Bir Kulaklık Ekosisteminin Anatomisi: BLE Stack Analizi, *Cihaz–İstemci İletişim Katmanının Tersine Mühendisliği* ve AWS Altyapısı Üzerinden Firmware Dağıtım Kanallarının Analizi

Giriş:

Bu çalışma, modern bir Bluetooth kulaklık ekosisteminin görünmeyen tarafta nasıl iletişim kurduğunu ele alan bir tersine mühendislik incelemesidir.

Yüzeyden bakıldığında yalnızca ses üreten bir cihaz gibi görünen kulaklıkların, Bluetooth Low Energy protokolü üzerinden nasıl kontrol edildiği; servis düzeyi iletişim, durum telemetrisi ve komut mekanizmaları üzerinden adım adım incelenmektedir.

İnceleme, cihaz–istemci iletişim katmanını merkeze alarak kulaklığın yalnızca bir donanım değil, çok katmanlı bir yazılım ve protokol sistemi olarak nasıl davrandığını ortaya koymayı amaçlamaktadır.

Bu araştırmanın çıkış noktası oldukça basit bir merakla başladı. Günlük olarak kullandığım Bluetooth kulaklığın kontrol uygulamasının üretici tarafından bir süre sonra uygulama marketlerinden kaldırıldığını fark ettim. Uygulama ortadan kalkınca, kulaklık hâlâ ses veriyor, bağlanıyor ve temel işlevlerini yerine getiriyordu. Ancak kulaklığın sunduğu bazı ayarların ve davranışların kontrolünü kaybetmiştim
Bu durum, kulaklığımın uygulama ile nasıl iletişim kurduğunu merak etmeme yol açtı. Basit bir kullanım sorusuyla başlayan bu merak, zamanla kulaklığın Bluetooth Low Energy üzerinden nasıl haberleştiğini, hangi verileri ürettiğini ve hangi mekanizmalarla komut aldığını anlamaya yönelik daha derin bir araştırmaya dönüştü.
Ortaya çıkan çalışma; planlanmış bir hedeften ziyade, bir cihazın “nasıl çalıştığını gerçekten görmek” isteğinin adım adım teknik bir incelemeye evrilmesinin sonucudur.

Sorumluluk Reddi ve Kapsam

Bu çalışma kapsamında gerçekleştirilen tüm analizler, yazarın kendi kullanımında bulunan donanım üzerinde ve gözlemsel/analitik yöntemlerle yürütülmüştür. Araştırma sürecinde herhangi bir sisteme zarar verilmemiş, hizmet kesintisine yol açılmamış veya üçüncü taraflara ait altyapılara müdahalede bulunulmamıştır.

Metin içerisinde yer alan ürün adları, cihaz kimlikleri, servis uç noktaları, adresler ve benzeri teknik bilgiler; çalışmanın anlaşılabilirliğini korumak amacıyla kısmen değiştirilmiş, maskelenmiş veya anonimleştirilmiştir. Bu bilgiler, gerçek sistemlerin birebir temsili olarak değerlendirilmemelidir.

Bu makalenin amacı; bir Bluetooth kulaklık ekosisteminin teknik işleyişini, tersine mühendislik perspektifinden incelemek ve iletişim katmanlarının nasıl çalıştığını ortaya koymaktır. İçerik, güvenlik açığı istismarı, yetkisiz erişim veya zararlı kullanım senaryolarını teşvik etmeyi amaçlamamaktadır.

Bluetooth Low Energy (BLE) Protokol Yığını ve Temel Kavramlar

Bluetooth, kısa menzilli kablosuz haberleşme için tasarlanmış, **2.4 GHz ISM bandı** üzerinde çalışan, düşük güç tüketimli bir dijital iletişim protokol ailesidir. Yüzeyde kullanıcıya “cihaz eşleştirme” ve “veri aktarımı” gibi basit işlevler sunsa da, teknik açıdan Bluetooth; **frekans atlamalı (FHSS)**, **zaman dilimli**, paket tabanlı ve olay temelli bir iletişim yapısına sahip karmaşık bir protokol yığınıdır.

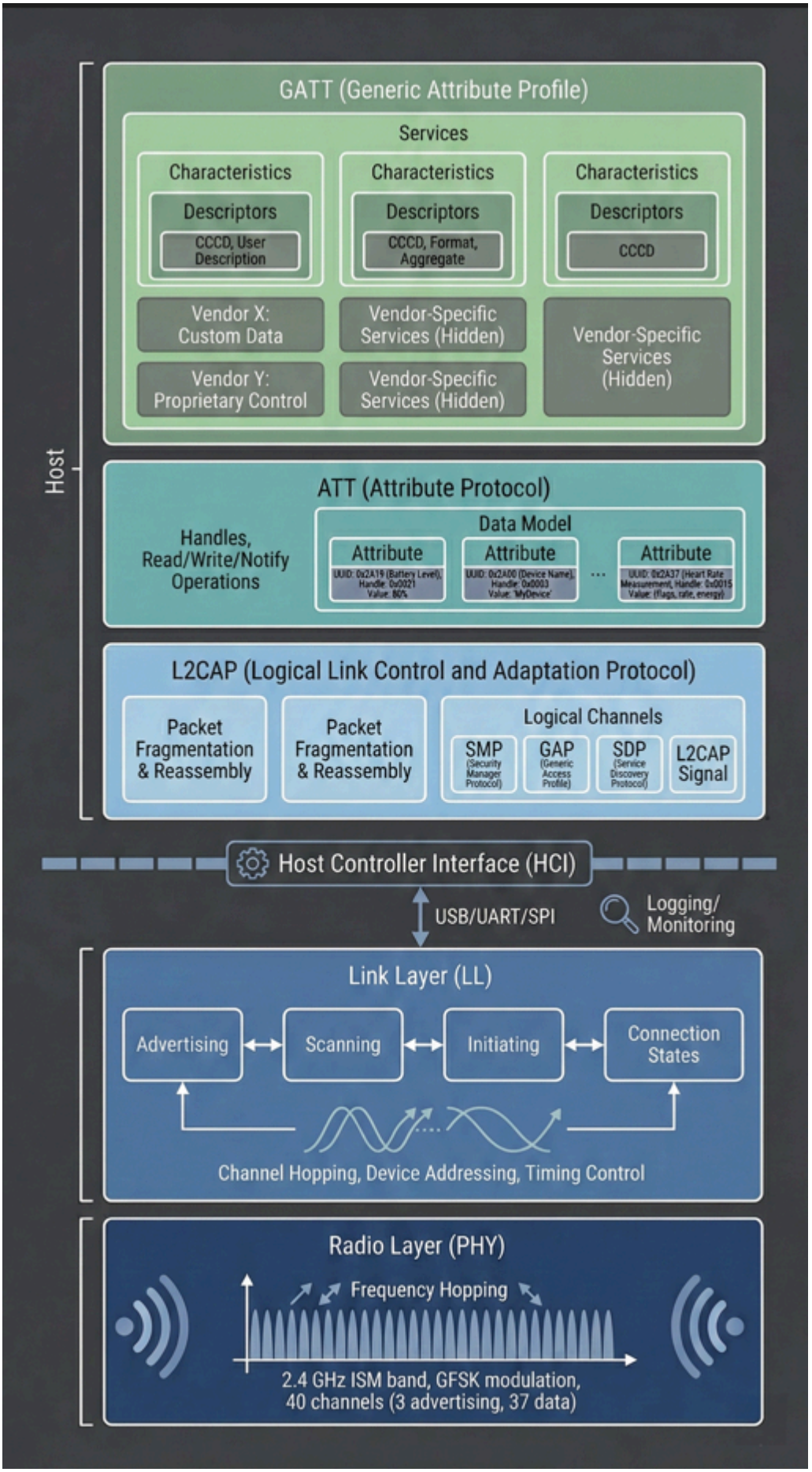
Bluetooth; hava arayüzü üzerinden çalışan, çekirdeği standartlara dayanan ancak üreticiye özgü genişletmelerle şekillenen bir haberleşme sistemidir. İletişimin temel zamanlama ve kanal yönetimi Link Layer’da gerçekleştirilirken, üst katmanlarda servis keşfi, güvenli bağlantı kurulumu ve uygulama verisi taşınır.

Bluetooth ekosisteminde, özellikle BLE tarafında, aktif yayın ve bağlantı temelli iki ana iletişim modeli öne çıkar. Advertising paketleri, cihaz kimliği, servis ipuçları ve üreticiye özel veriler içererek pasif dinleme (passive sniffing) için yüksek değerli bir istihbarat kaynağı oluşturur. Bağlantı kurulduğunda ise iletişim, kanal atlamaları ve zamanlama nedeniyle doğrudan dinlemeye karşı daha dirençlidir; ancak güvenlik açıkları veya vendor hataları hedef alınarak analiz edilebilir.

Tersine mühendislik perspektifinde Bluetooth, **standart dışı GATT karakteristikleri**, **özel komut setleri** ve **şifrelenmiş telemetri** üzerinden cihaz davranışlarının yeniden inşa edilmesine imkân tanır. Özellikle kulaklıklar, IoT cihazları ve giyilebilir teknolojiler; resmi uygulamalar olmaksızın, yalnızca Bluetooth trafiği analiz edilerek kontrol edilebilir veya modellenebilir.

> 💡 Bu yaklaşım, otomotiv dünyasında yaygın olarak bilinen; araçlarda donanımsal olarak mevcut olan ancak yazılım veya konfigürasyon seviyesinde devre dışı bırakılmış özelliklerin, üretici dışı araçlar kullanılarak etkinleştirilmesine benzetilebilir. Bu tür çalışmalar, yeni bir işlev eklemekten ziyade, mevcut sistemin nasıl çalıştığını anlamaya ve var olan davranışları görünür hâle getirmeye odaklanır.

Özetle Bluetooth, modern kablosuz ekosistemde; düşük güç ve kısa menzil hedefiyle tasarlanmış olmasına rağmen, tersine mühendislik ve davranışsal analiz açısından geniş bir inceleme yüzeyi sunan bir haberleşme protokolüdür.


GATT (Generic Attribute Profile)
Services
Characteristics
Descriptors
CCCD, User Description
Characteristics
Descriptors
CCCD, Format, Aggregate
Characteristics
Descriptors
CCCD
Vendor X: Custom Data
Vendor Y: Proprietary Control
Vendor-Specific Services (Hidden)
Vendor-Specific Services (Hidden)
Vendor-Specific Services (Hidden)
Host
ATT (Attribute Protocol)
Handles, Read/Write/Notify Operations
Data Model
Attribute
Attribute
Attribute
L2CAP (Logical Link Control and Adaptation Protocol)
Packet Fragmentation & Reassembly
Packet Fragmentation & Reassembly
Logical Channels
SMP (Security Manager Protocol)
GAP (Generic Access Profile)
SDP (Service Discovery Protocol)
L2CAP Signal
Host Controller Interface (HCI)
USB/UART/SPI
Logging/ Monitoring
Link Layer (LL)
Advertising
Scanning
Initiating
Connection States
Channel Hopping, Device Addressing, Timing Control
Radio Layer (PHY)
Frequency Hopping
2.4 GHz ISM band, GFSK modulation,
40 channels (3 advertising, 37 data)

💡 **Şekil 1.** Bluetooth Low Energy protokol yığınının katmanlı yapısı. Diyagram, radyo seviyesinden GATT profiline kadar uzanan iletişim katmanlarını ve üreticiye özgü servislerin genellikle üst katmanlarda konumlandığını göstermektedir. Tersine mühendislik açısından en fazla etkileşim yüzeyi ATT ve GATT katmanlarında ortaya çıkmaktadır.

Yukarıda tanımlanan BLE protokol katmanları, sahada belirli iletişim ve veri modelleri üzerinden kendini gösterir. Advertising, bağlantı temelli iletişim, attribute tabanlı veri modeli ve üreticiye özgü genişletmeler; bu katmanlı yapının pratikteki yansımalarıdır.

1. Advertising Mekanizması

Cihazlar periyodik olarak advertising paketleri yayınlar.

Bu paketler şunları içerebilir:

- Cihaz adresi
- Servis UUID'leri
- Üreticiye özel (Manufacturer Specific Data)

2. Connection-Oriented İletişim

Bağlantı kurulduktan sonra:

  - Kanal atlamaları senkronize edilir
  - Connection interval ve latency uygulanır
- Şifreleme bu aşamada devreye girer (LE Legacy / LE Secure Connections).
- Zayıf eşleştirme senaryoları (Just Works) analiz için zemin oluşturur.

3. Veri Modeli (Attribute-Centric Design)

- BLE'de "paket" değil **anlamlı veri nesneleri** ön plandadır.
- Her cihaz, kendini bir veri ağacı (GATT database) olarak sunar.
- Bu durum:Bilinmeyen cihazların davranışsal olarak yeniden inşa edilmesini ve Vendor uygulaması olmadan kontrol edilmesini mümkün kılar.

4. Vendor-Specific Genişletmeler

- Standart UUID’ler dışında özel UUID blokları kullanılır.
- ANC, EQ, gesture, firmware update gibi işlevler çoğu zaman belgelenmez.
- Reverse engineering süreci genellikle:
    1. GATT keşfi
    2. Karakteristik erişim denemeleri
    3. Notify/Write davranışlarının korelasyonu
    4. Uygulama ↔ cihaz trafiğinin karşılaştırılması

    adımlarından oluşur.

ANALİZ

Bu örnek senaryoda, kontrol uygulaması üretici tarafından platformlardan kaldırılmış olan ticari bir **TWS Bluetooth kulaklık** üzerinde BLE tabanlı iletişim incelenmektedir. Çalışma kapsamında, cihazın BLE üzerinden nasıl kontrol edildiğini anlamak amacıyla finalde sınırlı işlevlere sahip deneysel bir istemci uygulaması geliştirilmiştir.

