosure	Manual	iOS 13.3–13.5	Mar 31 2020	Not fixed
SMP1	SMP OOB	Partial PC Control	ToothPicker	iOS 13.3	Mar 31 2020	iOS 13.5, *CVE-2020-9838*
SIG1	Missing Checks	Crash	ToothPicker	iOS 13.3–13.5	Mar 31 2020	iOS 13.6, *CVE-2020-9931*

tension for FЯIDA was not available when initially building *ToothPicker* but has been released in July 2020 [30].

Intuitively, fuzzing on newer *iPhone* models should be faster. We ported *ToothPicker* to an *iPhone SE2*, released in April 2020, as well as the *iPhone 11*, released in September 2019, which are much newer than the *iPhone 7*, released in September 2016. However, the *iPhone SE2* and *iPhone 11* feature an *A13* CPU with Pointer Authentication (PAC). Thus, each FЯIDA `NativeFunction` call needs to be signed. These extra operations reduce the speed from 20 inputs/s to 14 inputs/s on average on both *A13* devices when the manager runs on an *Intel Core i7-6600U*.

4.3 Increasing Jetsam Limits

A general issue that arises due to the in-process fuzzing is that the resource utilization of `bluetoothd` is much higher than usual. *Jetsam*, *Apple*'s out-of-memory killer, terminates processes taking too many resources [20]. Thus, we change the *Jetsam* configuration file to increase `bluetoothd`'s memory limit and set its priority to the maximum of 19, reducing terminations due to resource consumption.

4.4 Supporting Multiple *iOS* Versions

ToothPicker supports *iOS 13.3*, *13.3.1*, *13.5 Beta 4*, and *13.5* on an *iPhone 7*, *iOS 13.5* on an *iPhone SE2*, and *iOS 13.3* on an *iPhone 11*. Symbol locations within `bluetoothd` change with each *iOS* version. For the current *ToothPicker* version, 19 symbols have to be defined. Since the changes between those versions are minimal and most of them even contain the same print statements, they can be easily identified using *BinDiff*, *Diaphora*, or the *Ghidra* versioning tool [19, 24, 32].

5 Identified Issues

An overview of identified vulnerabilities is shown in Table 3. Note that *ToothPicker* has been initially developed to analyze *MagicPairing*, and thus, bugs discovered within *MagicPairing* and L2CAP have been described previously [15]. The first issues discovered by *ToothPicker* are still unpatched months after reporting. As *Apple* does not communicate details about their patching timeline, we do not know when they will be fixed. However, the most severe bug discovered by *ToothPicker*, which allows RCE, was reported on March 31 and fixed in the *iOS 13.5* release on May 20 as *CVE-2020-9838*. Moreover, *Apple* started fixing the DoS issues and addressed one as *CVE-2020-9931* in the *iOS 13.6* release on July 16. In the following, more details about the findings are provided.

SMP1: Security Manager Protocol Out-of-Bounds Jump

This vulnerability occurs in the reception handler of the Security Manager Protocol (SMP), which, as of *iOS 13*, is accessible via both Classic Bluetooth and BLE. The cause for this flaw is an incorrect check of the received protocol opcode value, which leads to an out-of-bounds read. The value that is read is treated as a function pointer. Listing 2 shows a pseudocode representation of the flawed opcode check. If the opcode `0x0f` is sent, the bounds check is still valid. However, the global function table where the handlers for the specific opcodes reside only has 15 entries. As the table is indexed by zero, the 15$^{\text{th}}$ entry is out of bounds. The table is immediately followed by other data. As the tables for Classic Bluetooth and BLE are at different locations, the result is different, depending on which technology is chosen. We found that on *iOS 13.3*, the data following the Classic Bluetooth SMP function table is partially controllable by an attacker. More specifically,

```
opcode = data[0]     // Cet opcode from the input data
if (opcode <= 0xf):  // Flawed opcode check
  // The handler is resolved from the function table
  handler = GLOBAL_FUNCTION_TABLE[opcode]
  // The code checks if the resolved pointer is not null
  if handler != None:
     handler(...)
```

Listing 2: Flawed SMP opcode check includes `0xf`.

