Why does my security camera scream like a Banshee?

Signal analysis and RE of a proprietary audio-data encoding protocol

About Me

Rion Carter
www: 7thzero.com

Software Engineer

Love me some codes!

Hacker Mindset

Move fast and break things... then fix them!

Food

Recipe hacking with delicious results!

Opinionated

All opinions are my own and may not reflect those of past/present employers

TABLE OF CONTENTS

01

What Are We Doing Here?

And why are we talking about wireless security cameras?


Original Goal: Use an inexpensive wireless camera to monitor the garden

Inexpensive Camera

Sounds easy... What's the catch?

Cloud 'App' Lock-in

Vendor app requires a login

No docs

Good luck finding reliable docs on a $30 camera

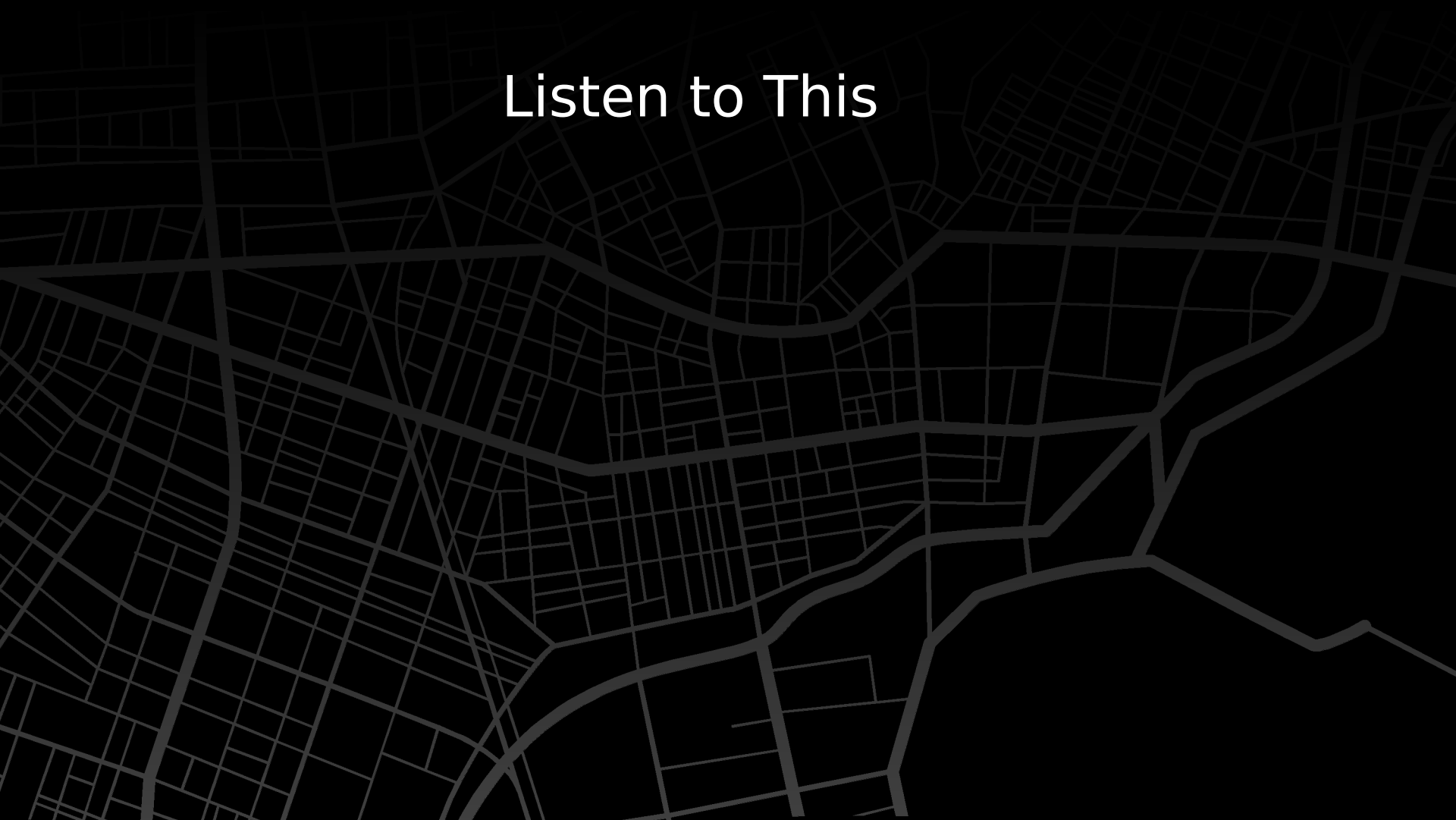
No self-setup

No way to directly configure the camera (lacks BT/AD-HOC WIFI)

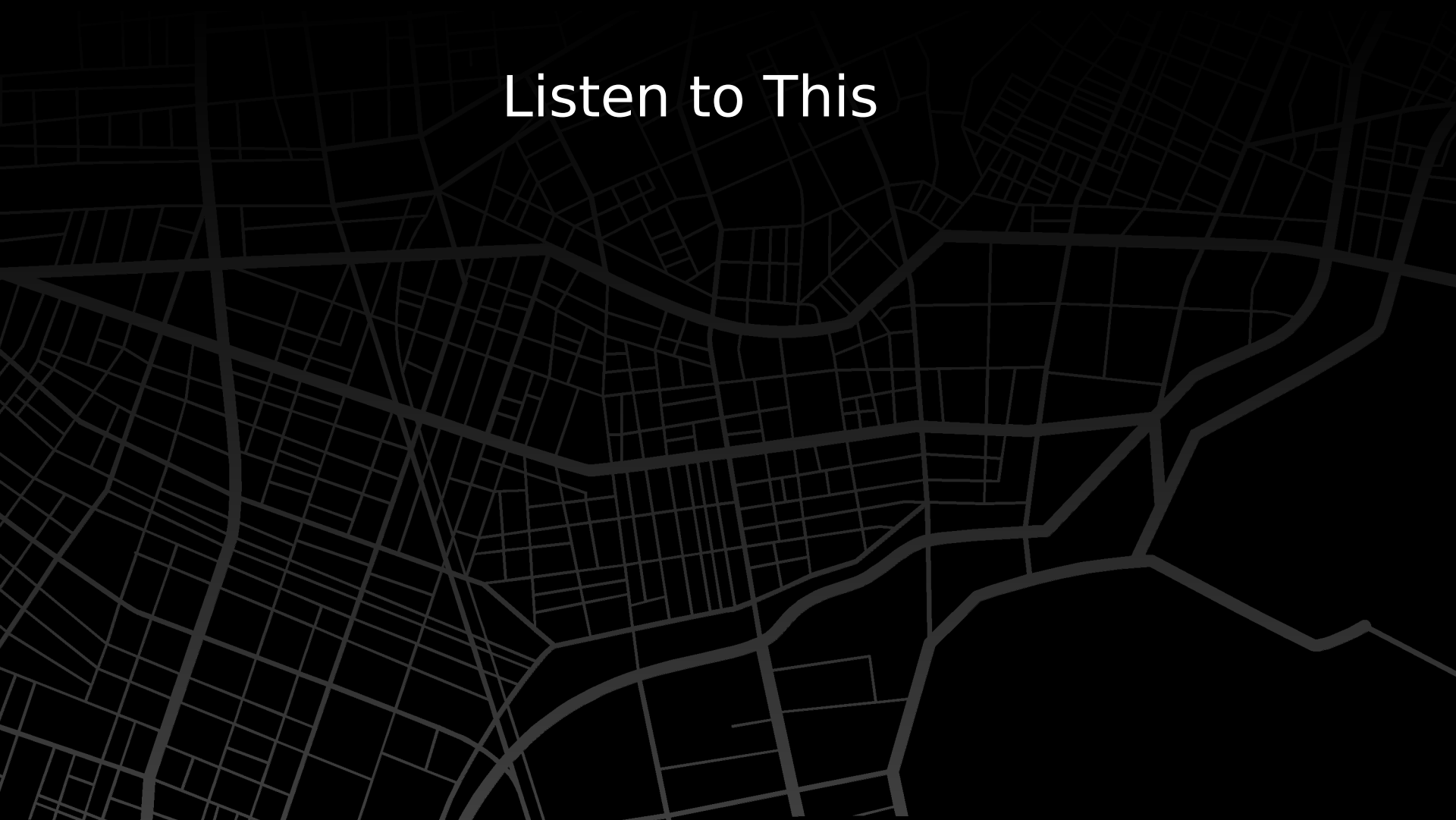
Bespoke Setup Protocol Using Soundwaves

Which is what brings us here today ;)