Kullanılan yöntemler güvenlik perspektifinden değerlendirildiğinde; bireysel kullanıcı açısından uzun süreli, kritik veya fiziksel bir zarara yol açma potansiyeline sahip değildir. Ancak pasif ve aktif gözlemler üzerinden, kullanım alışkanlıklarına dair çıkarımlar yapılabilmesi ve cihaz davranışlarının zaman içinde izlenebilmesi mümkündür.

Buna karşın, analiz edilen yaklaşımın kapsamı **giyilebilir teknolojiler** ve benzeri cihazlar ölçeğinde ele alındığında; akıllı kilitler, medikal cihazlar ve endüstriyel sistemler gibi alanlarda, hizmet erişilebilirliği ve sosyal mühendislik bağlamında daha anlamlı risk yüzeyleri ortaya çıkabilmektedir.

Analiz Ortamının Seçilmesi

Bu noktada analiz ortamı üzerinde kısaca durmak gerekir. İlk bakışta sanal bir analiz ortamı (ör. REMnux sanal makinesi) kullanmak cazip görünse de, Bluetooth tabanlı düşük seviye analizlerde bu yaklaşım pratikte sınırlıdır. VMware veya VirtualBox gibi sanallaştırma çözümlerinde çalışan bir işletim sistemi, Bluetooth donanımına doğrudan erişemez; iletişim ana işletim sistemi üzerinden soyutlanmış şekilde gerçekleşir.

Bu durum, HCI seviyesinde trafik gözlemi ve BLE davranış analizi gibi çalışmalarda sanal makine kullanımını işlevsiz hâle getirir. Bu nedenle analiz ortamı için iki temel seçenek bulunmaktadır: harici bir Bluetooth USB dongle kullanmak veya donanıma doğrudan erişim sağlayan bir canlı sistem (live USB) üzerinde çalışmak.

Bu çalışmada, analiz sürecine hızlı ve temiz bir başlangıç yapabilmek amacıyla canlı sistem yaklaşımı tercih edilmiştir. Güncel ve kararlı bir Ubuntu sürümü kullanılarak, Bluetooth yığınına doğrudan erişim sağlanan bir analiz ortamı oluşturulmuştur. Canlı ortamın hazırlanmasına ilişkin standart kurulum adımları, çalışmanın odağını dağıtmamak adına burada ele alınmamıştır.

Analiz ortamı hazırlandıktan sonra, Bluetooth denetleyicisinin durumunu doğrulamak için komut satırı incelemesine geçilmiştir.

bluetoothctl show

```
ubuntu@ubuntu:~/Desktop$ bluetoothctl show
Controller C0:A5:E8:XX:XX:XX (public)
Manufacturer: 0x0002 (2)
Version: 0x0b (11)
Name: ubuntu
Alias: ubuntu
Class: 0x006c010c (7078156)
Powered: yes
Discoverable: no
DiscoverableTimeout: 0x000000b4 (180)
Pairable: no
UUID: A/V Remote Control (0000110e-0000-1000-8000-00805f9b34fb)
UUID: Handsfree Audio Gateway (0000111f-0000-1000-8000-00805f9b34f
b)
UUID: PnP Information (00001200-0000-1000-8000-00805f9b34fb)
UUID: Audio Sink (0000110b-0000-1000-8000-00805f9b34fb)
UUID: Audio Source (0000110a-0000-1000-8000-00805f9b34fb)
UUID: A/V Remote Control Target (0000110c-0000-1000-8000-00805f9b3
4fb)
UUID: Generic Access Profile (00001800-0000-1000-8000-00805f9b34f
b)
```

```
UUID: Generic Attribute Profile (00001801-0000-1000-8000-00805f9b34f
b)
UUID: Device Information (0000180a-0000-1000-8000-00805f9b34fb)
UUID: Handsfree (0000111e-0000-1000-8000-00805f9b34fb)
Modalias: usb:XX:XX
Discovering: no
Roles: central
Roles: peripheral
Advertising Features:
ActiveInstances: 0x00 (0)
SupportedInstances: 0x0c (12)
Supported Includes: tx-power
Supported Includes: appearance
Supported Includes: local-name
SupportedSecondaryChannels: 1M
SupportedSecondaryChannels: 2M
SupportedSecondaryChannels: Coded
```

Şimdi hedef kulaklığı eşleştirme moduna (pairing mode) alıyoruz. Bu adımın uygulaması cihazdan cihaza farklılık gösterebilir; amaç cihazın **eşleştirilebilir** ve **keşfedilebilir** durumda olmasıdır.

Ardından Bluetooth yönetim oturumunu başlatmak için `bluetoothctl` çalıştırılır:

`bluetoothctl`

Araç başarıyla açıldığında aşağıdaki komut istemi görülür:

`[bluetooth]#`

Bu noktadan sonra pairing modunda olan kulaklığı tarayarak BLE kimliğini (LE kimliği/MAC) tespit edeceğiz.

şimdi Pairing mode açık olan kulaklığımızı tarayalım ve BLE Kimliğini alalım.

`scan on`

```
[bluetooth]# scan on
[bluetooth]# SetDiscoveryFilter success
[bluetooth]# Discovery started
[bluetooth]# [CHG] Controller C0:A5:E8:XX:XX:XX Discovering: yes
[bluetooth]# [NEW] Device D0:65:AE:XX:XX:XX TWS-DEVICE-01 [LE]
[bluetooth]# [NEW] Device 7D:C0:CE:XX:XX:XX 7D-C0-CE-XX-XX-XX
```

```
[bluetooth]# [NEW] Device 9C:0D:AC:XX:XX:XX TWS-DEVICE-01
[bluetooth]# [NEW] Device 78:02:B7:XX:XX:XX SMART-WATCH-LE
```

> 💡 Tarama çıktısında aynı fiziksel cihaza ait birden fazla girişin gözlemlenmesi, cihazın hem Bluetooth Classic hem de Bluetooth Low Energy bağlamında farklı advertising davranışları sergilediğini göstermektedir. Bu durum, özellikle TWS sınıfı kulaklıklarda yaygın olan çift modlu (dual-mode) mimarinin doğal bir sonucudur.

Bu aşamada hedef cihazın Bluetooth Low Energy reklam paketleri pasif olarak gözlemlenmiş ve cihazın BLE üzerinden aktif olarak advertise ettiği doğrulanmıştır. Dual-mode davranış sergileyen ve BLE üzerinden erişilebilir olan bu tür tüketici cihazlarında, uygulama-tabanlı kontrol ve durum telemetrisi işlevlerinin çoğunlukla BLE/GATT üzerinden sağlandığı bilinmektedir. Bu nedenle elde edilen gözlem, söz konusu işlevlerin varlığına dair güçlü bir ön kanıt sunmaktadır.

Şimdi hedef cihaza bağlanalım:

`connect D0:65:AE:XX:XX:XX`

Bağlantı sırasında bir eşleştirme isteği (pairing request) gelirse **yes** ile onaylayarak devam ediyoruz. Ardından cihazı güvenilir olarak işaretliyoruz:

`trust D0:65:AE:XX:XX:XX`

Cihaz durumunu ve sunulan servisleri görüntülemek için:

`info D0:65:AE:XX:XX:XX`

Örnek çıktı:

```
Device D0:65:AE:XX:XX:XX (random)
    Name: [TWS-DEVICE] [LE]
    Alias: [TWS-DEVICE] [LE]
    Appearance: 0x03c0 (960)
    Paired: yes
    Bonded: yes
    Trusted: yes
    Blocked: no
    Connected: yes
```

```
    LegacyPairing: no
    UUID: Generic Access Profile    (00001800-0000-1000-8000-00805f9b
34fb)
    UUID: Generic Attribute Profile (00001801-0000-1000-8000-00805f9b3
4fb)
    UUID: Device Information        (0000180a-0000-1000-8000-00805f9b34
fb)
    UUID: Battery Service           (0000180f-0000-1000-8000-00805f9b34f
b)
    UUID: Unknown                   (0000aa00-0000-1000-8000-00805f9b34f
b)
    UUID: Google                    (0000fe2c-0000-1000-8000-00805f9b34fb)
    UUID: Vendor specific           (XXXXXXXXXXXXXXXX-XXXX-XXXX-XXXX
-XXXXXXXXXXXX)
    ManufacturerData.Key: 0xXXXX
    ManufacturerData.Value:
      XX XX XX XX XX XX XX XX XX
    Battery Percentage: 0x5a (90)
```

Bu noktada `Connected: yes` ve BLE servis UUID'lerinin listelenmesi, hedef cihazla BLE bağlantısının kurulduğunu ve GATT üzerinden servis keşfi yapılabildiğini göstermektedir. **Dolayısıyla cihaz artık GATT servislerini tam olarak açabilir.**

Bir sonraki adımda, GATT menüsüne geçerek servis ve karakteristik envanterini çıkaracağız.

1) Battery Service (Standart BLE Servisi)

```
UUID: Battery Service (0000180f-0000-1000-8000-00805f9b34fb)
Battery Percentage: 0x5a (90)
```

Bu servis, cihazın **Bluetooth SIG tarafından tanımlanmış standart Battery Service**'i kullandığını göstermektedir. Batarya yüzdesinin doğrudan okunabilir olması, ilgili telemetri bilgisinin **şifrelenmemiş ve pasif biçimde erişilebilir** olduğunu ortaya koymaktadır.

2) Vendor-Specific Servisler (Asıl Analiz Yüzeyi)

```
UUID: Unknown        (0000aa00-0000-1000-8000-00805f9b34fb)
UUID: Google         (0000fe2c-0000-1000-8000-00805f9b34fb)
UUID: Vendor specific(5052494d-2dab-0341-6972-6f6861424c45)
```

Bu servisler, cihazın **standart BLE profillerinin ötesinde**, üreticiye özgü kontrol ve telemetri mekanizmaları barındırdığını göstermektedir.

- **0000aa00**

  Bluetooth ekosisteminde sıkça, üreticiye özgü **kontrol, durum ve telemetri verilerinin toplandığı merkezi bir servis** olarak kullanılmaktadır. ANC durumu, mod geçişleri ve sensör çıktıları gibi veriler çoğu zaman bu servis altında taşınır.

- **0000fe2c (Google)**

  Google tarafından tanımlanmış bu UUID, cihazın ***Fast Pair*** ve Android ekosistemi ile entegrasyon yeteneklerine sahip olabileceğine işaret etmektedir. Bu servis, eşleştirme kolaylığı ve kullanıcı deneyimi odaklı yardımcı veriler taşımak amacıyla kullanılmaktadır.

- **5052494d-...**

  Standart BLE UUID bloklarının dışında kalan bu servis, açık biçimde **vendor-proprietary** bir yapıyı temsil etmektedir. UUID'nin ASCII karşılığı incelendiğinde üreticiye özgü bir imza barındırdığı görülmekte olup, cihazın özgün kontrol protokolünün bu servis üzerinden çalıştığı güçlü biçimde değerlendirilmektedir.

Bu servisler birlikte ele alındığında; **ANC kontrolü, çalışma modları, kulaklık takılı algılama (wear detection) ve şarj kutusu durumu** gibi işlevlerin büyük olasılıkla bu katman üzerinden sağlandığı sonucuna varılmaktadır.

---

3) Manufacturer Specific Data

```
ManufacturerData.Key: 0x065a
ManufacturerData.Value: 06 00 9c 0d ac 09 a1 29 01
```

Manufacturer Specific Data alanı, üreticiye özgü verilerin BLE advertising ve bağlantı aşamalarında taşınmasına imkân tanır. Bu veri bloğu, cihazın **BLE kimliği ile Classic Bluetooth tarafındaki adresleme ve durum bilgilerinin ilişkilendirilebildiğini** göstermektedir.

Bu tür veriler, dual-mode çalışan cihazlarda BLE ve Classic kanalların birlikte nasıl koordine edildiğini ortaya koymak açısından değerlidir. Ayrıca bu alan, cihazların **davranışsal olarak ayırt edilmesi ve sınıflandırılması** için kullanılabilecek bir yüzey sunmaktadır.

GATT envanteri: servis ve karakteristiklerin çıkarılması

Bluetooth LE bağlantısı kurulduktan sonra cihazın GATT veritabanını envanterlemek için `bluetoothctl` üzerinde GATT menüsüne geçilir ve tüm attribute'lar listelenir:

`menu gatt`
`list-attributes`

`list-attributes` çıktısı çok uzun olduğu için, aşağıda yalnızca analizin belkemiğini oluşturan servislerin **temsili kesiti** verilmiştir. (Tam çıktı ek bölümde/supplementary materyalde saklanmıştır.)

GATT Haritası

```
Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service004d
    5052494d-2dab-0341-6972-6f6861424c45
    Vendor specific

Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service0044
    0000fe2c-0000-1000-8000-00805f9b34fb
    Google

Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service001f
    0000aa00-0000-1000-8000-00805f9b34fb
    Unknown

Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service001a
    0000180f-0000-1000-8000-00805f9b34fb
    Battery Service
```

```
Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service0015
    0000180f-0000-1000-8000-00805f9b34fb
    Battery Service

Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service000a
    0000180a-0000-1000-8000-00805f9b34fb
    Device Information

Primary Service
    /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service0001
    00001801-0000-1000-8000-00805f9b34fb
    Generic Attribute Profile
```

Bu harita, cihazın standart BLE servislerinin yanında üç ayrı vendor spesific servis sunduğunu gösterir: **0000aa00**, **0000fe2c** ve **5052494d-...**. Devam eden adımlarda analiz odağı bu servisler altında yer alan karakteristiklerin davranışsal sınıflandırılmasıdır (event/state/command).

Battery Service (0000180f) — iki ayrı instance

Çıktıda Battery Service'in **iki farklı instance** olarak göründüğü dikkat çekmektedir:

- `service001a/char001b` → **Battery Level (0x2A19)**
- `service0015/char0016` → **Battery Level (0x2A19)**

Bu durum genellikle cihazın birden fazla pil kaynağını (ör. **sol/sağ kulaklık** veya **kulaklık + şarj kutusu**) ayrı ayrı raporladığına işaret eder.