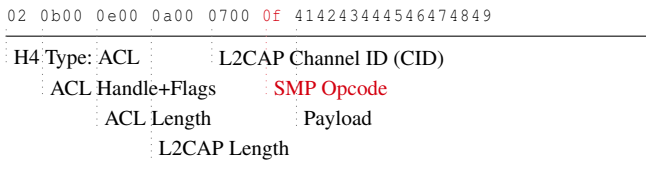


Figure 4: Malicious payload for *SMP1* aka *CVE-2020-9838*.

it is followed by two 4 B values, an unknown value that we observed to be `0x00000001` and a global counter that is incremented for each Bluetooth connection that is created. Due to its layout, the connection counter determines the MSB of the address. Therefore, an attacker could crash `bluetoothd` and then create a specific amount of connections to form the address. However, we did not find a way to influence the other value. This leads to the lower four bytes of the address to be equal to `0x00000001`, which is an unaligned address and thus crashes `bluetoothd`. In case it is possible to influence this value, or the layout of the memory following the function table changes, this flaw could potentially lead to a control flow hijack. The amount of preparation required for abusing this flaw is very high. This includes preparing or determining a useful address to jump to and also being able to precisely control the value.

This issue was fixed by Apple in *iOS 13.5* and was assigned *CVE-2020-9838*. Figure 4 shows an ACL payload that triggers the vulnerability. Note that the handle needs to be adapted to the actual connection's handle value. *SMP1* affects both SMP over Classic Bluetooth and BLE. Classic Bluetooth is using the Channel ID (CID) `0x0007`, while BLE is using the CID `0x0006` for SMP.

As shown in Table 2, we did not explicitly target the SMP protocol. The *SMP1* vulnerability was found by the ACL fuzzing pass, which had a corpus of only L2CAP *Signal Channel* frames. This also shows that the input generation and mutation mechanism is capable of identifying new paths and protocol messages.

SIG1: Signal Channel Missing Checks

The *Classic Signal Channel* fuzzing identified three crashes that all originate from the same flaw. These crashes are a result of accessing memory at the location `0x00`, `0x08`, and `0x10`. The crash occurs within the *Configuration Request*, and *Disconnection Response* frame handlers in the *Signal Channel*. These frames are all related to dynamic L2CAP channels. Thus, they contain a CID. To obtain the necessary data structure for an L2CAP channel with a given CID, a lookup is done via the `ChanMan_GetChannel` function. This returns the channel data structure for both dynamic and static L2CAP channels. If the L2CAP channel is a dynamic channel, the data structure contains a field that points to another structure with additional information about the channel. In case of a fixed L2CAP channel this additional data structure does not seem to exist and the field value is `0`. The *Signal Channel* parsing code in `bluetoothd` does not differentiate between static and dynamic L2CAP channels. Thus, the code falsely treats this null pointer as pointer to a structure and dereferences the fields at offset `0x00`, `0x08`, and `0x10`, leading to crashes due to invalid memory accesses. A pseudocode representation of this flaw is shown in Listing 3. The example is taken from the reception handler of the *Disconnection Response* frame handler. *Apple* fixed this issue in *iOS 13.6*, however, only the payloads we provided for offsets `0x08` and `0x10` stopped crashing, our payload for `0x00` still crashes `bluetoothd` and might be a different issue.

```
// L2CAP channel structure obtained by CID
status = ChanMan_GetChannel(channel_id, &channel);
// If CID exists, obtain dynamic structure via channel
if (status == 0) {
    dyn_structure = get_dyn_structure(channel);
    // The dyn_structure pointer is not checked before
         derefencing. The following line will try to
         dereference the value 0x10, leading to a crash.
    if (*(char *)(dyn_structure + 0x10) == 0x06 && [...]) {
        [...]
    }
}
```

Listing 3: Signal channel structure pointer dereference.