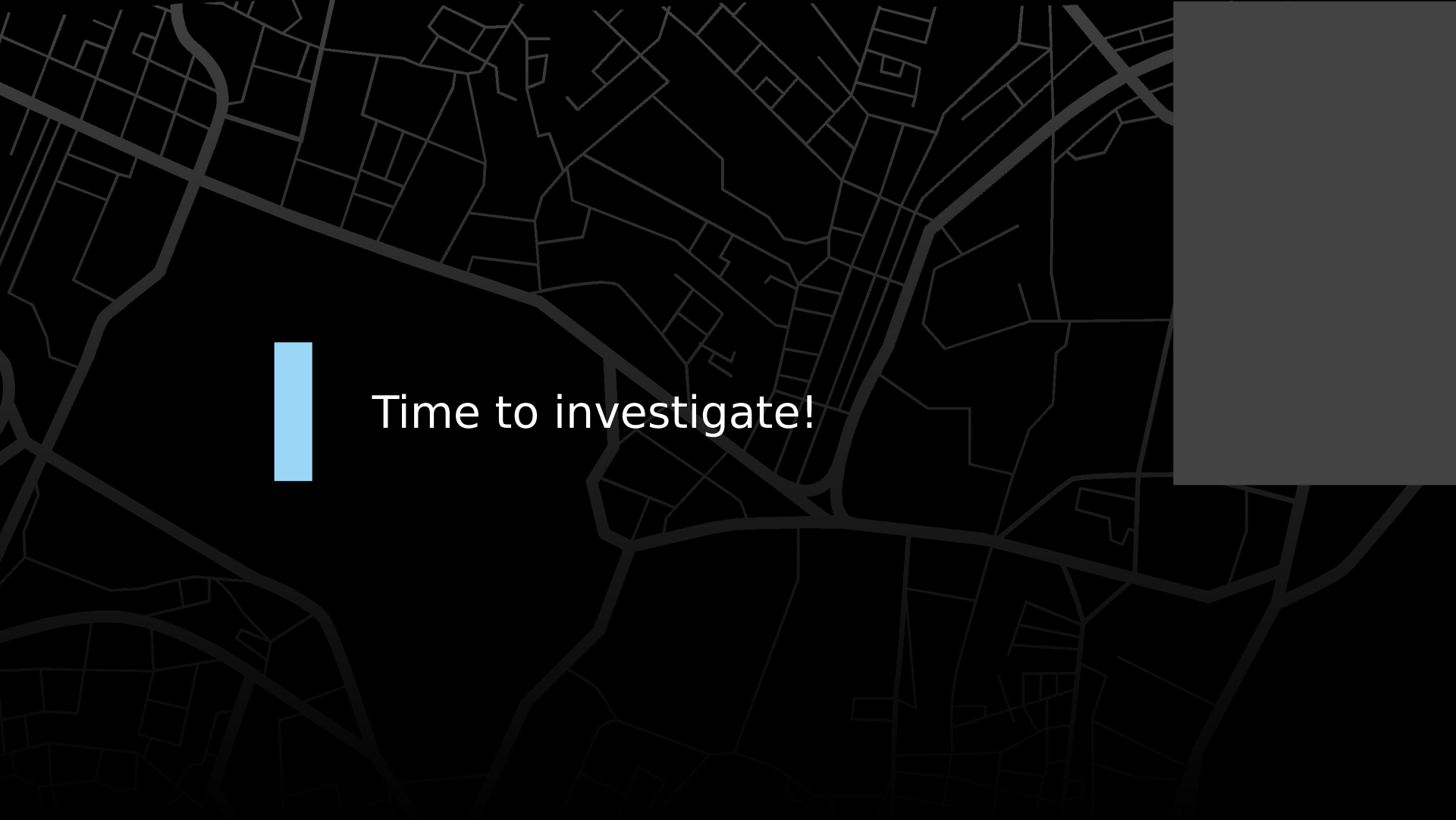
Listen to This

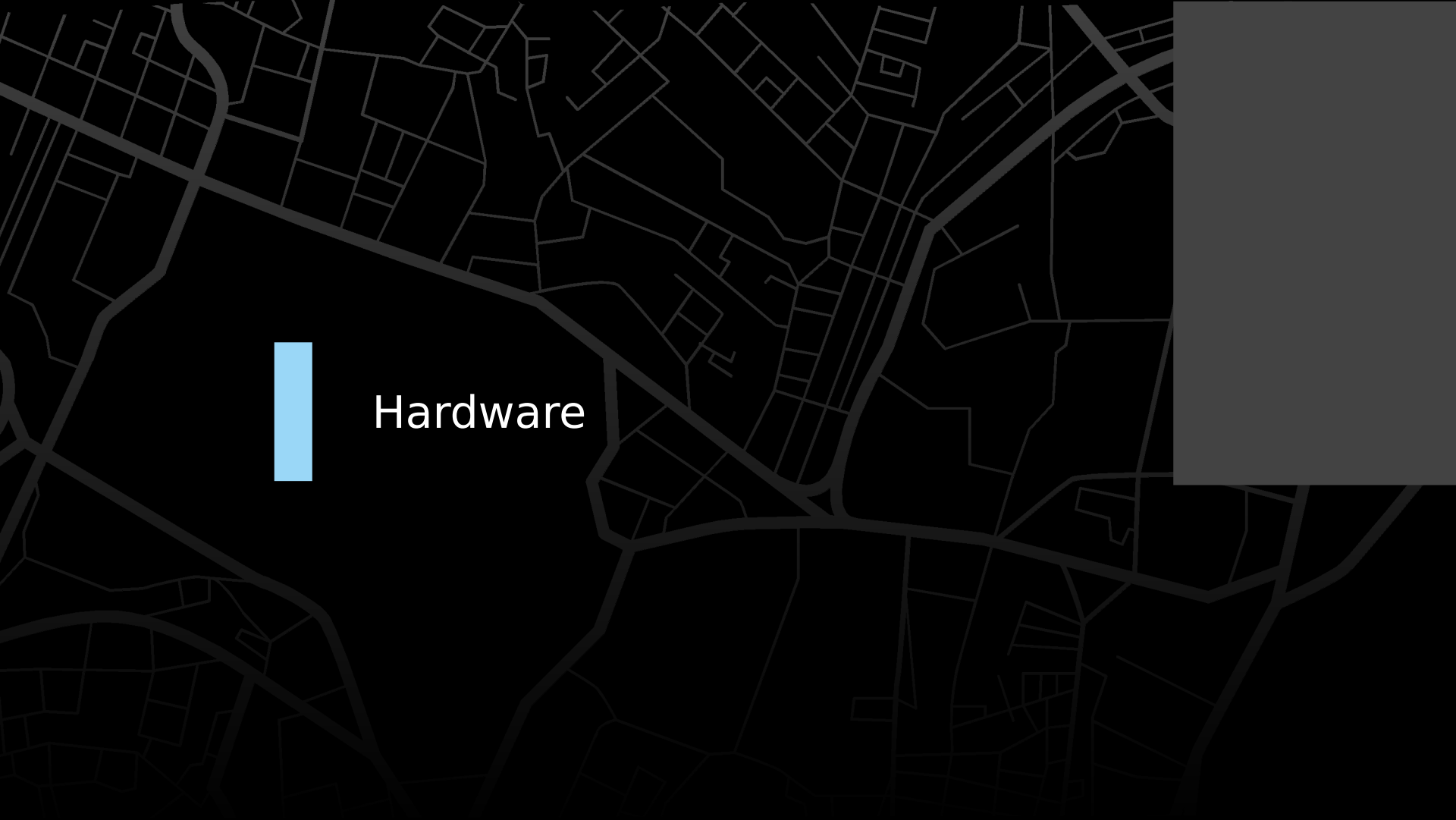


Listen to This




New Goal: Figure out what is going on during camera setup


Time to investigate!