Device Information (0000180a)

Device Information servisi altında donanım/firmware/model/seri gibi alanlar yer alır ve çoğu zaman **salt-okunur** yapıdadır. Bu bölüm, davranış analizi için referans oluşturur ancak kontrol yüzeyi değildir.

Generic Attribute Profile (00001801)

GATT altyapı servisidir. `Service Changed (0x2A05)` gibi karakteristikler burada bulunur ve BLE/GATT değişimlerini bildirim mekanizmasıyla taşımaya yarar; uygulama-

kontrol protokolünün kendisi değildir.

Vendor Specific analiz yüzeyi

Harita cihazın kontrol ve telemetri mantığının ağırlıklı olarak aşağıdaki servisler altında konumlandığını göstermektedir:

- **0000aa00** (Unknown) → üreticiye özgü kontrol/telemetri yüzeyi adayı
- **0000fe2c** (Google) → Fast Pair/Android entegrasyonu
- **5052494d-...** → vendor-proprietary protokol yüzeyi

Bir sonraki adımda, bu servislerin altındaki karakteristikler **notify/read/write** davranışlarına göre sınıflandırılacak ve hangi endpoint'in "event", hangisinin "state", hangisinin "command" rolü oynadığı deneysel olarak çıkarılacaktır.

Batarya Telemetrisi — Notify Mekanizmasının Doğrulanması

Bu aşamada, cihazın batarya telemetrisini BLE üzerinden **olay temelli (event-driven)** olarak iletip iletmediğini test ediyoruz. Bunun için standart **Battery Service** altında yer alan *Battery Level* karakteristiği dinlenmektedir.

`select-attribute /org/bluez/hci0/dev_D0_65_xx_xx_xx_xx/service0015/char0016`

Karakteristik seçildikten sonra bildirim (notify) mekanizması etkinleştirilir:

```
[DEVICE-LE]# notify on
[CHG] Attribute ... Notifying: yes
Notify started
```

Notify aktif hale geldikten sonra karakteristikten aşağıdaki değerler alınmıştır:

```
[CHG] Attribute ... Value:
  64
[CHG] Attribute ... Value:
  64
```

Gelen Verinin Yorumlanması

Gelen değerler **raw byte** olarak gösterilmektedir. Bu değerler ASCII karşılıklarına denk düşebilse de, **anlamsal olarak sayısal telemetri** ifade

etmektedir.

Hex → Decimal dönüşümü:

- `0x64` → 100
- `0x64` → 100

Neden Bu Önemli?

Batarya servisi, **en güvenli ve standart test yüzeyi** olduğu için seçilmiştir. Burada notify mekanizmasının çalıştığının doğrulanması, bir sonraki adımda:

- ANC durumu
- mod değişimleri
- sensör olayları

gibi **vendor-specific telemetri kanallarının** da benzer şekilde dinlenebileceğine işaret eder.

Vendor-Specific Telemetri — ANC Olay Akışının Gözlemlenmesi

Bu aşamada, daha önce envanteri çıkarılan vendor-specific servis altındaki bir karakteristik dinlenmiştir:

`select-attribute /org/bluez/hci0/dev_D0_65_AE_XX_XX_XX/service001f/char0037`
`notify on`

Notify mekanizması etkinleştirildikten sonra, kulaklığın **Active Noise Cancelling (ANC)** modu değiştirilirken aşağıdaki değerler gözlemlenmiştir:

```
[CHG] Attribute ... Value:
  02
[CHG] Attribute ... Value:
  00
[CHG] Attribute ... Value:
  01
```

İlgili karakteristik:

- `char0037`

- UUID: `0000aa20`
- Vendor-specific

Bu gözlem, söz konusu karakteristiğin **olay temelli (event-driven) bir telemetri hattı** olarak kullanıldığını göstermektedir. ANC ile etkileşim sırasında, cihazın BLE üzerinden **tek baytlık durum veya olay kodları** yayınladığı açıkça görülmektedir.

Gözlemlenen Veri Modeli

ANC etkileşimi sırasında elde edilen değerler:

- `0x02`
- `0x00`
- `0x01`

Her değer, ANC'nin mevcut durumunu temsil eder:

- `00` → Kapalı (Off)
- `01` → Açık (On)
- `02` → Alternatif mod (ör. ANC / Ambient / Transparency)

Bu yaklaşımda karakteristik, cihazın **anlık durumunu** periyodik veya olay temelli olarak bildirir.
Bu aşamada elimizde, ilk bir **davranış eşlemesi (initial mapping)** yapmak için yeterli veri bulunmaktadır. Ancak bu eşlemenin **kesinleştirilmesi** için, kontrollü ve tekrarlanabilir bir test senaryosu gereklidir.

💡 Bu aşamada elimizde, ilk bir **davranış eşlemesi (initial mapping)** yapmak için yeterli veri bulunmaktadır. Ancak bu eşlemenin **kesinleştirilmesi** için, kontrollü ve tekrarlanabilir bir test senaryosu gereklidir.

Diğer Servislerin İncelenmesi

GATT envanterinde yer alan diğer servis ve karakteristikler de benzer yöntemlerle incelenmiş; okuma (read), yazma (write) ve bildirim (notify) davranışları test edilmiştir. Ancak bu servisler altında elde edilen verilerin,

cihazın kontrol mantığı veya durum telemetrisi açısından **ayırt edici ya da yeni bir bilgi sunmadığı** gözlemlenmiştir.

Bu karakteristiklerin büyük bölümü ya **standart BLE profillerine** ait olup salt-okunur bilgilendirme amacı taşımakta, ya da vendor-specific olmakla birlikte **statik, düşük frekanslı veya anlamlı bir olay akışı üretmeyen** veri yapıları içermektedir. Dolayısıyla, analiz kapsamında bu servisler detaylandırılmamış ve çalışmanın odağı, cihaz davranışlarını doğrudan yansıtan aktif telemetri kanalları üzerinde tutulmuştur.

Bu yaklaşım, elde edilen bulguların **tekrarlanabilirliği ve analitik değeri** yüksek olan yüzeylere odaklanmayı amaçlamaktadır.

Trafik analizi

Classic Bluetooth (BR/EDR) trafiği, Wireshark ve `btmon` araçları kullanılarak hem bağlantı kurulumu hem de aktif kullanım senaryoları sırasında gözlemlenmiştir. Elde edilen veriler incelendiğinde, ses iletimi ve profil yönetimi (A2DP/HFP/AVRCP) dışında, **ANC kontrolü veya cihaz telemetrisiyle ilişkili herhangi bir uygulama-seviyesi veri taşınmadığı** doğrulanmıştır. Bu durum, kontrol ve durum bilgisinin Classic kanal yerine Bluetooth Low Energy üzerinden yürütüldüğünü teyit etmektedir.

Analiz sürecinde ayrıca, mobil istemci tarafında Bluetooth logları alınmış; Android ekosistemine özgü **Fast Pair** mekanizması ve ilişkili servisler incelenmiştir. Fast Pair kapsamında yayınlanan advertising paketleri, pairing akışı ve cihaz–istemci etkileşimi değerlendirilmiş; bilinen zafiyet senaryoları (ör. WhisperPair) açısından deneysel testler gerçekleştirilmiştir. Ancak bu incelemeler sonucunda, çalışmanın odağını değiştirecek ölçekte **kritik veya exploit edilebilir bir bulguya** rastlanmamıştır.

Bu nedenle çalışma, Classic Bluetooth trafiği veya Fast Pair mekanizmasının derinlemesine araştırılması yerine, **cihaz kontrol mantığının fiilen konumlandığı BLE katmanına** ve kontrol uygulamasının tersine mühendisliğine odaklanacak şekilde ilerletilmiştir.

Vendor Mobil Uygulamasının Tersine Mühendisliği

Bu noktadan itibaren analiz, cihaz tarafındaki gözlemleri doğrulamak ve BLE üzerindeki olay/komut akışını uygulama mantığıyla ilişkilendirmek amacıyla **vendor mobil uygulamasının** tersine mühendisliğine genişletilmektedir.

Çalışmada referans olarak kullanılan **v2.1.0** sürümlü APK dosyası, statik analiz için `JADX` ortamına aktarılmış ve uygulama içindeki BLE etkileşimleri (servis/karakteristik erişimleri, payload formatları ve olay işleme mantığı) incelenmeye başlanmıştır.

Bu bölümün amacı; cihaz üzerinde gözlemlenen event akışını, uygulama tarafındaki kod yolu ve veri formatlarıyla **kanıta dayalı** biçimde eşleştirmek; böylece

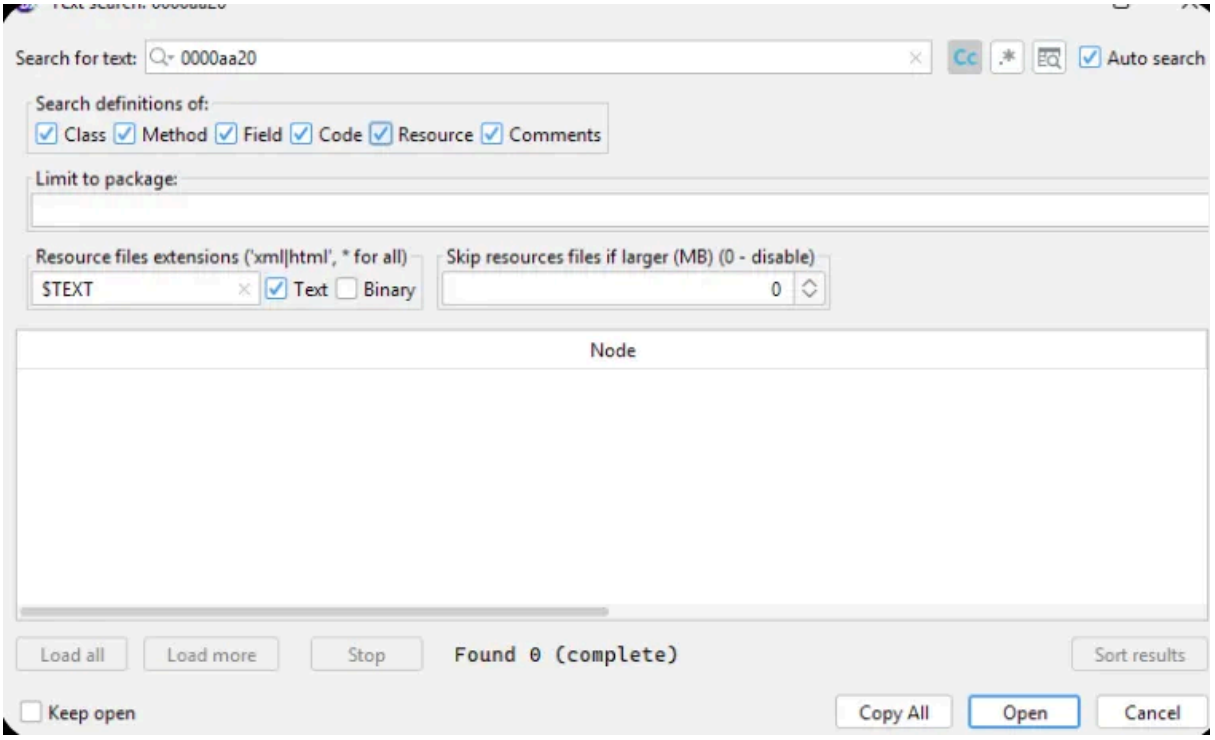
"gözlem → yorum" çizgisini "gözlem → uygulama doğrulaması" seviyesine taşımaktır.

`Bu bölümde yer alan paket adları ve tanımlayıcılar, çalışmanın teknik bütünlüğünü korumak amacıyla kısmen anonimleştirilmiştir.`

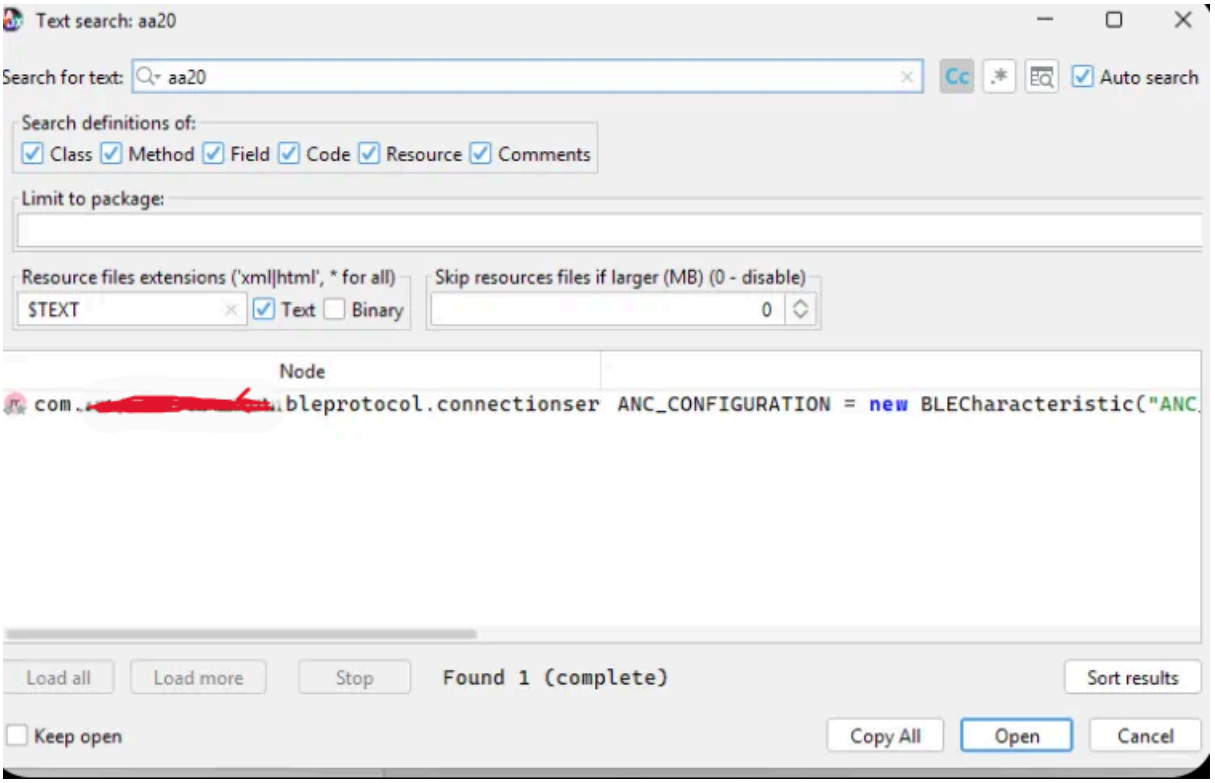
JADX ortamını açıp APK'yı içe aktardıktan sonra, daha önce cihaz üzerinde gözlemlediğimiz **ANC ile ilişkili characteristic UUID**'sini kod içinde arayarak başlangıç noktasını belirleyeceğiz. Bunun için global aramayı açıyoruz:

`Ctrl + Shift + F`

Arama kutusuna, sahada yakaladığımız UUID'yi (ör. `0000aa20` veya tam 128-bit formu) girerek uygulama içerisinde geçtiği sınıf ve metotları tespit ediyoruz.



