Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets

Jan Ruge and Jiska Classen, *Secure Mobile Networking Lab, TU Darmstadt;*
Francesco Gringoli, *Dept. of Information Engineering, University of Brescia;*
Matthias Hollick, *Secure Mobile Networking Lab, TU Darmstadt*

https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets

Jan Ruge
Secure Mobile Networking Lab
TU Darmstadt

Jiska Classen
Secure Mobile Networking Lab
TU Darmstadt

Francesco Gringoli
Dept. of Information Engineering
University of Brescia

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt

Abstract

Wireless communication standards and implementations have a troubled history regarding security. Since most implementations and firmwares are closed-source, fuzzing remains one of the main methods to uncover Remote Code Execution (RCE) vulnerabilities in deployed systems. Generic over-the-air fuzzing suffers from several shortcomings, such as constrained speed, limited repeatability, and restricted ability to debug. In this paper, we present *Frankenstein*, a fuzzing framework based on advanced firmware emulation, which addresses these shortcomings. *Frankenstein* brings firmware dumps "back to life", and provides fuzzed input to the chip's virtual modem. The speed-up of our new fuzzing method is sufficient to maintain interoperability with the attached operating system, hence triggering realistic full-stack behavior. We demonstrate the potential of *Frankenstein* by finding three zero-click vulnerabilities in the *Broadcom* and *Cypress* Bluetooth stack, which is used in most *Apple* devices, many *Samsung* smartphones, the *Raspberry Pis*, and many others.

Given RCE on a Bluetooth chip, attackers may escalate their privileges beyond the chip's boundary. We uncover a Wi-Fi/Bluetooth coexistence issue that crashes multiple operating system kernels and a design flaw in the Bluetooth 5.2 specification that allows link key extraction from the host. Turning off Bluetooth will not fully disable the chip, making it hard to defend against RCE attacks. Moreover, when testing our chip-based vulnerabilities on those devices, we find *BlueFrag*, a chip-independent *Android* RCE.

1 Introduction

Bluetooth is present in a lot of privacy-sensitive applications. These include headsets that we share contacts with, smartwatches, cars, medical devices, and all kinds of Internet of Things (IoT) products. Around 4.4 billion Bluetooth-enabled devices will be presumably shipped in 2020 alone, and annual device shipments are growing [11].

The overall zero-click attack surface is comparably large. For example, all *Apple* devices publicly expose connectable Bluetooth Low Energy (BLE) Generic Attribute (GATT) services whenever Bluetooth is enabled—even without prior pairing. Many devices have Bluetooth enabled by default, and quite a number of them advertise their identity [46]. Despite these identities being anonymous, an attacker might find interesting targets near airports or office buildings. Vulnerabilities are wormable, as most devices can initiate new connections.

In this work, we evaluate various attack vectors based on RCE. We consider attacks that are either compliant with the Bluetooth 5.2 specification [12], propagate into components outside of the Bluetooth chip, or brick the Bluetooth hardware. On *Broadcom* combo chips, Wi-Fi and Bluetooth run on separate Advanced RISC Machine (ARM) cores. As they share the 2.4 GHz antenna, they need to agree on access through *coexistence mechanisms*. Using coexistence, we escalate from Bluetooth into Wi-Fi components, block these, and then force reboot various devices, including the *iPhone 11*.

We gain Bluetooth zero-click RCE by systematically fuzzing those parts of the *Broadcom* firmware that can be reached prior to pairing. *Cypress* acquired parts of *Broadcom's* Bluetooth implementation in 2016 [17], and while both stacks diverged since then, they remain fuzzable and vulnerable using similar techniques. Emulation and fuzzing provide insights into an otherwise undocumented, proprietary firmware. We provide a *C* programming environment to interact with the firmware image that can test hypotheses on the firmware and narrow down the relevant code paths. Our main contributions are as follows:

- We design and implement the emulation framework *Frankenstein* to execute large portions of the firmware, including injection of raw wireless frames and interaction with the host,
- find three zero-click chip vulnerabilities, two for classic Bluetooth and one for BLE,
- find the *BlueFrag* RCE in *Android*,
- break the coexistence mechanism in Wi-Fi/Bluetooth combo chips requiring a full device reboot to restore functionality, with some devices kernel panicing,

- uncover a design flaw in the Bluetooth 5.2 specification [12] that allows attackers to extract link keys including inactive connections, and
- showcase that users cannot turn off Bluetooth as a defense on recent mobile operating systems, as the chip reset is not specified properly.

Frankenstein is publicly available on *GitHub*. The provided fuzzing examples for two Common Vulnerabilities and Exposures (CVEs) find these in a matter of seconds to a few minutes. Firmware dumps of other popular wireless systems are also good candidates to be analyzed with our solution. We were able to confirm portability of *Frankenstein* by porting it to another firmware, however, we cannot present further examples due to non-disclosure agreements.

This paper is structured as follows. Section 2 introduces attacks within Bluetooth stacks and clarifies the difference between the full-stack *Frankenstein* approach and existing wireless fuzzers. Section 3 showcases broader vulnerabilities and attack concepts that apply to Bluetooth chips of all manufacturers, including new exploitation techniques we found. Section 4 gives an overview of firmware and Bluetooth-specific internals. Based on this, we explain how *Frankenstein* works in Section 5. The identified RCEs are described in Section 6. Applicability to other firmware and vulnerability patching are discussed in Section 7. An overview of related work is given in Section 8. Section 9 concludes our findings.

2 Motivation for Frankenstein

In the following, we put the motivation for *Frankenstein* in a broader context. Thus, we explain the general attack paths within Bluetooth stacks in Section 2.1. Then, we outline how *Frankenstein* integrates into these stacks, how its full-stack capability differentiates it from other fuzzers, as well as its applicability to other firmware in Section 2.2.

More details about how to perform these attacks follow in Section 3. A technical description of *Frankenstein* is provided in Section 5. However, we recommend reading this motivation to those who are not familiar with Bluetooth and wireless fuzzing.

2.1 Bluetooth Attack Paths

Figure 1 shows the attacks uncovered with *Frankenstein*. While all attacks can be launched over-the-air, their capabilities and escalation strategies differ.

Operating System RCE The most severe attacks allow direct access to the operating system. Depending on the operating system, the Bluetooth daemon runs with limited privileges; thus, the attacker needs to escalate further. However, on most operating systems, these limited privileges include accessing files and contacts.

While vulnerabilities in the operating system are the most severe, they are the easiest to patch. All they require is an operating system update, as they are hardware-independent.

On-Chip RCE The firmware running on the Bluetooth chip can be vulnerable as well. In general, it is easier to exploit—the protection mechanisms of the Real-Time Operating System (RTOS) running on it and the chip's hardware are rudimentary compared to what modern operating systems and architectures provide. An attacker with control over the firmware can access data processed within the chip and perform specification-compliant requests to the operating system. However, to also gain code execution on the operating system, further vulnerabilities on the host stack are required.

Thus, despite high exploitability, full system compromise requires additional escalation. Nonetheless, on-chip vulnerabilities are a security risk that often remains unpatched as security fixes require patches provided by the hardware vendor that, in turn, are shipped with an operating system update.

Inter-Chip Escalation An attack path that excludes mitigation by the operating system is inter-chip escalation. On *Broadcom* chips, Bluetooth and Wi-Fi run on two separate ARM cores. However, to coordinate spectrum access by means of coexistence mechanisms, they directly communicate with each other without the operating system being involved into this. Using inter-chip escalation, a Bluetooth RCE can then escalate into Wi-Fi components.

Depending on the type of inter-chip escalation, this communication channel exists in hardware and might be unpatchable. Thus, the firmware running on both cores must mitigate against this type of attack, and the operating system drivers should take action where possible.

Within our work on *Frankenstein*, we uncover all these vulnerability types, as shown in Figure 1. The focus of *Frankenstein* is to find on-chip RCE. We show that on-chip RCE can be used to break confidentiality in a specification-compliant manner by extracting the link keys used by Bluetooth encryption. During attempts to trigger the *Frankenstein* vulnerabilities over-the-air, one of our Proofs of Concept (PoCs) triggers *BlueFrag*, an *Android* operating system RCE. Moreover, we explore inter-chip escalations and find that we can crash the Wi-Fi firmware, which, in turn, produces kernel panics on *Android* and *iOS*.

2.2 Frankenstein

Frankenstein creates a physical device snapshot and then emulates it in Quick Emulator (QEMU) to fuzz the full stack: over-the-air data is provided by a virtual modem, the emulated firmware implements thread and task switches to fuzz multiple handlers, and it attaches to a real *Linux* host. It utilizes QEMU in user mode without further customizations.

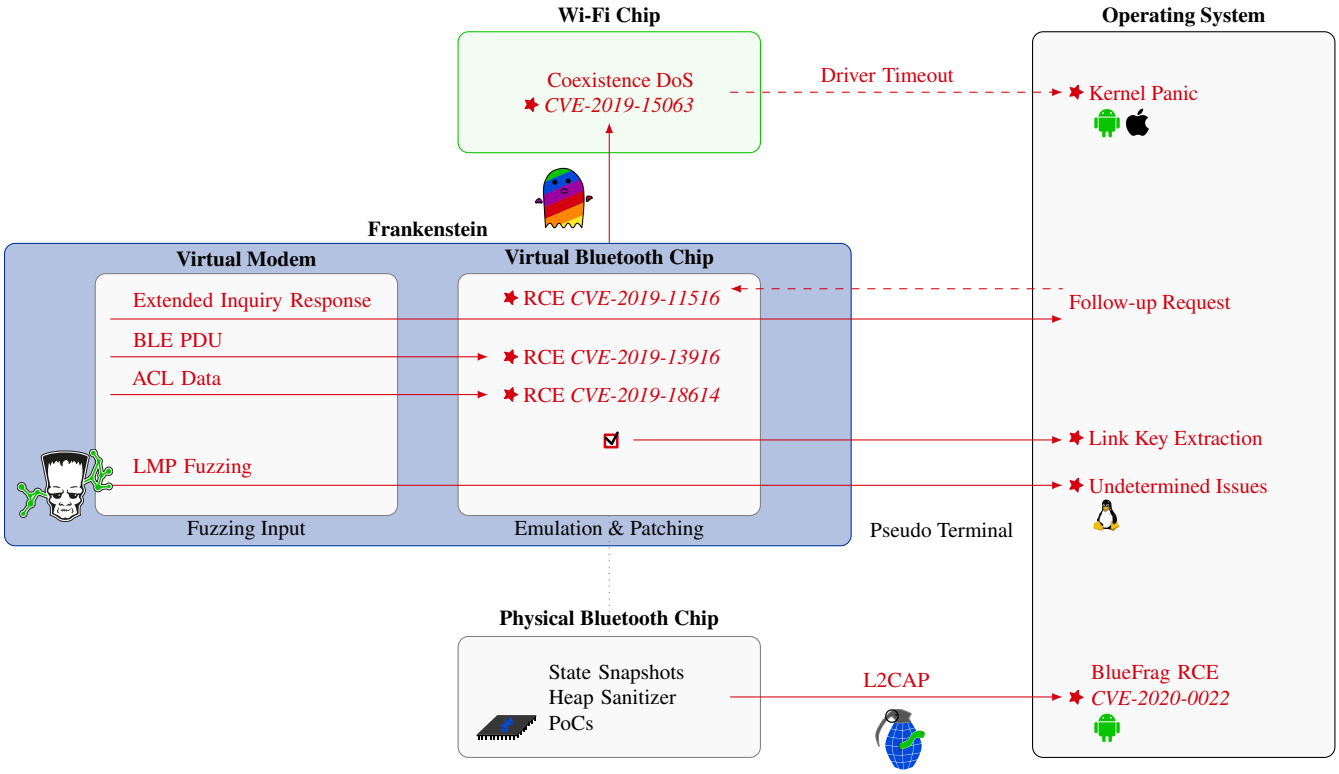


Figure 1: Bluetooth attacker model and *Frankenstein* integration, vulnerabilities discovered by us marked with ✸.

