Attacks on Wireless Coexistence: Exploiting Cross-Technology Performance Features for Inter-Chip Privilege Escalation

Jiska Classen*
jclassen@seemoo.de

Francesco Gringoli†
francesco.gringoli@unibs.it

Michael Hermann*
mhermann@seemoo.de

Matthias Hollick*
mhollick@seemoo.de

* Technical University of Darmstadt, Secure Mobile Networking Lab
† University of Brescia, CNIT

Abstract—Modern mobile devices feature multiple wireless technologies, such as Bluetooth, Wi-Fi, and LTE. Each of them is implemented within a separate wireless chip, sometimes packaged as combo chips. However, these chips share components and resources, such as the same antenna or wireless spectrum. Wireless coexistence interfaces enable them to schedule packets without collisions despite shared resources, essential to maximizing networking performance. Today's hardwired coexistence interfaces hinder clear security boundaries and separation between chips and chip components. This paper shows practical coexistence attacks on *Broadcom*, *Cypress*, and *Silicon Labs* chips deployed in billions of devices. For example, we demonstrate that a Bluetooth chip can directly extract network passwords and manipulate traffic on a Wi-Fi chip. Coexistence attacks enable a novel type of lateral privilege escalation across chip boundaries. We responsibly disclosed the vulnerabilities to the vendors. Yet, only partial fixes were released for existing hardware since wireless chips would need to be redesigned from the ground up to prevent the presented attacks on coexistence.

I. INTRODUCTION

Wireless communication is enabled by Systems on a Chip (SoC), implementing technologies such as Wi-Fi, Bluetooth, LTE, and 5G. While SoCs are constantly optimized towards energy efficiency, high throughput, and low latency communication, their security has not always been prioritized. New exploits are published continuously [1], [6], [27], [30], [35], [39], [48]. Firmware patching to mitigate flaws requires strong collaboration between vendors and manufacturers, leading to asynchronous, incomplete, and slow patch cycles [48]. In addition, firmware patch diffing can provide attackers with SoC vulnerabilities multiple months before public disclosure [22].

Mobile device vendors account for potentially insecure wireless SoCs by isolating them from the Operating System (OS) and hardening the OS against escalation strategies. For example, on *Android*, the Bluetooth daemon residing on top of OS drivers runs with limited privileges, is sandboxed, and is currently being reimplemented in a memory-safe language [3]. As a result, recent wireless exploit chains targeting the mobile OS instead of the SoC are rather complex and need to find a bypass for each mitigation [8], [9], [47].

We provide empirical evidence that coexistence, i.e., the coordination of cross-technology wireless transmissions, is an unexplored attack surface. Instead of escalating directly into the mobile OS, wireless chips can escalate their privileges into other wireless chips by exploiting the same mechanisms they use to arbitrate their access to the resources they share, i.e., the transmitting antenna and the wireless medium. This new model of wireless system exploitation is comparable to well-known threats that occur when multiple threads or users can share resources like processors or memory [33], [34], [38]. This paper demonstrates lateral privilege escalations from a Bluetooth chip to code execution on a Wi-Fi chip. The Wi-Fi chip encrypts network traffic and holds the current Wi-Fi credentials, thereby providing the attacker with further information. Moreover, an attacker can execute code on a Wi-Fi chip even if it is not connected to a wireless network. In the opposite direction, we observe Bluetooth packet types from a Wi-Fi chip. This allows determining keystroke timings on Bluetooth keyboards, which can allow reconstructing texts entered on the keyboard [43].

Since wireless chips communicate directly through hardwired coexistence interfaces, the **OS drivers cannot filter any events to prevent this novel attack**. Despite reporting the first security issues on these interfaces more than two years ago, the inter-chip interfaces remain vulnerable to most of our attacks. For instance, Bluetooth→Wi-Fi code execution is still possible on *iOS 14.7* and *Android 11*.

Wireless coexistence is indispensable for high-performance wireless transmissions on any modern device [63]. We experimentally confirm that these interfaces exist and are vulnerable:

- We explore a *Broadcom* and *Cypress* Bluetooth↔Wi-Fi interface, which is present in all *iPhones* and *MacBooks*, the *Samsung Galaxy S* series, *Raspberry Pis* and IoT devices.
- We successfully launch Bluetooth→Wi-Fi code execution on all recent *Broadcom* and *Cypress* combo chips.
- We implement packet type information disclosure and Denial of Service (DoS) on the standardized IEEE 802.15.2 coexistence interface, used by *Silicon Labs* and further vendors, to show that this novel attack type is generally applicable.

We received nine Common Vulnerabilities and Exposure (CVE) identifiers for the implemented attacks (see Table I). Since the underlying issue is broader than our practical attack implementations, we informed the *Bluetooth SIG*, responsible for the Mobile Wireless Standards (MWS) specification. Moreover, we included wireless chip manufacturers, such as *Intel*, *Marvell*, *MediaTek*, *NXP*, *Qualcomm*, and *Texas Instruments*, in the responsible disclosure process, if their datasheets mentioned coexistence interfaces [32], [42], [44], [45], [57].

The rest of the paper is structured as follows. We provide a background on wireless coexistence in Section II. Based on the threat model presented in Section III, we introduce coexistence attack concepts in Section IV. Then, we detail practical coexistence attacks on various chips and technologies in Section V. In Section VI, we discuss patch timelines and mitigation strategies while hardware-based fixes are not available. We discuss other wireless side channels, which are related to our novel coexistence-based attack scheme, in Section VII. We conclude our work in Section VIII.

II. WIRELESS COEXISTENCE

Minimizing the number of collisions is a fundamental goal of all wireless communication systems. Technologies like Bluetooth and LTE prevent collisions by assigning to each node a fixed schedule of reserved time slots [11]. Wi-Fi nodes, instead, try to avoid collisions by repeatedly deferring their transmissions if the channel is busy [10]. While these mechanisms are effective when working in isolation, different channel access mechanisms operating in close proximity over the same frequencies can cause severe interference [55]. Wireless coexistence solves this problem by introducing an additional control layer that arbitrates transmissions from different transceivers collocated in the same node [63].

A. Working Principle

We show in Figure 1 Bluetooth Low Energy (BLE) (blue) and Wi-Fi (green) frames transmitted and received at a node that adopts a wireless coexistence mechanism. When BLE is operating using in its assigned time slots, the Wi-Fi transceiver is literally blocked. As soon as the BLE transceiver releases the channel, the Wi-Fi chip may start transmitting a data frame and afterward wait for the acknowledgment that it expects from the receiver. Meanwhile, the BLE transceiver cannot use any time slot that may occur (shadowed gray frames in the middle of the Wi-Fi one) and rests until the next Wi-Fi-free slot. Such coordination is fundamental to *i)* prevent collisions that neither BLE nor Wi-Fi would have been able to avoid on their own and hence *ii)* increase the aggregated throughput.
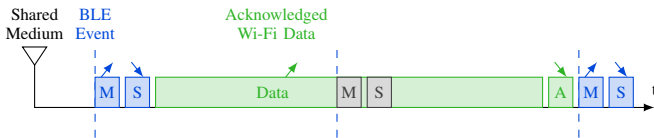

Fig. 1: Wireless packet scheduling example for Wi-Fi and BLE on a shared medium.

Also, technologies that use different frequencies can collide and must adopt some coexistence mechanism. For example, 3GPP allocated for the uplink channel of LTE band 7 frequencies above $2.5\,\mathrm{GHz}$ that are very close to the upper side of the $2.4\,\mathrm{GHz}$ ISM band used by Wi-Fi and Bluetooth: interference is still possible because of spurious harmonics even if there is no actual spectral overlapping [37].
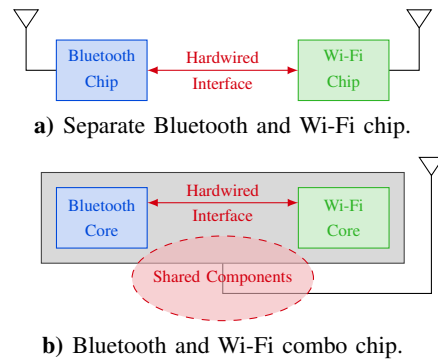
B. Coexistence Architecture

Coexistence implementations depend on the underlying chip architecture: separate chips or combo chips. On separate chips (see Figure 2a), all scheduling information is exchanged using an external hardwired interface. Sometimes, such interfaces are documented for interoperability between chips by different vendors. Combo chips (see Figure 2b) typically embed separate cores dedicated to the different protocols for performance reasons. The combined design allows sharing redundant components such as the antenna. However, additional shared components, such as transmission and reception buffers, could also be involved in coexistence coordination. It is up to the vendor to optimize the hardware design and add proprietary features.

Both concepts are common in practice. For example, we reverse-engineered coexistence interfaces on an *iPhone 11*. We found that Bluetooth and Wi-Fi use a *Broadcom* combo chip with proprietary coexistence features, while the *Intel* LTE baseband chip is interconnected with a serial MWS interface.

C. Involved Components

Coexistence interfaces are part of the chip's hardware because packet scheduling requires real-time information from multiple chips or cores. The firmware running on the chip can send data to hardwired interfaces by writing to Memory Mapped Input/Output (MMIO) addresses. Thus, some coexistence behavior can be firmware-defined. Configurability depends on the initial hardware design by the vendor.

In general, coexistence is managed directly by the chips without involving the OS, e.g. *Android* or *iOS*, which does not see real-time packet scheduling information. However, coexistence information that is not timing-critical can be coordinated externally. For example, on an *iPhone* with MWS, frequency band configuration information from the LTE chip is processed by *iOS* and then forwarded to the Bluetooth chip.

a) Separate Bluetooth and Wi-Fi chip.

b) Bluetooth and Wi-Fi combo chip.

Fig. 2: Coexistence architectures.

TABLE I: Practical coexistence attacks demonstrated in this paper.

Attack Type	Component	Chip Type	Vendor	CVE	Impact	Patchable
Architectural	Shared Memory	Combo	Broadcom, Cypress	CVE-2020-10368	Bluetooth→Wi-Fi memory readout	✗
				CVE-2020-10367	Bluetooth→Wi-Fi code execution	✗
Protocol-based	SECI (Proprietary)	Combo, Separate	Broadcom, Cypress	CVE-2019-15063	Bluetooth→Wi-Fi denial of service	✓
				CVE-2020-10370	Wi-Fi→Bluetooth denial of service	✓
				CVE-2020-10369	Wi-Fi→Bluetooth packet timing & metadata	?
	PTA (Standardized)	Separate	Silicon Labs	CVE-2020-29531	Bluetooth→Wi-Fi denial of service	✓
				CVE-2020-29533	Bluetooth→Wi-Fi packet timing & metadata	?
				CVE-2020-29532	Wi-Fi→Bluetooth denial of service	✓
				CVE-2020-29530	Wi-Fi→Bluetooth packet timing & metadata	?

D. Relevance

Due to backward compatibility, coexistence features are bound to be available on future chips, even if chunks of Wi-Fi traffic are offloaded to the 5 GHz or 60 GHz bands. Even now, recent standards such as 802.11ax continue to operate in the 2.4 GHz band.