Direkt tam UUID aratmak yerine aa20 şeklinde arama yaparak daha net sonuçlar edilebilir.



`0000aa20` UUID'sinin geçtiği yerde `ANC_CONFIGURATION` ve `BLE` anahtar kelimelerinin görülmesi, bu karakteristiğin uygulama tarafında ANC konfigürasyonu ile ilişkilendirildiğini doğrulamaktadır. Bu bulgu, cihaz gözlemleri ile APK kod yolu arasında doğrudan bir bağ kurmamızı sağlar.

Bu aşamada, `com.[XXXXX].bleprotocol` paketi altındaki ilgili class üzerinden BLE kontrol akışı analiz edilmektedir.

BLECharacteristic

```
    static {
        UUID r12 = null;
        int r13 = 2;
        u uVar = null;
        MANUFACTURER_NAME = new BLECharacteristic("MANUFACTURER_NAME", 1, a.c("2a29"), r12, r13, uVar);
        UUID r52 = null;
        int r62 = 2;
        u uVar2 = null;
        MODEL_NAME = new BLECharacteristic("MODEL_NAME", 2, a.c("2a24"), r52, r62, uVar2);
        SERIAL_NUMBER = new BLECharacteristic("SERIAL_NUMBER", 3, a.c("2a25"), r12, r13, uVar);
        FIRMWARE_REVISION = new BLECharacteristic("FIRMWARE_REVISION", 4, a.c("2a26"), r52, r62, uVar2);
        HARDWARE_REVISION = new BLECharacteristic("HARDWARE_REVISION", 5, a.c("2a27"), r12, r13, uVar);
        BATTERY_LEVEL = new BLECharacteristic("BATTERY_LEVEL", 6, a.c("2a19"), r52, r62, uVar2);
        a.Companion c0543a = k7.a.INSTANCE;
        RIGHT_EARBUD_BATTERY_LEVEL = new BLECharacteristic("RIGHT_EARBUD_BATTERY_LEVEL", 7, c0543a.c(), r12, r13, uVar);
        LEFT_EARBUD_BATTERY_LEVEL = new BLECharacteristic("LEFT_EARBUD_BATTERY_LEVEL", 8, c0543a.a(), null, 2, null);
        MAIN_CASE_BATTERY_LEVEL = new BLECharacteristic("MAIN_CASE_BATTERY_LEVEL", 9, c0543a.b(), null, 2, null);
        BLUETOOTH_CLASSIC_PAIRING_MODE = new BLECharacteristic("BLUETOOTH_CLASSIC_PAIRING_MODE", 10, a.d("0001"), a.c("aa02"));
        RENAME = new BLECharacteristic("RENAME", 11, a.d("0003"), a.c("aa04"));
        AUTO_POWER_OFF = new BLECharacteristic("AUTO_POWER_OFF", 12, a.d("0004"), a.c("aa05"));
        VOLUME = new BLECharacteristic("VOLUME", 13, a.d("0007"), a.c("aa08"));
        AUDIO_CONTROL = new BLECharacteristic("AUDIO_CONTROL", 14, a.d("0009"), a.c("aa10"));
        AUDIO_NOW_PLAYING = new BLECharacteristic("AUDIO_NOW_PLAYING", 15, a.d("000a"), a.c("aa11"));
        UI_SOUNDS = new BLECharacteristic("UI_SOUNDS", 16, a.d("000b"), a.c("aa12"));
        ACTION_BUTTON_EVENT = new BLECharacteristic("ACTION_BUTTON_EVENT", 17, a.d("000c"), a.c("aa13"));
        ACTION_BUTTON_CONFIGURATION = new BLECharacteristic("ACTION_BUTTON_CONFIGURATION", 18, a.d("000d"), a.c("aa14"));
        SPEED_MODE = new BLECharacteristic("SPEED_MODE", 19, a.d("000e"), a.c("aa15"));
        GRAPHICAL_EQUALIZER = new BLECharacteristic("GRAPHICAL_EQUALIZER", 20, a.d("000f"), a.c("aa16"));
        ANC_CONFIGURATION = new BLECharacteristic("ANC_CONFIGURATION", 21, a.d("0013"), a.c("aa20"));
        TOUCH_LOCK = new BLECharacteristic("TOUCH_LOCK", 22, a.d("0014"), a.c("aa21"));
        EQUALIZER_SETTINGS = new BLECharacteristic("EQUALIZER_SETTINGS", 23, a.d("0017"), a.c("aa25"));
        EQUALIZER_SETTINGS_CUSTOM_PRESET = new BLECharacteristic("EQUALIZER_SETTINGS_CUSTOM_PRESET", 24, a.d("0018"), a.c("aa26"));
        ANC_TRANSPARENCY_VALUE = new BLECharacteristic("ANC_TRANSPARENCY_VALUE", 25, a.d("0019"), a.c("aa27"));
        ANC_NOISE_CANCELLING_VALUE = new BLECharacteristic("ANC_NOISE_CANCELLING_VALUE", 26, a.d("001a"), a.c("aa28"));
        AUDIO_SOURCE = new BLECharacteristic("AUDIO_SOURCE", 27, a.d("001b"), a.c("aa29"));
        PARTY_MODE = new BLECharacteristic("PARTY_MODE", 28, a.d("001c"), a.c("aa2a"));
        ECO_CHARGING = new BLECharacteristic("ECO_CHARGING", 29, a.d("001d"), a.c("aa2b"));
        ROOM_PLACEMENT = new BLECharacteristic("ROOM_PLACEMENT", 30, a.d("001e"), a.c("aa2c"));
        NIGHT_MODE = new BLECharacteristic("NIGHT_MODE", 31, a.d("001f"), a.c("aa2d"));
        SOLAR_DATA = new BLECharacteristic("SOLAR_DATA", 32, a.d("0022"), a.c("aa30"));
        HISTORIC_SOLAR_DATA = new BLECharacteristic("HISTORIC_SOLAR_DATA", 33, a.d("0023"), a.c("aa31"));
        TONE_CONTROL = new BLECharacteristic("TONE_CONTROL", 34, a.d("0025"), a.c("aa33"));
        CHARGING_CABLE_STATUS = new BLECharacteristic("CHARGING_CABLE_STATUS", 35, a.d("0026"), a.c("aa37"));
        WEAR_SENSOR_STATUS = new BLECharacteristic("WEAR_SENSOR_STATUS", 36, a.d("0027"), null);
        WEAR_SENSOR_ACTION = new BLECharacteristic("WEAR_SENSOR_ACTION", 37, a.d("0028"), null);
        $VALUES = $values();
        INSTANCE = new Companion(null);
    }

    private BLECharacteristic(String str, int r22, UUID r32, UUID r42) {
        this.uuid = r32;
        this.legacyUuid = r42;
        this.supportedUuids = CollectionsKt__CollectionsKt.O(r32, r42);
    }

```

Görülebileceği üzere, uygulama içindeki `BLECharacteristic` tanımları ile GATT envanterinin birleştirilmesi sonucunda, uzun süredir tespit etmeye çalıştığımız **GATT haritası** anlamlı bir şekilde ortaya çıkmaktadır. Bu noktadan sonra hedef, bu karakteristikleri davranışlarına göre sınıflandırmak ve hangi işlevleri temsil ettiklerini kanıta dayalı biçimde notlandırmaktır.

Bu servisler Bluetooth SIG tarafından standartlaştırılmıştır ve tipik bir BLE istemcisi tarafından kolayca keşfedilebilir.

Servis	UUID	Örnek Karakteristikler	İşlev
Device Information	0x180A	0x2A29 / 0x2A24 / 0x2A25 / 0x2A26	Üretici, model, seri/firmware bilgileri (salt-okunur)
Battery Service	0x180F	0x2A19	Batarya seviyesi (%)
Generic Access	0x1800	0x2A00	Cihazın görünen adı2) Kritik Kontrol Servisi: 0xAA00 (Vendor-Specific)

Diğer Özel Servisler

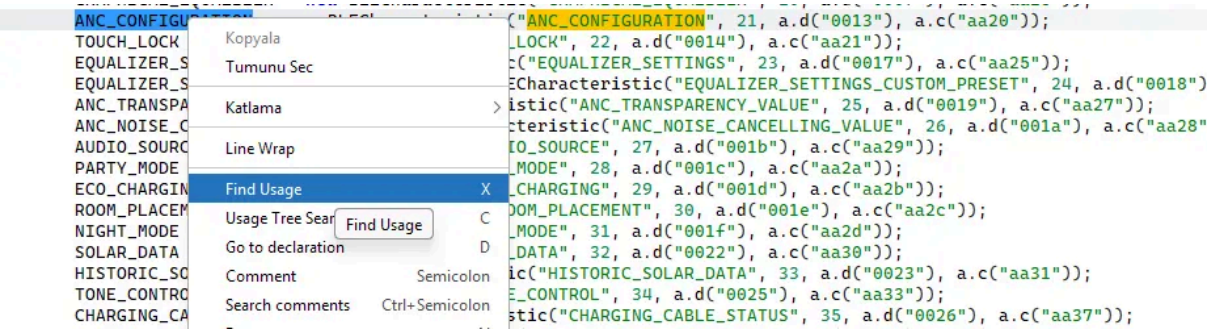
- **Google Fast Pair (`0xFE2C`)**: Android cihazlarla hızlı eşleşme akışını destekleyen servis ailesidir. Bu servis altındaki bazı karakteristikler, standart istemcilerle doğrudan okunmaya çalışıldığında `NotPermitted` hatası döndürebilir; bu durum Fast Pair akışının özel bir erişim modeli veya oturum bağlamı gerektirdiğini düşündürür.
- **Airoha Proprietary (`5052494d...`)**: Vendor-proprietary bir servis olup, üreticiye özgü düşük seviyeli yönetim işlevleri için kullanılan bir protokol yüzeyi olabilir. Bu çalışmada doğrudan kontrol/telemetri hattı olarak değil, ekosistemi tamamlayan ikincil bir katman olarak değerlendirilmiştir.

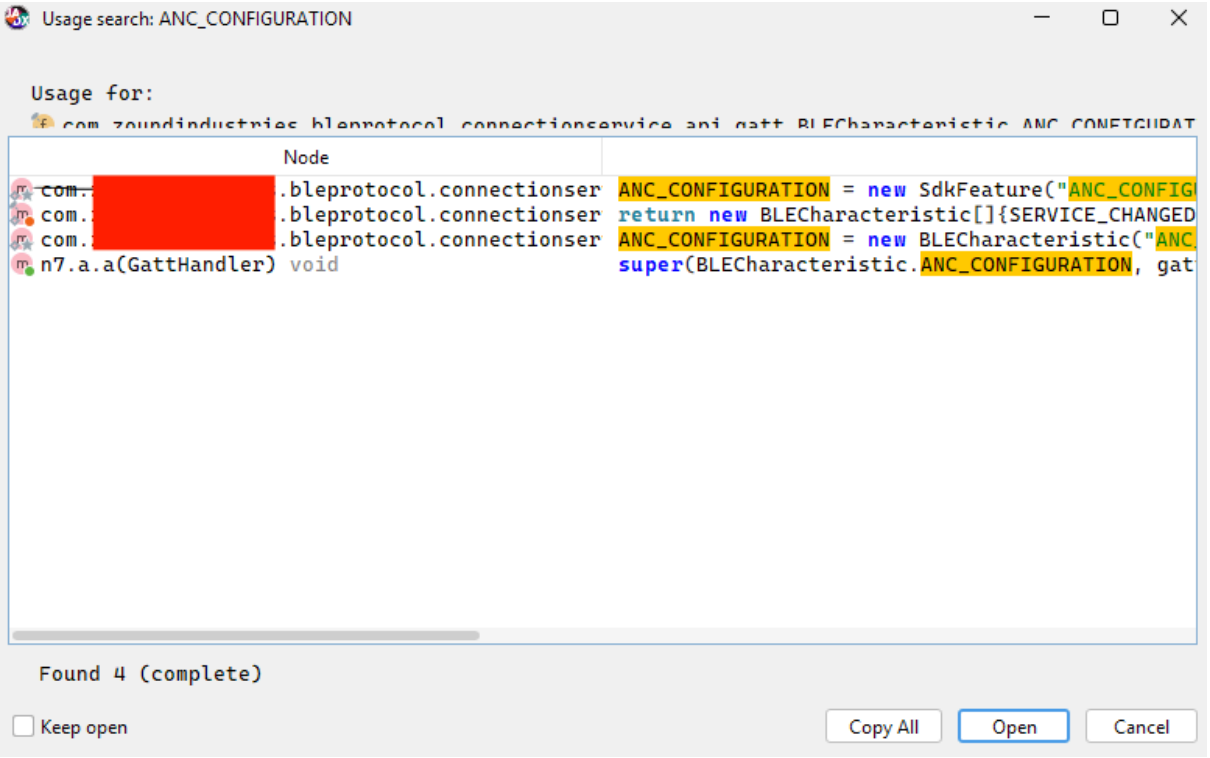
Mimari Özet ve Uygulama Geliştirme Notu

Elde edilen bulgular, cihazın genel olarak **state-driven** (durum odaklı) bir kontrol modeli benimsediğini göstermektedir:

- **Girdi (Input):** Tek tıklama (*single tap*) gibi bazı kullanıcı etkileşimleri cihaz firmware'i içerisinde yorumlanır ve BLE tarafında her zaman ayrı bir "olay" olarak görünmeyebilir.
- **Kontrol:** ANC gibi bazı işlevlerin, belirli bir karakteristiğe yazma yapılarak (örn. `aa20`) değiştirilebildiği gözlemlenmiştir. Ancak bu kontrolün hangi değer aralıklarıyla ve hangi payload formatıyla çalıştığının kesinleştirilmesi, kontrollü test ve uygulama analizi ile doğrulanmalıdır.
- **Olay (Event):** Uzun basma gibi etkileşimler, notify tabanlı bir event kanalında (örn. `aa13`) tek bayt veya kısa frame'ler halinde gözlemlenebilir. Bu event kodlarının anlamı, tekrar eden testlerle ve uygulama tarafındaki işleme mantığıyla eşleştirilerek netleştirilecektir.