Hardware

Camera Inspection

USB

Unfortunately, this is not an avenue of investigation

The cable supplies power to the camera

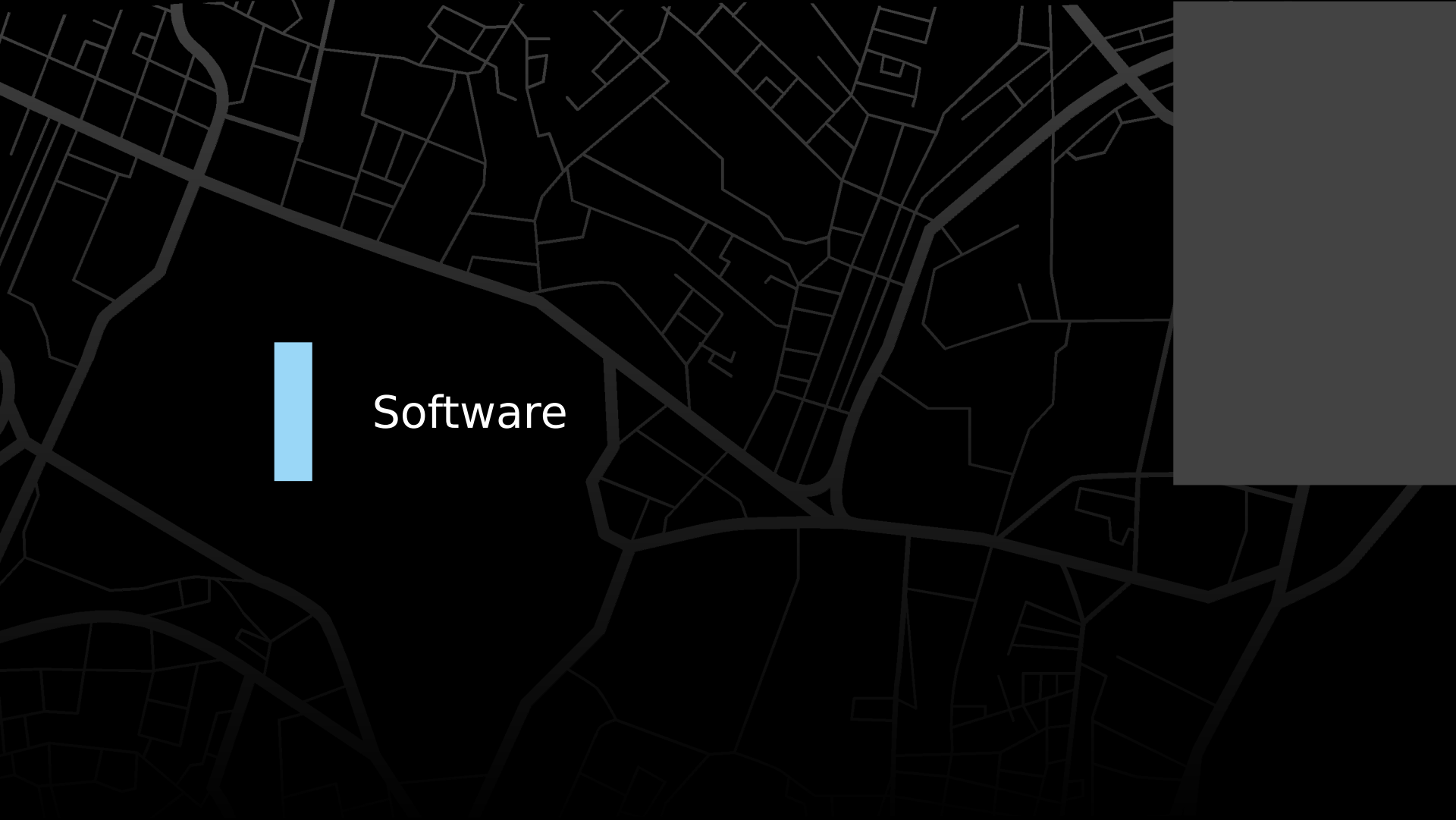
Camera Inspection

BT/WIFI

On boot, the camera enters ‘setup mode’

Setup mode requires the app for pairing

The camera does not advertise an Ad-Hoc network or show up as a bluetooth device


Software

Vendor App

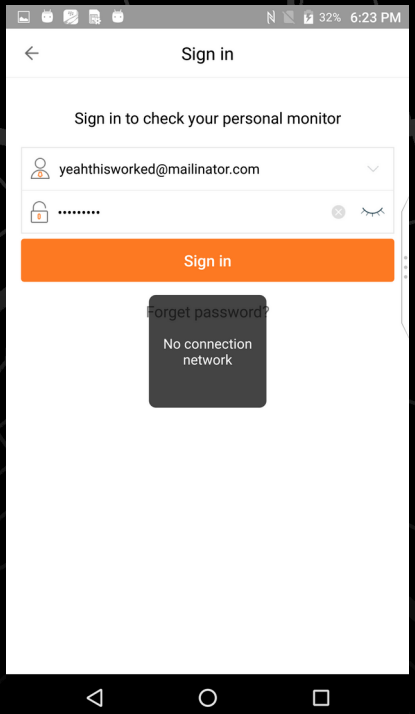
Meet JAWA

JAWA is the app used to configure the ‘Cloud Camera’

Vendor App

The App needs something...

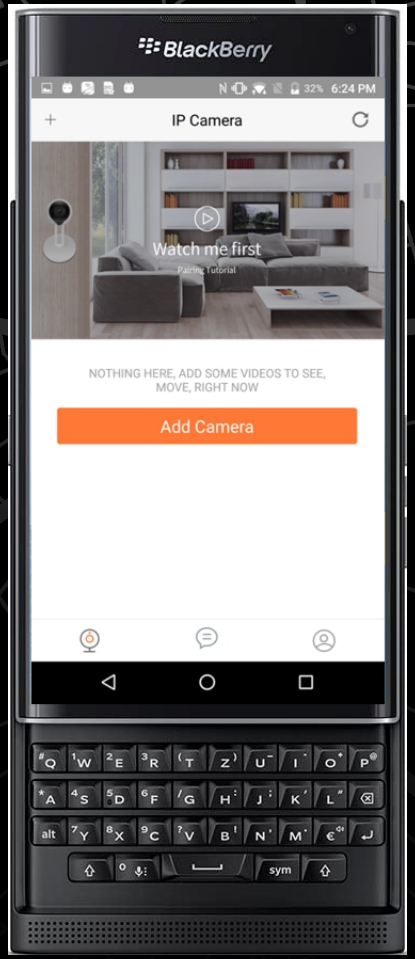
Unfettered internet access



Vendor App

Time for a ‘test’ device

All vendor-app analysis performed on a ‘vintage’ BlackBerry Priv



Not much online...

Searching for terms like ‘Amiccom’, ‘Smart IP Camera’, and ‘audio pairing wifi camera’ didn’t turn up a whole lot. Expanding the search found additional cameras that may use a similar pairing technique, though...

- Imou Cue 2, Ranger 2, LOOC, Bullet Lite, Ranger Pro

Interestingly enough, I found examples of cheap cameras that use a less-flaky approach (QR Code Scanning) in place of audio pairing:

- BESDER 1080P Cloud Storage Wireless PTZ IP Camera
- Swann Indoor/Outdoor Wireless 1080p WiFi Security Camera

Signal Analysis

02

It sounds... Intriguing?

What should we try?

Capture & Visualize

Capture the signal, use tools to visualize the signal

Reptition

Are there any parts of the signal that repeat?