III. THREAT MODEL

In the following, we explain the attacker's goals and prerequisites for launching coexistence attacks.

A. Attacker Goals

When exploiting coexistence, the attacker aims at escalating privileges laterally from one wireless chip or core into another. This way, the attacker can extract secret information only available to this other chip (e.g., steal a Wi-Fi password via Bluetooth) and generally widens the attack surface for further escalating into the OS (e.g., use a Bluetooth chip vulnerability to escalate into an *Android* Wi-Fi driver).

B. Attacker Prerequisites

We assume an attacker with code execution on one wireless component. Thus, one wireless core or chip is untrusted. While this is a strong attack precondition, it is similar to other side-channel attacks that escalate privileges for information extraction or code execution, e.g., executing *JavaScript* on a website [28], [49].

Controlling a wireless component requires one of the following efforts. An attacker can execute code by *i)* exploiting an unpatched or new security issue over-the-air, or *ii)* use the local OS firmware update mechanism [40], [51]. Coexistence behavior can also be observed on higher layers, e.g., *iii)* untrusted applications on a smartphone can observe side-effects of packet scheduling of either Bluetooth or Wi-Fi. Such observations do not require code execution within a wireless chip but exploit the mere existence of coexistence protocols. In the remainder of this paper, we consider an over-the-air attacker *i)*. We will practically confirm our attacks using the code execution methods *i)* and *ii)*. We neglect *iii)*, since it does not directly involve wireless chips.

Typically, when a new exploitable wireless bug or attack class is found, it only affects one wireless technology, such as Bluetooth [23], [24], [48], [36], [40], Wi-Fi [1], [6], [9], [21], [26], or cellular basebands [13], [25], [27], [29], [30], [35], [39], [41]. Coexistence attacks change this in favor of

the attacker: vulnerabilities of a single technology can affect multiple technologies due to the lateral privilege escalation.

Even without finding a new bug, an attacker has a chance to gain initial over-the-air code execution due to patch gapping. Wireless firmware patching requires close collaboration between OS maintainers and hardware manufacturers. In practice, rolling out such patches takes multiple months, and some devices receive patches earlier than others [48]. Despite coordinated disclosure processes, even in 2021, Bluetooth chip vulnerability descriptions have been released prior to patches [23], resulting in a patch gap of approximately 1–2 months, depending on the vendor. Similarly, a Wi-Fi fuzzing tool capable of finding new bugs was released in 2021 prior to rolling out all patches [21]. After such publications, the specific technology is at high risk of being exploited, until the vendor rolls out patches and users install them.

IV. COEXISTENCE ATTACK CONCEPTS

We implement practical coexistence attacks against popular chips, as listed in Table I. Prior to implementing these attacks, we provide a brief description of their root cause and impact. The first two vulnerabilities are *architectural*, while the remaining vulnerabilities are *protocol-based*.

A. Architectural Vulnerabilities

Wireless cores or chips can share otherwise redundant components. Even though there is no detailed documentation, some datasheets briefly mention shared components like the antenna [44], [45], [57], the USB to host interface [42], or memory [18]. Chip architectures might contain further undocumented shared components.

Within a shared component, data of multiple wireless technologies is mixed, e.g., in a shared buffer that caches raw samples prior to transmission via the antenna, or in a shared interface to the OS such as USB or PCIe. If these components miss separation, one technology can access data of the other technology. Moreover, the OS sends data to the chips in plaintext, and chips perform encryption, meaning that **unencrypted data can leak** over shared components (CVE-2020-10368). Additionally, if shared components hold firmware data or are involved in firmware updates, this can lead to **code execution** (CVE-2020-10367).

B. Protocol-based Vulnerabilities

Separate wireless chips use hardwired serial interfaces to coordinate packet transmissions. Such interfaces can also be part of a combo chip design, in addition to shared components.

Even a simple serial protocol, which only allows or defers transmissions, can be susceptible to **DoS**, e.g., the coexistence interface can be misconfigured on one end (CVE-2019-15063), transmission requests can be ignored (CVE-2020-10370), or priority transmissions can be requested permanently (CVE-2020-29531, CVE-2020-29532).

Most serial coexistence protocols contain additional information to optimize scheduling, such as transmission slot timings and packet priorities. This can **leak information** across wireless chips: depending on the underlying serial protocol an attacker can determine current packet types and activity (CVE-2020-29533, CVE-2020-29530) or even understand if a Bluetooth packet contained keystrokes (CVE-2020-10369).

C. Stealthiness

Since wireless chip code execution flaws are published so frequently, OSes harden their wireless stacks against privilege escalations from the chip towards the OS, e.g., by reimplementing wireless daemons in *Rust* [3]. However, instead of escalating into higher-level components of wireless stacks, our attacker aims at escalating laterally between wireless technologies within the same level (wireless chip). On this level, components are hardwired, and the OS can neither observe nor filter an ongoing attack. Thus, coexistence flaws bypass OS-level protections like sandboxing wireless daemons.

D. Patchability

Some issues can only be patched by releasing a new hardware revision. For example, a new firmware version will not physically remove shared memory from a chip or adjust for arbitrary jitter in a serial protocol. Moreover, some packet timing and metadata cannot be removed without negatively impacting packet coordination performance. More details on patches and the disclosure timeline follow in Section VI-B.

V. PRACTICAL COEXISTENCE ATTACKS

We implement coexistence attacks on multiple chip types, components, and protocols. This demonstrates that coexistence is exploitable in practice. First, we explain standardized coexistence mechanisms (Section V-A). Then, we outline methods for finding coexistence mechanisms within proprietary chips for analysis (Section V-B). Based on this knowledge, we exploit a proprietary memory sharing mechanism (Section V-C), a proprietary coexistence protocol (Section V-D), as well as a standardized serial protocol (Section V-E).

A. Standardized Coexistence Mechanisms

As a first step for testing coexistence implementations, we need to understand which chips implement which coexistence mechanisms and if these could be vulnerable. Two hardwired inter-chip coexistence mechanisms are both openly standardized and actually used by wireless chips or mentioned in their datasheets. These mechanisms are called Packet Traffic Arbitration (PTA) and Mobile Wireless Standards (MWS).

1) Packet Traffic Arbitration: PTA, specified by IEEE 802.15.2, addresses the coexistence of a co-located Wi-Fi and Bluetooth transmitter [62]. Based on timing and packet type information of both Wi-Fi and Bluetooth, it prevents potential collisions and assigns priorities by packet types. For example, Bluetooth voice transmissions and Wi-Fi reception acknowledgment frames are given priority. The standard defines three collaborative coexistence mechanisms between Wi-Fi and Bluetooth [62]: *i)* A frequency notch filter on the physical layer, *ii)* traffic type prioritization called PTA, and *iii)* time-based transmission division called Alternating Wireless Medium Access (AWMA) on the MAC layer. All of these mechanisms can be combined. The standard does not propose any DoS mitigation.

PTA was specified in 2003, but is still mentioned in recent wireless combo chip datasheets [19], [44], [52]. *Silicon Labs* chips support a basic PTA mode without proprietary extensions, which we will analyze later in Section V-E.

2) Mobile Wireless Standards: The Bluetooth specification defines MWS, which targets Wi-Fi and LTE [11, p. 290ff]. MWS defines inter-chip logical signals, which indicate packet priorities [11, p. 3227ff]. The logical coexistence signals are exchanged using Universal Asynchronous Receiver Transmitter (UART). Bluetooth indicates priority receptions and, thus, can request that the LTE chip ceases its transmission or refrains from starting a transmission. Even though MWS is part of the Bluetooth specification, it also considers chips that integrate Wi-Fi functionality. Since such chips know the Wi-Fi state within the $2.4\,\text{GHz}$ frequency band, there is also a Wi-Fi priority signal. According to the specification, it is not guaranteed that the LTE chip honors these priority signals. However, if it does, this enables DoS attacks. If vendor-specific messages are used [11, p. 3252], information disclosure side channels might also be possible.

In general, *Broadcom* and *Cypress* Bluetooth chips as well as *Intel* LTE chips implement MWS command handlers. We confirm that MWS is used on recent *iPhones* with *Intel* baseband chips. MWS messages can be found in *PacketLogger* traces as well as the system log [4], for example, on the *iPhone 7*, *8*, *11*, and *SE2*.

B. Identifying and Testing Coexistence Features in Practice

In the following, we use a real-world example to illustrate how coexistence interfaces are implemented and how to analyze them for security issues. Most datasheets contain little to no information about coexistence features. Thus, we focus on the previously confidential datasheet of the slightly outdated *BCM4339* Bluetooth/Wi-Fi combo chip by *Broadcom*. The datasheet was released upon acquisition by *Cypress* [17]. We highlight the most important parts of this datasheet in Figure 3. In general, any shared component or connection between the two cores of the combo chip could be attacked:

- Global Coexistence Interface (GCI), which is mentioned to be similar to PTA but with vendor-specific extensions,

- unidirectional WLAN RAM sharing,
- bidirectional WLAN/Bluetooth access, with the exact purpose being undocumented,
- Low-Noise Amplifier (LNA) control,
- Front-End Module (FEM) or Single Pole, Triple Throw (SP3T) switch circuit connecting the antenna.

The datasheet also includes "other" coexistence interfaces without any further specification. All interfaces are undocumented, e.g., commands and addresses to interact with them are missing. The most interesting features are GCI, which is protocol-based and can also be addressed externally, and WLAN RAM sharing, which could potentially be abused for escalating between cores. Further features, such as LNA control and antenna connections, are likely only susceptible to DoS attacks. Thus, we only focus on the first two features.

Proprietary coexistence interfaces in the *Broadcom* chip are not meant to be reprogrammed by external developers. Nonetheless, this is possible by using *Nexmon* for the Wi-Fi core [51] and *InternalBlue* for the Bluetooth core [40]. These two projects provide a patching framework and contain many resources assisting further reverse engineering.

C. WLAN RAM Sharing (Broadcom & Cypress)

In the following, analyze the WLAN RAM sharing feature and find that the Bluetooth core can abuse this feature for reading security-sensitive data from the Wi-Fi core and reliably gain code execution.

1) Reverse Engineering: We recover the *"WLAN RAM Sharing"* internals using leaked symbols from *Cypress WICED Studio 6.2*, a public wireless chip development platform by *Cypress* [20], [22]. It contains symbol information about the Bluetooth part of the *BCM20703A2* combo chip, embedded into 2015–2016 *MacBooks*. In these symbols, we find a few functions following the naming scheme `wlan_buff_*`. Based on these function names and a ROM dump from the *MacBook*, we reverse-engineer the Bluetooth→Wi-Fi memory mapping. When Wi-Fi is turned off, this memory area is all zeros. Otherwise—irrespective of whether Wi-Fi is currently connected to a network—this memory region is mapped. As detailed in Appendix A-A, Bluetooth can read and write data within a large memory region. The shared RAM address mapping is the same on all chips introduced since 2014.