6 Conclusion

The results of *ToothPicker* show that it is a powerful fuzzing framework that can discover new vulnerabilities within the *iOS* Bluetooth stack. Using virtual connections, it harnesses `bluetoothd` in a way that *Apple* likely did not apply when testing it internally. Thus, we were able to discover new vulnerabilities despite the speed limitations. Since *ToothPicker* runs as FЯIDA in-process fuzzer on an *iPhone* and dynamically rewrites code during runtime, it is comparably slow to what *Apple* could reach with internal testing tools and recompiling the stack. Our findings are quite concerning with regards to the omnipresence of Bluetooth within their ecosystem. We hope that the publication of tools like *ToothPicker* will improve the state of Bluetooth security in the long term.

Acknowledgments

We thank *Apple* for handling our responsible disclosure requests. Moreover, we thank Kristoffer Schneider for continuing the work on *ToothPicker*, Mathias Payer for shepherding this paper, Ole André V. Ravnås for the FЯIDA support, 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.

Availability

ToothPicker is publicly available on *GitHub*: https://github.com/seemoo-lab/toothpicker

References

[1] Jonathan Afek. Simplifying iOS Research: Booting the iOS Kernel to an Interactive Bash Shell on QEMU. https://www.offensivecon.org/speakers/2020/jonathan-afek.html, Feb 2020.

[2] Apple. Accessory Design Guidelines for Apple Devices. https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf, 2020.

[3] Apple. Bluetooth – Apple Developer. https://developer.apple.com/bluetooth/, 2020.

[4] Apple. Set up Wireless Audio Sync with Apple TV. https://support.apple.com/en-us/HT210526, 2020.

[5] Apple. Use Continuity to connect your Mac, iPhone, iPad, iPod touch, and Apple Watch. https://support.apple.com/en-us/HT204681, 2020.

[6] Inc. Armis. The Attack Vector 'BlueBorne' Exposes Almost Every Connected Device. https://www.armis.com/blueborne/, 2017.

[7] Brandon Azad. KTRW: The journey to build a debuggable iPhone. https://googleprojectzero.blogspot.com/2019/10/ktrw-journey-to-build-debuggable-iphone.html, Oct 2019.

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

[9] Damien Cauquil. Bluetooth Low Energy Swiss-army knife. https://github.com/virtualabs/btlejack, 2019.

[10] Guillaume Celosia and Mathieu Cunche. Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols. *Proceedings on Privacy Enhancing Technologies*, 2020(1):26–46, 2020.

[11] Jianjun Dai. Take Down MacOS Bluetooth with Zero-click RCE. https://blogs.360.cn/post/macOS_Bluetoothd_0-click.html, 2020.

[12] Apple Google. Exposure Notification, Bluetooth Specification, v1.1. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ExposureNotification-BluetoothSpecificationv1.1.pdf, April 2020.

[13] Samuel Groß. Remote iPhone Exploitation Part 2: Bringing Light into the Darkness – a Remote ASLR Bypass. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-2.html, Jan 2020.

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

[15] Dennis Heinze, Jiska Classen, and Felix Rohrbach. MagicPairing: Apple's Take on Securing Bluetooth Peripherals. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul 2020.

[16] Aki Helin. radamsa - a general-purpose fuzzer. https://gitlab.com/akihe/radamsa, 2020.

[17] Marc Heuse. radamsa lib is slooooow. https://gitlab.com/akihe/radamsa/-/issues/66, Oct 2019.

[18] Kim Jong Cracks. checkra1n—iPhone 5s – iPhone X, iOS 12.3 and up. https://checkra.in/, 2020.

[19] Joxean Koret. Diaphora. http://diaphora.re/, 2020.

[20] Jonathan Levin. *New OSX Book, Volume II, *iOS Internals::Kernel Mode*. 2019.

[21] Dennis Mantz. Frida-based General Purpose Fuzzer. https://github.com/demantz/frizzer, 2019.

[22] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, Jun 2019.

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

[24] National Security Agency. Ghidra. https://ghidra-sre.org/, 2020.

[25] Nordic Semiconductor. nRF Connect for Mobile. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile, 2020.

[26] Jack Purcher. A European Patent Filing from Apple Describes Voice Controls Destined for a Television and the Apple TV Set Top Box. https://www.patentlyapple.com/patently-apple/2014/11/a-european-patent-filing-from-apple-describes-voice-controls-destined-for-a-television-and-the-apple-tv-set-top-box.html, 11 2014.