Chip Integration and Emulation Firmware running on a physical chip is difficult to access, monitor, and modify. *Broadcom* provides vendor-specific commands that can be used to extract firmware from the ROM. Moreover, the ROM can be temporarily patched with breakpoints, the so-called *Patchram* mechanism. The *InternalBlue* experimentation framework enables ROM extraction and patching [35].

The *Patchram* mechanism and monitoring on the hardware itself are very limited. Even with an over-the-air Software-Defined Radio (SDR) fuzzer, which would require to re-implement all the logic and formats defined in the 3256 pages of the Bluetooth specification, analyzing the results would be infeasible. Thus, *Frankenstein* fuzzes the firmware in emulation. This provides higher speed than over-the-air fuzzing and enables coverage feedback through QEMU.

Emulating a firmware dump comes with various challenges. These include memory map generation, chip state extraction including hardware registers, and working with only partial symbols. The common approach to handle these challenges is to reverse-engineer firmware in order to identify protocol parsers that pose a potential zero-click attack surface. Then, these specific protocol handlers can be manually analyzed or automatically fuzzed. However, *Frankenstein* emulates and fuzzes the firmware as a whole—including a virtual modem for input generation and the ability to attach it to the *Linux BlueZ* Bluetooth stack. Internally, this requires the implementation of interrupt handling and thread switches.

Instead of using the emulator for most of these tasks, *Frankenstein* applies these features as *C* hooks within the firmware. This enables running a selection of these hooks on the physical chip, such as *Frankenstein* heap sanitizer.

Full-Stack Approach The virtual modem and the ability to interact with an operating system mean that *Frankenstein* triggers realistic full-stack behavior. For example, *Frankenstein* generates various pairing dialogs on an *Ubuntu* desktop installation when fuzzing the Link Management Protocol (LMP). In fact, we uncover one complex vulnerability during device scanning, where the host asks for an Extended Inquiry Response (EIR), and the over-the-air reply triggers the bug. The EIR issue is triggered by a specification-compliant message flow, meaning that it works on both *Android* and *Linux* hosts.

Frankenstein is not only faster than over-the-air fuzzing, but our measurements show that it also provides significantly higher hooking performance than the state-of-the-art *Unicorn* engine [48]. This speedup is required for the full-stack capability. If the fuzzer is too slow and runs into timeouts of the operating system driver or cannot handle interrupts and thread switches properly, attaching it to a host is impossible.

In principle, *Frankenstein* could also be attached to other operating systems that support running QEMU locally. As of June 2020, we are working on adding further operating systems.

Another, more complex application would be to replace the

virtual modem with an SDR. While this would only be possible with a high-speed variant that supports at least 80 MHz bandwidth, this would result in a fully software-controllable Bluetooth stack starting at the physical layer. Current SDR-based Bluetooth implementations primarily support physical-layer decoding but do not provide a full stack.

Portability The main focus of this paper is the emulation of the *CYW20735* Bluetooth evaluation board. This board runs on an ARM *Cortex M4* [19]. The underling RTOS is *ThreadX* [22]. A more technical description of similar platforms is provided in Section 7.1.

Frankenstein requires custom hooks inside the firmware. Not accounting for Bluetooth-specific hooks, supporting interrupts and thread switches on ARM with *ThreadX* are approximately 100 custom hooks.

As of June 2020, *Frankenstein* also partially supports the *CYW20819* evaluation board as well as the *Samsung Galaxy S10/S20* firmware. For the latter, no symbols are available at all. However, symbols are required only for the hooks. The emulation itself runs without symbols as it simply interprets and executes binary code based on an initial state—thus, identifying all relevant functions is sufficient. Moreover, we used *Frankenstein* for a non-public project that is not a *Broadcom* or *Cypress* Bluetooth chip. Although this additional project is non-public, we pushed all code changes that enable easier integration of new projects to *GitHub*.

3 RCE-enabled Bluetooth Attacks

In this section, we present various novel attack scenarios enabled by on-chip Remote Code Execution (RCE). Details on how we found and exploited on-chip RCE in the first place are provided in Section 6. Our attacks are practical and apply to the specification, a wide variety of operating systems, or also affect the chips other than *Broadcom*.

A specification-compliant attack to extract link keys is described in Section 3.1. Bluetooth capabilities are typically combined with Wi-Fi within one chip, and with LTE on the same smartphone. We exploit this fact to escalate into the Wi-Fi chip component and cause kernel panics across various smartphone models and outline how to lower LTE performance in Section 3.2. An attacker might be able to brick Bluetooth chips forever, as shown in Section 3.3. In general, it is hard to defend against RCE, as turning off Bluetooth is not guaranteed to reset the chip's memory (see Section 3.4).

This section only discusses on-chip attacks and inter-chip escalations, as these attacks have a potential lifetime of multiple operating system major releases. Escalations into the operating system are highly platform-dependent and rather short-lived. Nonetheless, such escalations pose a significant threat, which has already been demonstrated as an attack for the *Broadcom* Wi-Fi implementation [1, 5, 6]. Since the *iPhone XS*, the Host Controller Interface (HCI) is attached via Peripheral Component Interconnect Express (PCIe), exposing similar escalation targets.

3.1 Link Key Extraction

During initial pairing between two devices, a link key is negotiated. It will ensure the security of all follow-up connections between the two paired devices. If the link key of a user's headset leaks, an attacker can listen to calls and access the user's phone book. Paired keyboards and mice can generate arbitrary input or be eavesdropped. *Smart Lock*, introduced in *Android 5* and still present in *Android 10* [30], enables users to unlock their smartphone with nearby paired devices.

A Bluetooth implementation can either hold the link keys within the controller or on the host. The *Broadcom* chip has no permanent storage except the ROM. Thus, the host stores link keys for all connections. According to the Bluetooth specification [12, p. 1948], the controller can ask the host for a link key associated with a Media Access Control (MAC) address. The host will send back different message types depending on whether it has a link key for a requested MAC address. This separation into two message types simplifies exploitation. For example, an attacker can hook the reply function inside the *Broadcom* chip to copy the link key to the global device name variable. Reusing existing firmware functions makes this patch require around 128 B in practice [16].

The ability of the controller to request any encryption key differs a lot from other wireless standards. It is very specific to Bluetooth, because the simple pairing concept of Trust On First Use (TOFU) also means that there is no additional verification by certificates or other external dependencies. In contrast to existing attacks on pairing and key negotiation [2, 8], our link key extraction does *not* require an active Machine-in-the-Middle (MITM) setup, but RCE.

Our tests on real devices showed that even the link key for inactive connections could be requested. As a link key extraction countermeasure, the host should only return link keys if proper HCI messages were exchanged previously. For example, *BTstack* only copies the link key if it has an active connection [10]. Moreover, the stack should introduce a short delay in link key request replies to prevent MAC address brute-force attacks.

However, this attack can only be made harder, but cannot be prevented completely while keeping the host's implementation Bluetooth specification-compliant. As any proper mitigation would break compatibility with the current specification, including the whole TOFU concept that enables Bluetooth pairing without certificate checks, we did not report this issue to the Bluetooth SIG but only to the vendors. In general, vendors are aware of this—*Apple* even designed *MagicPairing* to secure pairing of their proprietary Bluetooth peripherals and integrate them into *iCloud* [27].

Table 1: Exploiting Wi-Fi through Bluetooth coexistence on combo chips (*CVE-2019-15063*).

Chip	Device	OS	Build Date	Address	Value	Effect
BCM4335C0	Nexus 5	Android 6.0.1	Dec 11 2012	`0x650440`, `0x650600`	`0x00`	Disconnects from 2.4 GHz and 5 GHz Wi-Fi, Wi-Fi can be re-connected.
BCM4345B0	iPhone 6	iOS 12.4	Jul 15 2013	`0x650000`–`0x6507ff`		Disables 2.4 GHz Wi-Fi until restarting Bluetooth.
BCM4345C0	Raspberry Pi 3+/4	Raspbian Buster	Aug 19 2014	`0x650000`–`0x6507ff`	Random	Full and partial Wi-Fi crashes, including Secure Digital Input Output (SDIO), ability to scan for Wi-Fis, speed reduction. Reboot required to restore functionality.
BCM4358A3	Nexus 6P	Android 7.1.2	Oct 23 2014	`0x650000`–`0x6507ff`		Disables all Wi-Fi until restarting Bluetooth.
BCM4358A3	Samsung Galaxy S6	Lineage OS 14.1	Oct 23 2014	`0x650000`–`0x6507ff`		Disables all Wi-Fi until restarting Bluetooth.
BCM4345C1	iPhone SE	iOS 12.4–13.3.1	Jan 27 2015	`0x650200`	`0xff`	Kernel panic, resulting in a reboot.
BCM4355C0	iPhone 7	iOS 12.4–13.3.1	Sep 14 2015	`0x650200`		Kernel panic, resulting in a reboot.
BCM4347B0	Samsung Galaxy S8	Android 8.0.0	Jun 3 2016	`0x650200`		Disables 2.4 GHz and 5 GHz Wi-Fi, kernel panic and reboot when re-enabling Wi-Fi.
BCM4347B0	Samsung Galaxy S8	LineageOS 16.0	Jun 3 2016	`0x650200`		Temporarily disables 2.4 GHz and 5 GHz Wi-Fi, freezes system for a couple of seconds when re-enabling Wi-Fi.
BCM4347B1	iPhone 8/X/XR	iOS 12.4–13.3.1	Oct 11 2016	`0x650200`		Kernel panic, resulting in a reboot.
BCM4375B1	Samsung Galaxy S10/S10e/S10+	Android 9	Apr 13 2018	`0x650200`		Disables 2.4 GHz and 5 GHz Wi-Fi. Reboot required to re-enable Wi-Fi.
BCM4377B3	MacBook Pro/Air 2019–2020	macOS 10.15.1–10.15.5	Feb 28 2018	`0x650400`		Kernel panic, resulting in a reboot.
BCM4378B1	iPhone 11	iOS 13.3	Oct 25 2018	`0x650400`		Kernel panic, resulting in a reboot.