The shared RAM region seems to be a **legacy feature**. Bluetooth firmware compiled after 2016 does not cross-reference the shared RAM, as listed in Table II. The few symbol leaks let us assume that the shared RAM was introduced for audio buffer sharing. It remains unclear why such a feature would require access to almost the full Wi-Fi RAM region via Bluetooth and why it was never removed in newer chips.

2) RAM Sharing Code Execution: Bluetooth can write to the Wi-Fi shared RAM area. We confirm that this is sufficient for Bluetooth to execute code on the Wi-Fi core. Due to its firmware patching mechanism, Wi-Fi executes code in writable memory regions [51]. There are two suitable approaches to identify Wi-Fi RAM areas that lead to code execution:

- Statically reverse-engineering Wi-Fi patches provided by the vendor, which are applied to the RAM region, and find a location that is executed regularly, or
- dynamically writing pre-defined chunks to the Wi-Fi RAM region until they appear in crash logs.
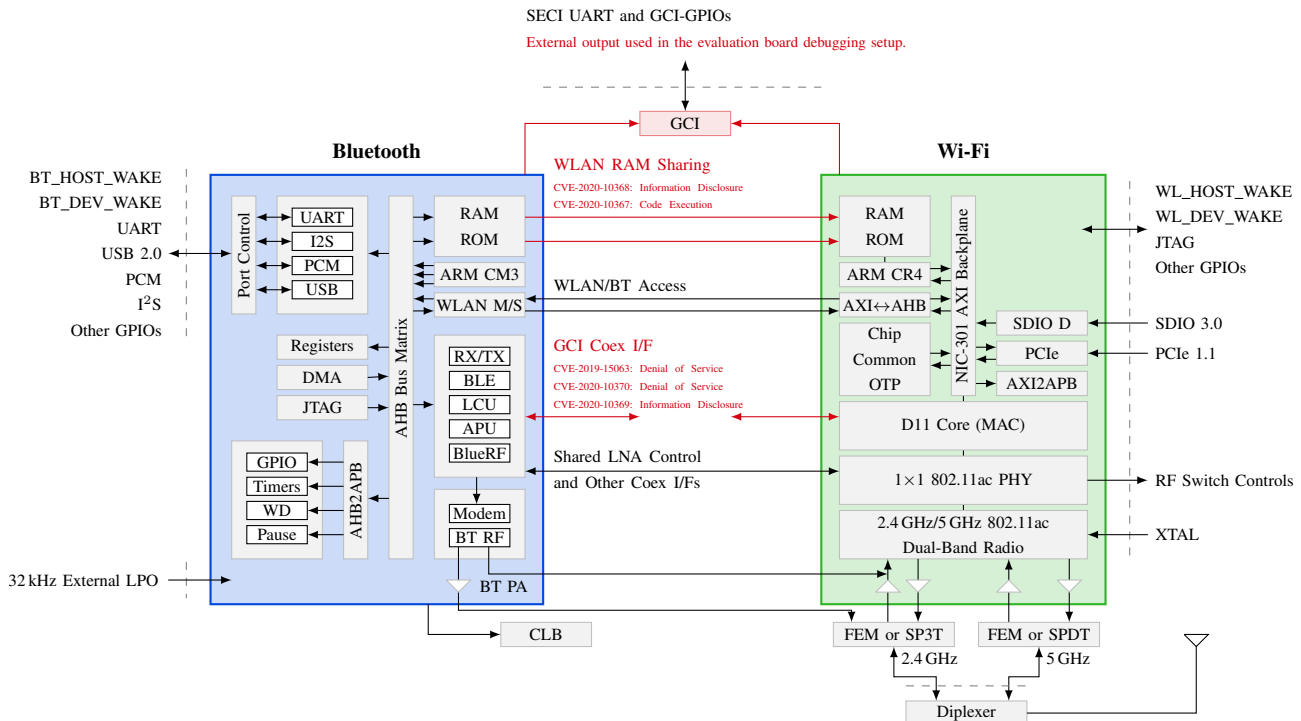

Fig. 3: Coexistence interfaces as documented for the *Google Nexus 5* chip [18], discovered vulnerabilities marked in red.

TABLE II: Wi-Fi code execution and data leak through Wi-Fi/Bluetooth shared RAM (CVE-2020-10368, CVE-2020-10367).

Chip	Device	Tested OS with FW Updates	FW Build Date	FW Accesses `wl_buff`	Code Execution
BCM4335C0	Nexus 5	Android 6.0.1	Dec 11 2012	×	?
BCM4345B0	iPhone 6	iOS 12.4–12.5.1	Jul 15 2013	×	✓★
BCM43430A1	Raspberry Pi 3	Raspbian Buster	Jun 2 2014	×	?
BCM4345C0	Raspberry Pi 3B+/4	Raspbian Buster	Aug 19 2014	✓	✓
BCM4358A3	Samsung Galaxy S6, Google Nexus 6P	Lineage OS 14.1	Oct 23 2014	×	✓★
BCM20703A2	MacBook Pro 2016	–	Oct 22 2015	✓	•
BCM4355C0	iPhone 7	iOS 13.3–14.3	Sep 14 2015	×	✓
BCM4347B0	Samsung Galaxy S8/S8+/Note 8	Android 8.0.0	Jun 3 2016	✓	✓★
BCM4347B1	iPhone 8/X/XR	iOS 13.3–14.7	Oct 11 2016	✓	✓ (★ <14.3)
BCM4375B1	Samsung Galaxy S10/S10e/S10+	Android 9	Apr 13 2018	×	✓★
BCM4375B1	Samsung Galaxy S10/S10e/S10+/S20/Note 20 5G	Android 10	Apr 13 2018	×	✓
BCM4375B1	Samsung Galaxy S20/Note 20 5G	Android 11	Apr 13 2018	×	✓
BCM4377B3	MacBook Pro+Air, 2019–2020 (PCIe)	macOS 10.15.1–10.15.7	Feb 28 2018	×	✓ (★ <10.5.7)
BCM4364B3	MacBook Pro+Air, 2019–2020 (UART)	macOS 10.15.4–10.15.6	May 9 2018	×	✓
BCM4378B1	iPhone 11/SE2	iOS 13.3–13.5	Oct 25 2018	×	✓

? Mentioned in datasheet but probably different mapping, did not work in our test.
• Likely vulnerable but no physical device available for testing.
✓ Code execution within Wi-Fi successfully tested.
★ Kernel panic observed on the OS (Android/iOS/macOS).

Issues persist on all tested up-to-date devices, but not all OS updates were tested due to limited device and jailbreak availability.

The Wi-Fi firmware is always loaded to the same addresses. Thus, once code execution was confirmed on a single chip and firmware patch level, the identified region is valid on all devices of the same model and patch level.

We confirm code execution by analyzing crash logs. However, after initial confirmation and exploit development, logging is no longer required. *Android*, *macOS* and *iOS* generate Wi-Fi crash logs, including a full chip RAM dump. We use RAM dumps to confirm that we can write to the shared RAM and got code execution. On *Android*, Wi-Fi crash logs and memory dumps are written to `/data/vendor/log/wifi/`. On some models, memory dumps are disabled, but the Wi-Fi chip still logs to the kernel, observable via `/dev/kmsg`. *macOS* and *iOS* use a different debug format. On *iOS*, creating these logs requires an additional Wi-Fi debug profile [4]. *macOS* and *iOS* both write to the common log directory. The folder name already contains the crash cause and the file `SoC_RAM.bin` contains the RAM dump.

Finding exploitable regions can be automated by sending randomized branch instructions as bytecode. For example, if the Bluetooth chip writes the instruction `b 0xcafebabe` to a shared RAM address and Wi-Fi executes it, it will crash since there is no valid code at the branch target. The Wi-Fi firmware fault handler forwards the exception to the Wi-Fi driver prior to resetting. Thus, the address `0xcafebabe` appears in the OS Wi-Fi crash logs. The last address byte differs due to ARM-specific branch handling. We find shared RAM regions that **immediately trigger Wi-Fi chip code execution**.

While using this method to identify executable code regions, the Bluetooth chip frequently continues sending bytes to the shared RAM region even after the Wi-Fi chip crashes. The Bluetooth chip can already control the shared RAM during early Wi-Fi chip initialization. In this state, the Wi-Fi chip is not connected to any network. After each chip crash, the

code execution finder should wait at least 10 s, ensuring that executable regions are reached during a regular chip state.

The Wi-Fi crash logs contain further insights into the chip's current state and information. For example, when the chip crashes while being connected to a network, the RAM dump contains the network name and password. Thus, an attacker is able to access this information.

This attack only affects combo chips. Separate chips lack the bus used for memory sharing, and thus, coexistence coordination via pure serial interfaces is not affected. Almost all chips manufactured by *Broadcom* are combo chips. Some *Cypress*-branded chips, such as in the *Raspberry Pi*, are similarly affected by shared RAM attacks [16].

3) Over-the-Air Code Execution: Bluetooth→Wi-Fi code execution can be triggered over the air. We show the potential of coexistence exploitation by building an over-the-air Proof of Concept (PoC) on a *Samsung Galaxy S8*. The authors of *Frankenstein* published CVE-2019-11516 [48], an over-the-air Bluetooth code execution vulnerability. This issue was reported in April 2019, and patches were rolled out in fall 2019. We downgrade the *S8* Bluetooth firmware to a January 2019 patchlevel to ensure it is still vulnerable. We keep the remaining system on an up-to-date *Samsung* stock ROM, including the Wi-Fi firmware. The following equipment is required to perform the attack:

- *CYW20735* evaluation board with a firmware modification to set shellcode device names and change the Bluetooth MAC address, taken from the *Frankenstein* repository [46].
- *Raspberry Pi* or other *Linux* device with the *BlueZ* stack.
- A battery pack (optional).

The total costs of this portable setup are below USD 100. Such a mobile attacker is hard to locate. While the original PoC was meant to write arbitrary memory on the

Bluetooth chip, combining this with the shared Wi-Fi RAM allows us to control arbitrary Wi-Fi memory, resulting in Wi-Fi code execution. On the specific Wi-Fi version running on the *Samsung Galaxy S8*, the Bluetooth address `0x6841d2` is mapped to the Wi-Fi address `0x1841d2`, and the code at this address is executed regularly. When writing the shellcode `b 0xcafebabe` to this address via Bluetooth, the Wi-Fi console output read from `/dev/kmsg` shows that the code is executed:

```
CONSOLE: 000000.454 wl1: Broadcom BCM4361 802.11 Wireless
    Controller 13.38.63 (B0 Network/rsdb)
CONSOLE: 000000.456 ThreadX v5.6 initialized
CONSOLE: TRAP 3(2bfea0): pc
    cafebabc, lr 1843ef, sp 2bfef8, ...
```

Note that *CVE-2019-11516* was only fixed on devices that still received official security updates in fall 2019. Devices like the *Google Nexus 5* or *Samsung Galaxy S6* remain unpatched. Unofficial *Android* images, e.g., *LineageOS*, only contain OS updates—updated firmware patches require a collaboration between *Broadcom* and, respectively, *Samsung* and *Google*.