Daha önce doğrulanan **ANC konfigürasyon UUID'si** üzerinden `JADX` içinde **Find Usage** yaparak, bu karakteristiğin uygulama tarafındaki kullanım noktalarını incelemeye başlıyoruz.





Uygulama kodunda sınıf adları seviyesinde belirgin bir **obfuscation** (ad gizleme) gözlemlenmektedir. Bu tür isimlendirme kalıpları (`a.a.a`, `n7.a.a` vb.) genellikle R8/DexGuard benzeri araçlarla yapılan küçültme/obfuscation süreçlerinin bir sonucudur. Kullanılan paketleyici/koruma katmanına dair göstergeler, `Detect It Easy (DIE)` gibi araçlarla yapılan statik incelemeyle desteklenebilir.

Bu durum analizi zorlaştırsa da, **yapısal olarak bir engel değildir**; çünkü BLE ile ilgili sınıflar çoğu zaman sabit stringler (UUID'ler), alan adları veya çağrı

zincirleri üzerinden izlenebilir. Bu nedenle bir sonraki adımda, `n7.a.a` adlı sınıf üzerinden ilerleyerek ilgili BLE çağrı akışını takip edeceğiz.

```
package n7;

import com.google.firebase.messaging.e;
import com.vendor.bleprotocol.connectionservice.api.gatt.BLECharacteristi
c;
import com.vendor.bleprotocol.connectionservice.api.gatt.GattHandler;
import com.vendor.bleprotocol.connectionservice.api.gatt.wrappers.vendo
r.anc.AncConfigData;
import kotlin.Metadata;
import kotlin.jvm.internal.f0;
import org.jetbrains.annotations.NotNull;

@Metadata(
   bv = {},
   d1 = {"\u0000\u001c\n\u0002\u0018\u0002\n\u0002\u0018\u0002\n\u0
002\u0018\u0002\n\u0002\u0010\u0012\n\u0002\b\u0003\n\u0002\u001
8\u0002\n\u0002\b\u0004\u0018\u00002\b\u0012\u0004\u0012\u00020
\u00020\u0001B\u000f\u0012\u0006\u0010\b\u001a\u00020\u0007¢\u00
06\u0004\b\t\u0010\nJ\u0010\u0010\u0005\u001a\u00020\u00022\u000
6\u0010\u0004\u001a\u00020\u0003H\u0016J\u0010\u0010\u0006\u001
a\u00020\u00032\u0006\u0010\u0004\u001a\u00020\u0002H\u0016¨\u0
006\u000b"},
   d2 = {
      "Ln7/a;",
      "Lcom/vendor/bleprotocol/connectionservice/api/gatt/wrappers/b;",
      "Lcom/vendor/bleprotocol/connectionservice/api/gatt/wrappers/vend
or/anc/AncConfigData;",
      "",
      e.f.a.f32040c0,
      "v",
      "w",
      "Lcom/vendor/bleprotocol/connectionservice/api/gatt/GattHandler;",
      "gattHandler",
      "<init>",
      "(Lcom/vendor/bleprotocol/connectionservice/api/gatt/GattHandler;)
```

```
V",
        "bt-sdk_release"
    },
    k = 1,
    mv = {1, 7, 1}
)
/* loaded from: classes3.dex */
public final class a extends com.vendor.bleprotocol.connectionservice.api.
gatt.wrappers.b<AncConfigData> {

    /* JADX WARN: 'super' call moved to the top of the method (can break c
ode semantics) */
    public a(@NotNull GattHandler gattHandler) {
        super(BLECharacteristic.ANC_CONFIGURATION, gattHandler, false, fal
se, 0L, 28, null);
        f0.p(gattHandler, "gattHandler");
    }

    @Override // com.vendor.bleprotocol.connectionservice.api.gatt.wrappe
rs.b
    @NotNull
    /* renamed from: v, reason: merged with bridge method [inline-methods]
*/
    public AncConfigData r(@NotNull byte[] data) {
        f0.p(data, "data");
        if ((data.length == 0) || data.length > 1) {
            return AncConfigData.PLAYBACK_ONLY;
        }
        AncConfigData ancConfigDataA = AncConfigData.INSTANCE.a(data
[0]);
        return ancConfigDataA == null ? AncConfigData.PLAYBACK_ONLY : an
cConfigDataA;
    }

    @Override // com.vendor.bleprotocol.connectionservice.api.gatt.wrappe
rs.b
    @NotNull
    /* renamed from: w, reason: merged with bridge method [inline-method
```

```
s] */
    public byte[] s(@NotNull AncConfigData data) {
        f0.p(data, "data");
        return new byte[]{(byte) data.getValue()};
    }
}
```

Cihazdan Uygulamaya: ANC Durumunun Çözülmesi (Read / Notify Yolu)

Cihaz tarafından uygulamaya gönderilen veri, aşağıdaki mantıkla işlenmektedir:

- BLE karakteristiğinden gelen `byte[]` veri alınır.
- Veri uzunluğu **1 byte değilse**, durum geçersiz kabul edilir.
- Geçersiz veya tanımsız durumda uygulama, güvenli bir varsayılan olarak `PLAYBACK_ONLY` moduna düşer.

Bu davranış birkaç önemli noktaya işaret eder:

- ANC durumu **event bazlı** veya **notify** ile güncelleniyor olabilir.
- Uygulama, cihazdan gelen verinin **mutlak doğruluğuna güvenmez**.
- Vendor tarafı, protokolde ileriye dönük genişleme ihtimaline karşı uygulamayı savunmacı (defensive) biçimde yazmıştır.

Burada özellikle dikkat edilmesi gereken nokta şudur:

Uygulama, cihazdan gelen değeri **doğrudan kullanmaz**; önce bu değeri `AncConfigData` enum/mapper yapısına sorar. Eşleşme bulunamazsa, durum geçersiz kabul edilir.

Bu yaklaşım, bilinmeyen veya yeni firmware değerlerinin uygulamayı kırmaması için bilinçli bir tasarım tercihidir.

Uygulamadan Cihaza: ANC Komutunun Kodlanması (Write Yolu)

Kullanıcı arayüzü üzerinden bir ANC modu seçildiğinde, ters yönde bir işlem gerçekleşir:

- Seçilen ANC durumu, `AncConfigData` tipiyle temsil edilir.

- Bu tip, kendi içinde **sayısal bir karşılığa** sahiptir.
- Uygulama, bu sayısal değeri **tek baytlık bir dizi** hâline getirerek ilgili BLE karakteristiğine yazar.

Bu noktada önemli olan şudur:

- Uygulama, BLE tarafına **hiçbir ek metadata** göndermez.
- CRC, length field veya flag bulunmaz.
- Komut, yalnızca `1 byte` üzerinden ifade edilir.

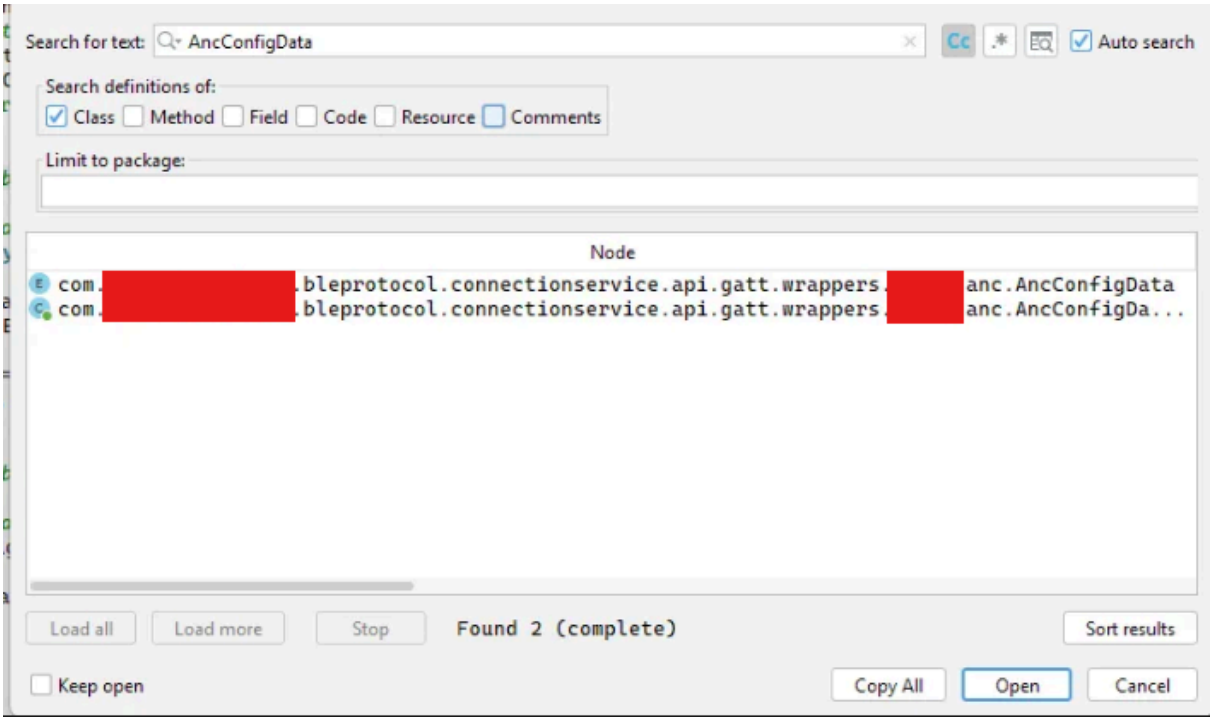
Bu da bize ANC kontrolünün:

- düşük bant genişliği gerektiren,
- latency'ye duyarlı,
- gerçek zamanlı kullanıcı etkileşimine uygun bir şekilde tasarlandığını göstermektedir.

💡 Bu aşamada hâlâ açık kalan soru şudur:

> Bu tek baytlık değerlerin (0, 1, 2, ...) hangi ANC modlarına karşılık geldiği
> ve bu eşlemenin nasıl tanımlandığı.

Bu sorunun yanıtı, bir sonraki adımda incelenecek olan `AncConfigData` sınıfında aranacaktır.



AncConfigData içinde yapılan analiz sonucunda ANC modlarının **sayısal karşılıkları açık biçimde tanımlanmış durumdadır**. İlgili enum eşlemesi aşağıdaki gibidir:

- `PLAYBACK_ONLY (0)`
- `CANCELLING (1)`
- `TRANSPARENCY (2)`

💡 **`k7.a` sınıfı araştırması**

k7.a Sınıfında yapılan analiz sonucunda not almaya değer aşağıdaki verileri bulabiliriz ancak makalenin ana odağını bozabileceği için bu kısa süreç dahil edilmeyecektir.

Uygulama k7.a sınıfının j isimli metodunu kullanarak cihazdan gelen verinin hangi kulaklığa ait olduğunu `instanceId` üzerinden ayırt ediyor.

Cihaz Parçası	Tam UUID Adresi	Değişken Adı
Sağ Kulaklık	`7a573e5d-9330-4d9b-8660-63c33fc5d401`	`f41419i` (RIGHT)
Sol Kulaklık	`7a573e5d-9330-4d9b-8660-63c33fc5d402`	`f41420j` (LEFT)
Şarj Kutusu	`7a573e5d-9330-4d9b-8660-63c33fc5d403`	`f41421k` (MAIN_CASE)

(k7.a sınıfına “BLECharacteristic” üzerinden ulaşabilirsiniz.)

Bu aşamada analiz `BLECharacteristic` içinde yer alan ve **ANC kontrolü için kullanılan** `aa20` **kısa UUID değerine** geri döner. Daha önce incelenen wrapper ve enum sınıfları, bu karakteristiğin **hangi veriyi taşıdığını** ve **nasıl yorumlandığını** ortaya koymuştur; ancak karakteristiğin BLE seviyesindeki **tam adreslemesi** bu noktada netleştirilmelidir.

```
SPEED_MODE = new BLECharacteristic("SPEED_MODE", 19, a.d("000e"), a.c("aa15"));
GRAPHICAL_EQUALIZER = new BLECharacteristic("GRAPHICAL_EQUALIZER", 20, a.d("000f"), a.c("aa16")
ANC_CONFIGURATION = new BLECharacteristic("ANC_CONFIGURATION", 21, a.d("0013"), a.c("aa20"));
TOUCH_LOCK = new BLECharacteristic("TOUCH_LOCK", 22, a.d("0014"), a.c("aa21"));
EQUALIZER_SETTINGS = new BLECharacteristic("EQUALIZER_SETTINGS", 23, a.d("0017"), a.c("aa25"));
```

`BLECharacteristic` tanımında yer alan `aa20` ifadesi, doğrudan bir `UUID` nesnesi değil; uygulama içinde daha alt seviyede çözümlenen bir **kısa tanımdır**. Bu nedenle analiz sırasında, bu değerin nasıl genişletildiğini görmek için ilgili referans takip edilmiştir.