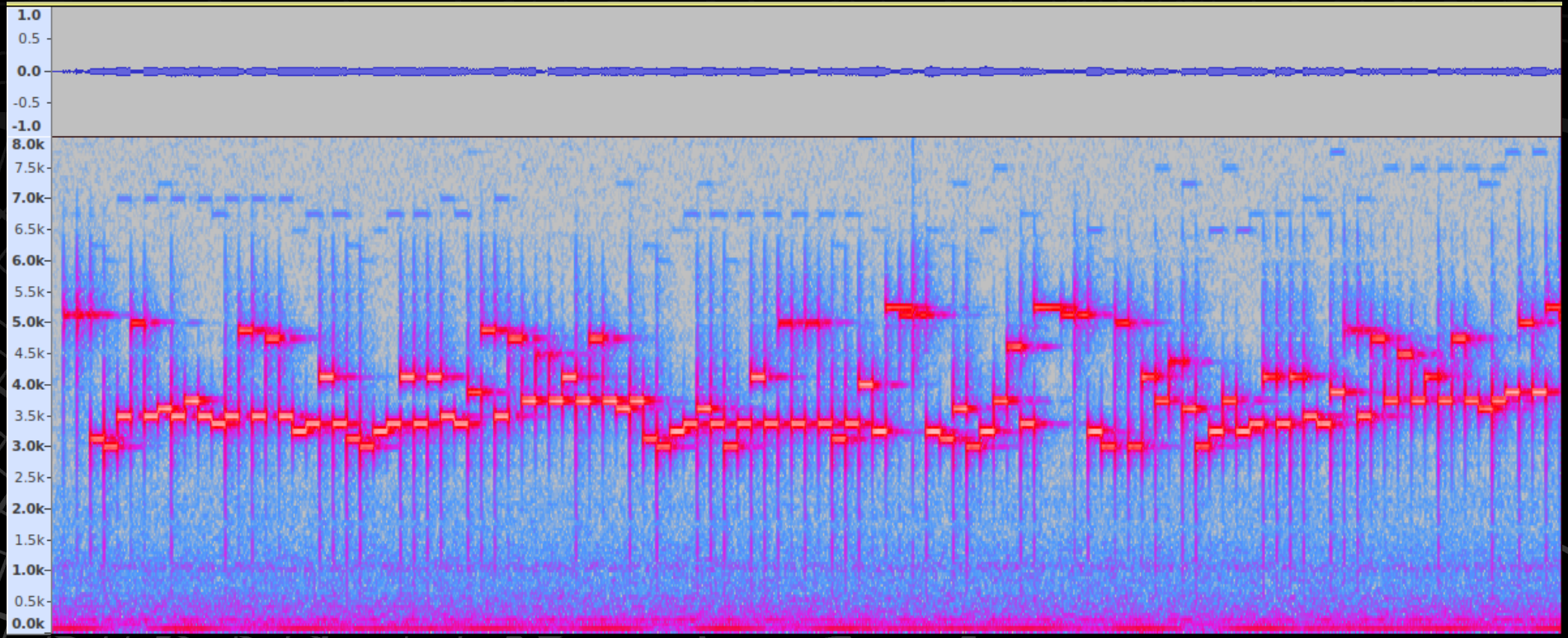
Replay & Variation

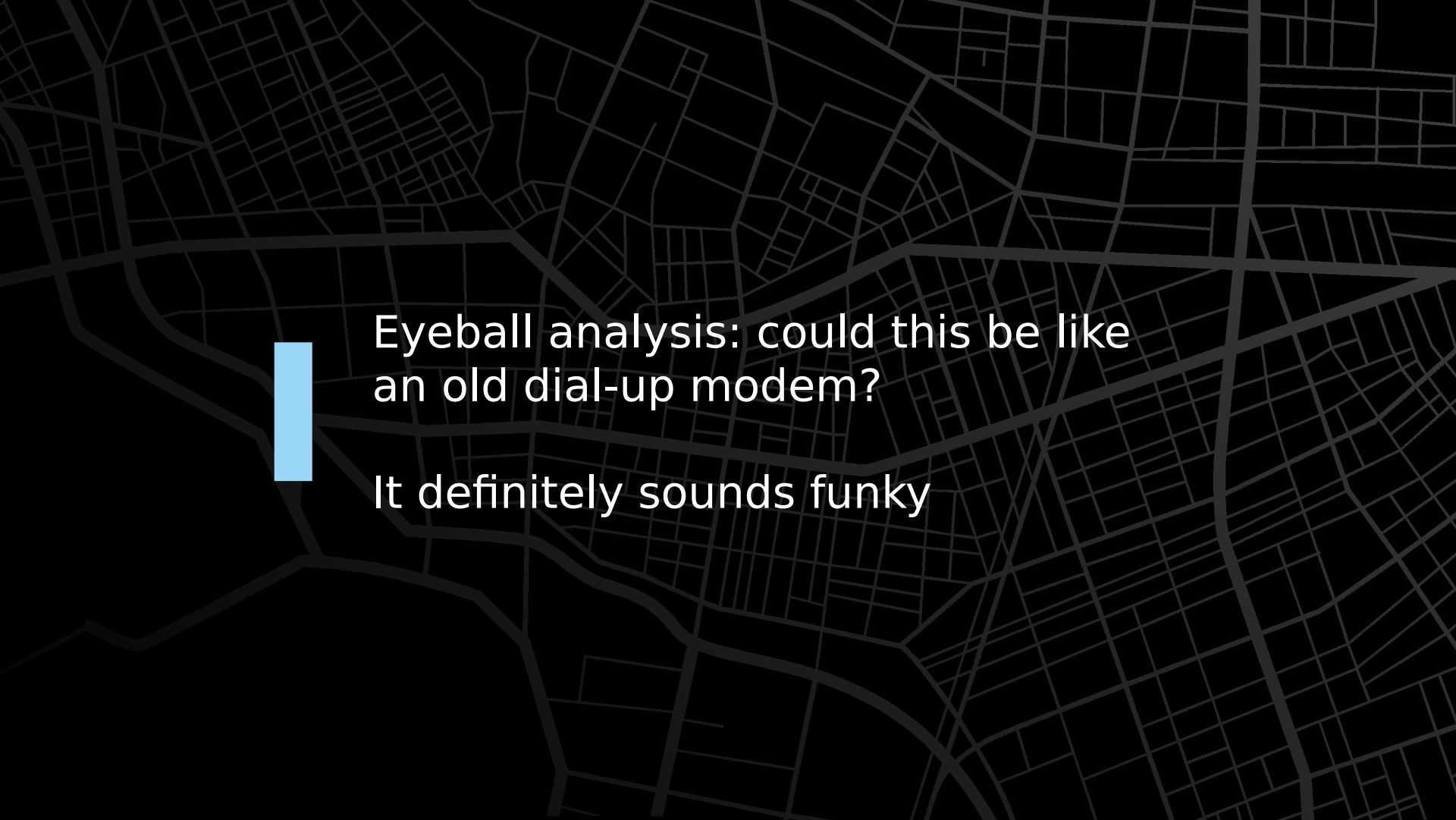
What differences stand out? Any outliers? Patterns where variated output is observed?

Fuzzing / Simulation

Take control of the input and examine the output

Capture viewed in Audacity

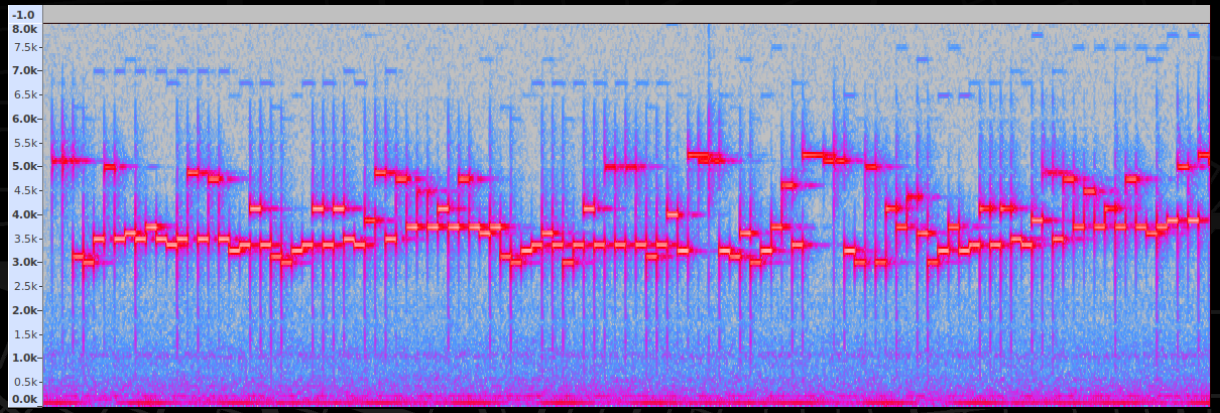




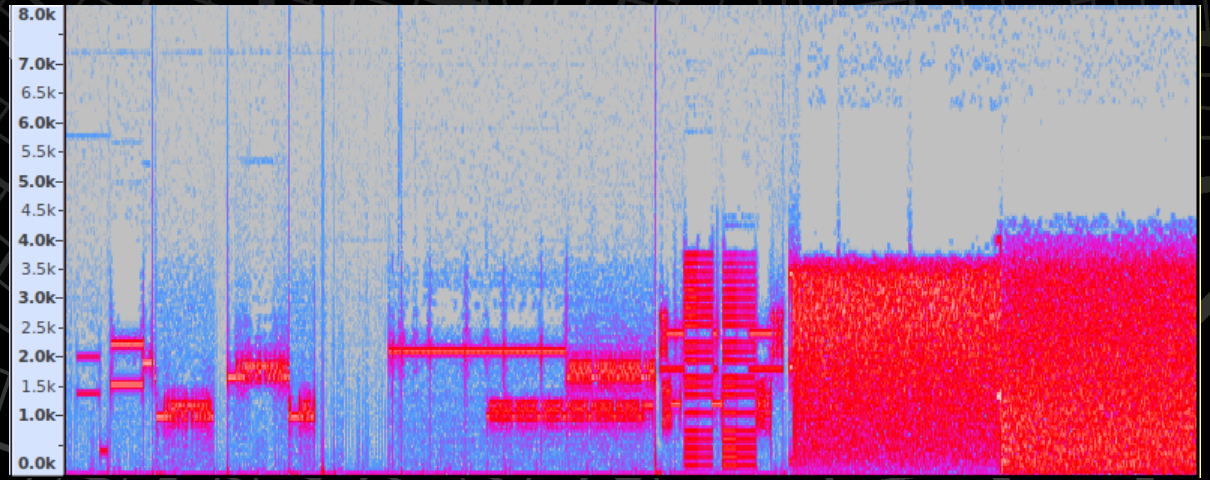
Eyeball analysis: could this be like an old dial-up modem?