4) Wi-Fi Kernel Driver Issues: Finding Wi-Fi kernel vulnerabilities is not a focus of this paper and has conceptually been covered by previous work [8], [9], [60]. Yet, our Bluetooth→Wi-Fi coexistence test setup triggers kernel panics across multiple OSes. After our reports, the vendors fixed kernel panics caused by missing PCIe and Wi-Fi state management. Kernel panics and fixes are listed in Table II.

All these crashes were produced while probing the Bluetooth→Wi-Fi shared RAM interface. This indirectly fuzzes the Wi-Fi→host interface, most of the time by crashing the Wi-Fi chip when the OS driver does not expect it. Thus, the majority of these kernel panics are caused by PCIe bus timeouts and failed attempts to bring up the Wi-Fi chip again on *Android*, *macOS*, and *iOS*. Interestingly, some of these crashes indicate more substantial issues—e.g., a malformed PCIe Input–Output Memory Management Unit (IOMMU) request on *iOS*. This means that the fuzzer also manipulates data sent to the host via PCIe.

Overall, the stability issues found in the kernels indicate that Wi-Fi drivers were not tested sufficiently. We assume that especially unexpected states that over-the-air setups cannot trigger but via coexistence were not considered, such as invalid replies during driver initialization or network connection setup. While we were working on coexistence issues inside the chips, other researchers found various issues within *Apple*'s Wi-Fi stack in parallel [8], [60].

D. Serial Enhanced Coexistence (Broadcom & Cypress)

The proprietary Serial Enhanced Coexistence Interface (SECI) by *Broadcom* and *Cypress* is used internally in combo chips. In addition, it is exposed on chips that only support Bluetooth or Wi-Fi, and it can be manually connected. SECI is largely undocumented, except for the information covered in Section V-D1. Its functionality can be observed with a logic analyzer as described in Section V-D2, resulting in understanding the SECI physical layer in Section V-D3. After understanding the protocol, we can mount DoS and information disclosure attacks from the firmware—without a logic analyzer or other hardware modifications. A Bluetooth DoS on the grant reject scheme is demonstrated in Section V-D4. A more severe attack, where Wi-Fi observes packet types and timings of Bluetooth keyboards to determine keystroke timings, is shown in Section V-D5.

1) Documentation: The *BCM4339* datasheet in Figure 3 shows multiple coexistence interfaces [18]. GCI supports various coexistence mechanisms, such as standardized MWS and proprietary SECI. SECI uses UART to transmit 64 bit coexistence data [19], [58]. Wi-Fi sends its current transmit channels to Bluetooth, which blocklists them. Moreover, SECI contains timeout parameters, such as Asynchronous Connection-Less (ACL) and Synchronous Connection Oriented (SCO) timeout limits, powersave and idle timers, as well as medium request and grant timers. Priorities of Wi-Fi and Bluetooth are hardcoded. Audio, video, BLE, and Human Interface Devices (HIDs) have the highest priority, while file transfer has the lowest priority. During Wi-Fi powersave, all Bluetooth requests are granted. Otherwise, coexistence polls between Wi-Fi and Bluetooth.

These features extend PTA. As the *Cypress* coexistence application note states, the chips already implement PTA [19]. SECI augments PTA with additional information to enable more advanced coexistence methods.

2) Reverse-engineering: SECI uses a Bluetooth→Wi-Fi wire and a Wi-Fi→Bluetooth wire. Both wires operate at 3 MBaud and indicate all packets, including metadata, such as the packet type. For comparison, the Bluetooth↔Host UART interface that carries all Bluetooth data also transmits with 3 MBaud. Both the Wi-Fi firmware and *D11* core as well as the Bluetooth firmware access SECI. The Wi-Fi *D11* core is
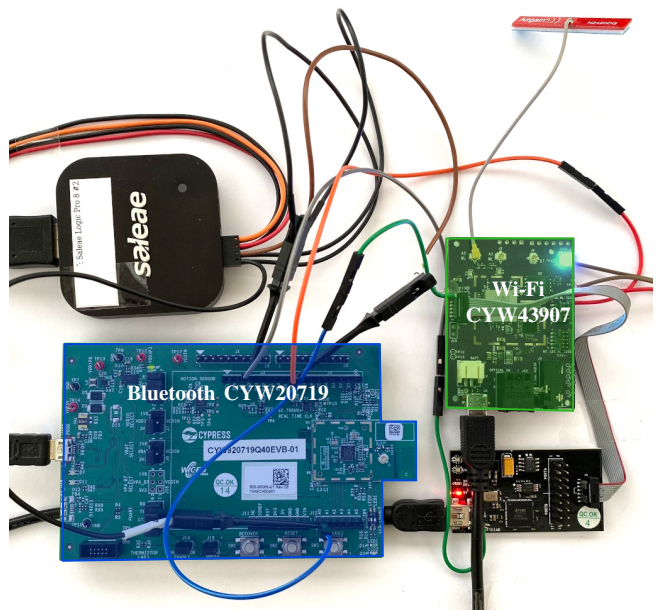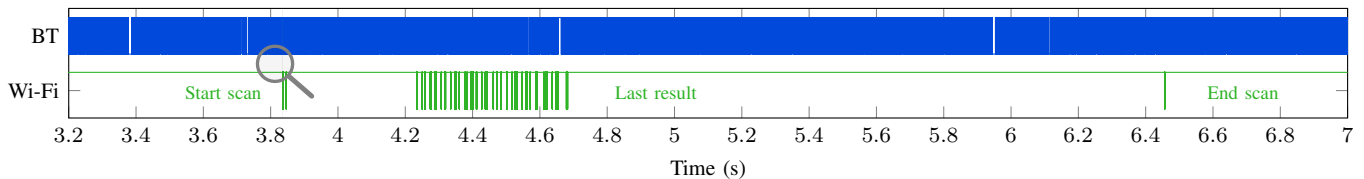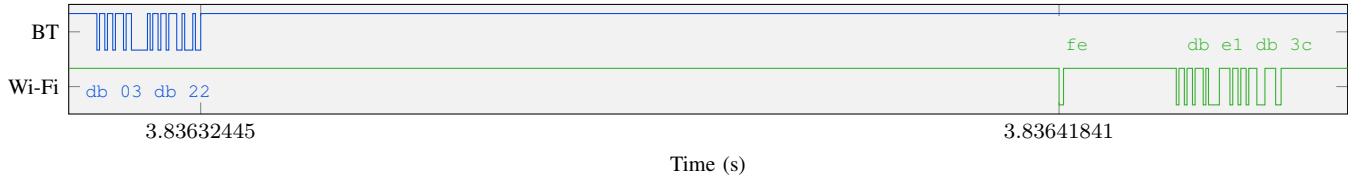

Fig. 4: Debugging setup for the *Cypress* SECI between a separate Wi-Fi and Bluetooth chip.

a) SECI while streaming music via Bluetooth and scanning for Wi-Fi access points.

b) SECI while streaming music via Bluetooth and scanning for Wi-Fi access points, zoomed in to first Wi-Fi peak.

Fig. 5: Initial SECI observations.

a reprogrammable low-level processor that implements a state machine for the first packet processing stage [51].

Combo chips have an integrated SECI connection, but even separate Wi-Fi and Bluetooth modules can be connected with SECI. Each chip generation implements a slightly different version, and only specific chips can be connected [19]. Major differences are 48 bit or 64 bit data size and 3 MBaud or 4 MBaud data rate. Irrespective of physical layer details, SECI is abstracted as hardware-mapped registers. Once a fresh value is written to an output register on one wireless core, it is received as input on the other wireless core.

For the initial analysis, we reproduce a coexistence setup with a separate Wi-Fi and Bluetooth chip [59]. Then, we intercept the exposed SECI with a logic analyzer, as shown in Figure 4. Wi-Fi `TP17` and `TP18` are connected to Bluetooth `D8` and `A0`, respectively. The black board is required for Wi-Fi programming and debugging but has no further wireless functionality. In this setup, all information exchanged between Wi-Fi and Bluetooth only originates from SECI. Antenna locks and other signal-related effects can be excluded.

With this setup, SECI messages can be observed, and their meanings can be reverse-engineered. SECI is not implemented inside the firmware but an external hardware component. The Bluetooth and Wi-Fi firmware only use hardware register mappings to write values over the serial protocol (mapping details for reproducing our experiments are provided in Appendix A-B). According to our threat model, the attacker only has code execution on one of the wireless core, but cannot attach cables. Thus, the attacker can only observe values in these registers at a maximum granularity of fixed clock cycles. We measure that the SECI message timing has 200 ns standard deviation on the Wi-Fi *D11* core. This jitter is way below the *D11* core SECI sampling rate, meaning that **the attacker cannot observe side-channels based on jitter.**

3) Physical Layer Working Principle: Even though the high-level documentation states the basic features of SECI and that it is probably transmitting 48 bit or 64 bit blocks, we do not know its exact working. Thus, we run an initial test series. The Bluetooth input is supplied by a *Linux* host

that is decoupled from the Wi-Fi chip, such that Wi-Fi and Bluetooth can be measured independently. On the *Linux* host, we attach the *CYW20719* evaluation board to *BlueZ* and then use it for audio streaming and keyboard input. Both use ACL packets for data transmission but at different rates. The Wi-Fi evaluation board Real-Time Operating System (RTOS) provides a command-line interface to join an access point as a station, acting as an access point itself or sending pings across the network. Figure 5 shows two captures with this setup.

In the first capture (Figure 5a), Bluetooth streams music while Wi-Fi scans for access points. The pattern indicating a scan has two peaks in the beginning and one in the end. The intermediate peaks differ slightly depending on the number of scan results. Each peak carries serial data (see Figure 5b). The logic analyzer decodes a serial protocol. Most protocol values start with `db`. Each value indicates a certain action. For example, when Wi-Fi starts scanning, the first peak decodes as `fe db e1 db 3c`. Each packet has a variable length, with a maximum of up to 64 bit.

4) Bluetooth Grant and Reject DoS (CVE-2020-10370): Once SECI is set up, Bluetooth waits for grants to send packets. In case 2.4 GHz Wi-Fi is enabled, Wi-Fi might reject Bluetooth packets, as they operate on the same frequency band. Note that Wi-Fi does not reject Bluetooth packets if it is connected to a 5 GHz access point or if it is disabled.

Wi-Fi can abuse this grant and reject pattern. If Wi-Fi does not signal that it is inactive but also stops sending grants, Bluetooth pauses sending packets containing data because its requests are rejected. We build this as *D11* core PoC for the *Nexus 5* as well as the *CYW439037+CYW20719* evaluation board. For the latter, the PoC can temporarily be enabled and disabled by sending a Wi-Fi packet with a particular payload. As shown in Figure 6, Wi-Fi pauses sending SECI messages during this attack period. In the depicted example, Bluetooth is streaming audio, but pauses in case where Wi-Fi does not send any further SECI messages. If the period only lasts a few seconds, Bluetooth is able to maintain active connections with keep-alive null packets but stops sending data. Otherwise, connections time out.
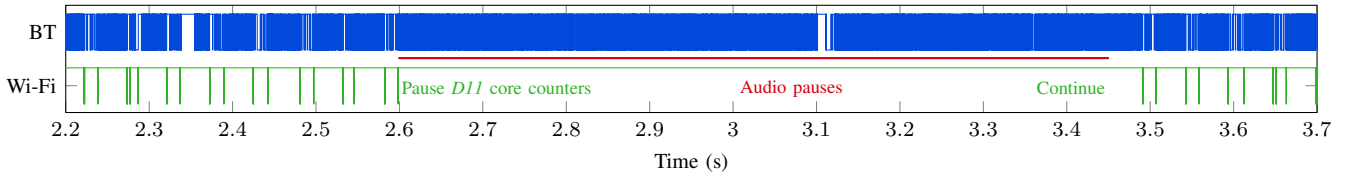
Fig. 6: Temporary Wi-Fi DoS attack on Bluetooth grant requests that pauses audio playback.