[27] Ole André V. Ravnås. Anatomy of a code tracer. https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8, Oct 2014.

[28] Ole André V. Ravnås. Frida - A world-class dynamic instrumentation framework. https://frida.re/, 2020.

[29] Ole André V. Ravnås. Frida Stalker Documentation. https://frida.re/docs/stalker/, 2020.

[30] Keegan S. hotwax. https://github.com/meme/hotwax, Jul 2020.

[31] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 37–54, Santa Clara, CA, August 2019. USENIX Association.

[32] zynamics. BinDiff. https://www.zynamics.com/bindiff.html, 2020.

Appendix

This appendix contains detailed descriptions of all proprietary protocols we identified within the *iOS 13* `bluetoothd`. We start with fixed L2CAP channels in Section A.1, because they are accessible prior to authentication, continue with dynamic L2CAP channels in Section A.2, and then describe the remaining protocols in Section A.3.

A.1 Fixed L2CAP Channels

In the following, we describe custom L2CAP responses, which are part of the *FastConnect* protocol. *FastConnect* has both fixed and dynamic channels, but as fixed channels have a higher attack potential, we include it in this section. Moreover, we describe *Apple*'s implementation of LEA, which was introduced before the Bluetooth specification added it in the most recent version 5.2 [8]. Then, we describe *MagicPairing*, which is used by the *AirPods*, and continue with *Magnet*, used by the *Apple Watch*. Finally, we describe *DoAP*, which is specific to *Siri*.

A.1.1 Custom L2CAP Echo Responses

L2CAP *Echo Requests* and *Echo Responses* have the purpose to implement a *ping*-like pattern between two Bluetooth peers [8, p. 1059-1060]. One device sends an *Echo Request* with a certain payload and the other responds with an *Echo Response*, which includes the same payload. Interestingly, *Apple* deviates from the Bluetooth specification in two details.

First, if the *Echo Request* has a specific format, *Apple* treats this as a *FastConnect Discovery Request* (see Section A.1.2). Second, if the request payload is `Apple`, the response is a custom string that differs from `Apple`. This behavior even differs between the various operating system versions, which allows to disclose the device's operating system:

- *iOS* responds with `AppleComputerInc.`,
- *RTKit* on the *AirPods* responds with `Apple` followed by `AppleComputerInc.`, and
- *macOS* does not have this feature and responds specification-compliant with `Apple`.

A.1.2 FastConnect and FastConnect Discovery

FastConnect increases connection setup speed. It simplifies configuration that would otherwise be exchanged through various SDP messages. Usually, dynamic L2CAP channels need to be negotiated to exchange information. *FastConnect* simplifies this negotiation for L2CAP channels that have been opened in previous connections to the device. There are two parts of the *FastConnect* protocol, one that is used prior to pairing and one that is used after the devices are paired.

The first part of *FastConnect*, the *Discovery* messages, are implemented on top of L2CAP *Echo Requests*. There are two properties that indicate a *Discovery Request* as opposed to a normal *Echo Request*. The first indication is a payload length greater than 23 B. As soon as the L2CAP *Echo Request* payload is longer than 23 B, it is no longer interpreted as an *Echo Request* but as a *Fast Connect Discovery Request*. The second indication is the value `0x01` at byte position 6 in the L2CAP payload, which is required for the payload to be properly parsed. We do not know why *Apple* chose a length to indicate packet types rather than adding another opcode. Either case, this is not specification-compliant.

The second part of the *FastConnect Protocol* is implemented over a dynamic L2CAP channel. This channel is created after sending a *FastConnect Discovery Request* followed by a successful pairing (e.g., via *MagicPairing*, see Section A.1.4, which can be specified as a pairing `sink`). The CID to be used in the following is specified in the *FastConnect Discovery Response*. The protocol then continues to exchange four messages, two for each peer: a *FastConnect Service Descriptor* followed by a *FastConnect Service Descriptor Response*. Afterward, a *FastConnect Service Configure* is followed by a *FastConnect Setup Complete*. The *FastConnect Service Descriptor* message contains a list of protocols the sender supports along with their configuration and allocated CID. The receiver acknowledges this list by replying with the protocols it supports and its local CID. If required, the sender can then send further configuration options via the *Service Configure* message. The protocol concludes with the *Setup Complete* message, where the receiver acknowledges the received configuration. Finally, the negotiated protocol CID are opened and can be used by the peers.