3.2 Inter-Chip Escalation (*CVE-2019-15063*)

In the following, we analyze possibilities to escalate from Bluetooth into further wireless components. This is possible because Wi-Fi and Bluetooth are combined in the same chip, and reside with LTE on the same smartphone. On *Broadcom* Wi-Fi/Bluetooth combo chips, each protocol runs on a separate ARM core, but they share parts of the transceiver. They have a common interface to communicate their needs, which we exploit to shut down Wi-Fi persistently. The operating system cannot prevent this type of inter-chip escalation.

Coexistence between Bluetooth and Wi-Fi is usually realized by applying an Adaptive Frequency Hopping (AFH) channel map [12, p. 289], which can blacklist overlapping 2.4 GHz channels. Vendors can implement proprietary coexistence additions for better performance [12, p. 290]. Simply blacklisting channels is not sufficient on *Broadcom* Bluetooth combo chips—they add their own Enhanced Coexistence Interface (ECI) protocol. ECI optimizes priorities for different types of Wi-Fi and Bluetooth packets. Each protocol stack collaboratively waits for the other, depending on the scenario.

Our practical tests disabling coexistence confirm that *Broadcom* combo chip performance highly depends on it. When streaming a video and simultaneously listening to it with Bluetooth headphones, the video stutters while the sound is playing for a few seconds, and then the sound stops while the video continues buffering. This means, as a countermeasure against attacks on coexistence, *Broadcom* cannot simply disable it. 2.4 GHz Wi-Fi and Bluetooth would block each other significantly, even without any attacker being present.

Coexistence implementations vary a lot between chips. While there are different implementations, firmware compiled between 2012 and 2018[1] map coexistence registers to the same memory area. We crash or practically disable Wi-Fi by writing to those registers via Bluetooth, as listed in Table 1. Often, it is impossible to re-enable Wi-Fi, and the device needs to be rebooted to restore functionality. The *Samsung Galaxy S8* stock ROM tries to re-enable Wi-Fi five times until rebooting with a soft kernel panic. When installing a *LineageOS 16.0* unofficial nightly build from August 30 2019, and performing the same attack on the *Samsung Galaxy S8*, the log shows errors related to `WifiHAL`. While *LineageOS 16.0* does not reboot, the screen is still freezing for a couple of seconds, then turns off and leaves the user at the lock screen. We also observed a kernel panic on the *iPhone SE*, *7*, *8/X/XR*, and *11* related to a kernel mutex and `AppleBCMWLANBusInterfacePCIe`.

In general, coexistence can also be disabled in other ways, such as ignoring callbacks with channel blacklistings or packet transmission requests. The attack also works the other way round—we produced a Wi-Fi firmware that never allows Bluetooth to transmit on the *Nexus 5* with *Nexmon* [41].

Coexistence for shared or co-located antennas is also an issue across vendors. Various frequency bands of technologies used within one device are likely to interfere with Bluetooth, including LTE bands 40 and 7 uplink close to the 2.4 GHz band. In addition to those direct neighbors, harmonics can also interfere. Advanced measurement setups in shielded chambers allow measuring the exact interference within a given device [20].

Vendor-independent solutions enable coexistence between Bluetooth, Wi-Fi, and LTE chips. The Bluetooth specification

[1] Chips require at least a year to appear in the wild, and this is the newest firmware we had access to as of June 2020. The latest *iPhone SE2*, *MacBook Pro 2020*, and *Samsung Galaxy S20* all use firmware dating back to 2018.

outlines a generic Mobile Wireless Standards (MWS) scheme for coexistence with both LTE and Wi-Fi [12, p. 3227ff]. *Broadcom* implements all MWS HCI commands the specification proposes, along with vendor-specific additions. This enables LTE coexistence with chips of different manufacturers, such as *Intel* or *Qualcomm*. Since MWS coexistence is coupled less tightly to the hardware than ECI, we assume that tampering with MWS commands only leads to performance degradation, but no kernel panics. Performance issues highly depend on the chip-internal implementations as well as physical aspects such as the frequency and antenna location.

Indeed, MWS is used on *iPhones*. The `WirelessRadio Managerd` manages coexistence between LTE, Bluetooth, and Wi-Fi. We can observe MWS messages on various *iPhone* models. In contrast, we could not see any MWS messages on the *Samsung Galaxy S8* and *S10e*.

3.3 Bricking Hardware

At first sight, *Broadcom's* memory layout seems unbrickable. Firmware is stored in ROM, and patches are temporarily applied in Patchram. After a hard reset, all changes are gone. Though, there is a Non-Volatile Random-Access Memory (NVRAM) section that should only be written during manufacturing. It contains a per-device configuration like the MAC address and crystal trimming information.

The *WICED Studio* documentation warns users about writing to NVRAM slots below `0x200`. The *WICED* Hardware Abstraction Layer (HAL) only accepts higher slots. An attacker can skip this HAL safety mechanism and directly call the `nvram_write` function. We did not want to brick our Bluetooth devices, yet our experiments writing to NVRAM bricked one *Broadcom* Wi-Fi evaluation board.

While it might still be possible to recover a device to a non-bricked state, this requires system-level access to the Bluetooth controller. On a smartphone, this implies either a patch issued by the manufacturer or the user taking control over the device to unbrick Bluetooth. The latter is an obstacle on *iPhones*, which require to be jailbroken for this, and *Samsung* devices, which flip the *Knox* bit once rooted.

3.4 Ineffective Defense: Disabling Bluetooth

On recent mobile operating systems, turning off Bluetooth via the advanced settings menu will not turn the chip off. This is counter-intuitive because active connections to other devices are lost. We test RCE persistence by checking if memory is reset and timers continue running. The underlying flaw is in the Bluetooth specification, which allows a soft reset.

3.4.1 HCI Reset

According to the Bluetooth 5.2 specification, the `HCI_Reset` command will not necessarily perform a hardware reset [12, p. 2077]. On the *CYW20735* evaluation board, only some timers, current connections, link manager queues, and similar information are reset. No full hardware reset is performed.

3.4.2 Testing Chip Hard Reset

We analyze if a device was appropriately reset. On *Broadcom* and *Cypress* firmware, a `bootcheck` memory area is written during a hard `__reset` of any device under test. We insert custom values into this area. If they stay persistent, we know that no hard reset took place. This approach excludes that memory is persistent due to cold boot effects [47]. Moreover, timer registers can be used to confirm the hardware state. We issue `HCI_Reset` commands on chips ranging from 2012 to 2018. Indeed, the `bootcheck` memory area is never reset.

3.4.3 iOS Devices

On *iOS 12* and *13* devices, including *iOS 13.5*, the Bluetooth chip is neither hard reset when Bluetooth is disabled nor in flight mode. Under some circumstances, like a firmware crash, a hard reset can happen. When Bluetooth is disabled via the settings menu, we can still connect to other devices when issuing commands on the chip. Executing commands on the chip and getting HCI events passed to the host for processing connection establishments requires `btwake` to be active. We believe this to not be a showstopper when facing RCE, since it is implemented as interrupt on the firmware and can be reconfigured. Communication with the host is not necessarily required when adding functions inside the firmware to handle over-the-air requests.

While Bluetooth is enabled on an *iPhone*, it can be found using BLE device scanning. The MAC address is randomized, but an attacker can connect and request the firmware version. BLE advertisements contained a device name in *iOS 12* [46], which has been fixed in *iOS 13*. However, this anonymity does not stop attackers, as Bluetooth requires proximate targets either way. Moreover, Bluetooth has become an even more integral part of *iOS 13* due to features like *Find My* [3].

3.4.4 Android Devices

In contrast to *iOS*, *Android 8* and *9* on a *Samsung Galaxy S8* as well as *Android 9* and *10* on a *Samsung Galaxy S10e* will disable and hard reset Bluetooth in flight mode. However, when not in flight mode, the Bluetooth chip will not be reset by turning off Bluetooth. The latest version we tested is *Android 10* on the March 2020 patch level. This behavior does not change when disabling location services. Whenever a user turns off Bluetooth, only BLE and classic scanning for devices are disabled. No `HCI_Reset` is issued. It is still possible to connect to other devices.

Android 6 on a *Nexus 5* resets memory contents and also reloads the firmware patch file with each Bluetooth restart.

4 Proprietary Firmware Internals

Understanding firmware internals is essential to master emulation and find on-chip RCE vulnerabilities. Figure 2 depicts firmware internals, which we explain top-down in the following. The details described in this section were discovered and analyzed using the emulation techniques described later in Section 5. Our analysis is based on the *Cypress CYW20735* evaluation board and its firmware [18], which was shipped with partial symbols in the *WICED Studio 6.2* toolsuite [35]. For this firmware, no public documentation or source code is available.

4.1 Interaction Between Host and Controller

In Bluetooth terminology, the host is the operating system, and the controller is the chip running the firmware. A host communicates with the controller using the HCI. In the case of the *CYW20735* chip, HCI is sent via Universal Asynchronous Receiver Transmitter (UART) to the host. Data is sent via the same interface using Asynchronous Connection-Less (ACL) and Synchronous Connection-Oriented (SCO) packets. Data does not require any interpretation by the Link Manager (LM).

4.2 ThreadX

The firmware is based on *ThreadX*, a RTOS optimized for embedded devices [22]. *ThreadX* implements threads, events, queues, semaphores, and dynamic memory. The firmware uses several threads, such as the LM, UART state machine, and a special idle thread. Each thread implements a main loop, waiting for events to be processed. When an event for a waiting thread is created, a context switch is performed. Those events are mainly used for inter-thread communication, i.e., pass an HCI packet from the LM to the UART state machine. If all events are processed, the firmware enters an idle state. At this point, new events are only generated by interrupts, such as UART, the Bluetooth Core Scheduler (BCS), or timers.

4.3 Bluetooth Core Scheduler

The BCS is a separate component, handling time-critical Bluetooth events. The interrupt handler `bluetoothCoreInt_C` calls it every 312.5 µs. This timing is the smallest unit of the Bluetooth clock and corresponds to ½ slot length [12, p. 415]. The BCS kernel holds a pool of various tasks, whereas only a single task can be active at any point in time. Tasks implement ACL, device inquiry, paging, and more. They directly access the hardware packet buffer and registers holding packet information. For classic Bluetooth, which supports higher throughput rates than BLE, the packet buffer is mapped into RAM using a Direct Memory Access (DMA) mechanism. On reception, the packet is copied to dynamic memory and handed to the corresponding thread.