5) Inferring Keypress Timings (CVE-2020-10369): In what follows, we show that an attacker with control over the Wi-Fi core can infer exact Bluetooth packet timings and their content type. This can be used to determine if a packet contained data and, thus, observe if a packet sent by a Bluetooth keyboard contains a keypress. Even though we only demonstrate this for a keyboard, further information disclosure attacks based on metadata might be possible.

a) Keypress Attacker Model: Keypress timings, as observable by an attacker, can be analyzed statistically. This becomes interesting for inferring passwords and password lengths. For example, after a long idle time, the user likely first enters login credentials. Previous work has shown that timing attacks on keyboard-based input is possible and developed working statistical methods [43], [54]. Thus, keypress timings should not leak via coexistence protocols.

b) Setup: We pair a *Linux* host with a *CYW20719* Bluetooth chip with a keyboard. On the *Linux* host, we can record a ground truth about what is happening on the Bluetooth interface using *Wireshark*. Each keypress is represented as Asynchronous Connection-Less (ACL) data. Then, we compare these recordings to SECI.

c) Observing Bluetooth via Wi-Fi: We record traces of multiple HID keyboards. Typical HID devices operate at 15 ms. The minimum interval in classic Bluetooth is 1.25 ms [11, p. 2318]. In practice, we find keyboards with timings of up to 30 ms, as shown in Table III. The Wi-Fi *D11* core polls every 1.25 ms and is able to observe each packet's metadata at the same rate as Bluetooth packets are sent.

HID devices like keyboards transfer their keystrokes using the ACL protocol. Thus, on the Bluetooth end, the ACL coexistence handler `_ecsi_gci_HandlerACL` is responsible. This handler writes values to the output register `gci_output`. Once a value is written to this register, the Bluetooth

TABLE III: Timings of Bluetooth HID keyboards.

Product	Timing
Apple Wireless Keyboard	12.5 ms
Apple Magic Keyboard	15 ms
Adafruit Mini Keyboard	30 ms

chip sends it over the SECI interface. Then, this value appears in the Wi-Fi *D11* core register.

We analyze which Bluetooth HID events the Wi-Fi *D11* core can capture. To this end, we hook the `_ecsi_gci_HandlerACL`. This enables us to observe the SECI values as written by the Bluetooth chip and align these with the actual keystrokes as decoded by the Bluetooth chip. In parallel, we attach a logic analyzer to SECI, which is exposed on the *CYW20719* evaluation board. Even though we already know the values that will be sent to the Wi-Fi *D11* core, this enables us to observe potential jitter. The different outputs produced by this experiment are depicted in Figure 7.

Using this setup, we find that keystroke events are indicated by the value 85 sent over the serial interface, while empty packets during inactivity are indicated by the value 05. Thus, the **Wi-Fi *D11* core is able to distinguish between packets with and without keystrokes**. The granularity of frames sent over the SECI physical layer is equal to the HID device events, which is 30 ms in our example. Moreover, we observe a jitter of 200 ns, which is way below the Wi-Fi *D11* core poll interval, and, thus, cannot be captured by the Wi-Fi core.

E. Packet Traffic Arbitration (Silicon Labs)

In the following, attacks on a PTA-based interface are shown. We analyze the *Silicon Labs* Wi-Fi↔Bluetooth implementation. Section V-E1 introduces the basic concept behind PTA. The analysis setup for the *Silicon Labs* coexistence development kit is described in Section V-E2. The plain PTA implementation is susceptible to DoS (see Section V-E3 and
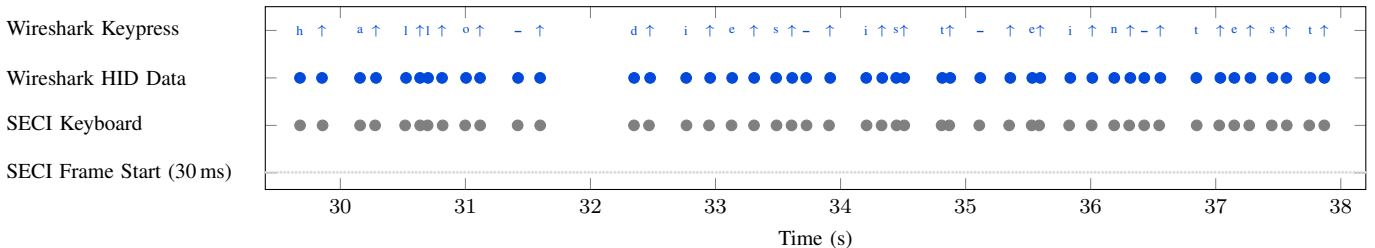

Fig. 7: Keypress timings as observed on logic analyzer (●). SECI time resolution is indicated by (·). which is each 30 ms for the analyzed keyboard. The aligned *Wireshark* trace is observed on the host and contains the decoded keypresses in addition to the slightly inaccurate timings (●).

V-E4), as expected according to the PTA specification. The *Silicon Labs* implementation introduces a significant jitter that is sufficient to let Wi-Fi infer basic Bluetooth protocol activities and vice versa (see Section V-E5 and V-E6).

1) PTA Working Principle: As briefly outlined in Section V-A1, PTA allows for prioritizing transmissions like Bluetooth voice and Wi-Fi acknowledgments. The Bluetooth voice mode in headsets is also known as SCO, which is optimized for low-latency voice, in contrast to music and data transmission. Since it is low-latency, it should be prioritized. In Wi-Fi, each data frame is immediately acknowledged by the receiver within a fixed time slot. If the acknowledgment is not received, the data frame must be retransmitted—causing significantly more congestion.

PTA allows two wireless chips to coordinate their transmissions. Both ends get the ultimate power to request a priority transmission. In an over-the-air scenario, if both ends send their data at once, it would collide, and both transmissions are lost. Thus, PTA decides which transmitter wins the competition and tells the other end to wait. PTA is a simple scheme that only allows to request a transmission slot and optionally mark its priority. No metadata about packet types is included. Each transmitter decides on its own what should be prioritized.

2) Experimental Setup: As shown in Figure 8, PTA coexistence is based on three signals: REQUEST and PRIORITY sent by Bluetooth and GRANT sent by Wi-Fi. *Silicon Labs* integrates the PTA coexistence controller into the Wi-Fi chip. Thus, Wi-Fi knows whether it has a priority transmission and schedules it without asking permission. However, Bluetooth can request priority for urgent transmissions.

Silicon Labs offers a separate coexistence development kit to be plugged into a Wi-Fi and a Bluetooth development board. The coexistence kit provides PTA and additional debug outputs. Parts of the documentation are non-public, but a public application note explains the basic configuration and logic analyzer outputs [52].

Fig. 8: PTA implementation by *Silicon Labs*.

3) Wi-Fi→Bluetooth DoS: In the first experiment, the Bluetooth development board is configured as BLE beacon, which regularly advertises its identity. The beacon sends coexistence REQUEST signals during this time. Once Wi-Fi changes the GRANT signal from 0 to 1 (inverted logic), the requests sent by Bluetooth become shorter. Moreover, the beacon disappears in a Bluetooth scanner app, meaning that it stops sending packets. While this is the expected coexistence behavior, Bluetooth does not continue sending BLE advertisements, even if GRANT is set to 1 for multiple minutes—causing a permanent DoS.

Note that the PTA configuration does not make any difference for the Bluetooth side, since it only observes the resulting GRANT signal generated by the PTA controller.

4) Bluetooth→Wi-Fi DoS: Before turning on the Wi-Fi radio, five different scheduling priorities can be configured in the integrated PTA controller by setting them in the software development platform [53]. The lowest priority for Wi-Fi and highest for the coexistence device, which is Bluetooth in this setup, is COEX_MAXIMIZED. In this mode, Wi-Fi connections might be dropped. The opposite, WLAN_MAXIMIZED, maximizes Wi-Fi priority at the cost of Bluetooth. The default configuration is BALANCED.

As summarized in Table IV, we test if any of these configurations are susceptible to DoS attacks. The PTA priority changes how often Bluetooth coexistence requests are granted. If Bluetooth makes excessive prioritized requests and is given a priority by the PTA configuration, this influences the Wi-Fi throughput. When configuring the Wi-Fi device as an access point and pinging it as a client, prioritized Bluetooth requests lead to 100 % packet loss for almost all settings. Even with the Wi-Fi priority set to WLAN_HIGH, packets are significantly delayed. Only with WLAN_MAXIMIZED priority, a Bluetooth PRIORITY request cannot override the Wi-Fi priority.

TABLE IV: *Silicon Labs* Wi-Fi DoS via Bluetooth requests.

PTA Priority	REQUEST, no PRIO set	REQUEST, PRIO set
COEX_MAXIMIZED	DoS	DoS
COEX_HIGH	DoS	DoS
BALANCED	DoS	DoS
WLAN_HIGH	Grant periodically denied	DoS
WLAN_MAXIMIZED	Wi-Fi gets priority	Wi-Fi gets priority

5) Wi-Fi→Bluetooth Traffic Type Disclosure: The three PTA wires are set to high or low, and do not contain information about Wi-Fi or Bluetooth packets. However, the *Silicon Labs* implementation introduces a significant jitter in Bluetooth REQUEST signals, which Wi-Fi can observe. According to the *Silicon Labs* documentation, coexistence signals are sampled at a rate of $10\,\mu s$. The measurement with a logic analyzer is more precise. However, an attacker with Wi-Fi code execution is limited to this rate. Interestingly, the jitter exceeds $10\,\mu s$ by up to $+302\,\mu s$. The jitter stays within $\pm312.5\,\mu s$, which is the maximum acceptable jitter for BLE coexistence to work at all, since BLE uses $625\,\mu s$ time slots.

BLE traffic can be sent unacknowledged as notification or acknowledged as indication [11, p. 1584ff]. As shown in Figure 9, there are major differences in slot offsets depending
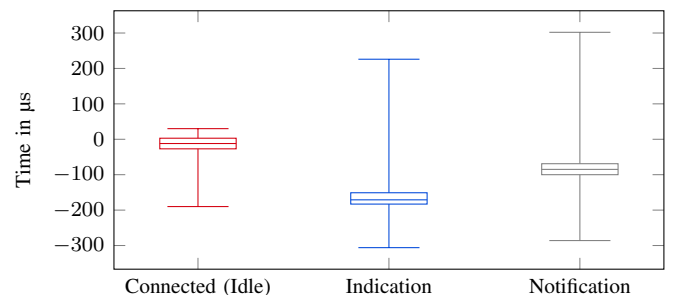

Fig. 9: REQUEST offset compared to $625\,\mu s$ Bluetooth slots.

on the traffic type. The interquantile range of the observed offsets stays within approximately $30\,\mu s$ for all types, but the median shifts drastically.