Bu amaçla, `BLECharacteristic` içinde kullanılan yardımcı metoda (`a(...)`) gidilmiş ve bu çağrının hangi sınıfta gerçekleştiği incelenmiştir.

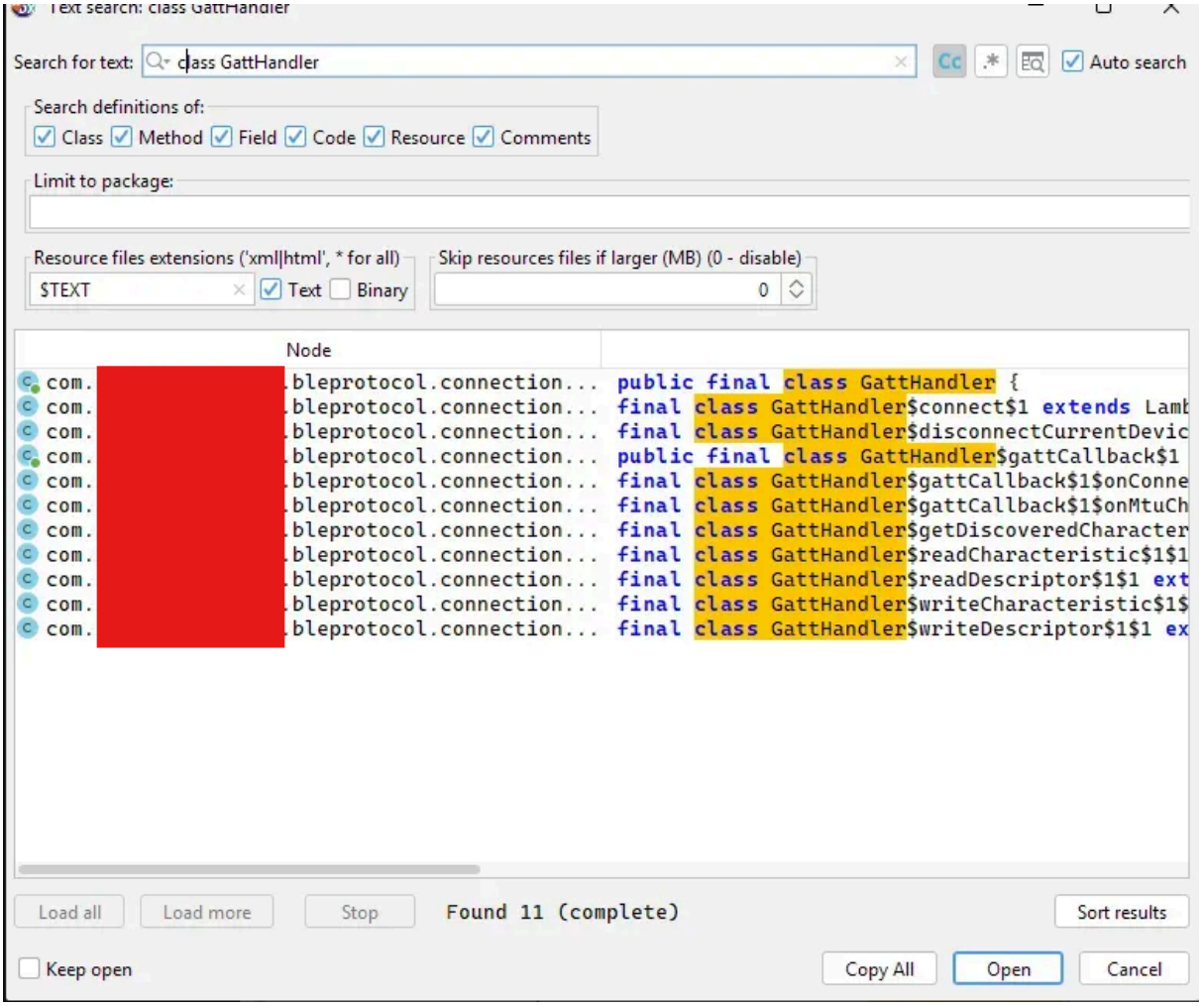
```
package com.[redacted].bleprotocol.connectionservice.api.gatt;

import java.util.UUID;
import kotlin.Metadata;

@Metadata(bv = {}, d1 = {"\u0000\u000e\n\u0002\u0010\u000e\n\u0000\n\u0002\u0018\u0002\n\u0002\b\u0003\u001a\u0010\u0010\u0003\u001a\u00020\u00022\u0006\u001
/* loaded from: classes3.dex */
public final class a {
    /* JADX INFO: Access modifiers changed from: private */
    public static final UUID c(String str) {
        return GattHandler.INSTANCE.a(str);
    }

    /* JADX INFO: Access modifiers changed from: private */
    public static final UUID d(String str) {
        return GattHandler.INSTANCE.b(str);
    }
}
```

Bu aşamada karşılaşılan yardımcı sınıf (`a`), UUID üretim mantığını kendi içinde barındırmaktan ziyade, işlemi merkezi bir bileşene devreden **yönlendirme katmanı** olarak konumlanmaktadır. Sınıfın içerdiği metotlar, yalnızca gelen kısa tanımı (`str`) alıp `GattHandler` üzerindeki ilgili çözücü fonksiyonlara iletmektedir.

Kod seviyesinde gözlem şu şekildedir:

- `c(String str)` → `GattHandler.INSTANCE.a(str)` çağrısını yapar
- `d(String str)` → `GattHandler.INSTANCE.b(str)` çağrısını yapar

Bu tasarım, **kısa UUID → tam UUID** dönüşümünün bu sınıfta değil, doğrudan `GattHandler` içinde tanımlandığını göstermektedir. Dolayısıyla "Base UUID" (ör. standart Bluetooth SIG tabanı `0000...-0000-1000-8000-00805f9b34fb` veya vendor'a özgü bir 128-bit taban) gibi kritik sabitlerin ve birleştirme mantığının **`GattHandler` içerisinde bulunması beklenir**.



`BLECharacteristic` ve ara sınıflar üzerinden yapılan yönlendirme takip edildiğinde, UUID üretim zincirinin nihai olarak `GattHandler` sınıfında sonlandığı görülmektedir. Kod tabanında bu sınıf, bin satırı aşan kapsamıyla kulaklığın BLE protokol davranışlarının **merkezi karar noktası** konumundadır. Bu nedenle sınıfın tamamı paylaşılmadan, yalnızca adresleme mantığını belirleyen kritik bölümler incelenmiştir.

Bu noktada `GattHandler`, uygulamanın Bluetooth tarafındaki tüm servis ve karakteristik UUID'lerini **tek bir yerden üreten** ve yöneten ana bileşen olarak ortaya çıkmaktadır.

`GattHandler` sınıfı içinde yer alan `Companion` nesnesi, kısa UUID (header) değerlerinin **hangi Base UUID ile genişletileceğini** belirleyen üç ayrı üretim yolunu açık biçimde tanımlamaktadır. Bu yapı, kulaklığın farklı protokol katmanlarıyla aynı anda çalışabilmesini sağlamaktadır.

1. Standart / Legacy UUID Üretimi (`a(header)`)

Bu yol, Bluetooth SIG tarafından tanımlı **standart Base UUID**'yi kullanır:

```
0000+header+0000-1000-8000-00805f9b34fb
```

Bu mekanizma, klasik ve geriye dönük uyumlu (legacy) BLE karakteristikleri için kullanılmaktadır. Dokümanda yer alan ve daha önce analiz edilen `aa20` **(ANC konfigürasyonu)** karakteristiği bu kategoriye dahildir.

---

2. Vendor-Spesifik / Modern UUID Üretimi (`b(header)`)

İkinci yol, standart Bluetooth tabanı yerine vendor'a özgü bir Base UUID kullanmaktadır:

```
0000 +header +g.f37787b
```

Bu yapı, üreticiye ait **modern BLE protokolünü** temsil etmektedir. APK içindeki daha yeni ANC komutlarının (örneğin `0013`) bu yol üzerinden adreslendiği gözlemlenmiştir. Bu durum, legacy ve modern protokollerin aynı uygulama içinde paralel olarak desteklendiğini göstermektedir.

---

3. OTA (Firmware Güncelleme) UUID Üretimi (`c(header)`)

Üçüncü yol ise yalnızca firmware güncelleme (OTA) süreçlerine ayrılmış sabit bir Base UUID kullanmaktadır:

```
0000 +header + d102-11e1-9b23-00025b00a5a5
```

Bu adresleme, normal çalışma karakteristiklerinden ayrıştırılmıştır ve güncelleme işlemlerinin güvenli biçimde yürütülmesini amaçlar.

Bu aşamada modern protokolün tam 128-bit UUID'sini elde etmek için geriye tek bir belirsizlik kalmaktadır:

Vendor'a özgü UUID üretiminde kullanılan g.f37787b değişkeninin içeriği.

Bu değerin, daha önce pil (battery) karakteristiklerinde görülen

- `9330-4d9b-8660-63c33fc5d400` benzeri bir **vendor tail (kuyruk)** içermesi kuvvetle muhtemeldir. Dolayısıyla bir sonraki adım, `g` sınıfının veya bu sabitin

tanımlandığı noktayı bularak modern ANC protokolünün **tam BLE adresini kesin olarak çıkarmaktır**.

```
@NotNull
public final UUID b(@NotNull String header) {
    f0.p(header, "header");
    UUID r32 = UUID.fromString("0000" + header + g.f37787b);
    f0.o(r32, "fromString(\"0000$header$[redacted]_GATT_UUID\")");
    return r32;
}

@NotNull
```

`GattHandler` içinde kullanılan `g.f37787b` referansı takip edildiğinde, ilgili sabitin `g` sınıfında doğrudan string olarak tanımlandığı görülür. Bu sınıf, uygulamanın BLE adresleme şemasında kullanılan üç farklı Base UUID "kuyruğunu" (tail) merkezi olarak tutmaktadır:

```
public final class g {
   @NotNull
   public static final String f37786a = "-0000-1000-8000-00805f9b34fb";
   @NotNull
   public static final String f37787b = "-1337-1dea-feed-c0ffee70c0de";
   @NotNull
   public static final String f37788c = "-d102-11e1-9b23-00025b00a5a5";
   private static final int f37789d = 2;
   private static final long f37790e = 500;
   private static final long f37791f = 10000;
   private static final int f37792g = 5;
   private static final int f37793h = 6;
   private static final int f37794i = 13;
   private static final int f37795j = 14;
   private static final int f37796k = 185;
   @NotNull
   private static final UUID f37797l = GattHandler.INSTANCE.a("2902");
   @NotNull
   public static final UUID a() {
      return f37797l;
   }
}
```

`Tam UUID = "0000" + <header> + <ilgili Base UUID kuyruğu>` mantığını kullanarak aşağıdaki tablolar oluşturulmuştur:

Servis Adı	Tam UUID Adresi	Açıklama
Vendor Modern Service	`0000fccd-0000-1000-8000-00805f9b34fb`	Yeni nesil komutlar için ana servis.
Legacy Vendor Service	`0000aa00-0000-1000-8000-00805f9b34fb`	Eski tip (aaXX) karakteristikler için.
Battery Service	`0000180f-0000-1000-8000-00805f9b34fb`	Standart pil servisi.

Fonksiyon	Tam UUID Adresi	Değerler (Byte)
Modern ANC Kontrol	`00000013-1337-1dea-feed-c0ffee70c0de`	`00`: Kapalı, `01`: ANC Açık, `02`: Şeffaf
Legacy ANC Kontrol	`0000aa20-0000-1000-8000-00805f9b34fb`	(Üsttekiyle aynı mod değerlerini kullanır)
Ekolayzır (EQ)	`0000aa25-0000-1000-8000-00805f9b34fb`	EQ profilleri
Düğme Olayları	`0000aa13-0000-1000-8000-00805f9b34fb`	Fiziksel buton basışlarını buradan okunur

Donanım	Tam UUID Adresi	Okuma Tipi
Sağ Kulaklık	`7a573e5d-9330-4d9b-8660-63c33fc5d401`	Read / Notify (0-100 arası)
Sol Kulaklık	`7a573e5d-9330-4d9b-8660-63c33fc5d402`	Read / Notify (0-100 arası)
Şarj Kutusu	`7a573e5d-9330-4d9b-8660-63c33fc5d403`	Read / Notify (0-100 arası)

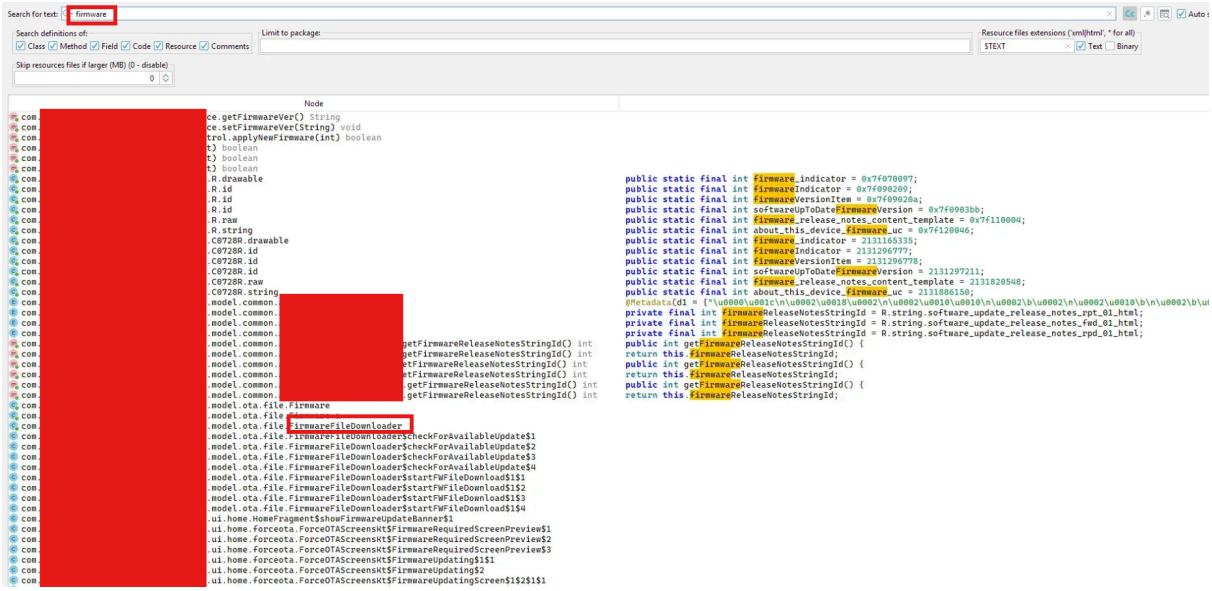
Uygulama Üzerinden Firmware Güncelleme Kanalının Analizi

Bu aşamaya kadar yapılan inceleme, mobil uygulamanın cihazla kurduğu BLE iletişimini; kullanılan servis ve karakteristik UUID'lerini, legacy ve modern protokol ayrımını ve temel kontrol mekanizmalarını ortaya koymuştur. Bu analizler tamamen **uygulama tarafındaki incelemelere** dayanmaktadır.