The discovery part of the *FastConnect* protocol introduces an additional attack surface, as it is reachable via a fixed L2CAP channel, and, thus, prior to pairing. However, the attack surface is rather small as it only concerns two message types.

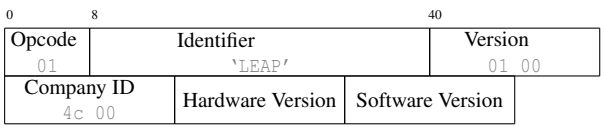


Figure 5: *LEAP* version message.

Table 4: *LEAP* version messages indicating first digit of the *iOS* version.

Device	iOS Version	HW Version	SW Version
iPhone 7	iOS 13.3	7005	0D30
iPhone 8	iOS 13.3	710D	0D30
iPhone 11	iOS 13.3	730D	0D30
iPhone 11 Pro	iOS 13.3.1	7308	0D30
iPhone 11 Pro	iOS 13.4.1	7308	0D40
iPad Mini 2 Retina	iOS 12.4.5	6D0B	0C40

A.1.3 LE Audio

LEA enables audio streaming via BLE instead of Classic Bluetooth. It consists of two subprotocols, one for management and one for streaming. The subprotocols are called LE Audio Protocol (LEAP) and LE Audio Stream (LEAS). While we did look into the implementation, we did not find a device that is using this protocol. According to sources citing the patent *Apple* registered for this protocol, it is aimed to be used for hearing aids, remote controls, and set-top box devices [26]. LEA is a predecessor of the BLE audio protocol introduced in the latest Bluetooth 5.2 specification [8]. It exists at least since *iOS 9*, released in 2015 [6]. Thus, *Apple* had the ability to stream audio over BLE at least five years before this technology was included in the Bluetooth specification.

During the analysis of the LEAP protocol, we found that there exists a LEAP version message that can be used to exchange version information between the sender and the receiver. The structure of this version message is shown in Figure 5. It contains the LEAP version, divided in hardware and software version. While the hardware and software versions in these messages are encoded, they can be used to infer a device's *iOS* major version and the first digit of the minor version. The version message, and the LEAP protocol in general, do not require an authenticated connection and are reachable via BLE. Thus, this serves as a valuable information leak for a potential attacker. Table 4 shows an overview of LEAP version messages captured from test devices, along with their device and software versions.

A.1.4 MagicPairing

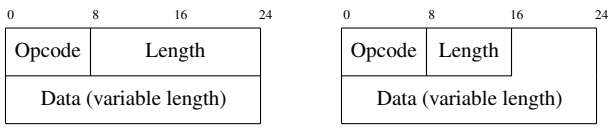
MagicPairing replaces the pairing mechanism defined by the Bluetooth specification while deriving the same type of key. Thus, *MagicPairing* remains compatible with the Bluetooth chip's encryption mechanisms. However, the derived key depends on the user's *iCloud* account and, thus, the Bluetooth accessory is instantly available on all *Apple* devices of the same user. We only observed it to be used to pair *AirPods*. Note that we skip a detailed description on *MagicPairing* here, as we already described it in detail in a separate paper [15].

A.1.5 Magnet

Magnet is primarily used between an *iPhone* and an *Apple Watch*. On *macOS*, `bluetoothd` seems to occasionally refer to the protocol as *CBPipes*. Previously, this protocol has been referred to as *Piped Dreams* [6]. Its purpose is similar to that of the L2CAP *Signal Channel*, which manages connections and channels [8, p. 1046ff]. Furthermore, it can create and manage L2CAP channels. In addition to that, it offers synchronization features. *Magnet* is tightly coupled with *Terminus*, which in turn is a service responsible to communicate with the *Apple Watch* over multiple transports with different encryption methods.