It definitely sounds funky

Confirmed: Not a Modem



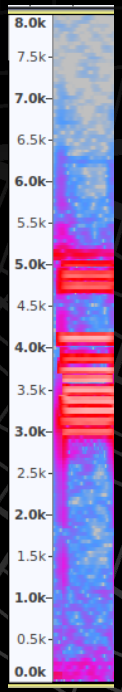
Camera Comm Signal

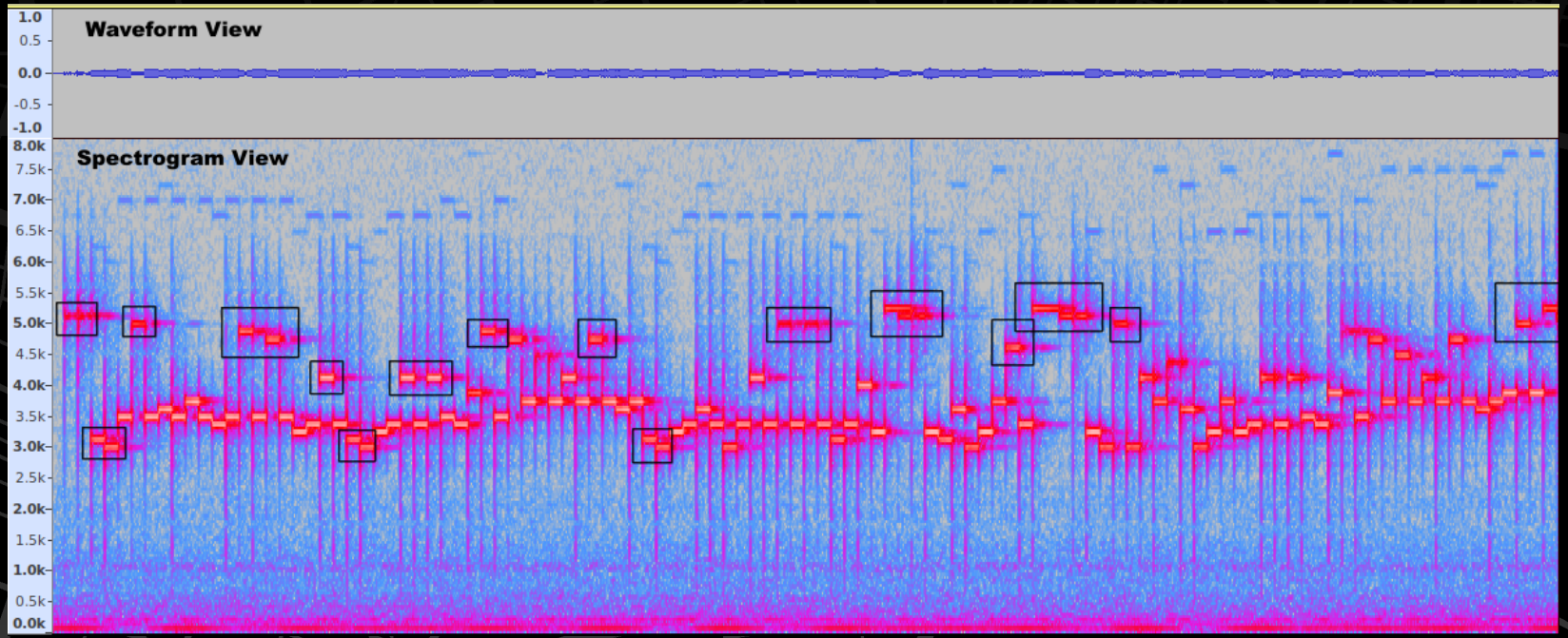


56k Modem

Spectrographs are substantially different

Eyeball analysis



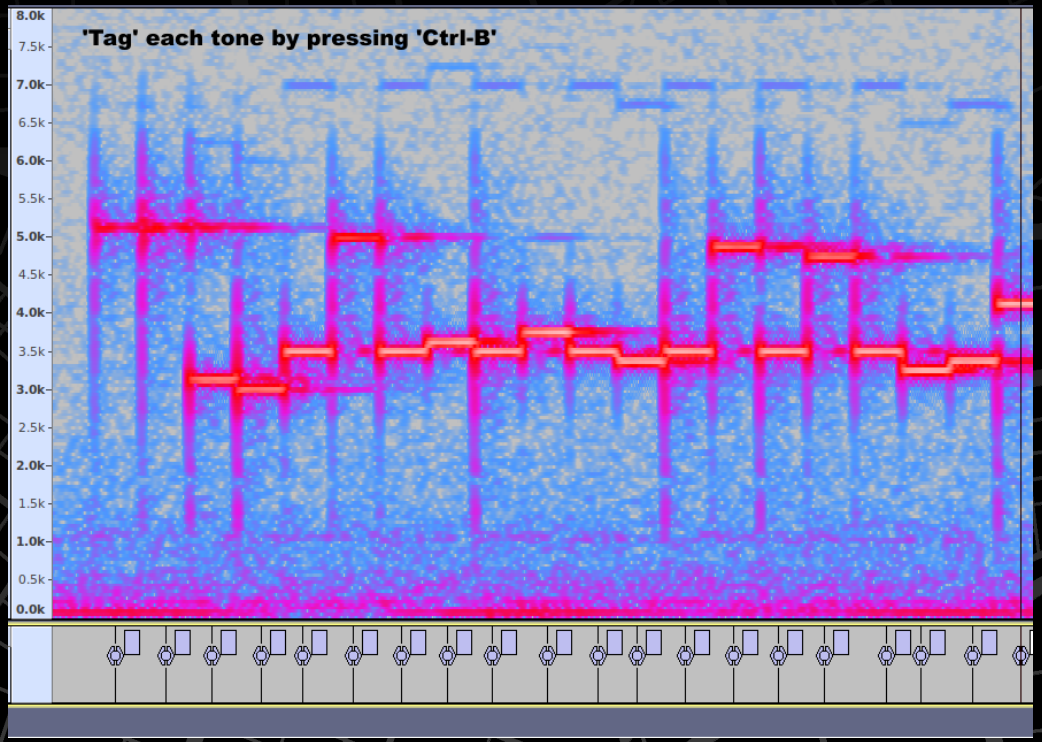


- Collapsed-Spectrograph view shows output in 3000Hz → 5000Hz range
- Quick eyeballing of the graph shows a few areas of interest

Can we get more precise?

Pretty pictures can only take us so far

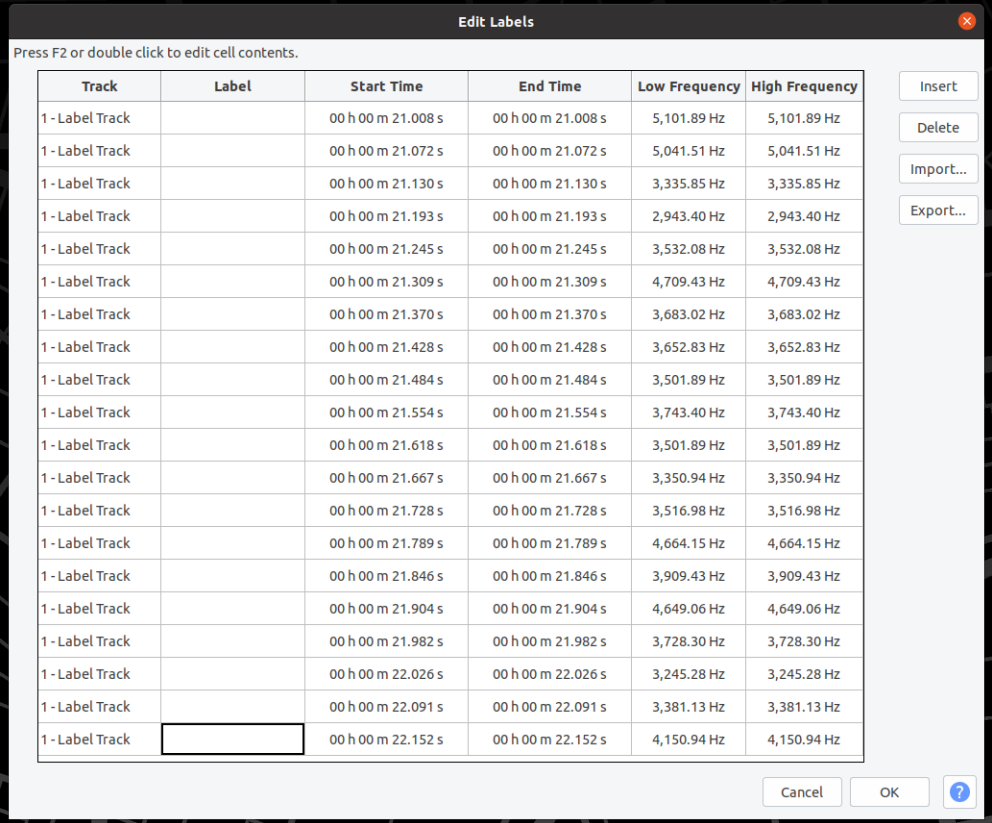
Manual Mode



Use Labels in Audacity:

- Position the cursor over each frequency time-slice
- Press **Ctrl-B** to add the label
- This will cause Audacity to calculate the frequency of each tone

Manual Mode



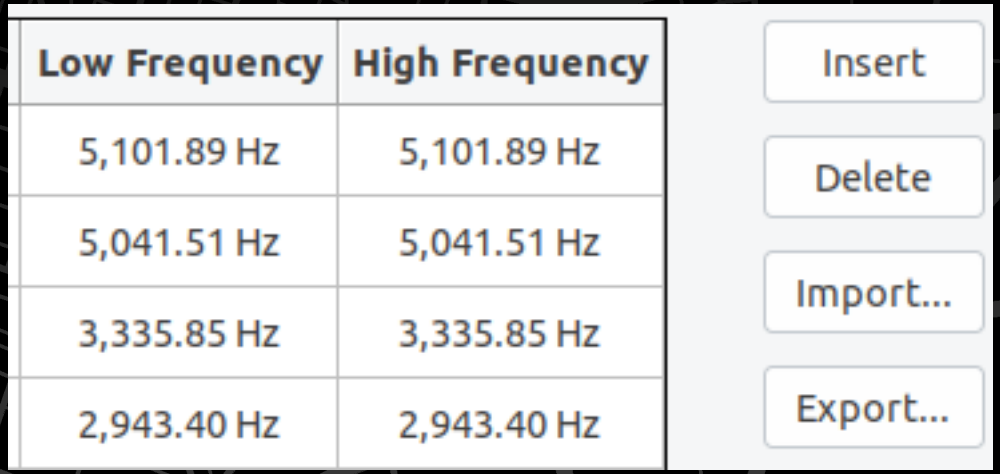
Track	Label	Start Time	End Time	Low Frequency	High Frequency
1 - Label Track		00 h 00 m 21.008 s	00 h 00 m 21.008 s	5,101.89 Hz	5,101.89 Hz
1 - Label Track		00 h 00 m 21.072 s	00 h 00 m 21.072 s	5,041.51 Hz	5,041.51 Hz
1 - Label Track		00 h 00 m 21.130 s	00 h 00 m 21.130 s	3,335.85 Hz	3,335.85 Hz
1 - Label Track		00 h 00 m 21.193 s	00 h 00 m 21.193 s	2,943.40 Hz	2,943.40 Hz
1 - Label Track		00 h 00 m 21.245 s	00 h 00 m 21.245 s	3,532.08 Hz	3,532.08 Hz
1 - Label Track		00 h 00 m 21.309 s	00 h 00 m 21.309 s	4,709.43 Hz	4,709.43 Hz
1 - Label Track		00 h 00 m 21.370 s	00 h 00 m 21.370 s	3,683.02 Hz	3,683.02 Hz
1 - Label Track		00 h 00 m 21.428 s	00 h 00 m 21.428 s	3,652.83 Hz	3,652.83 Hz
1 - Label Track		00 h 00 m 21.484 s	00 h 00 m 21.484 s	3,501.89 Hz	3,501.89 Hz
1 - Label Track		00 h 00 m 21.554 s	00 h 00 m 21.554 s	3,743.40 Hz	3,743.40 Hz
1 - Label Track		00 h 00 m 21.618 s	00 h 00 m 21.618 s	3,501.89 Hz	3,501.89 Hz
1 - Label Track		00 h 00 m 21.667 s	00 h 00 m 21.667 s	3,350.94 Hz	3,350.94 Hz
1 - Label Track		00 h 00 m 21.728 s	00 h 00 m 21.728 s	3,516.98 Hz	3,516.98 Hz
1 - Label Track		00 h 00 m 21.789 s	00 h 00 m 21.789 s	4,664.15 Hz	4,664.15 Hz
1 - Label Track		00 h 00 m 21.846 s	00 h 00 m 21.846 s	3,909.43 Hz	3,909.43 Hz
1 - Label Track		00 h 00 m 21.904 s	00 h 00 m 21.904 s	4,649.06 Hz	4,649.06 Hz
1 - Label Track		00 h 00 m 21.982 s	00 h 00 m 21.982 s	3,728.30 Hz	3,728.30 Hz
1 - Label Track		00 h 00 m 22.026 s	00 h 00 m 22.026 s	3,245.28 Hz	3,245.28 Hz
1 - Label Track		00 h 00 m 22.091 s	00 h 00 m 22.091 s	3,381.13 Hz	3,381.13 Hz
1 - Label Track		00 h 00 m 22.152 s	00 h 00 m 22.152 s	4,150.94 Hz	4,150.94 Hz

View the Labels in Audacity:

- Click on Edit → Labels → Edit Labels...
- Audacity will show a list of labels
- This list of labels will include frequency information
- Press the Export button to get this in a text file for analysis

Manual Mode

Zoomed-view of frequency detection



What do we know now?