5 The Frankenstein Framework

We call our firmware emulation framework *Frankenstein* because it modifies a firmware image to bring it back to life in a different environment. Snapshots of the state of the physical hardware during normal operation can be integrated and ported to the emulated environment. In the following, we showcase the capabilities of our approach on the *CYW20735* Bluetooth controller. However, other firmwares are also supported. The emulated virtual Bluetooth chip can even be attached to a sophisticated operating system like *Linux*, but in principle also to other operating systems that support UART Bluetooth, such as *macOS*. All steps to revive the *CYW20735* firmware are explained in the following.

5.1 Bringing Firmware Images Back to Life

Emulation either requires firmware initialization or a clean memory snapshot containing all registers. Memory snapshots simplify the process for complex firmware. Initially, it might be undocumented how memory is mapped. Thus, *Frankenstein* comes with a `map_memory` hook that overwrites the ARM memory fault handler and sweeps through the whole address range. Once the memory map is known, a snapshot of the memory is obtained from a physical device by executing an `xmit_state` hook, which can be placed within arbitrary functions. The `xmit_state` hook pauses interrupts and disables the watchdog while copying all memory via HCI, which takes several minutes. Since snapshot hooks are placed within functions, the snapshot state is comparably deterministic. For example, snapshots can be taken while the chip has an active connection within a selected protocol handler.

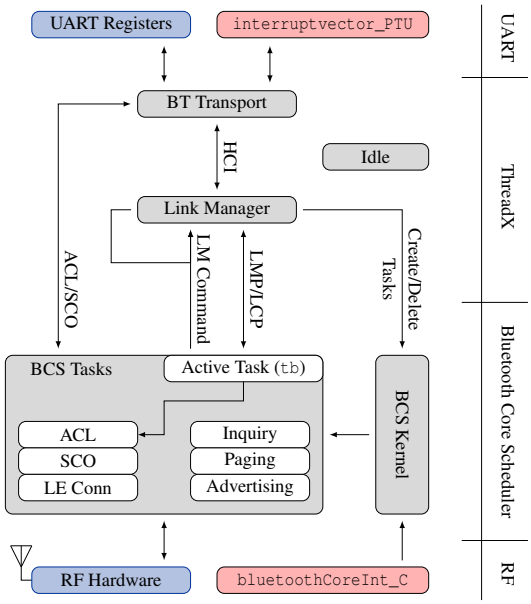


Figure 2: *Broadcom/Cypress* Bluetooth firmware internals.

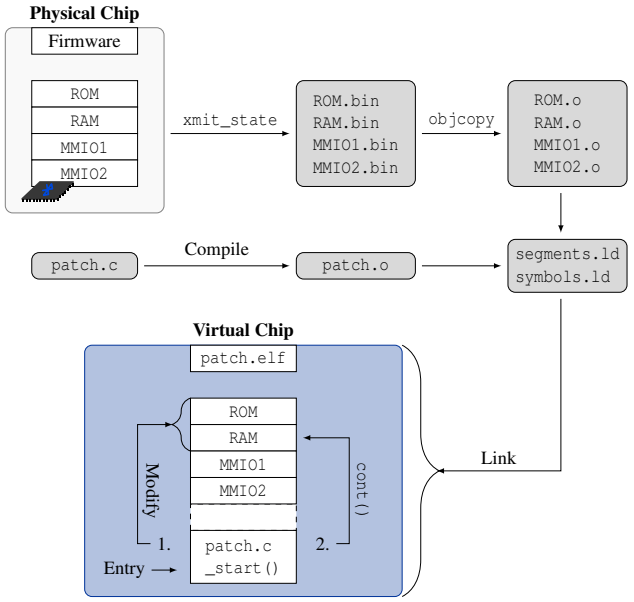


Figure 3: Reassembling the firmware image and live snapshot to an executable ELF file.

We use an unmodified QEMU in user mode for emulation. However, the snapshot is a raw binary without symbols. We reassemble it to an Executable and Linking Format (ELF) file, as illustrated in Figure 3. User-defined code is then compiled and linked against the firmware image. The compiled code is stored in a separate page and provides the initial entry point `_start` for the emulation. It shares the same address space as the firmware, hence it can call functions and parse data structures within the image. The syntax is equivalent to any *C* code written for the firmware. It also adds new features and makes modifications to substitute missing physical hardware.

Frankenstein runs in *Linux* user mode, which does not support interrupts. Thus, we disable functions responsible for enabling and disabling interrupts. Timing-related functions, such as delay, use special purpose hardware and are also replaced. *ThreadX* uses a Supervisor Call (SVC) to perform a context switch between threads. On ARM this is a software interrupt, with a handler located at a known location. As an SVC has special calling conventions that cannot be emulated in user-mode, we re-implemented the handler.

After these modifications, the firmware is executed until the idle thread returns from the interrupt.[2] We replace that return address on the stack with a pointer to our own function. Within this function, we can invoke interrupt handlers like a normal function call to preserve the threading behavior. Thereby, we can inject HCI traffic or Bluetooth frames, as described in Section 5.4 and Section 5.5.

[2]On ARM, returning from exceptions is done by loading a special value to the Program Counter (PC). The idle thread will return to `0xfffffffd`, showing that an interrupt invoked it.

5.2 Hooking for Portability

We implement a lightweight hooking mechanism that can be used to modify the emulated firmware as well as the firmware running on the device. Any code written in *Frankenstein* can also be compiled for the firmware and injected like a shellcode. Even though the firmware is in ROM, it can be patched temporarily. *Broadcom* uses a Patchram mechanism to do so [35]. Each Patchram slot contains a 4 B overlay in ROM and can be used to branch to the actual patch. The number of Patchram slots is very limited, but we use this mechanism as it allows us to install patches on the virtual and the physical firmware.

As the number of modifications to the ROM is limited to 256 Patchram slots on the *CYW20735* chip, we use a trampoline-based approach, similar to the *Nexmon* hook patch variant [41]. More advanced approaches like *RetroWrite* that pose less overhead are completely infeasible, as they rewrite the whole firmware and require position-independent code [21]. Instead, we modify the prologue of the target function to branch to our code. Once our hook is executed, we restore the original prologue and call the target function. On return, we execute a post-hook function to reinstall the hook and continue normal execution.

This hooking mechanism enables *Frankenstein* to trace function calls and analyze interrupt handlers and the corresponding status registers running on QEMU and the physical device. It also supports writing PoCs for over-the-air firmware vulnerabilities running on the physical hardware.

For example, a basic LMP protocol fuzzer requires the following hooks:

1. context switches between threads,
2. Host Controller Interface (HCI) support,
3. hardware interrupt based timers, and
4. ∼100 hooks for debugging and implementation.

5.3 Heap Sanitizer Hook Performance

ThreadX has a custom implementation for dynamic memory called `BLOC` buffer. Each `BLOC` is a continuous chunk of memory, divided into several chunks of equal size. Free chunks are managed using a singly linked list.

The sanitizer iterates over the free list and validates that all pointers are within the `BLOC` pool. *Frankenstein* hooks various functions such as `memcpy` and `dynamic_memory_Release` to integrate this check without further modifications to the heap itself. Thus, the *Frankenstein* sanitizer can also be added during runtime to the firmware running on the physical device.

Unicorn, which is the state-of-the-art firmware hooking tool, allows setting callbacks for each executed basic block, instruction, or memory access. It relies on external function calls [48]. Since *Frankenstein* hooks are modifications to

the firmware itself, no external libraries are called. In addition, the *Frankenstein* hook payload is executed within the instrumented firmware and implemented in *C*. Therefore, it outperforms the *Unicorn* hooking mechanism.

We re-implement the same heap sanitizer with *Unicorn Python* bindings for comparison and run it on a *Thinkpad T430* with an *i5-3320M* CPU. Figure 4 shows the results. The baseline runtime of the instrumented firmware without heap sanitizer is 24.8 ms on average. When sanitizing the heap during LMP fuzzing, *Frankenstein* comes with a performance overhead of 11.6 ms (46.8 %) on the mean average compared to the baseline. The same implementation using *Unicorn* increases the runtime by 145.2 ms (585.5 %) compared to the baseline. Therefore, firmware instrumentation using *Frankenstein* outperforms *Unicorn* by a factor of 12.5 in the heap sanitizer scenario. Performance of other use cases varies depending on the number of hooks.

While the exact speedup depends on the scenario, it is sufficient to overcome the break-even point for the full-stack fuzzing use case. *Frankenstein* emulates the firmware fast enough to enable interaction with an unmodified Bluetooth stack on the host and, thus, attaching it to *Linux BlueZ* [13].

5.4 Talking to an Operating System

Attaching *Frankenstein* to an operating system Bluetooth implementation enables full-stack fuzzing. For example, *CVE-2019-11516* (see Section 6.2) is triggered by the host asking for additional information. On the physical device, HCI traffic is sent to the host via UART. In the emulation, we connect UART to a *Linux* host using a pseudo-terminal device [33]. Opening a Pseudo Terminal Master (PTM) creates a file descriptor, used in the emulator via *Linux* `read` and `write` system calls. The operating system then creates a corresponding Pseudo Terminal Slave (PTS), which is similar to a virtual serial interface. The PTS is then passed to `btattach` to attach the emulator to the *Linux BlueZ* Bluetooth stack.

HCI events generated by the firmware are extracted by hooking `uart_SendAsynch`. This function is a central component of the transmit state machine and gets called for every

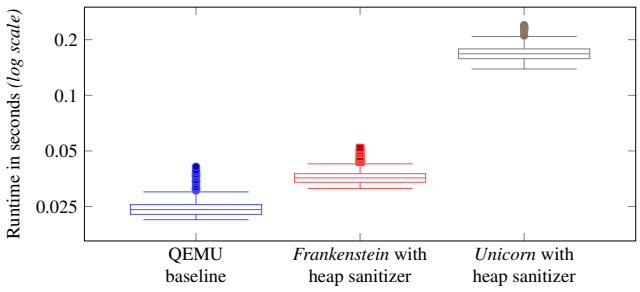


Figure 4: Performance comparison of heap sanitizer with *Frankenstein* and *Unicorn* hooks in LMP fuzzing.

HCI event. Those events are sent to the host using the `write` system call. The opposite direction, injecting HCI commands, requires two steps. We replace functions that read data from UART packet buffers with `read` system calls and analyze the status registers triggering the UART interrupt handler. This will invoke the UART receive state machine implemented in the `bttransport` thread. Note that ACL and SCO data traffic is also passed over the UART interface.