Another use of the *Magnet* protocol we have been able to observe is the *Wireless Audio Sync* feature of the *Apple TV* [4]. It allows a user to synchronize and correct timing offsets between the *Apple TV* and its connected speakers. For this purpose, the *Magnet* protocol is used to open another L2CAP channel which then handles the data exchange for the synchronization feature.

The *Magnet* message structure is shown in Figure 6. Depending on the version that was negotiated in the *Magnet* version message, the structure differs slightly. For a version newer than *8*, the length field is 2 B instead of one. While observing the protocol in the wild, between an *iPhone* and *Apple Watch* as well as an *Apple TV*, *iOS 12* used version *6* and *iOS 13* used version *9*.



a) Version > *8*. **b)** Version < *8*.

Figure 6: *Magnet* message structure.

A.1.6 DoAP

DoAP is a protocol implemented on top of GATT. Its responsibilities include configuring and triggering *Siri*. Most of the relevant *DoAP* code resides within the `BTLEServer` binary. As *DoAP* protocol is only available after pairing, we did not look further into it. However, it might be an attractive target for future work.

A.2 Dynamic L2CAP Channels

In the following, we describe Apple AirPods Protocol (AAP), which is used to configure *AirPods*. The *Apple Pencil* that can write on *iPads* has a specific GATT extension. Moreover, Made for Apple (MFi) accessories use a special protocol.

A.2.1 AAP Protocol

AAP is implemented on top of L2CAP. It is used for the communication between an *Apple* host device and the *AirPods*. The protocol offers multiple different services, which all revolve around configuring the *AirPods* and obtaining information and data from them. Examples are firmware updates,

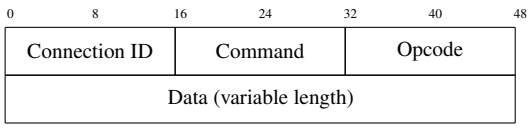


Figure 7: AAP message structure.

getting and setting tapping actions, or exchanging *MagicPairing* key material. The protocol is undocumented. However, analyzing `bluetoothd` on both *iOS* and *macOS*, as well as *Apple*'s Bluetooth *PacketLogger*, helps to reconstruct the protocol. The general structure of an AAP message is shown in Figure 7.

A.2.2 Apple Pencil GATT

During a dynamic analysis using *PacketLogger*, we found that the communication between the *Apple Pencil* and an *iPad* is implemented over GATT. Every movement of the pencil is transmitted as a GATT *Value Notification*. We were only able to test the first generation of the *Apple Pencil*. As the second generation only works with the newest *iPad* generations, the protocol might differ.

A.2.3 External Accessory

The *External Accessory* protocol, also called *iAP2*, can be used as an application-layer protocol to communicate with external hardware. The underlying frameworks abstract the actual transport layer. It supports connections over Classic Bluetooth or the Lightning connector. To use the protocol with an external device, it needs to be certified within the MFi program [2].

A.3 Other Protocols

The *AirPods* use a special protocol between the left and the right pod, that is exclusively used by them. Moreover, the *Magic Keyboard* supports out-of-band pairing, which is not Bluetooth per se, but the negotiated key is later used by Bluetooth.

A.3.1 BRO and UTP

BRO and *UTP* are both protocols used between two individual *AirPods* buds. Their purpose includes synchronization and primary to secondary switching. Currently, the only source for information about these is found in *PacketLogger*. Some versions of it contain a human-readable file called `ACI_HCILib_2xml`, which includes a basic documentation of *BRO* and *UTP* commands.

A.3.2 USB Out-of-Band Pairing

The *Apple Magic Keyboard* supports an out-of-band pairing mechanism. The keyboard first needs to be connected to the host device via USB. Then the Bluetooth Link Key (LK) is generated on the keyboard and sent to the host. The LK is transmitted through the Human Interface Device (HID) configuration endpoint. Afterward, the devices are automatically paired. While this is not a wireless protocol per se, we still want to mention this, as the established LK is used in a wireless context later.