What do we know now?

Tone-encoded

Despite the audible-medium, the signal is digitized in coded tones

Consistent Range

Tones are in the 3Khz to 5Khz range

Control Signals…?

Outliers at the top and bottom of the spectrum warrant a closer look

Repetition

The complete sequence repeats several times

Not ‘Binary’

The signal isn’t two-toned or ‘directly’ sending 1’s and 0’s

Not 56k Modem!

Spectral analysis does not match

Can we go further with Manual Mode?

Can we go further manually?

Yes, with caveats:

- **Variability** of audio playback and capture gets old
  - You'll tire of this quickly
- It would be great to identify **exact** frequencies used
  - Audacity frequency analysis gets you 'close'
  - You'll find variation in the detected frequency depending on which tone you check
- **No API** is readily available to leverage this Audacity functionality
  - Also no readily available CLI option for us to script...

What are the manual mode options?

Black-box signal reversing

- Brute-force reproduction of tones
- Attempt to match generated tones with the spectrograph
- Fuzz/generate permutations of the signal until you find a match

We have a better way...


Next Up: Reverse-engineer the App to understand the signal protocol

App Analysis: Reversing the APK

03

We have the vendor app, so
let’s take a closer look

What are our options?

What are our options?

Execute and Log

Leverage Android debug log to picture the runtime state

Decompile

Reverse compiled code

De-Obfuscate

Attribute meaning to code that’s lost its context

String Search

Look for strings or constants to aid in our understanding

Key Methods

Obfuscation may not strip out all meaning

High-Speed Fuzz

Leverage code to automate brute-force where necessary

Preparation

Prepare Your Computer to Pull the APK

1. Enable **Developer Mode** on your Android test device
   - Go to ‘Settings’ and find the ‘Software Information’ for your device
   - Tap the **Build number** field 7 to 10 times

2. Allow **USB Debugging**
   - Go to ‘Settings’ and find ‘Developer options’ (which appears after step 1)
   - Flip the slider to enable/allow ‘USB debugging’ when USB is connected

3. Ensure you have **Android Studio** installed (includes the adb command)

4. Verify that **adb** is correctly installed and that USB debugging is propery enabled on your test device:
   - Plug in your device
   - Run this command:
     - adb shell pm list packages | tail

       (You should see a list of packages)

Extract the APK

To find and extract the camera app APK, follow these commands:

1. Open a terminal which can execute adb
2. Execute: adb shell pm list packages | awk -F':' '{print $2}' | grep jawa
3. With the package name from step 2, execute: adb shell pm path com.jawa.cam | awk -F':' '{print $2}'
4. Use output from step 3, execute: adb pull /data/app/com.jawa.cam-1/base.apk
   (Note: The path varies per-device, so use your output!)
5. Rename the APK to something usable:
   mv base.apk com.jawa.cam.apk
6. Sample Output:

```
rion@0x:~/Desktop$ adb shell pm list packages | awk -F':' '{print $2}' | grep jawa
com.jawa.cam
rion@0x:~/Desktop$ adb shell pm path com.jawa.cam | awk -F':' '{print $2}'
/data/app/com.jawa.cam-1/base.apk
rion@0x:~/Desktop$ adb pull /data/app/com.jawa.cam-1/base.apk
/data/app/com.jawa.cam-1/base.apk: 1 f.... 4.1 MB/s (44495777 bytes in 10.413s)
rion@0x:~/Desktop$ mv base.apk com.jawa.cam.apk
```

Decompile the APK: JADX

For this analysis, I used JADX

1. Visit https://github.com/skylot/jadx/releases
2. Download the latest release
3. Extract the archive
4. ./jadx ~/tmp/apk/com.jawa.cam.apk -d ~/tmp/apk/jawa-decompiled

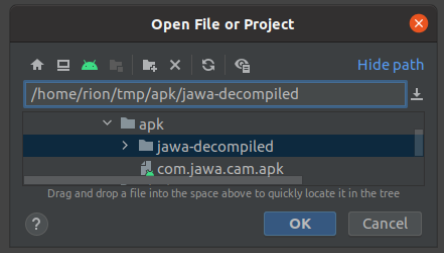
Example

```
rion@0x: ./jadx ~/tmp/apk/com.jawa.cam.apk -d ~/tmp/apk/jawa-decompiled
INFO   - loading ...
INFO   - processing ...
ERROR - finished with errors, count: 15
rion@0x:
```

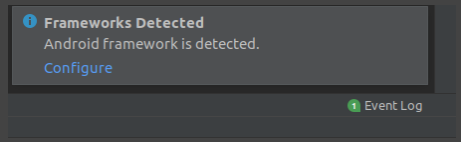
Decompile the APK: Android Studio

Load the decompiled APK into an IDE

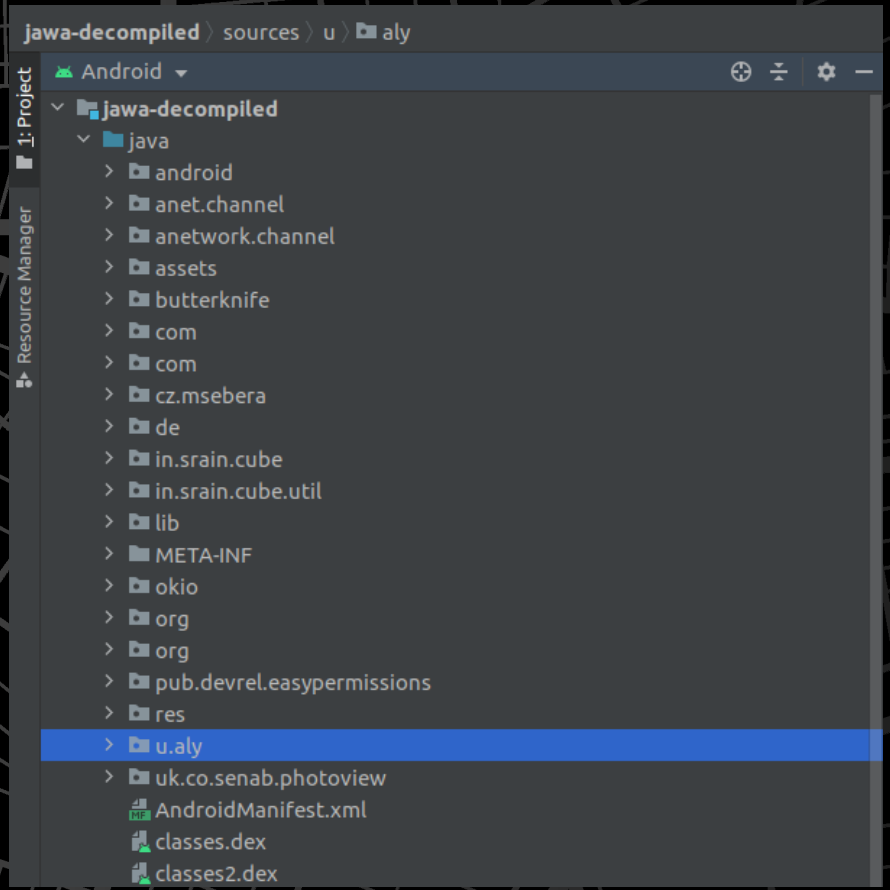
1. Ensure you have Android Studio installed
2. Select File -> Open and select your decompiled sources