The jitter only leaves a very short reaction time for Wi-Fi to stay within one Bluetooth time slot from a coexistence performance perspective. Jitter is not only a performance issue but also **Wi-Fi can observe additional information about Bluetooth traffic types by approximating the jitter**. This is surprising because the raw PTA protocol does not contain any metadata. However, the information extracted using the jitter is less fine-grained than metadata from vendor-specific extensions. Even though this measurement focuses on extracting information about packets sent by the Bluetooth chip, further attacks like side-channels on the Bluetooth chip's calculation might be possible. Yet, such attacks are likely prevented when restricting observations to the PTA sampling window of $10\,\mu s$ on the Wi-Fi chip.

6) Bluetooth→Wi-Fi Activity Disclosure: In the opposite direction, the Wi-Fi `GRANT` signals also have significant time variances if Wi-Fi is under high load. However, we assume the same hardware-based sampling constraints for an attacker with on-chip code execution. These prevent fine-grained measurements for effective side channels.

In our experiment, the Wi-Fi development board runs a minimal network stack, including a few testing tools. It includes an `iperf` implementation that can measure TCP throughput. An `iperf` client connects to the Wi-Fi development board, configured as an access point, and sends TCP traffic with a throughput of $7\,\mathrm{Mbit/s}$. At the same time, Bluetooth is configured as BLE beacon, regularly sending `REQUEST` signals.

On the logic analyzer installed in the coexistence interface, **frequent glitches in the `GRANT` signal can be observed while Wi-Fi is under high load**, as shown in Figure 10. They only occur under traffic and if the coexistence priority mode is set to `WLAN_HIGH` or `WLAN_MAXIMIZED`. While this looks like a power supply issue at first sight, setting the logic analyzer output to analog mode reveals that the power level drops to $0\,\mathrm{V}$. Thus, this is likely a programming error.
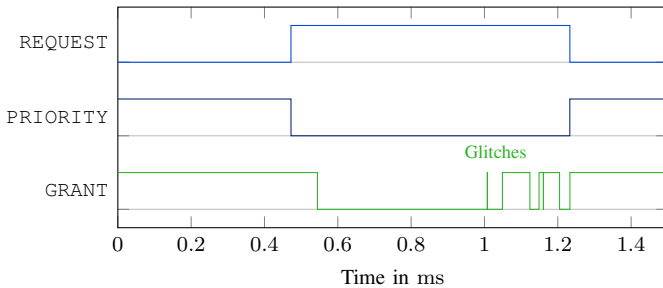

Fig. 10: `GRANT` glitches for `WLAN_HIGH`/`MAXIMIZED`.

Such bugs are surprising given that *Silicon Labs* should have made the same observations with their coexistence development kit. Optimizing time variation and glitches should significantly improve their Bluetooth and Wi-Fi performance.

VI. Mitigation

In the following, we address how coexistence security should be improved and compare this to what has been already patched by manufacturers. Moreover, we discuss how users can minimize their personal risk of wireless exploitation.

A. Improving Coexistence Security

Addressing coexistence issues depends on the affected component (hardware or firmware) and how to control it (e.g., reconfigurable hardware interfaces). Possible mitigations are:

- Fixing chip architectures with shared memory components (hardware),
- stripping metadata and unnecessary information from coexistence protocols (hardware/firmware), and
- adding plausibility checks on resource claims (firmware).

Addressing Hardware Issues: Since our threat model assumes code execution on one wireless core, firmware patches are only effective when they prevents another core from escalating privileges into the patched core. However, the untrusted core could re-enable access to shared hardware components that are temporarily disabled by firmware configurations. Thus, approaches like firmware debloating have limited effect [61]. Ideally, coexistence flaws are patched in hardware.

Fixing chip architectures and hardware will take a lot of time until it reaches customers. Rolling out hardware patches requires a new generation of chips, and as of November 2021, we have not seen a new *Broadcom* or *Cypress* chip generation that addresses the shared memory issue. The chips in the latest *iPhone 12* and *13* have a firmware compile date of October 2019, which is prior to our report of the shared memory code execution flaw. Thus, we expect hardware-based patches not earlier than the *iPhone 14* release.

Changing Coexistence Protocols: Changes to protocol implementations should not reduce the performance under normal operation. However, additional plausibility checks and countermeasures on suspected attack attempts and generally stripping information from these protocols pose a vast **potential to impact performance negatively**.

Moreover, protocol changes require that all wireless chips update protocol implementations at once to maintain inter-chip compatibility. Even if they might be patchable in firmware, parts of these protocols might be implemented in hardware, i.e., firmware can only access abstract packet type and time information but is not able to manipulate raw coexistence protocol signals or adjust jitter to packet time slots.

B. Vendor Patches and Timeline

The complexity of coexistence patches, including the parts that are theoretically patchable as they are implemented in firmware and not hardware, raises an important question: How did vendors apply patches?

Responsible Disclosure Timeline: We reported the first coexistence DoS in August 2019. *Broadcom* replied that they would add protected register access to prevent coexistence reconfiguration via Bluetooth. However, ARM Cortex M3/M4 these chips are based on does not have such a feature. In

January 2020, the issue was still unpatched on an up-to-date *iOS*. Thus, we added *Apple* to the loop, one of the largest customers of *Broadcom* wireless chips, showing that the bug was not fixed, potentially unpatchable, and according to first measurements, even information leakage was plausible. Once we had working PoCs and descriptions for information leakage and code execution, we started the next round of responsible disclosure in March 2020, which also included *Cypress* and further selected customers like *Google* and *Samsung*. Moreover, we informed other chip manufacturers that seemed to have similar issues according to data sheets about the more general nature of coexistence issues. Later on, we built PoCs for *Silicon Labs* chips, which we reported in November 2020 and included the Bluetooth SIG.

Broadcom Patches: As of November 2021, more than two years after reporting the first coexistence bug, coexistence attacks, including code execution, still work on up-to-date *Broadcom* chips. This highlights how hard these issues are to fix in practice. In the following, we outline the usual patch timelines and patches.

Broadcom prioritizes customers by the number of affected chips, with mobile devices patched after 2–4 months. The usual order is *iOS* with irregular patch releases, then *Samsung*-flavored *Android* with a monthly patch cycle, followed by *macOS*. To the best of our knowledge, patches need to be requested by *Broadcom's* customers. This seems to cause a very slow patch timeline for *Linux*-based devices. Thus, *iPhones* and the newest *Samsung Galaxy S* series are the preferred devices to check for patches—and have dedicated security teams to contact in case expected patches are missing.

Cypress Patches: The *Cypress* patch process is independent of *Broadcom* patches. *Cypress* acquired parts of *Broadcom's* wireless division [17], more precisely the IoT devices. After the acquisition, code was developed independently. *Cypress* is not necessarily informed by *Broadcom* if there are vulnerabilities, despite still sharing large parts of the codebase, and patches might be developed independently. To the best of our knowledge, *Broadcom* never released any publicly visible patch release notes—but *Cypress* released some in June 2020 and updated the status in October as follows [16]:

- They claim that the shared RAM feature causing code execution has only been *"enabled by development tools for testing mobile phone platforms"*. They plan to remove stack support for this in the future.
- The keystroke information leakage is remarked as solved without a patch because *"keyboard packets can be identified through other means"*.
- DoS resistance is not yet resolved but is in development. For this, *"Cypress plans to implement a monitor feature in the Wi-Fi and Bluetooth stacks to enable a system response to abnormal traffic patterns"*.

Dividing chips into mobile devices and IoT devices is misleading, since the actual distinction is single combo chip versus separate SECI-connected chips. The wireless chip in the *Raspberry Pi 3B+/4* series has a shared RAM and is part of the chips *Cypress* acquired.

Escalation Risk Reduction: Modern operating systems separate the Bluetooth from the Wi-Fi daemon via sandboxing. Thus, if the Bluetooth daemon is able writing into the Wi-Fi chip RAM, this can already be considered a threat. This is fixed by preventing the Bluetooth daemon to write into the Bluetooth chip's RAM after loading firmware patches during driver initialization, which prevents the exploit chain *Bluetooth daemon→Bluetooth chip→Wi-Fi chip* as well as a potential follow-up escalation into the Wi-Fi daemon. This protection has been put into place since *iOS 13.5*, *Android 10* since the March 2020 release, and *macOS Big Sur*.

Yet, our primary threat defined in Section III, which are **over-the-air attacks via the Bluetooth chip, is not mitigated by current patches**. Only the interface *Bluetooth daemon→Bluetooth chip* is hardened, not the shared RAM interface that enables *Bluetooth chip→Wi-Fi chip* code execution. It is important to note that the **daemon→chip interface was never designed to be secure** against attacks. For example, the initial patch could be bypassed with a UART interface overflow (*CVE-2021-22492*) in the chip's firmware until a recent patch, which was at least applied by *Samsung* in January 2021. Moreover, while writing to the Bluetooth RAM via this interface has been disabled on *iOS* devices, the *iPhone 7* on *iOS 14.3* would still allow another command to execute arbitrary addresses in RAM.

Details on how we removed the Bluetooth RAM write protection in order to check if the Bluetooth chip has any other mitigations in place are provided in Appendix A-C. This protection also prevents vendors from using tools like *InternalBlue* to check patches provided by *Broadcom*. Overall, this mitigation prevents security research while marginally improving security.

C. Personal Risk Minimization

While hardware-related issues remain unpatched, there are simple measures that **significantly reduce the risk of wireless attacks** every user can take:

- Delete unnecessary Bluetooth device pairings,
- remove unused Wi-Fi networks from the settings, and
- use cellular instead of Wi-Fi at public spaces.

An attacker within over-the-air proximity needs to gain code execution on one wireless component initially. This means that they need to exchange malicious data packets that corrupt memory. The fewer data exchange possible, the lower the risk of attacks.

Paired Bluetooth devices have special permissions, such as keyboard input capabilities. Bluetooth is known for issues in the encryption scheme—in the past three years, five critical bugs were published by the Bluetooth SIG [12]. Yet, even after encryption schemes are broken, an initial pairing still requires user interaction. When an attacker gets code execution on a Bluetooth chip, they can get the capabilities of already paired devices—but cannot add new devices. Hence, it is also recommended to delete unnecessary Bluetooth pairings. This reduces the risk that an attacker gains capabilities like keyboard input to use these for further escalation.

Due to the current pandemic situation, many users have Bluetooth enabled for privacy-preserving contact tracing. The Google Apple Exposure Notification (GAEN) API reduces the risk of exploitation by using short, fixed-length packets, which are sent without feedback about reception [5]. Typical memory corruption bugs rely on overflows due to missing length checks and need a feedback channel to bypass security mechanisms like Address Space Layout Randomization (ASLR) [48]. Thus, we consider exposure notifications to be reasonably secure.