5.5 Non-Wireless Wireless Packet Injection

The virtual modem calls the Bluetooth Core Scheduler (BCS) interrupt handler and generates specific packets for these. For most task types, the packets can be entirely random for reaching maximum coverage. The Link Management Protocol (LMP) was fuzzed coverage-based due to the complexity of the Link Manager (LM) state machine. The most interesting fuzzing optimizations are as follows.

Paging This task accepts any connection attempt.

LMP The LM handles a lot of logic within the firmware and is fuzzed coverage-based, as described in Section 5.6.

We have to analyze the calling convention of `bluetooth CoreInt_C` to implement a virtual modem injecting custom packets. On the device, it is important not to alter the 312.5 µs timing at which `bluetoothCoreInt_C` is called. Hence, only a limited number of debugging techniques can be used. We hook this function on the physical device and dump the hardware registers of interest to a ring buffer. Those are mainly `phy_status` and `sr_status`, which control the BCS kernel. `phy_status` controls which function is executed by `bluetoothCoreInt_C` and depends on the Bluetooth clock.

An example of `phy_status` within an active ACL slave connection is shown in Table 2. Prior to a reception in the *Slot11* interrupt, the receive buffer located in RAM is mapped to the hardware receive buffer using DMA. This memory overlay technique is used to prevent the use of `memcpy` and therefore save CPU resources. Packet data is written to RAM instead of writing it directly to the hardware receive buffer. Within the *receive header done* interrupt, the packet header is available. Besides, it is checked whether the remote device acknowledged the previous transmission. If no retransmission is required, the next packet is put into the ACL task storage for transmission. Those LMP packets which the remote device acknowledged are passed back to the LM for final processing.

Table 2: Calling convention for an ACL slave connection.

Bluetooth Clock	`phy_status`
`0b??00`	Receive header done
`0b??01`	Receive done, *Slot01* interrupt
`0b??10`	Transmit done
`0b??11`	*Slot11* interrupt

Once the packet has been received, *receive done* is called. The receive buffer is unmapped, and the packet is saved to the ACL task storage. During the *Slot01* interrupt, the hardware is configured to transmit the next packet. In addition, the received packet is handed to the corresponding thread. LMP packets are passed to the LM thread. ACL packets are passed to the `bttransport` thread. *Transmit done* will unmap the transmit buffer. This process repeats with the *Slot11* interrupt.

5.6 Code Coverage

We came up with a different representation for coverage-guided fuzzing of protocols. Instead of handling all the input data as a single Binary Large Object (BLOB), we represent it as a sequence of packets, where packets and sequences are mutated separately. This enables the fuzzer to reorder already known packets to increase coverage. We start with a single sequence containing only one packet that consists of null-bytes. For each execution, a random sequence is chosen from the population and mutated. To distinguish the effect of mutating sequences from mutating packets, only one of those is performed per measurement. If the mutation of a single packet increased the code coverage, the sequence containing the new packet is added to the population. If mutating a sequence increased the code coverage, it is also added to the population. Sequence mutations are reordering packets, inserting known packets from other sequences, or merging two sequences.

Both approaches were compared with the same set of mutations over two million test cases, as shown in Figure 5. We previously implemented this reference implementation [39].

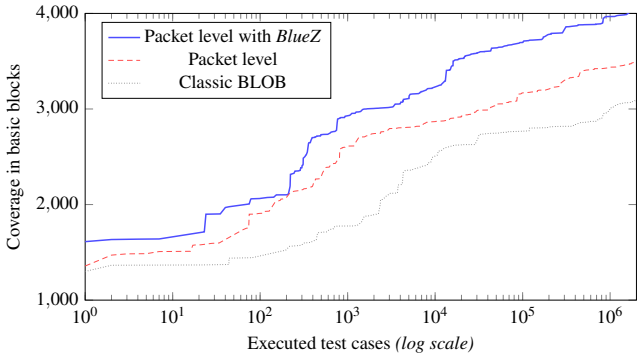


Figure 5: LMP fuzzing strategy comparison.

Table 3: Coverage increase by new zero-click attack surfaces.

Scenario	Coverage
LMP fuzzing	2.76 %
LMP fuzzing with *BlueZ* HCI	0.56 %
Attach to stack `hciconfig hci1 up`	2.04 %
Attach to stack `hcitool scan`	0.59 %
Attach to stack `hcitool cc`	1.04 %
Attach to stack `hcitool lescan`	0.57 %
Attach to stack `hcitool lecc`	1.85 %
Total	9.40 %

Our adaptive approach finds more blocks in a shorter amount of time. The total coverage for LMP fuzzing converges to 2.76 %. Introducing HCI support increases the coverage further by 0.56 %, as HCI handlers and the UART receive state machine are invoked.

We evaluate the total code coverage during LMP and BCS task fuzzing. This was obtained by using QEMU with the `translate_block` trace option. The total code coverage is then loaded to *IDA Lighthouse* plugin [24] to determine the percentage coverage shown in Table 3. Each row shows the amount of new code reached using the described method.

The total code coverage we reached so far is 9.40 %. However, we only analyzed specific scenarios prior to pairing, which enable potential zero-click attacks. This focus is reasonable as the Patchram is limited and *Broadcom* will likely not fix issues that require pairing. The code coverage reached is comparable to the size of the related parts within the Bluetooth specification. For example, we reached 3.32 % code coverage by fuzzing LMP, and the chapter containing LMP in the specification is only 4.05 % of the total Bluetooth specification in pages [12, p. 567ff]. Also, *Broadcom* provides vendor-specific additions and utilizes the *ThreadX* operating system, which are not part of the specification.

As also shown in Table 3, coverage increases by attaching the firmware to the host stack. This realistic behavior is possible due to *Frankenstein*'s full-stack approach. Fuzzers that do not implement thread switches and only focus on one specific protocol handler cannot reach these protocol parts by design. This includes *CVE-2019-11516* that requires interaction between the *BCS* kernel and link manager. Moreover, as *Frankenstein* includes host stack behavior, the identified issues will likely reproduce on physical devices.

Coverage also offers further insights. Even with the partial symbols, identifying relevant functions is complex. Simply calling a function in emulation and observing the execution can help to gain valuable high-level insights into the code. For example, 420 functions end on `Rx` and potentially receive data. Observing coverage enables us to determine which of these functions are important and in which reception handler context they are called.

5.7 Adding New Firmware

Apart from this use case described here, *Frankenstein* is also capable of fully emulating firmware if no memory snapshot is available but only the compiled firmware including debug symbols. ELF is a common format of these images that can be directly imported into *Frankenstein*. Without a memory snapshot, hardware initialization needs to be performed, which is challenging in complex environments. In the use case not described here we were able to set up a working emulation within half a week, including support of buttons, Serial Peripheral Interface (SPI), and Controller Area Network (CAN) interfaces of a smaller firmware.

Our workflow for integrating new firmware looks as follows. The firmware is executed until a fault—such as infinite loop or illegal instruction–occurs. Then, we fix the root cause of this. We add function tracing hooks to function calls that seem to be relevant. Those function calls are displayed during emulation to show the program flow. Prior to functions or interrupt handlers, hardware registers and buffers can be modified, e.g., using `read`. This includes clock values and receive buffers of external hardware. Then, coverage-guided fuzzing can be used to verify how the input is processed by the firmware.

6 Fuzzing Results and Exploitation

This section describes the heap exploitation technique and documents three heap overflows.

6.1 Heap Corruption

None of the observed devices implements any exploit mitigation, such as Data Execution Prevention (DEP) or Address Space Layout Randomization (ASLR). The memory allocator described in Section 5.3 can be easily exploited. With a heap overflow, an attacker can control a free list pointer to point to any location. This pointer is treated as a valid `BLOC` buffer due to repetitive allocations, as depicted in Figure 6. This leads to a *write-what-where* gadget and allows for Remote Code Execution (RCE). The technique has already been discussed for the exploitation of *Marvell* Wi-Fi controllers, although it was not used in the actual exploit [43].

6.2 Classic Bluetooth Device Scanning EIR (*CVE-2019-11516*)

This section describes a heap overflow exploit in device inquiry, utilizing the full stack [12, p. 513]. As a device scans for other devices, these can respond with an EIR. An EIR contains additional information such as the device name, which is copied into an HCI event to be displayed to the user to list available devices for pairing. The EIR length is extracted from the payload header and subject to the same physical-layer constraints such as data rate and maximum packet duration. Due to these physical-layer restrictions, the firmware skips further length checks prior to copying an EIR.

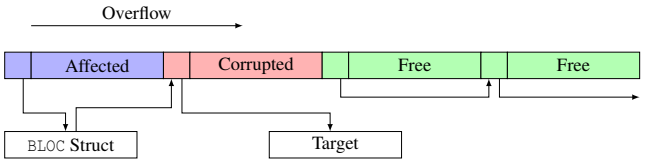
Figure 7 shows the ACL header format [12, p. 482]. The packet length is followed by Reserved for Future Use (RFU) bits, which should be set to zero. The firmware includes these bits in the packet length. Non-zero RFU bits exceed the buffer length of the HCI event.

The hardware buffer holding the payload is not restricted to the payload length of the specific packet being parsed. Even worse, it contains a duplicate of the packet payload, as depicted in Figure 8. This makes memory located after the original packet's payload predictable.

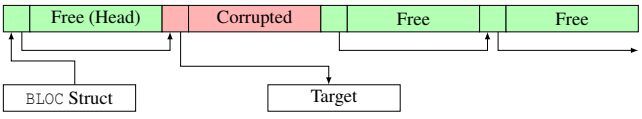
We allocate three buffers in a row within the affected `BLOC` pool to exploit this heap overflow with a write-what-where gadget. This cannot be achieved using the EIR packets, as the data rate is too low compared to the UART connection to the host—the `BLOC` pool would be cleared faster than filled.

We exploit that the host issues an `HCI_Remote_Name_Request` command when an unknown device connects [12, p. 1815ff]. The returned `HCI_Remote_Name_Request_Complete` event has the correct size to be allocated in the affected `BLOC` pool. The attacker-controlled remote name is read via LMP in multiple packets into that buffer. By omitting the last packet and silently dropping the connection, the buffer is kept for several seconds until a timeout occurs. Repeating this process, we can write arbitrary memory, resulting in RCE.

The over-the-air PoC works on various devices, as listed in Table 4. By overflowing the `BLOC` header with an invalid address, the Bluetooth chip of the device under test crashes. The PoC running on the *CYW20735* evaluation board changes the device name to the payload and MAC address to pretend to be multiple physical devices. This method works well against *Android* and *Linux* hosts.



(a) Layout immediately after a heap overflow.