3. Click the Configure button to setup the Android framework

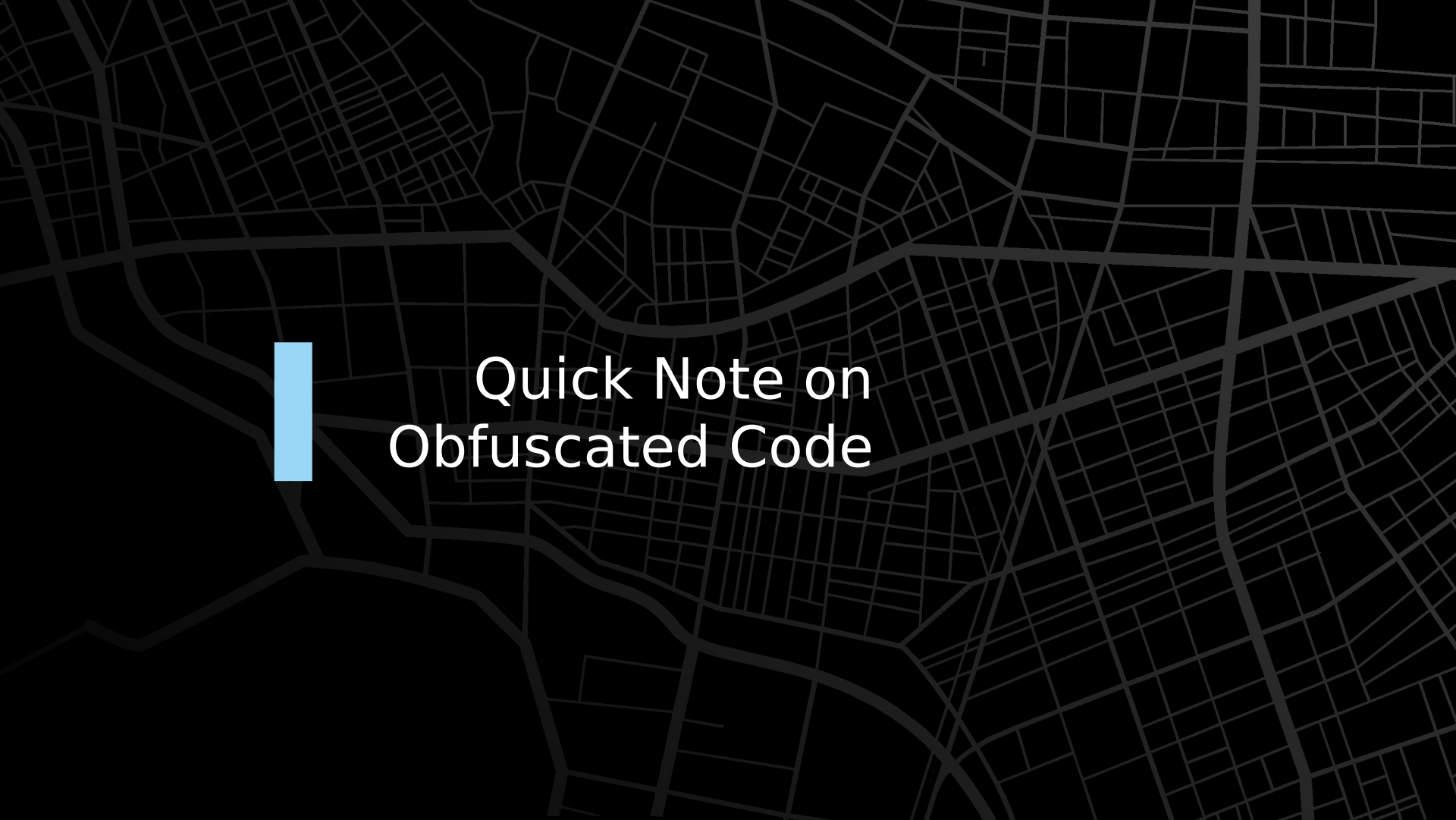


Decompile the APK: Android Studio



Behold: the loaded project!

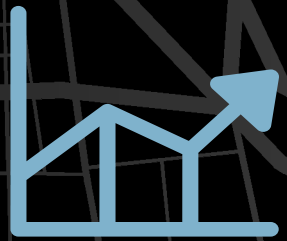
It holds the Secrets of the Grid


Quick Note on
Obfuscated Code

A Note on Obfuscation

What is obfuscation?

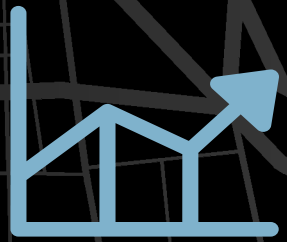
- Software makers try to hide what their software does
- With higher level languages you get terse, ‘randomly’ generated identifiers
- Harder to obfuscate the use of system libraries than their own code



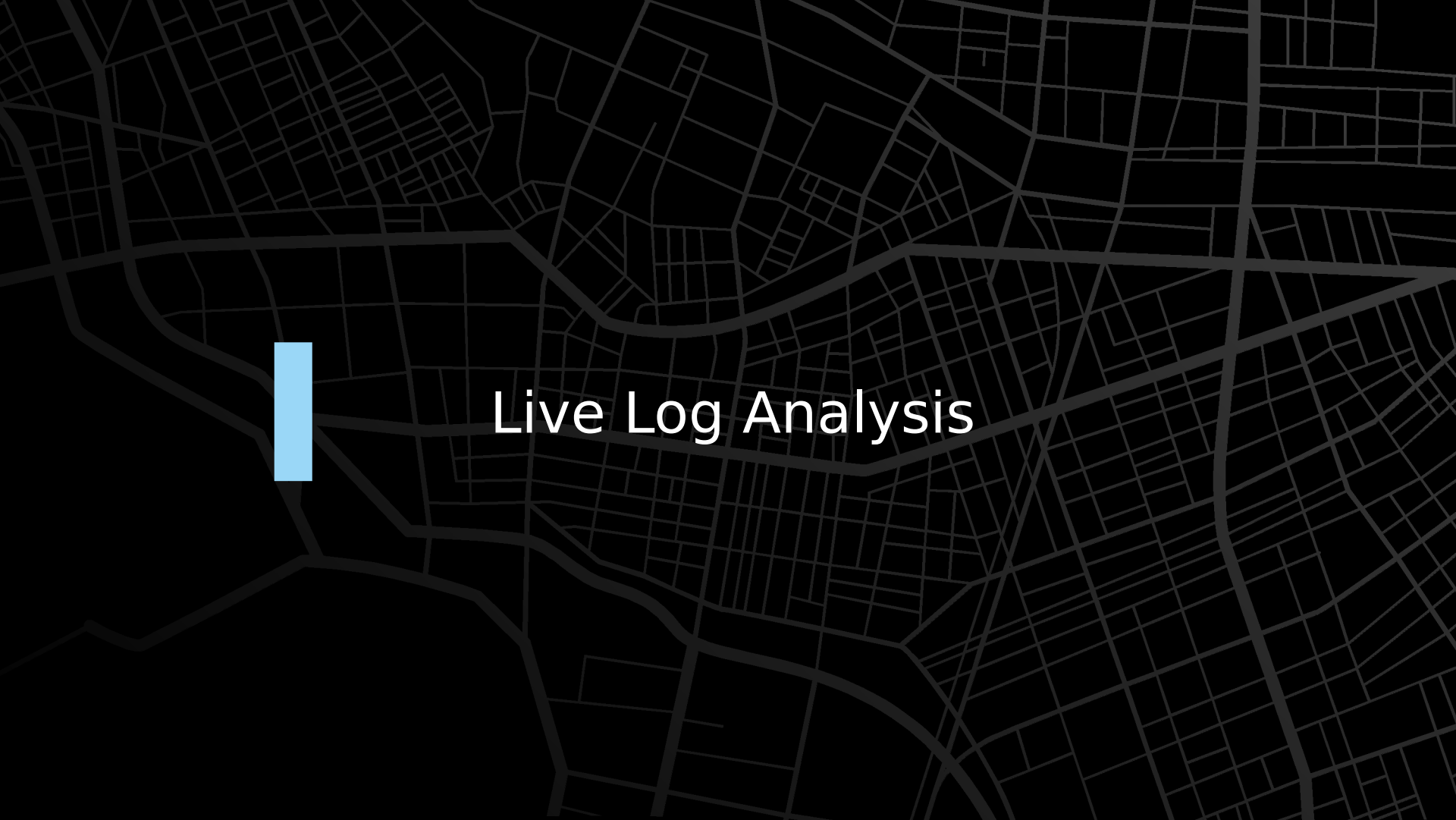
A Note on Obfuscation

Why Android Studio for de-obfuscation?

- Easily re-factor throughout the project
- Find Usages and Go To Declaration
- Integrated LogCat search bar