Smartphones permanently scan for Wi-Fi networks in the background and try to join them if the network name matches. The most common encryption scheme, WPA2, only verifies that the client has the correct password [31]. If an attacker knows a single valid network name and password configuration of a device, they can spawn a new access point and get active connections. Even worse, networks with captive portals do not require any initial password at all. Thus, users should remove unused Wi-Fi networks to reduce the risk of data exchange that could lead to Wi-Fi firmware exploitation.

Some services run in the background even while Wi-Fi and Bluetooth are disconnected. One example of this is the Apple Wireless Direct Link (AWDL) protocol on *iOS* and *macOS* devices, which is used to seamlessly transfer files using *AirDrop* and similar features [56]. The initial device scan uses Bluetooth, and the data transfer takes place via Wi-Fi. Since a lot of AWDL functionality is based in the kernel, it poses an interesting attack surface, and an exploit for it was published recently [8]. Disabling Wi-Fi via the settings menu on *iOS* disables AWDL. This is also a good measure to minimize the risk of joining malicious Wi-Fi networks.

Cellular data plans got more affordable during recent years and cellular network coverage increased. Disabling Wi-Fi by default and only enabling it when using trusted networks can be considered a good security practice, even if cumbersome.

VII. RELATED CROSS-TECHNOLOGY ATTACKS & SIDE CHANNELS

Shared resources and performance optimizations are well-known to introduce side channels on processors and memory [33], [34], [38].

Screaming Channels exploit electromagnetic side channels on mixed-signal chips [14]. A mixed-signal chip is different from our definition of a combo chip—it just has one digital processing core, such as an ARM Cortex M4. More precisely, the authors attack a *Nordic Semiconductor* chip with an analog Bluetooth radio frontend. The digital processing core is running firmware that calculates AES-128 in software, thereby causing an electromagnetic field. This could already be attacked with classic side-channel attacks in very close proximity. However, the electromagnetic field couples into the digital radio frontend and is amplified along with the intended Bluetooth signal, thereby creating a *Screaming Channel* that leaks similar information over a distance of up to $10\,\text{m}$. Compared to our coexistence attacks, observing *Screaming Channels* requires Software-Defined Radio (SDR)-based measurement equipment, closer and permanent physical

proximity. It can only lead to information leakage instead of code execution.

BLURtooth exploits the fact that BLE and Classic Bluetooth, two modes of operation in Bluetooth with different lower-layer protocols, support cross-transport key derivation [2]. Since BLE and Classic Bluetooth run on the same chip, there is no unauthorized data extraction across chip boundaries compared to coexistence attacks.

Moreover, it is possible to add cross-technology capabilities to a chip. For example, *WazaBee* repurposes a Bluetooth radio frontend by patching its firmware to support Zigbee transmissions [15]. This is possible because BLE and Zigbee are very similar on the physical layer. While such modifications technically enable running two technologies on the same chip, no chip boundaries are violated.

VIII. CONCLUSION

This paper shows that wireless coexistence comes with a huge attack surface and opens up various novel attack vectors, which even enable code execution across chips. While the code execution vulnerability is rooted in architectural issues of specific chips and uncovering required reverse-engineering efforts, DoS and information disclosure attacks of a more general nature can directly be derived from the openly available coexistence specifications.

Wireless coexistence enables new escalation strategies based on hardwired inter-chip components. Since the attack vector lies directly between the chips, it bypasses the main operating system. A full fix will require chip redesigns—current firmware fixes are incomplete.

AVAILABILITY

Our PoCs are based on *InternalBlue* scripts and *Nexmon* patches. They are publicly available as part of the *InternalBlue* project on https://github.com/seemoo-lab/internalblue.

REFERENCES

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

[2] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. Payer, "BLUR-tooth: Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy ," *arXiv preprint arXiv:2009.11776*, 2020.

[3] AOSP, "Gabeldorsche Bluetooth Stack," https://android.googlesource.com/platform/system/bt/+/master/gd, Apr 2021.

[4] Apple, "Profiles and Logs - Bug Reporting - Apple Developer," https://developer.apple.com/bug-reporting/profiles-and-logs/.

[5] Apple and Google, "Privacy-Preserving Contact Tracing," https://covid19.apple.com/contacttracing, May 2020.

[6] N. 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.

[7] Asus, "RT-AC86U Source Code," https://dlcdnets.asus.com/pub/ASUS/wireless/RT-AC86U/GPL_RT_AC86U_300438481049.zip, 2020.

[8] I. Beer, "An iOS zero-click radio proximity exploit odyssey," https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html, Dec. 2020.

[9] G. 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, Apr. 2017.

[10] G. Bianchi, "Performance Analysis of the IEEE 802.11 Distributed Coordination Function," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 3, pp. 535–547, 2000.

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

[12] ——, "Bluetooth Security Notices," https://www.bluetooth.com/learn-about-bluetooth/key-attributes/bluetooth-security/reporting-security/, Apr 2021.

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

[14] G. Camurati, S. Poeplau, M. Muench, T. Hayes, and A. Francillon, "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–177. [Online]. Available: https://doi.org/10.1145/3243734.3243802

[15] R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, and G. Marconato, "WazaBee: Attacking Zigbee Networks by Diverting Bluetooth Low Energy Chips," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[16] Cypress, "Security Bulletin: Potential Wi-Fi + Bluetooth Combo Security Vulnerabilities," https://community.cypress.com/t5/Security-Bulletin/Security-Bulletin-Potential-Wi-Fi-Bluetooth-Combo-Security/ba-p/246960, Jun 2020.

[17] Cypress Semiconductor Corporation, "Cypress to Acquire Broadcom's Wireless Internet of Things Business," https://www.cypress.com/news/cypress-acquire-broadcom-s-wireless-internet-things-business-0, Jun. 2016.

[18] *BCM4339: Single-Chip 5G WiFi IEEE 802.11ac MAC/Baseband/Radio with Integrated Bluetooth 4.1 and FM Receiver*, Cypress Semiconductor Corporation, Mar. 2017, rev. *H.

[19] Cypress Semiconductor Corporation, "AN214852 - Collaborative Coexistence Interface Between Cypress-to-Cypress Solutions and Cypress-to-third-party Chips," https://www.cypress.com/file/298336/download, Mar. 2018.

[20] ——, "WICED Software," https://www.cypress.com/products/wiced-software, Cypress Semiconductor Corporation, 2021.

[21] E7mer, "owfuzz," https://github.com/alipay/Owfuzz, Nov 2021.

[22] J. Friebertshäuser, F. Kosterhon, J. Classen, and M. Hollick, "Polypyus– the firmware historian," *Workshop on Binary Analysis Research (BAR) 2021*, Feb 2020.

[23] M. E. Garbelini, S. Chattopadhyay, V. Bedi, S. Sun, and E. Kurniawan, "BrakTooth: Causing Havoc on Bluetooth Link Manager," https://asset-group.github.io/disclosures/braktooth/, Aug 2021.

[24] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurni-awan, "SweynTooth: Unleashing Mayhem over Bluetooth Low Energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925.

[25] N. Golde, "There's Life in the Old Dog Yet: Tearing New Holes into Intel/iPhone Cellular Modems," https://comsecuris.com/blog/posts/theres_life_in_the_old_dog_yet_tearing_new_holes_into_inteliphone_cellular_modems/, Apr. 2018.

[26] X. Gong and P. Pi, "Exploiting Qualcomm WLAN and Modem Over The Air," in *DEF CON 27*, Aug 2019.

[27] M. Grassi and X. Chen, "Over the Air Baseband Exploit: Gaining Remote Code Execution on 5G Smartphones," *BlackHat USA 2021*, Aug. 2021.

[28] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

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

[30] G. Hernandez and M. Muench, "Emulating Samsung's Baseband for Security Testing," *BlackHat USA 2020*, Aug. 2020.

[31] IEEE, "IEEE 802.11i-2004: Amendment 6: Medium Access Control (MAC) Security Enhancements," 2004.

[32] Intel, "Intel Wireless-AC 9560 Product Brief," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/dual-band-wireless-ac-9560-brief.pdf, 2020.

[33] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, p. 361–372.

[34] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[35] D. Komaromy and L. Szabo, "How To Tame Your Unicorn - Exploring and Exploiting Zero-Click Remote Interfaces of Modern Huawei Smartphones," *BlackHat USA 2021*, Aug. 2021.

[36] V. Kovah, "Finding New Bluetooth Low Energy Exploits via Reverse Engineering Multiple Vendors' Firmwares," *BlackHat USA 2020*, Aug. 2020.

[37] D. Liebl and B. Schulz, "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, Rhode & Schwarz.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv.org e-Print archive*, Jan. 2018.

[39] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul. 2020.

[40] D. Mantz, J. Classen, M. Schulz, and M. Hollick, "InternalBlue - Bluetooth Binary Patching and Experimentation Framework," in *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, Jun. 2019.

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

[42] MediaTek, "MT7632U Datasheet," http://trolink.cn/UploadFiles/Product/20160419152038_94108.pdf, Apr. 2016.

[43] G. Oligeri, S. Sciancalepore, S. Raponi, and R. Di Pietro, "Broken-Strokes: On the (in)Security of Wireless Keyboards," *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, 2020.

[44] Panasonic, "PAN9026 Wi-Fi/BT Module with Marvell 88W8977 WLAN 2.4/5 GHz and Bluetooth single-chip solution inside (Rev 1.2)," https://media.digikey.com/pdf/Data%20Sheets/Panasonic%20Semiconductors%20ICs%20PDFs/PAN9026_Series_Rev1.2.pdf, Jan. 2020.

[45] Qualcomm, "QCA6234 Integrated Dual-Band 2x2 80211n + Bluetooth 4.0," https://developer.qualcomm.com/qfile/28871/lm80-p0598-12_qca6234_datasheet.pdf, 2016.

[46] J. Ruge, "CVE 2019-11516 PoC," https://github.com/seemoo-lab/frankenstein/blob/master/projects/CYW20735B1/patch/CVE_2019_11516.c, Sep 2019.

[47] ——, "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.

[48] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *29th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Aug. 2020.

[49] S. Röttger, https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, Aug 2021.

[50] M. Schulz, "Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications," Ph.D. dissertation, Technische Universität, 2018.

[51] M. Schulz, D. Wegemer, and M. Hollick, "Nexmon: The C-based Firmware Patching Framework," https://nexmon.org, 2017.

[52] Silicon Labs, "AN1128: Bluetooth Coexistence with Wi-Fi, Rev. 1.6," https://www.silabs.com/documents/public/application-notes/an1128-bluetooth-coexistence-with-wifi.pdf, Dec. 2020.

[53] ——, "Wi-Fi Coexistence with Other 2.4 GHz Radio Protocols," https://docs.silabs.com/wifi/wf200/content-source/application-note/wifi-coexistence, Jan. 2021.

[54] D. X. Song, D. A. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.

[55] M. Spörk, J. Classen, C. A. Boano, M. Hollick, and K. Römer, "Improving the Reliability of Bluetooth Low Energy Connections." in *EWSN*, 2020, pp. 144–155.