(b) Layout after a heap overflow and a free of the affected buffer.

Figure 6: Effect of overflowing a free `BLOC` buffer.

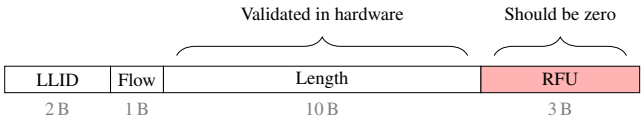


Figure 7: Payload header format for multi-slot ACL packets and all Enhanced Data Rate (EDR) ACL packets.

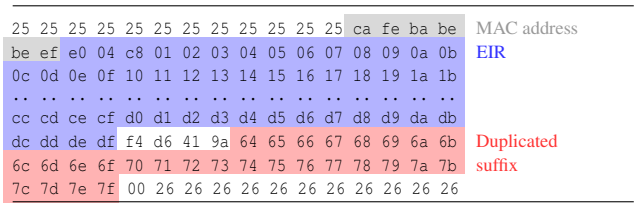


Figure 8: Overflowing hardware buffer that allows control over more bytes than the actual payload length.

Table 4: Devices vulnerable to *CVE-2019-11516*.

Chip	Device	Build Date	Vuln
BCM20702	Thinkpad T430	< 2010?	Yes
BCM4335C0	Nexus 5, Xperia Z3 Compact, Samsung Galaxy Note 3, LG G4	Dec 11 2012	Yes
BCM4345B0	iPhone 6 (unfixed in *iOS 12.4*)	Jul 15 2013	Yes
BCM4358A3	Samsung Galaxy S6, Nexus 6P	Oct 23 2014	Yes
BCM4345C1	iPhone SE (prior *iOS 12.4*)	Jan 27 2015	Yes
Unknown	Samsung Galaxy A3 (2016)	Unknown	Yes
BCM20707	Fitbit Ionic	Unknown	Yes
BCM4347B0	Samsung Galaxy S8	Jun 3 2016	Yes
BCM4347B1	iPhone 8/X/XR (prior *iOS 12.4*)	Oct 11 2016	Yes
BCM4357	Samsung Galaxy 9+/Note 9	Unknown	Yes
CYW20735B1	Evaluation Board	Jan 18 2018	Yes
BCM4375B1	Samsung Galaxy S10e/S10/S10+	Apr 13 2018	No
CYW20819A1	Evaluation Board	May 22 2018	Yes

Table 5: Devices vulnerable to *CVE-2019-13916*.

Chip	Device	Build Date	Vuln
< Bluetooth 4.2	—	< 2014	No
BCM4345C0	Raspberry Pi 3+/4	Aug 19 2014	Yes
BCM4347B0	Samsung Galaxy S8	Jun 3 2016	Yes
CYW20719B1	Evaluation Board	Jan 17 2017	Yes
CYW20735B1	Evaluation Board	Jan 18 2018	Yes
BCM4375B1	Samsung Galaxy S10e/S10/S10+	Apr 13 2018	Crash
CYW20819A1	Evaluation Board	May 22 2018	Yes

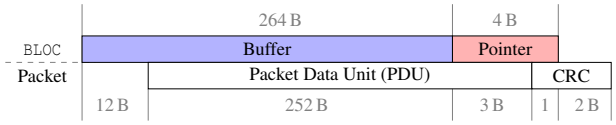


Figure 9: BLE PDU violating a BLOC buffer.

The Bluetooth stack on *Apple* devices does not allow for multiple unauthenticated connections simultaneously and is not covered by our PoC. We extracted ROM and Patchram from jailbroken *iOS 12.4* devices with *InternalBlue* and can confirm that the *iPhone SE*, *7*, and *8/X/XR* received a patch in August 2019 or earlier. On the *iPhone 6*, the vulnerability is still unpatched, but all Patchram slots are already occupied.

Since *Android* needs to support a lot of different hardware, and vendors need to apply individual fixes, patches take a bit longer. A fix was issued on August 5 2019, and it took *Samsung* until mid-September to roll out these patches for their devices.

The EIR vulnerability requires users to scan for devices. We were able to observe device scanning in practice, for example, every few hours inside a residential accommodation. However, we do not know which apps or user actions did trigger device scanning. Some apps require frequent scanning. Bhaskar et al. built a smartphone app used by law enforcement that scans for credit card skimmers using classic Bluetooth [7]. In our observations, location services only use BLE device scanning, but no classic device scanning.

6.3 Any BLE Packet (*CVE-2019-13916*)

This section describes a heap overflow in the reception of Bluetooth Low Energy (LE) Protocol Data Units (PDUs). In Bluetooth 4.2 the maximum PDU length was extended from 20 B to 255 B. A PDU is stored in a special purpose BLOC pool with a buffer size of 264 B, as shown in Figure 9. Besides the PDU payload, the buffer also contains 12 B for headers. Therefore, the buffer is 3 B too small to hold the maximum total PDU length of 255 B. These 3 B are part of the pointer to the next free BLOC buffer, as previously depicted in Figure 6.

However, the fourth overflowing byte is determined by the Cyclic Redundancy Check (CRC) and also copied. It is stored in the receive buffer, despite previously being validated in hardware. An attacker has to adapt the payload, including the CRC, to take control over the heap. The initial CRC state is randomized for each connection. Malicious packets with a chosen CRC need to be produced within the tight Bluetooth clock to prevent connection termination. The attacker can pre-calculate the header and first 248 B payload. The payload can be static for this attack. The next 4 B payload are used to adjust the CRC. After this, 3 B of the BLOC buffer header are inserted. These are followed by the chosen CRC, which manipulates the remaining 1 B of the header.

Our current PoC works over-the-air, but only allocates two BLOC buffers at once by sending a fragmented GATT notification within one BLE event. One additional buffer allocation is needed to gain RCE with a write-what-where gadget. Since the affected BLOC buffer is one of the largest, we assume that there is a standard-compliant way to execute this attack, i.e., using the 1M or 2M PHY modes. Table 5 shows a list of tested devices, which was validated with the partial PoC and local buffer debugging on the device using *InternalBlue*.

Interestingly, the *Samsung Galaxy S10e* is differently affected by exactly the same heap corruption. Bluetooth crashes over-the-air with our PoC because a new heap check was introduced. It checks for overflows by saving the Link Register (LR) and a static 1 B canary at the end of each BLOC buffer element. If the check fails, it crashes gracefully. When this happens, only one heap element is allocated, and we could not deploy a write-what-where gadget. The heap check protects from RCE with *CVE-2019-13916* despite the bug still being present. We were able to produce correct data for the heap check, which already requires all 4 B of our overflow. To control the next element header, an 8 B overflow would be required. Such a new RCE might be found by either patching *CVE-2019-13916* manually on the *CYW20735* firmware and continue fuzzing with *Frankenstein* or by porting it to the non-vulnerable *Samsung Galaxy S10e* firmware.

As we did not provide a full PoC and *Patchram* is limited, *CVE-2019-13916* has not been fixed on any RCE exploitable device to the best of our knowledge, despite reporting it in July 2019.

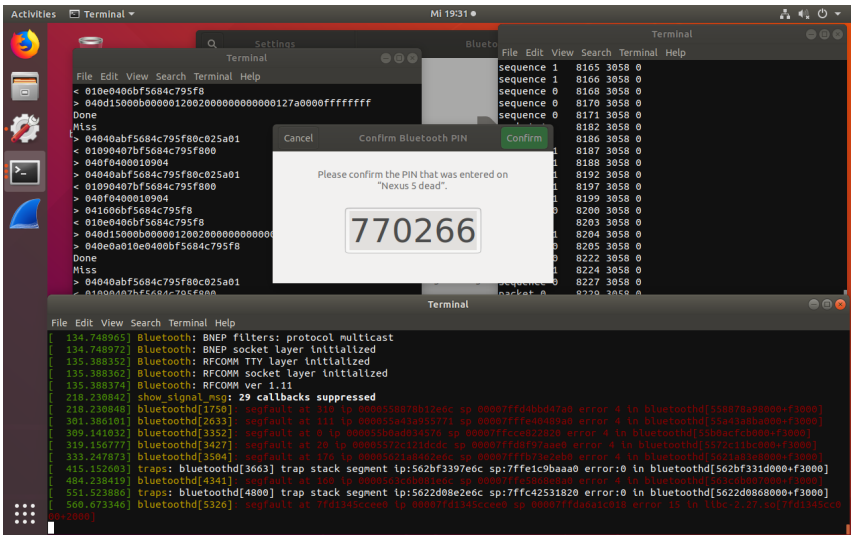


Figure 10: LMP fuzzing results on *Ubuntu* with *BlueZ*.

6.4 Any ACL Packet (*CVE-2019-18614*)

Within classic Bluetooth, Asynchronous Connection-Less (ACL) mode is used for data transfer, such as tethering or music streaming. Similar to HCI, it is sent to the host using UART, but with a different data prefix.

Upon driver initialization by the operating system, the Bluetooth chip signals the maximum packet and buffer size using the `HCI_Read_Buffer_Size` command [12, p. 795]. Broadcom chips are configured for an ACL length of 1021 B and 8 packets. If this buffer is exceeded, this causes a heap overflow. It is important to note that this overflow cannot be exploited without bypassing the driver and operating system Bluetooth stack, which requires privileged access either way.

Yet, on the *CYW20735* chip only, there is a buffer misconfiguration that makes ACL exploitable. The global variables `BT_ACL_HOST_TO_DEVICE_DEFAULT_SIZE` and `BT_ACL_DEVICE_TO_HOST_DEFAULT_SIZE` are set to 384 B, while the chip still signals a size of 1021 B to the host. Thus, just setting up a regular headset for audio streaming as a user immediately results in a heap overflow. As the misconfiguration affects both directions, the heap overflow can also be triggered over-the-air by sending a few *L2Ping* packets exceeding 384 B. When reconfiguring the buffer size in *WICED Studio 6.2*, this bricks the board's capability of flashing new firmware.

This vulnerability stopped us from further ACL fuzzing with the emulated *CYW20735* firmware. It is impossible to take a snapshot during music streaming or tethering before the firmware crashes. However, the *CYW20819* firmware does not have this issue—and *Frankenstein* is almost completely ported to this newer firmware as of June 2020.

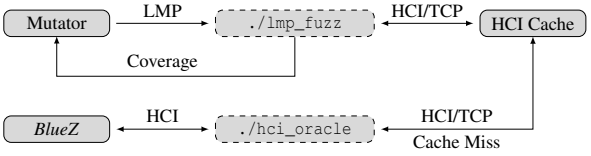


Figure 11: Testing of emulated LMP fuzzing against the *Linux BlueZ* Bluetooth stack.