Araştırmanın bir sonraki adımı da aynı yaklaşımı sürdürerek, yine uygulama üzerinden yürütülecektir. Bu kapsamda analiz, mobil uygulamanın **firmware güncelleme sürecini** nasıl yönettiğine odaklanacaktır. Özellikle uygulamanın hangi güncelleme uç noktalarıyla iletişim kurduğu, firmware paketlerini nasıl

tanımladığı ve bu dosyaları hangi indirme kanalı üzerinden temin ettiği incelenecektir.

Bu doğrultuda, uygulamanın firmware sunucusundan cihaza ait güncelleme dosyasını indirme süreci analiz edilecek ve elde edilen firmware paketi, sonraki bölümde yapısal olarak değerlendirilmek üzere ayrıştırılacaktır.

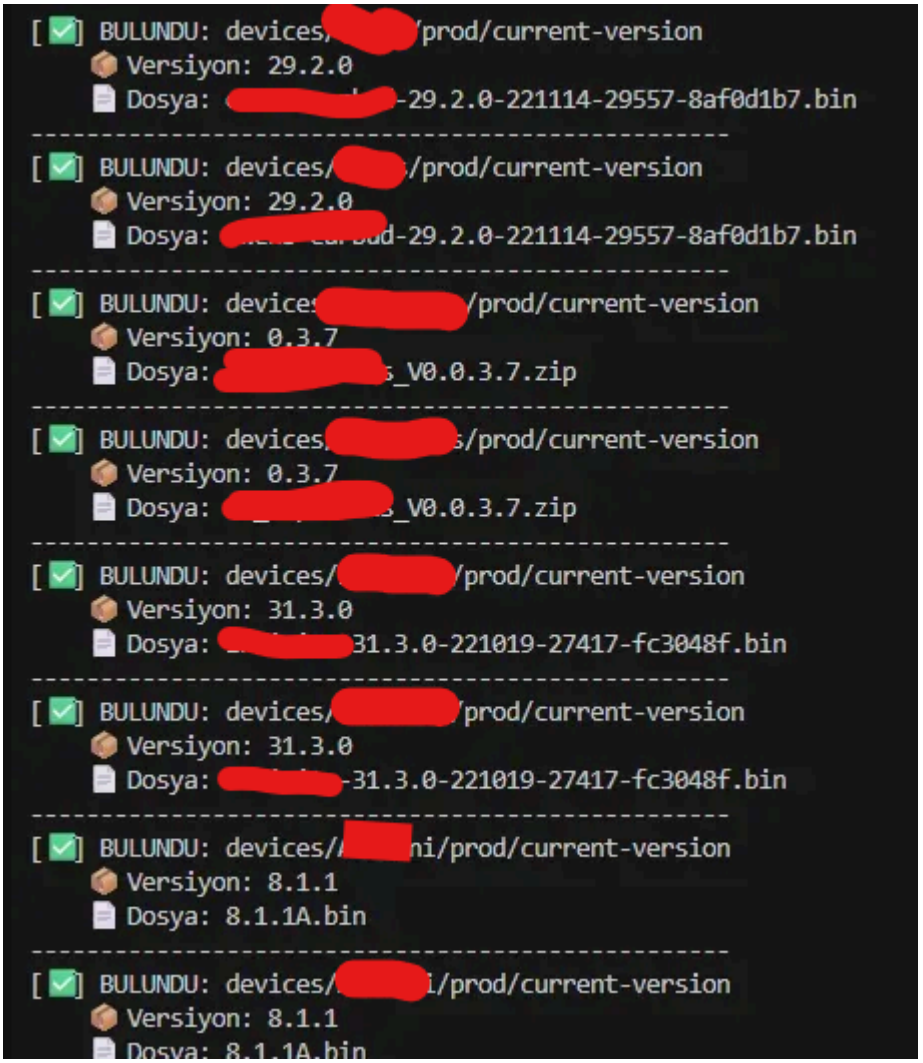


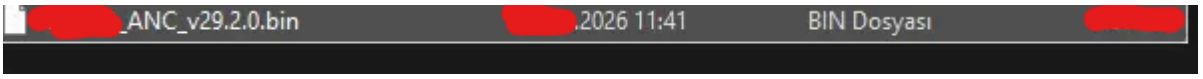
Firmware indirme sürecinde kullanılan istemci, gerekli kimlik ve yetkilendirme bilgilerini uygulama içindeki merkezi bir yapıdan dinamik olarak almaktadır. Bu bilgiler, sabit olarak kodlanmak yerine, konfigürasyon sağlayıcı bir nesne üzerinden (`a.f35480a.a()`) elde edilmektedir.

Firmware güncelleme kanalının analizi, uygulama içinde “firmware” referanslarının izlenmesiyle başlatılmış ve güncelleme sürecinde kullanılan kritik kimlik bilgisinin düz metin olarak değil, obfuske edilmiş bir `byte[]` dizisi halinde saklandığı tespit edilmiştir. Bu dizinin çalışma anında çözümlendiği metot zinciri takip edildiğinde, dönüşümün standart kriptografik API’ler kullanılarak gerçekleştirildiği; anahtar materyali ve başlangıç vektörünün uygulama içindeki sabitlerden üretildiği ve çözümleme sonrası çıktının ek bir Base64 decode adımından geçirilerek okunabilir metne dönüştürüldüğü görülmüştür. Analiz sırasında, decompiler’ın sabit bayt dizilerini bazı durumlarda hatalı temsil

edebildiği fark edilmiş; özellikle IV tarafında Base64 alfabesi dışı değerler içeren diziler nedeniyle doğrulamalı şifrelemenin (AES-GCM) tek bitlik hatada dahi başarısız olabileceği anlaşılmıştır. Bu nedenle çözümleme mantığı Smali seviyesinde .array-data blokları üzerinden doğrulanmış, Android'in Base64 çözümleyicisinin geçersiz karakterleri esnek biçimde ele alma davranışı dikkate alınarak gerçek bayt akışı teyit edilmiştir. Bu doğrulama zinciri sonucunda, uygulamanın firmware dağıtım altyapısında kullandığı kimlik bilgisinin obfuske edilme biçimi, çözümleme adımları ve yetkilendirme akışındaki yeri, tamamen uygulama içi kod akışına dayalı ve kanıta dayalı olarak ortaya konmuştur.

Firmware dağıtım sürecinin uygulama tarafındaki işleyişi incelendiğinde, cihaz için indirilecek yazılım paketinin rastgele veya serbest keşif yoluyla değil, **uygulama içinde tanımlı bir mantık zinciri** üzerinden belirlendiği görülmektedir. Uygulama, BLE tarama sırasında elde edilen üretici verileri, cihaz adı ve model kodlarını kullanarak cihazın hangi ürün ailesine ve varyanta ait olduğunu sınıflandırmakta; bu sınıflandırma sonucu, firmware sunucusundaki **dizin yapısının** hangi dalının kullanılacağını belirlemektedir. Cihaz modeli, üretim ortamı (ör. üretim/stabil kanal) ve sürüm bilgisi birleştirilerek hedef firmware dosyasının tam yolu hesaplanmakta, uygulama bu yolu doğrudan kullanarak dosyaya erişmektedir. Yetkilendirme için gereken kimlik bilgileri, uygulama içinde düz metin olarak saklanmamakta; bunun yerine obfuske edilmiş sabitlerden çalışma anında çözümlenen geçici kimlik materyali ile erişim sağlanmaktadır. İndirme tamamlandıktan sonra, dosyanın bütünlüğü uygulama tarafında kriptografik bir özet (hash) üzerinden doğrulanmakta ve yalnızca beklenen değerle eşleşen paketlerin geçerli kabul edildiği görülmektedir. Bu akış, firmware güncelleme mekanizmasının; cihaz tanımlama, yetkilendirme, hedef dosya seçimi ve bütünlük doğrulamasını kapsayan, kapalı ve deterministik bir dağıtım modeli üzerine kurulduğunu açık biçimde ortaya koymaktadır.





Analiz Sürecinin Kapanışı ve Uygulama Geliştirme Aşamasına Geçiş

Bu çalışma kapsamında yürütülen analizler sonucunda, cihaz–uygulama etkileşiminin temel bileşenleri, firmware dağıtım mantığı ve güncelleme sürecinde kullanılan yapı taşları uygulama içi kod akışı üzerinden ortaya konmuştur. Elde edilen bu bulgular, artık tersine mühendislik ve keşif aşamasından çıkılarak, pratik bir uygulama geliştirme sürecine geçilebilecek olgunluğa ulaşmıştır.

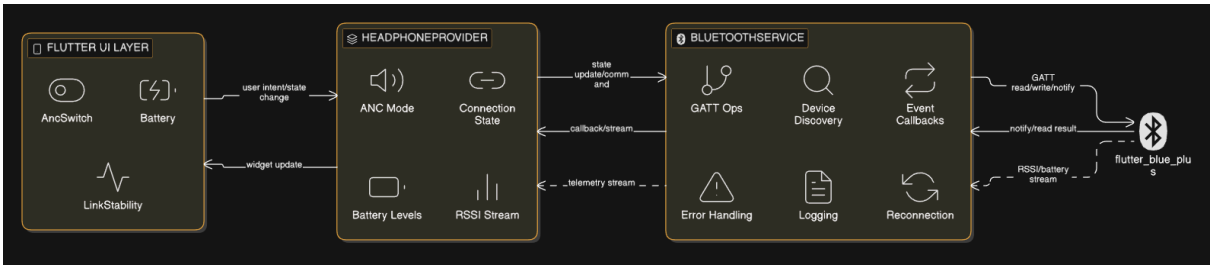
Bu doğrultuda, sonraki adım olarak elde edilen teknik bilgiler temel alınarak **Flutter** kullanımıyla platformdan bağımsız bir istemci uygulama geliştirilmesi hedeflenmiştir. Bu uygulamanın amacı, cihazla kurulan iletişimi ve güncelleme süreçlerini kontrollü ve kullanıcı odaklı bir arayüz üzerinden yönetmektir. Analiz

sırasında elde edilen firmware dosyasının iç yapısına yönelik detaylı incelemeler ise, bu çalışmanın kapsamı dışında bırakılmış olup; gerek görülmesi hâlinde, firmware paketinin formatı, içerdiği bölümler ve doğrulama mekanizmalarının ele alındığı ayrı bir teknik makalede değerlendirilmesi planlanabilir.

Uygulamayı geliştirmek amacıyla referans alınacak teknik dokümantasyon aşağıda sunulmuştur.

Bu doküman; BLE iletişim mimarisini, bağlantı yönetimini, servis ve karakteristik eşlemlerini, veri akışlarını ve platforma özgü gereksinimleri (Android bonding, MTU yapılandırması vb.) kapsar ve uygulama geliştirme sürecinde doğrudan teknik referans olarak kullanılmak üzere hazırlanmıştır.