[56] M. Stute, S. Narain, A. Mariotto, A. Heinrich, D. Kreitschmann, G. Noubir, and M. 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)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 37–54.

[57] Texas Instruments, "CC3235S and CC3235SF SimpleLink Wi-Fi, Dual-Band, Single-Chip Solution Datasheet (Rev. B)," http://www.ti.com/lit/ds/symlink/cc3235s.pdf, Jan. 2020.

[58] VinayakS_26, "Overview of SECI," https://community.cypress.com/community/wiced-wifi/wiced-wifi-forums/blog/2018/04/03/overview-of-seci, Cypress Semiconductor Corporation, Apr. 2018.

[59] ——, "Coexistence Throughput Test," https://community.cypress.com/community/wiced-wifi/wiced-wifi-forums/blog/2019/03/29/coexistence-throughput-test, Cypress Semiconductor Corporation, Mar. 2019.

[60] Y. Wang, "Dive into Apple IO80211FamilyV2," *BlackHat USA 2020*, Aug. 2020.

[61] J. Wu, R. Wu, D. Antonioli, M. Payer, N. O. Tippenhauer, D. Xu, D. J. Tian, and A. Bianchi, "LightBlue: Automatic Profile-Aware Debloating of Bluetooth Stacks," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 339–356. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wu-jianliang

[62] Y. Xiao and Y. Pan, *Overview of IEEE 802.15.2: Coexistence of Wireless Personal Area Networks with Other Unlicensed Frequency Bands Operating Wireless Devices*. IEEE, 2009, pp. 135–150.

[63] J. Zhu, A. Waltho, X. Yang, and X. Guo, "Multi-Radio Coexistence: Challenges and Opportunities," in *2007 16th International Conference on Computer Communications and Networks*, 2007, pp. 358–364.

A. APPENDIX

In this appendix, we provide additional information about the *Broadcom* and *Cypress* vulnerabilities presented in this paper. The details in this appendix include vendor-specific address ranges to leak information or execute code (see Section A-A) as well as SECI protocol details (see Section A-B). Finally, we provide information about how we analyzed and removed the Bluetooth write RAM mitigation on various firmware (see Section A-C).

A. WLAN RAM Sharing

The so-called *"WLAN RAM Sharing"*, previously discussed in Section V-C, enables code execution. The address range of this mapping is shown in Table V. Wi-Fi RAM is mapped into Bluetooth starting at `wlan_mem_base`.

Reading RAM: In the reading direction, the full memory region is not mapped all the time. When just reading in the memory area starting at `wlan_mem_base`, it tends to show some repeating chunks and a lot of zero padding. If a block is currently not ready but read by Bluetooth, this sometimes crashes the Bluetooth firmware. The exact behavior of this memory region depends on the chip. While we are able to read valid Wi-Fi memory chunks on some devices, reading access is not always stable.

Writing RAM: The underlying hardware abstracts writing to shared RAM much more transparently. Without having to deal with any special control registers, it is possible to write to the memory-mapped area via Bluetooth and the written values will immediately appear in the Wi-Fi RAM.

When writing to the Bluetooth chip addresses starting from `0x680000`, these values appear in the Wi-Fi memory dump starting from `0x10000`, the `wlan_mem` area. The debug memory dump is relative to the Wi-Fi chip's RAM address, it is not an absolute address. The memory dump has an offset of `0x170000`, meaning that Bluetooth can control the Wi-Fi area, mapped at `0x180000`.

Code Execution Examples: On the *BCM4377B3* chip of a *MacBook 2020* model on *macOS 10.15.7*, the latest version of *Catalina*, code written to the Bluetooth address `0x68cbfc` is executed by Wi-Fi. The Wi-Fi firmware running on a *BCM43475B1* chip with the *Android 10* patch level of March 2020 is exploitable by writing an arbitrary value to the address `0x681024` via Bluetooth. These are firmware version dependent examples, and generally, a crash caused by writing to this region indicates exploitability on a chip.

Kernel Panics: Writing random bytes to the shared memory on the *BCM43475B1* chip in a *Samsung Galaxy S10/S20* causes various crashes, including a kernel panic due to the locked PCIe Wi-Fi communication. We can produce kernel panics on *Android 9* with a patch level of May 2019. On a

more recent March 2020 *Android 10* release, less severe issues within the driver occur. The phone is no longer able to transmit packets without a manual reboot. At least, the kernel does not panic and the device does not reboot unintended.

The slightly older *BCM4347B0* chip in *Samsung Galaxy S8* devices on *Android 9* reacts similarly including kernel panics. The kernel panics are due to multiple failed attempts of trying to power up Wi-Fi. However, most of the time, only the Wi-Fi core crashes, and sometimes also `wpa_supplicant`, which manages Wi-Fi connections on *Android*. Note that the *Samsung Galaxy S8* still receives quarterly security updates but is not supported beyond *Android 9*.

On *iOS*, the majority of kernel panics occurs due to Wi-Fi driver hangs. The most unstable Wi-Fi implementation is on the *iPhone 6*, which panics immediately on *iOS 12.5.1* when writing random bytes to the shared memory. On an *iPhone 8* on *iOS 13*, we could create malformed PCIe IOMMU requests. Hardware components like the IOMMU are model-specific, and hangs in the kernel can be fixed in software. We could not reproduce kernel panics on the *iPhone 11* on *iOS 14*.

B. SECI Protocol Internals

In the following, we detail SECI protocol reverse-engineering results required for reproducing our findings. More specifically, we detail how SECI is mapped and accessed within the Bluetooth and Wi-Fi firmware.

Bluetooth Hardware Mapping: Symbols belonging to the *CYW20719* Bluetooth module can be extracted from *WICED Studio 6.2* [20]. The most important mappings extracted from these symbols are listed in Table VI and explained in the following.

For each packet sent by Wi-Fi, Bluetooth receives a $64\,\text{bit}$ value in `gci_input`. In the opposite direction, Bluetooth sends information to Wi-Fi via `gci_output`. The values written to `gci_output` and received via `gci_input` similar to those that are visible when intercepting the protocol with a logic analyzer. Thus, instead of wiretapping the SECI physical layer, an attacker can also intercept the coexistence registers and obtain the same information.

Wi-Fi Hardware Mapping: *Broadcom* Wi-Fi firmware is well-documented within the *Nexmon* binary patching framework, which is specifically designed for those chips [50]. Moreover, source code releases for some devices exist, such as the one by *Asus* for the *RT-AC86U* router [7]. Based on this knowledge, we analyze the Wi-Fi coexistence implementation.

The main Wi-Fi firmware running on an ARM core maps the SECI registers in the same order as in Bluetooth in a struct called `chipcregs_t`. The firmware uses the macro `NOTIFY_BT_CHL` to notify Wi-Fi about its current $2.4\,\text{GHz}$

TABLE V: Shared RAM mapping.

Bluetooth Symbol	Bluetooth	Wi-Fi	Wi-Fi Region
`wlan_mem_base`	`0x680000`	`0x180000`	`shared_base`
`wlan_mem` area	`0x680000–...`	`0x180000–...`	`wlan_mem` area

TABLE VI: Coexistence register mapping in Bluetooth.

Address	Name	Function
`0x650000–0x6507ff`	–	GCI region.
`0x650060`	`gci_input`	Value received from Wi-Fi.
`0x650160`	`gci_output`	Value sent to Wi-Fi.

channel and applies the channel bandwidth with the macro `NOTIFY_BT_BW_20`. These values are combined into one byte, with the first half representing Wi-Fi channels 1–11 or 0 for no channel, and the second half being 2 or 4 for 20 MHz or 40 MHz. Writing these values then triggers the process of channel blocklisting in the Bluetooth firmware.

In addition to the firmware running on the ARM core, Wi-Fi has a high-performance *D11* core. This proprietary core uses a special assembly language. It implements a state machine parsing all timing-critical low-level information. The *D11* core maps the Bluetooth `gci_output` register as input to `0xaeb` and `0xaec`. It polls them every two Bluetooth clock cycles, with one cycle being defined as 0.625 ms [11, p. 2318]. Thus, this is the maximum time resolution we can observe via Wi-Fi.

C. Write RAM Mitigation

The base firmware in the ROM is temporarily patched in the RAM by the operating system's driver during chip initialization. The operating system can issue a `Write_RAM` Host Controller Interface (HCI) command, which is required to install patches. The *InternalBlue* experimentation framework features the same functionality [40]. Using *InternalBlue*, we can directly write to the Bluetooth RAM that jumps to the Wi-Fi RAM. We require this test setup—we responsibly disclose all our bugs and, thus, do not have a working over-the-air exploit at the time of testing.

After releasing our escalation path Bluetooth daemon→Bluetooth chip→Wi-Fi, *Broadcom* disabled the `Write_RAM` after firmware initialization. We can remove this patch to confirm that the actual privilege escalation remains unpatched. HCI commands are handled by the function `bthci_cmd_GetDefaultCommandHandler`. Since ROM patches are limited and all vendors have individual HCI patches for their proprietary features, the HCI command handler first checks for a variable function table in RAM. Binary diffing can automatically locate the HCI command handler. The vendor-specific command handlers differ a lot, however, they typically use similar assembler instructions for skipping the `Write_RAM` command. These can be replaced by a non-existent handler.

On *iOS*, we first noticed `Write_RAM` being disabled on *iOS 13.5*, but it was still present on *iOS 13.3*. In *iOS 13.5*, the patches still ship as separate `.hcd` files, which is the common *Broadcom* patch format. Since *iOS 13.6* and also in the current *iOS 14.3*, the `.hcd` files are embedded into the `BlueTool` binary, which applies these patches to the chip. Writing to Wi-Fi RAM via Bluetooth is still possible on *iOS 14.3* after removing the `Write_RAM` patch.

Samsung also uses the `.hcd` format. Starting on the *Android 10* March 2020 and *Android 9* June 2020 release or slightly earlier, they removed the `Write_RAM` and `Read_RAM` handlers. The *Android 11* January 2021 release for the *Samsung Galaxy S20 5G* integrates further validations. The `BRCMcfgD` patch configuration region, which can still be changed, enforces checks on the patch region. Thus, we manually craft a `BRCMcfgD`-based patch, which confirms that the shared RAM region still is present.

macOS removed `Write_RAM` the slowest. On all *Catalina* versions, as tested on three different chips, `Write_RAM` still works. This handler was removed in the *Big Sur* release. Patches are stored in a slightly different `.hex` format on the read-only `/System` volume. Changing files on this volume is possible but requires disabling System Integrity Protection (SIP), which should only be done on testing devices not containing any user data.

For the *Raspberry Pi*, which is the most popular platform on *Linux*, the situation is even worse. The patch slots for this specific chip are exhausted. Patches are typically not shipped for *Raspberry Pis*.

The minimal `Write_RAM` patch against coexistence escalation only covers operating system to chip attacks. It slows down security testing and patch confirmation outside of *Broadcom*. **Over-the-air attackers can still escalate their privileges from Bluetooth to Wi-Fi.**