6.5 BlueFrag (*CVE-2020-0022*)

Nonetheless, we tried to create a PoC for *CVE-2019-18614* based on the assumption that a chip might cache ACL packets if sent using Logical Link Control and Adaptation Protocol (L2CAP) fragments. Instead of crashing the chip, it crashed within `bluetoothd` of an up-to-date *Samsung Galaxy S10e* as of November 2019. After the report, which contained a PoC including a Control Flow Integrity (CFI) bypass to create a reverse shell using Bluetooth within 2 min, this was fixed in the *Android* February 2020 patches as *CVE-2020-0022*. The details of this are covered in our blog post [40].

6.6 Link Management Protocol State Failures

The Link Management Protocol (LMP) in classic Bluetooth is managing connection and encryption setup. The protocol itself is rather simple. However, the most recent attacks affecting a large fraction of Bluetooth devices were located in the LM logic [2, 8]. Each packet type has a fixed length, with the maximum length being 17 B [12, p. 679].

We attach the emulated firmware to a *Linux* host to systematically test LMP with *Frankenstein*, as depicted in Figure 11. The firmware processes LMP packets generated by coverage-guided fuzzing, which in turn causes valid HCI events. A cache answers known event sequences, and un-

known sequences are forwarded to the *Linux BlueZ* host implementation. This differs from code coverage based tools like `syzkaller` [31], because only valid management-related events are passed to the host. Moreover, we aim at increasing coverage within the firmware and not within the host.

This interplay with a real system generates various interesting outputs, as depicted in Figure 10. The user interface shows a lot of weird pairing requests. We even observed faults that produced `dmesg` error outputs and one system freeze. However, they were hard to debug in practice, and we were not able to file specific bug reports.

7 Discussion

This section discusses *Frankenstein* and patching of discovered vulnerabilities on a broader scope. Section 7.1 provides an overview of other firmwares that could be fuzzed with *Frankenstein*. We show the current state of *Broadcom* firmware patches on multiple generations of devices in Section 7.2. Mitigation techniques against our attacks are discussed in Section 7.3 and Section 7.4.

7.1 Applicability to Other Systems

The general idea of emulating firmware to facilitate wireless fuzzing can also be applied to other chips. An emulator similar to QEMU and a basic understanding of the firmware binary are required, though.

Our emulation framework is tailored to ARM chips and *ThreadX*. *ThreadX* is the number one RTOS, which runs on over 6.2 billion devices and provides multiple ARM implementations [22]. Wireless firmware designed for this combination is wide-spread. The other firmware that we internally ported for *Frankenstein* is ARM-based and does not use any operating system at all.

In the following, we provide an overview of wireless firmware based on similar technologies. We assume that more similar wireless platforms exist, however, confirming this requires an extensive analysis of the respective firmware binaries. Due to the popularity of ARM and *ThreadX*, we assume that there are further *Frankenstein* targets.

A platform that uses ARM and *ThreadX* and implements a wireless standard is *Marvell Avastar* Wi-Fi [43]. Moreover, the *Huawei* baseband, as well as the *Shannon* baseband in *Samsung* smartphones, are ARM-based [14]. *Broadcom's* Wi-Fi chips are ARM-based, but the operating system is *HNDRTE* [6]. We took a deeper look into the *Raspberry Pi 3+/4* and *Samsung Galaxy S9* Wi-Fi firmware and compared them to the Bluetooth firmware with known symbols. We found that the `main` function in Wi-Fi and Bluetooth calls `_tx_initialize_kernel_enter`. Thus, both *Broadcom* wireless stacks use *ThreadX* for threading, timers, and events. Yet, Wi-Fi uses *HNDRTE* functions instead of *ThreadX* functions for memory management.

7.2 Patching Bluetooth Vulnerabilities

Broadcom Bluetooth chips are released with a fixed ROM image. Patches are applied using a special Patchram mechanism [35]. Each Patchram slot is temporarily stored in a remapped RAM section and consists of 4 B. This is sufficient to insert a branch instruction to code stored in a regular RAM section. The operating system applies device-specific patches during driver initialization.

Depending on the chip, there can be 128 or 256 Patchram slots. This increasing number shows the need to be able to apply more patches. Analysis of operating system patches reveals that 256 Patchram slots are by far not sufficient. An overview is shown in Table 6. Moreover, the RAM area containing the code each patch jumps into is limited. Overall, even recently released devices only allow for a few more patches. Manufacturers like *Apple*, who support devices for multiple years, cannot include all patches. For example, *CVE-2019-11516* was fixed in *iOS 12.4* on all devices except the *iPhone 6*, which already uses all Patchram slots.

Broadcom claimed *CVE-2019-13916* would not be an issue despite producing a heap overflow. Thus, we assume that *Broadcom* only ships security updates for issues that are publicly known and that they consider exploitable. The limited Patchram slots force them into this decision. To this end, expanding the *Frankenstein* fuzz cases beyond zero-click attacks would likely result in further issues that *Broadcom* would decide not to patch.

When initially finding *CVE-2019-11516*, it was exploitable on any *Broadcom* chip we tested. Surprisingly, during responsible disclosure, *Broadcom* stated that they knew about the issue since February 2018. We could confirm this because the *Samsung Galaxy S10e* ROM contains a fix and has a compile date of April 2018. Interestingly, the most recent *Cypress* evaluation board *CYW20819* with firmware from May 2018 does not contain a fix.

Device manufacturers need to trust *Broadcom* to include proper patches. One of the device manufacturers claimed that *Broadcom* assured them the devices had been patched, despite being vulnerable in our tests. Dissecting and confirming patches at large scale is very hard for anyone besides *Broadcom*. Binary diffing tools perform poorly on raw ARM binaries, as correct function identification due to duplicate meanings in *Thumb* mode at 2 B offsets is challenging [23]. Advanced graph analysis methods fail on this firmware because state-of-the-art disassemblers miss a significant amount of functions, thus, corrupting call graphs. Despite only differing in ARM *Cortex M3* versus *M4*, having comparable compiler options, and similar hardware register locations, less than 6 % of the functions could be identified in practice between the *Nexus 5* firmware and the *CYW20735* evaluation board firmware using *BinDiff* [35].

Table 6: Patchram slots used on various *Broadcom* devices.

Chip	Device	OS	Slots
BCM4345B0	iPhone 6	iOS 12.4	128/128
BCM4345C0	Raspberry Pi 3+/4	Raspbian Buster	128/128
BCM4345C1	iPhone SE	iOS 12.4	127/128
BCM4347B0	Samsung Galaxy S8	Android 9	254/256
BCM4347B1	iPhone 8/X/XR	iOS 13.4.1	240/256
BCM4375B1	Samsung Galaxy S10/S10+	Android 9	212/256

7.3 Memory Protection in Broadcom Chips

Broadcom announced the introduction of *critical area access* memory protection to prevent attacks like *CVE-2019-15063*. The idea is that special purpose registers, such as those for coexistence, can only be configured during device initialization and are locked afterward. Despite reporting *CVE-2019-15063* in August 2019, we did not see *critical area access* as a patch in any firmware as of February 2020. We assume that this feature is infeasible because the underlying ARM chip is a *Cortex M3* on chips prior to 2016 and a *Cortex M4* on newer chips [23], neither of which support such a feature.

After further questions to the *Broadcom* security team about how and when *critical area access* will be applied, we finally saw something potentially related to this feature in *iOS 13.4.1* and the March 2020 *Samsung Android* release. Instead of protecting memory at the chip-level, the HCI commands to read and write memory are restricted, including the undocumented super duper peek poke command. After driver initialization, these commands are blocked. While this helps against misusing `bluetoothd` to block the Wi-Fi chip causing Denial of Service (DoS), it does not protect from over-the-air RCE on the Bluetooth chip and further escalation into the Wi-Fi chip.

7.4 Heap Management in ThreadX

CVE-2019-11516, *CVE-2019-13916*, and *CVE-2019-18614* exploit the heap structure in the underlying operating system. Patching this would secure 6.2 billion systems running *ThreadX*. We proposed *Express Logic* to integrate a heap sanitizer. As the `BLOC` structure contains fixed sizes, these checks run in constant time and could have fully mitigated our exploit technique and helped developers to detect vulnerabilities. They responded that we are not the first to exploit the *ThreadX* heap—a similar attack was published a few months before against *Marvell Avastar* Wi-Fi chips [43]. Nonetheless, they do not plan to integrate any mitigation, stating that applications are responsible for secure heap access.

Despite this statement, the *Samsung Galaxy S10e* performs a very basic heap check. We do not know whether *Broadcom* or *Express Logic* introduced it. Crafting valid payloads is possible with the new check, but the payload needs to be adapted for each firmware version. This is already a requirement for all attacks that rely on calling functions and do not only write to special hardware registers.

8 Related Work

In the following, we summarize existing work on wireless chip exploitation as well as Bluetooth fuzzing.

To the best of our knowledge, publicly available work on Bluetooth fuzzing only covers host implementations. Firmware has not been extensively fuzzed or systematically tested. Vendors might have non-public testing mechanisms. Yet, the previously listed findings in wireless firmware show that vendors do not have sufficient techniques to prevent heap and buffer overflows.

So far, Bluetooth firmware research has been limited to extend chip functionality. *btlejack* builds on the documented *Nordic Semiconductor* BLE firmware [15]. It supports passive and active MITM attacks including BLE 5 hopping. In contrast, *InternalBlue* is based on reverse-engineered *Broadcom* chips [35]. While it does not support MITM attacks, it can read and modify lower layer packets for both BLE and classic Bluetooth. During the implementation of *InternalBlue*, the authors manually detected a security issue on various *Broadcom* chips. Despite the existing works on *Nordic Semiconductor* and *Broadcom* firmware, there has not been any public, systematic security testing on these chips.

An over-the-air fuzzing on top of HCI was implemented in [37]. This black-box testing approach only detects crashes. These crashes might happen in the firmware, however, due to the implementation focusing on host layer protocols, crashes are most likely to happen in the operating system. The remaining fuzzing implementations focus on the driver and operating system level and do not involve any over-the-air packets. For example, `syzkaller` supports fuzzing HCI on Linux [31]. Moreover, *kAFL* and its successors support fuzzing the Linux kernel [9,42]. Implementation faults in operating system components handling Bluetooth can lead to RCE across various operating systems, as the *Blueborne* attacks demonstrated [4]. Such—even wormable—escalations still exist in recent implementations as *CVE-2020-0022* alias *BlueFrag* shows [40].

Broadcom's Wi-Fi chips were initially exploited in 2017 by two independent research teams [5,6]. Heap exploitation was documented in [6], however, the heap is structured differently in the *HNDRTE* operating system. Recently, new *Broadcom* Wi-Fi vulnerabilities have been revealed [1].