1. BLE İLETİŞİM MİMARİSİ

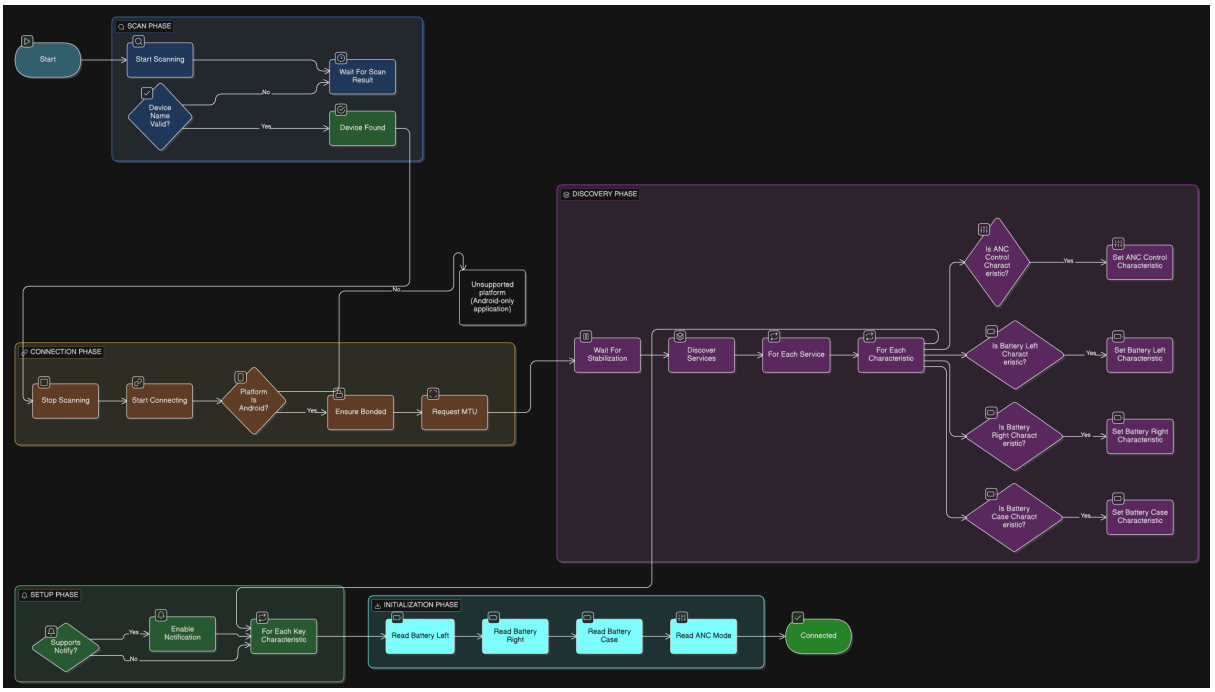


2. UUID TANIMLAMALARI VE KARAKTERİSTİKLER

```
/// VENDOR SERVİSİ
static const String vendorService = "0000aa00-0000-1000-8000-00805f9b34fb";
/// ANC KONTROL KARAKTERİSTİĞİ
/// Özellikler: READ + WRITE + NOTIFY
static const String ancControl = "0000aa20-0000-1000-8000-00805f9b34fb";

/* ANC BYTE DEĞERLERİ:

- 0x00 → Kapalı (Off/Neutral)
- 0x01 → ANC Aktif (Noise Cancellation)
- 0x02 → Şeffaflık/Farkındalık (Transparency/Awareness)
*/
```

```
/// BATARYA KARAKTERİSTİKLERİ
// Standart BLE Battery Service (0000180f servisi altında)
// İlk 2a19 → Sol Kulaklık, İkinci 2a19 → Sağ Kulaklık
static const String standardBattery = "00002a19-0000-1000-8000-00805f
9b34fb";

// Vendor Specific - Kutu Bataryası
static const String batteryCase = "0000aa16-0000-1000-8000-00805f9b3
4fb";
```

3. BAĞLANTI YÖNETİMİ (CONNECTION FLOW)




```
Future<bool> connect(BluetoothDevice device) async {
 try {
  // 1. Bağlan
  await device.connect(
   timeout: const Duration(seconds: 15),
   autoConnect: false,
```

```
      license: License.free,
    );

    // 2. Android Bonding
    if (Platform.isAndroid) {
      await ensureBonded(device);
    }

    // 3. MTU Request (Android) - KRİTİK: 247 byte
    if (Platform.isAndroid) {
      await device.requestMtu(247);
    }

    // 4. Stabilizasyon
    await Future.delayed(const Duration(milliseconds: 300));

    // 5. Service Discovery
    await _discoverServices(device);

    return true;
  } catch (e) {
    return false;
  }
}

Future<bool> connect(BluetoothDevice device) async {
  try {
    // 1. Bağlan
    await device.connect(
      timeout: const Duration(seconds: 15),
      autoConnect: false,
      license: License.free,
    );

    // 2. Android Bonding
    if (Platform.isAndroid) {
      await ensureBonded(device);
    }
```

```
    // 3. MTU Request (Android) - KRİTİK: 247 byte
    if (Platform.isAndroid) {
      await device.requestMtu(247);
    }

    // 4. Stabilizasyon
    await Future.delayed(const Duration(milliseconds: 300));

    // 5. Service Discovery
    await _discoverServices(device);

    return true;
  } catch (e) {
    return false;
  }
}
```

4. ANC (ACTIVE NOISE CANCELLATION) KONTROLÜ

```
Future<bool> setAncMode(int mode) async {
  if (_ancCharacteristic == null) return false;

  try {
    // writeWithoutResponse tercih et (daha hızlı)
    if (_ancCharacteristic!.properties.writeWithoutResponse) {
      await _ancCharacteristic!.write([mode], withoutResponse: true);
    } else {
      await _ancCharacteristic!.write([mode], withoutResponse: false);
    }

    onAncStateChanged?.call(mode);
    return true;
  } catch (e) {
    return false;
  }
}
```

```
// ANC Modunu Oku
Future<int> readAncMode() async {
  if (_ancCharacteristic == null) return -1;
  final value = await _ancCharacteristic!.read();
  return value.isNotEmpty ? value[0] : -1;
}

// ANC Değişikliklerini Dinle (Notify)
Future<void> subscribeToAncChanges() async {
  if (_ancCharacteristic?.properties.notify == true) {
    await _ancCharacteristic!.setNotifyValue(true);
    _ancCharacteristic!.lastValueStream.listen((value) {
      if (value.isNotEmpty) {
        onAncStateChanged?.call(value[0]);
      }
    });
  }
}
```

5. Batarya Servisi İzleme

```
// KARAKTERISTIK TANIMLAMA (Service Discovery sırasında)
Future<void> _discoverServices(BluetoothDevice device) async {
  final services = await device.discoverServices();

  for (var service in services) {
    for (var char in service.characteristics) {
      final charUuid = char.uuid.toString().toLowerCase();

      // ANC Kontrolü
      if (charUuid.contains("aa20") && char.properties.write) {
        _ancCharacteristic = char;
      }

      // Batarya Kontrolü (Standart 2a19)
      if (charUuid.contains("2a19")) {
        if (_batteryLeftChar == null) {
          _batteryLeftChar = char;
```

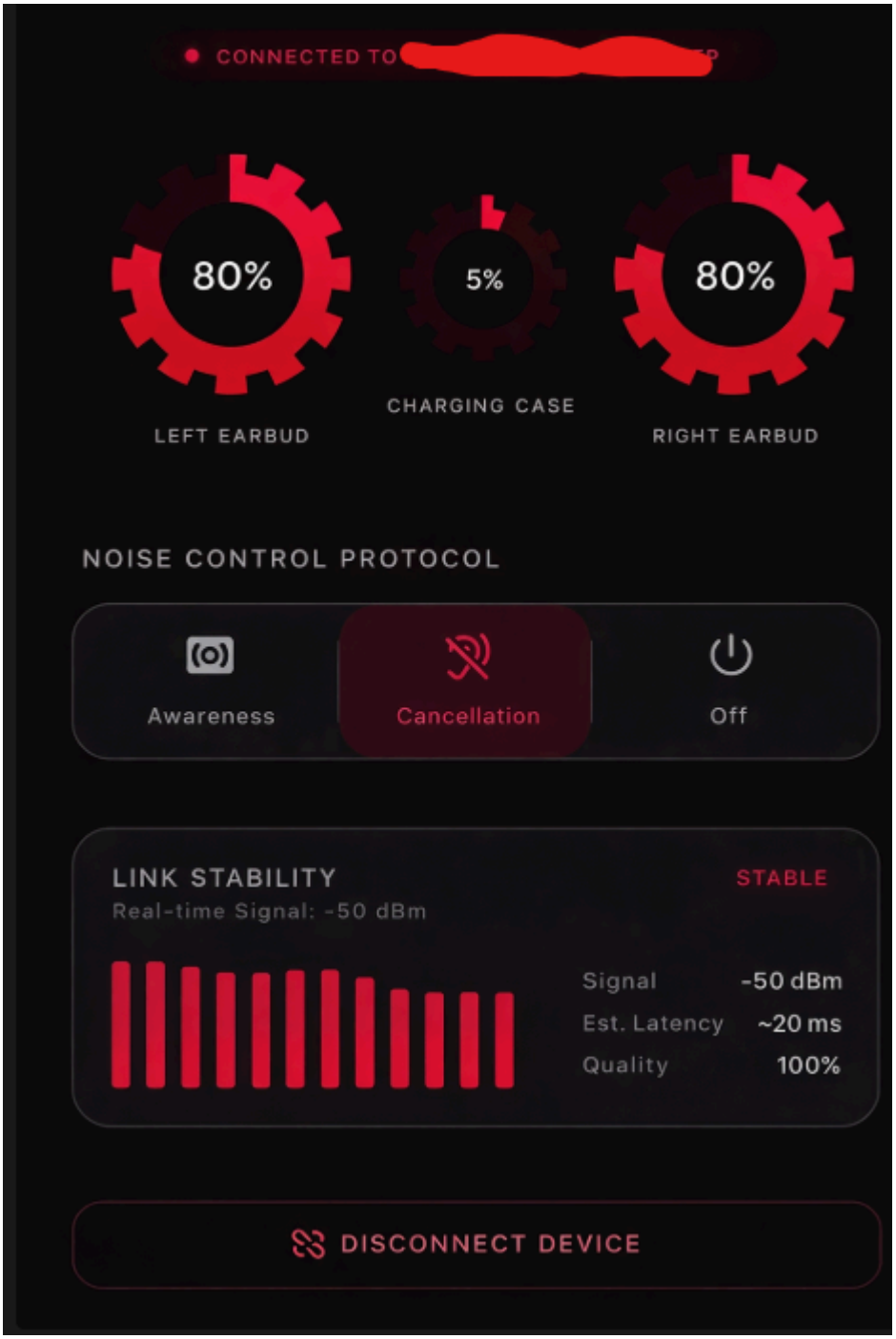
```
      } else if (_batteryRightChar == null) {
        _batteryRightChar = char;
      }
    }

    // Kutu Bataryası (aa16)
    if (charUuid.contains("aa16")) {
      _batteryCaseChar = char;
    }

    // Notify destekleyen karakteristikleri etkinleştir
    if (char.properties.notify) {
      await char.setNotifyValue(true);
    }
   }
  }
}

// BATARYA BİLDİRİMLERİNİ AKTİFLEŞTİRME
Future<void> enableBatteryNotifications() async {
  // Sol Kulaklık
  if (_batteryLeftChar?.properties.notify == true) {
    await _batteryLeftChar!.setNotifyValue(true);
    _batteryLeftChar!.lastValueStream.listen((value) {
      if (value.isNotEmpty) {
        onBatteryLeftChanged?.call(value[0]);
      }
    });
  }

  // Sağ Kulaklık
  if (_batteryRightChar?.properties.notify == true) {
    await _batteryRightChar!.setNotifyValue(true);
    _batteryRightChar!.lastValueStream.listen((value) {
      if (value.isNotEmpty) {
        onBatteryRightChanged?.call(value[0]);
      }
    });
```

```
  }

  // Kutu
  if (_batteryCaseChar?.properties.notify == true) {
    await _batteryCaseChar!.setNotifyValue(true);
    _batteryCaseChar!.lastValueStream.listen((value) {
      if (value.isNotEmpty) {
        onBatteryCaseChanged?.call(value[0]);
      }
    });
  }
}

// MANUEL OKUMA
Future<int> readBatteryLeft() async {
  if (_batteryLeftChar == null) return -1;
  final value = await _batteryLeftChar!.read();
  return value.isNotEmpty ? value[0] : -1;
}

Future<int> readBatteryRight() async {
  if (_batteryRightChar == null) return -1;
  final value = await _batteryRightChar!.read();
  return value.isNotEmpty ? value[0] : -1;
}

Future<int> readBatteryCase() async {
  if (_batteryCaseChar == null) return -1;
  final value = await _batteryCaseChar!.read();
  return value.isNotEmpty ? value[0] : -1;
}
```

6. Sinyal Kalitesi

```
// Stream Controller
final _rssiController = StreamController<int>.broadcast();
Stream<int> get rssiStream ⇒ _rssiController.stream;
Timer? _rssiTimer;
```

```
// RSSI İzlemeyi Başlat
void startRssiMonitoring() {
  _rssiTimer?.cancel();
  _rssiTimer = Timer.periodic(const Duration(seconds: 2), (timer) async {
    if (connectedDevice != null && isConnected) {
      try {
        final rssi = await connectedDevice!.readRssi();
        _rssiController.add(rssi);
      } catch (e) {
        // RSSI okuma hatası
      }
    } else {
      stopRssiMonitoring();
    }
  });
}

// RSSI İzlemeyi Durdur
void stopRssiMonitoring() {
  _rssiTimer?.cancel();
  _rssiTimer = null;
}
```



Final

Bu çalışma boyunca kapalı ve dokümantasyonsuz bir Bluetooth kulaklık ekosistemi çok katmanlı olarak çözümlendi ve ciddi teknik engeller sistematik biçimde aşıldı. BLE tarafında vendor-specific GATT servisleri ve karakteristikler, **bluetoothctl**, **btmon** ve **Wireshark (HCI/BLE logları)** birlikte kullanılarak ayrıştırıldı; kontrol, durum ve telemetri kanalları güvenilir şekilde eşleştirildi.

Mobil uygulama katmanında **jadx** ve **smali/baksmali** analizleri ile obfuscate edilmiş akışlar, gizli sabitler ve cihaz-model mantığı ortaya çıkarıldı. Sunucu tarafına geçişte, uygulama içinde gömülü verilerin **AES-GCM** (authenticated encryption) kullanılarak şifrelendiği, anahtar ve IV'nin Base64 tabanlı türevlerle işlendiği tespit edildi; bütünlük ve doğrulama mekanizmalarında **MD5 / SHA-256 hash** kontrollerinin rolü belirlendi. Yetkilendirme ve içerik dağıtım sürecinde geçici kimlikler, imzalı istekler ve TLS korumalı aktarım zinciri analiz edilerek firmware metadata yapısı, sürümleme ve cihaz eşleştirme mantığı çözüldü. Sonuç olarak; gizlenmiş şifreleme katmanları, kapalı protokol tasarımı ve çok aşamalı doğrulama zinciri aşılmış; elde edilen veriler doğrulanabilir, yeniden üretilebilir ve mühendislik açısından kullanılabilir hale getirilmiştir. Bu noktada çalışma, belirsizlik içeren bir tersine inceleme olmaktan çıkmış; kullanılan araçlar, çözülen kriptografik yapılar ve elde edilen çıktılarla tamamlanmış, teknik olarak sağlam bir başarıya dönüşmüştür.

Okuduğunuz için teşekkürler.

Özhan Yıldırım