Other chipsets were also successfully exploited. The *Marvell Avastar* Wi-Fi uses similar technologies as *Broadcom* and had comparable heap vulnerabilities [43]. The author was using `afl-unicorn` for fuzzing [49], but did neither document the precise setup nor publish any source code. The *Intel* LTE stack is based on x86, and was successfully exploited despite memory protection mechanisms [26]. Moreover, the *MediaTek* baseband exists in an ARM and a MIPS variant and both were fuzzed based on the emulation of security-relevant protocol handlers [34,36]. *Qualcomm* is using their own architecture and assembly for Digital Signal Processing (DSP), *Hexagon*, and implements various memory protection mech-

anisms as well as secure boot. Nonetheless, an over-the-air Wi-Fi buffer overflow exploit that escalates into the Linux kernel driver's memory allocation was found [25]. Security analysis of the Wi-Fi firmware was done manually.

In general, emulation-based fuzzing is also supported by *TriforceAFL* [29]. However, *TriforceAFL* does not use QEMU user-mode emulation like *Frankenstein* but full-system emulation. Instead of adding hooks to the firmware, it modifies QEMU.

In contrast to static program analysis and emulation-based fuzzing, *LTEFuzz* performs over-the-air analysis on LTE and found vulnerabilities in various mobile devices and core network components [32]. Moreover, *SpikerXG* wirelessly fuzzes 2G on multiple smartphones in parallel, including a packet mutator using *YateBTS* [28, 45]. Such approaches are feasible for 2G and LTE, because open source projects like *OpenAirInterface* and *srsLTE* already implement a lot of common protocol features on SDRs [38, 44]. For Bluetooth, there is currently no comparable implementation. Moreover, over-the-air analysis is often unable to determine the precise causes of crashes.

9 Conclusion

In this paper, we demonstrate several security problems originating from Bluetooth RCE—ranging from issues with the Bluetooth specification to broken driver implementations in various operating systems. Our findings unveil the possibility to escalate beyond the Bluetooth circuit boundaries: attackers may take control of the chip over-the-air and, from there, start disturbing Wi-Fi and LTE communications or even crash the entire smartphone.

We create *Frankenstein*, a tool for non-wireless fuzzing of wireless firmware in an emulated environment. *Frankenstein* restarts emulation from snapshots of the device's physical state after frame reception. As it brings fuzzing speed to an unprecedented level, it can be attached to complex operating systems and find full-stack bugs.

Emulation allows understanding RCE vulnerabilities that do not immediately cause a crash but are potentially dangerous. The findings covered in this paper got us in contact with further chip manufacturers, confirming that there is high interest in and awareness of technologies that allow testing wireless implementations and help fixing vulnerabilities.

The vulnerability patching issues of *Broadcom* Bluetooth chips highlight the importance of building sustainable and secure update mechanisms. We found the overall responsible disclosure process quite alarming. One of our attacks, *CVE-2019-11516*, was internally discovered by *Broadcom* in February 2018, but when we informed them about our findings in April 2019, our PoC was working on all *Broadcom* chips we had access to. Usually, until a Bluetooth chip becomes available on off-the-shelf devices, it is at least one year old. Due to the patching mechanism constraints and ease of analyzing patches, *Broadcom* cannot patch all vulnerabilities on older chips. Each patch comes with a high risk of leaking a vulnerability. Despite monthly contact with *Samsung*, a fix for *CVE-2019-11516* took until mid September 2019 on the *Samsung Galaxy S8*, which is comparably well-supported.

Despite failing to fix Bluetooth firmware vulnerabilities, mobile operating systems integrate Bluetooth into critical components. With the overall presence of Bluetooth, even worms spreading wirelessly become feasible. Recent mobile operating systems do not reset and disable Bluetooth properly, even though they suggest to users that they do. The advice to turn off Bluetooth when not needed is insufficient. Always being connected is a very alarming trend regarding over-the-air attacks. Ideally, this trend can be reversed in the future, thus, giving back control over wireless technologies to the users.

Acknowledgments

We thank *Apple*, *Broadcom*, *Cypress*, *Express Logic*, *Fitbit*, *Google*, and *Samsung* for handling the responsible disclosure requests, and René Mayrhofer for assisting us in the responsible disclosure process. Moreover, we thank Dennis Heinze for porting *InternalBlue* to *iOS* and testing *CVE-2019-15063* on various *iPhones*, Dennis Mantz for testing it on the *iPhone X*, and Michael Spörk for the BLE expertise. We also thank Lars Almon, Oliver Pöllny, Bianca Mix, Tim Walter, Dominik Maier, and Teal Starsong for proofreading and Nils Ole Tippenhauer for shepherding this paper.

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

Frankenstein is publicly available on https://github.com/seemoo-lab/frankenstein.

References

[1] Hugues Anguelkov. Reverse-engineering Broadcom Wireless Chipsets. https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html, Apr 2019.

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

[3] Apple. Set up Find My on your iPhone, Mac, and other devices. https://support.apple.com/en-us/HT210400, 2019.

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

[5] Nitay Artenstein. Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets. https://blog.exodusintel.com/2017/07/26/broadpwn/, 2017.

[6] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017.

[7] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please Pay Inside: Evaluating Bluetooth-based Detection of Gas Pump Skimmers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 373–388, Santa Clara, CA, August 2019. USENIX Association.

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

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, August 2019. USENIX Association.

[10] BlueKitchen. BTstack. http://bluekitchen-gmbh.com/btstack/.

[11] Bluetooth SIG. Bluetooth Market Update 2019. https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019.

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

[13] BlueZ Project. BlueZ - Official Linux Bluetooth protocol stack. http://www.bluez.org/.

[14] Amat Cama. A walk with Shannon. https://downloads.immunityinc.com/infiltrate2018-slidepacks/amat-cama-a-walk-with-shannon/presentation.pdf, 2018.

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

[16] Jiska Classen and Dennis Mantz. Reversing and Exploiting Broadcom Bluetooth, June 2019.

[17] Cypress Semiconductor. Cypress to acquire broadcom's wireless internet of things business. https://www.cypress.com/news/cypress-acquire-broadcom-s-wireless-internet-things-business-0, June 2016.

[18] Cypress Semiconductor Corporation. CYW920735Q60EVB-01 Overview. http://cypress.com/CYW920735Q60EVB-01.

[19] Cypress Semiconductor Corporation. Bluetooth (BR + EDR + BLE) Connectivity Solution Families. https://www.cypress.com/products/ble-bluetooth, 2020.

[20] Bernhard Schulz Detlev Liebl. LTE and Bluetooth In-Device Coexistence with WLAN. Application Note. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_application/application_notes/1ma255/1MA255_2e_Coex_LTE_BT_WLAN.pdf.

[21] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[22] Express Logic. THREADX RTOS - Royalty Free Real-Time Operating System. https://rtos.com/solutions/threadx/real-time-operating-system/, August 2019.

[23] Jan Friebertshäuser. Polypyus – The Firmware Historian. https://github.com/seemoo-lab/polypyus/, 2020.

[24] Gaasedelen. Lighthouse - A Code Coverage Explorer for Reverse Engineers. https://github.com/gaasedelen/lighthouse.

[25] Xiling Gong and Peter Pi. Exploiting Qualcomm WLAN and Modem Over The Air. In *DEF CON 27*, Aug 2019.

[26] Guy. Burned in Ashes: Baseband Fairy Tale Stories. In *REcon*, Jun 2019.

[27] 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.

[28] Grant Hernandez and Kevin RB Butler. Basebads: Automated security analysis of baseband firmware: poster. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[29] Jesse Hertz and Tim Newsham. TriforceAFL: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2020.

[30] Jerry Hildenbrand. How to use Smart Lock to unlock your phone automatically. https://www.androidcentral.com/smart-lock, 2018.

[31] Google Inc., Baozeng Ding, Lorenzo Stoakes, Jeremy Huang, Shuai Bai, Alexander Popov, Jean-Baptiste Cayrou, Yuzhe Han, Thomas Garnier, Utkarsh Anand, Tobias Klauser, Tim Tianyang Chen, Ed Maste, Sumukha PK, Mitchell Horne, Hangbin Liu, Denis Efremov, Ondrej Mosnacek, Chi Pham, Anton Lindqvist, Greg Steuck, Shankara Pailoor, Michael Tuexen, Kamil Rytarowski, Siddharth Muralee, Dan Robertson, Mark Johnston, Mellanox Technologies, Cody Holliday, Jin-Woo Lee, and Andrew Turner. `syzkaller` is an Unsupervised, Coverage-Guided Kernel Fuzzer. https://github.com/google/syzkaller, 2019.

[32] Hongil Kim, Jiho Lee, Lee Eunkyu, and Yongdae Kim. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *Proceedings of the IEEE Symposium on Security & Privacy (SP)*. IEEE, May 2019.

[33] Linux. ptmx(4) - Linux man page. https://linux.die.net/man/4/ptmx.

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul 2020.

[35] 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.

[36] Marco Grassi and Kira. Exploring the MediaTek Baseband. https://www.offensivecon.org/speakers/2020/marco-grassi-kira.html, Feb 2020.

[37] Tommi Mäkilä and Jukka Taimisto. Intelligent Bluetooth Fuzzing - Why bother? https://www.youtube.com/watch?v=Rvzrr_jfH64, Nov 2011.

[38] OpenAirInterface. OpenAirInterface - 5G software alliance for democratising wireless innovation.

[39] Jan Ruge. randomFuzz. https://github.com/bolek42/randomFuzz/commit/8ecdd12d83959e7c923ef5e48abdec46bff2ec56.

[40] Jan Ruge. CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE – BlueFrag. https://insinuator.net/2020/04/cve-2020-0022-an-android-8-0-9-0-bluetooth-zero-click-rce-bluefrag/, Feb 2020.

[41] Matthias Schulz. *Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications*. PhD thesis, Technische Universität, 2018.

[42] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[43] Denis Selyanin. Remotely Compromise Devices by Using Bugs in Marvell Avastar Wi-Fi: From Zero Knowledge to Zero-Click RCE. https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, 2018.

[44] Software Radio Systems Limited. srsLTE. https://github.com/srsLTE/srsLTE, August 2019.

[45] SS7ware Inc. YateBTS - LTE and GSM mobile network components for MNO and MVNO. https://yatebts.com/.

[46] 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.

[47] Fabian Ullrich, Jiska Classen, Johannes Eger, and Matthias Hollick. Vacuums in the Cloud: Analyzing Security in a Hardened IoT Ecosystem. In *The 13th USENIX Workshop on Offensive Technologies (WOOT)*, August 2019.

[48] Unicorn. The Ultimate CPU Emulator. http://www.unicorn-engine.org/.

[49] Nathan Voss. afl-unicorn. https://github.com/Battelle/afl-unicorn, August 2019.