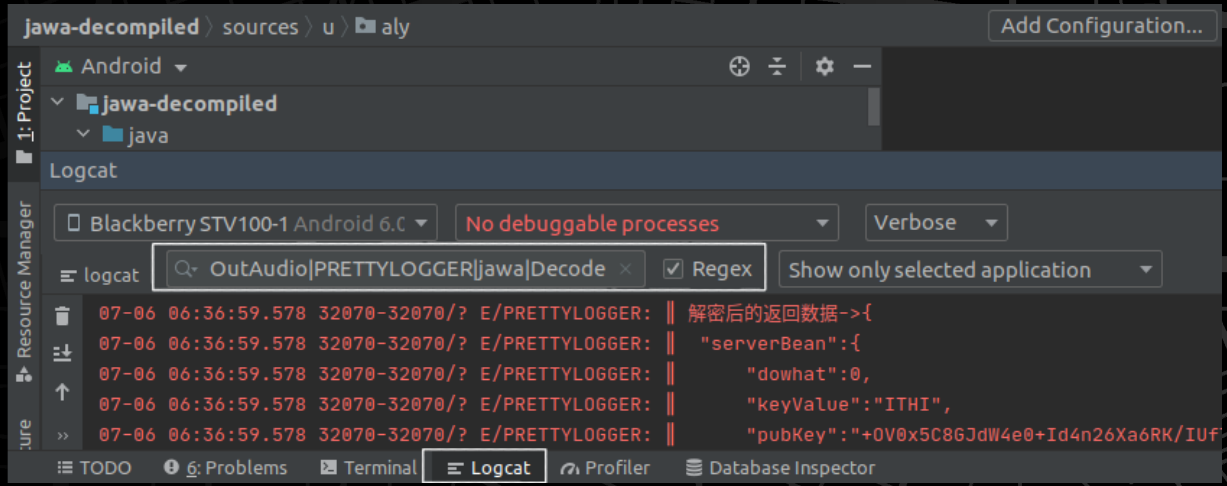


... Still a frustratingly manual proceess


Live Log Analysis

Live Log Analysis Setup

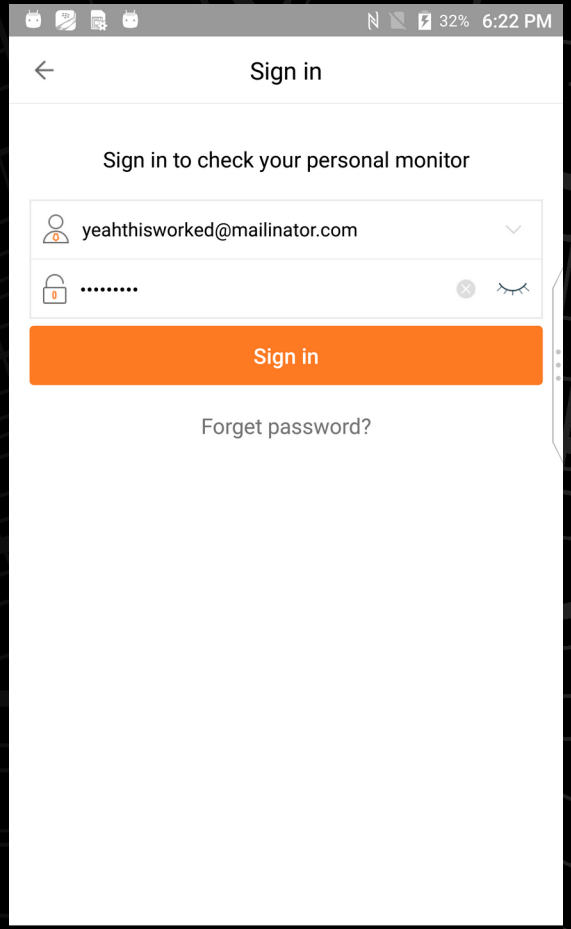
- Plug in your Android Test Device
- Select the LogCat tab in the bottom window
- Enter this in the Search field and check the Regex box:
  - OutAudio|PRETTYLOGGER|jawa|Decode




Logs on App Startup

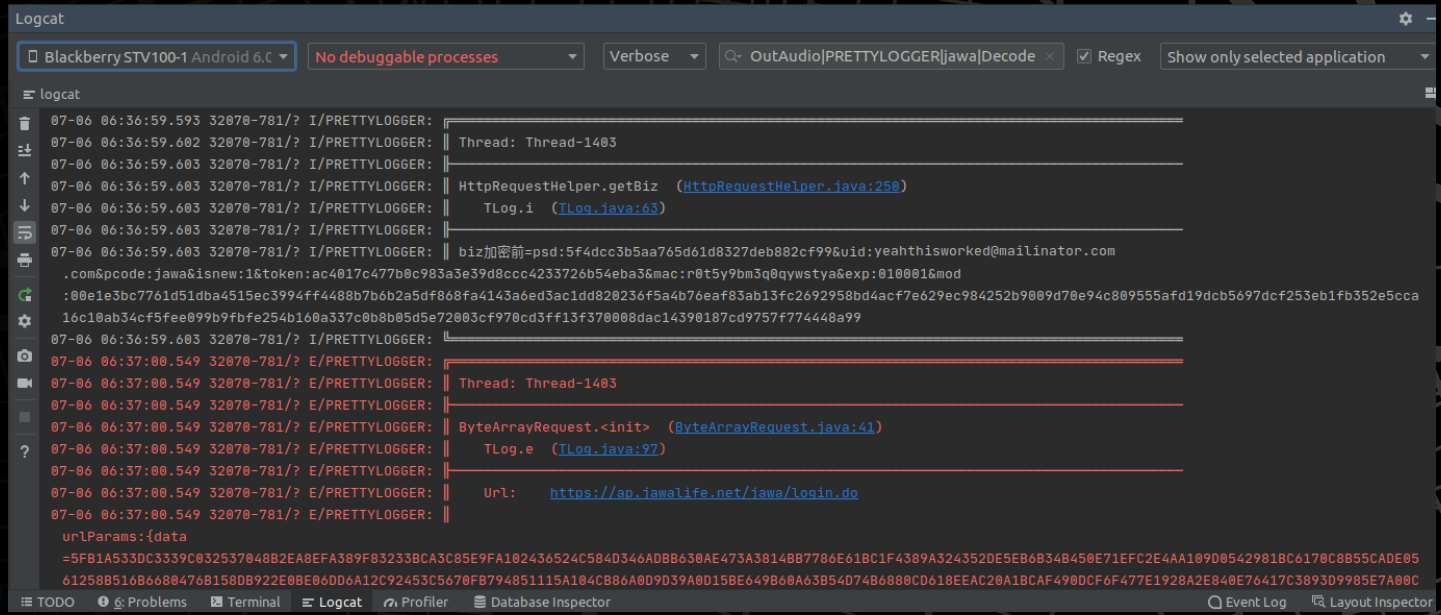
Live Log Analysis

Login Screen



Live Log Analysis: App Startup/Login

Hmm... looks interesting



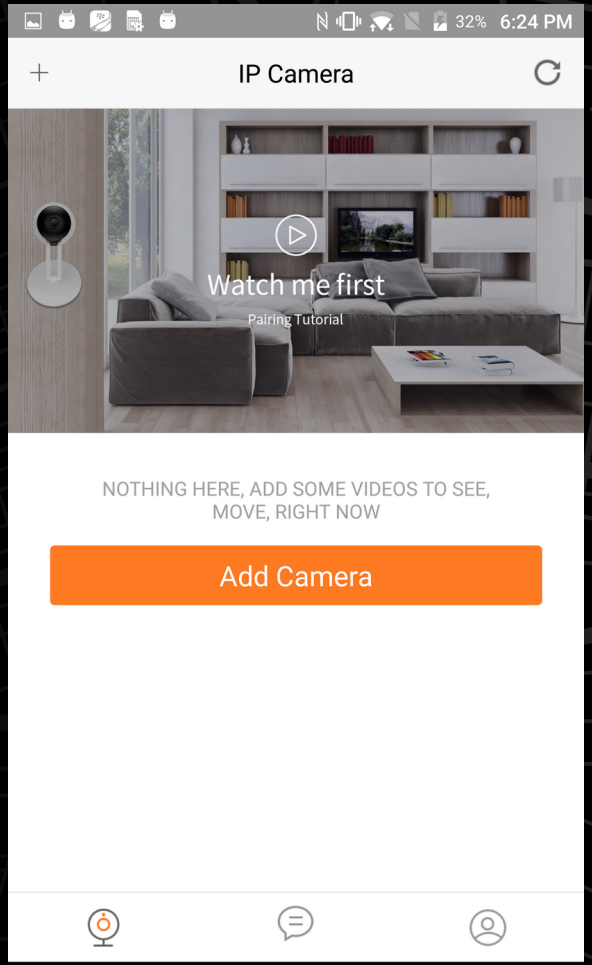
Live Log Analysis: App Startup/Login

On startup, we find log entries that could be useful:

- Distinctive characters: biz 加密前
- URL: https://ap.jawalife.net/jawa/login.do
- UrlParams with a lot of data
- And a lot more log messages off screen

Let’s see what else turns up

Logs on Camera Pairing

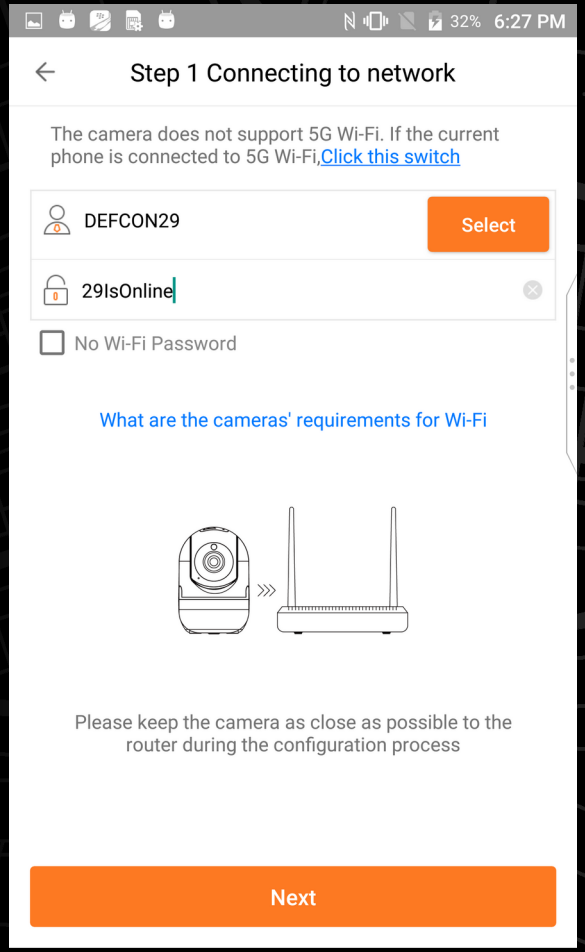


Live Log Analysis

Let’s see what we see!

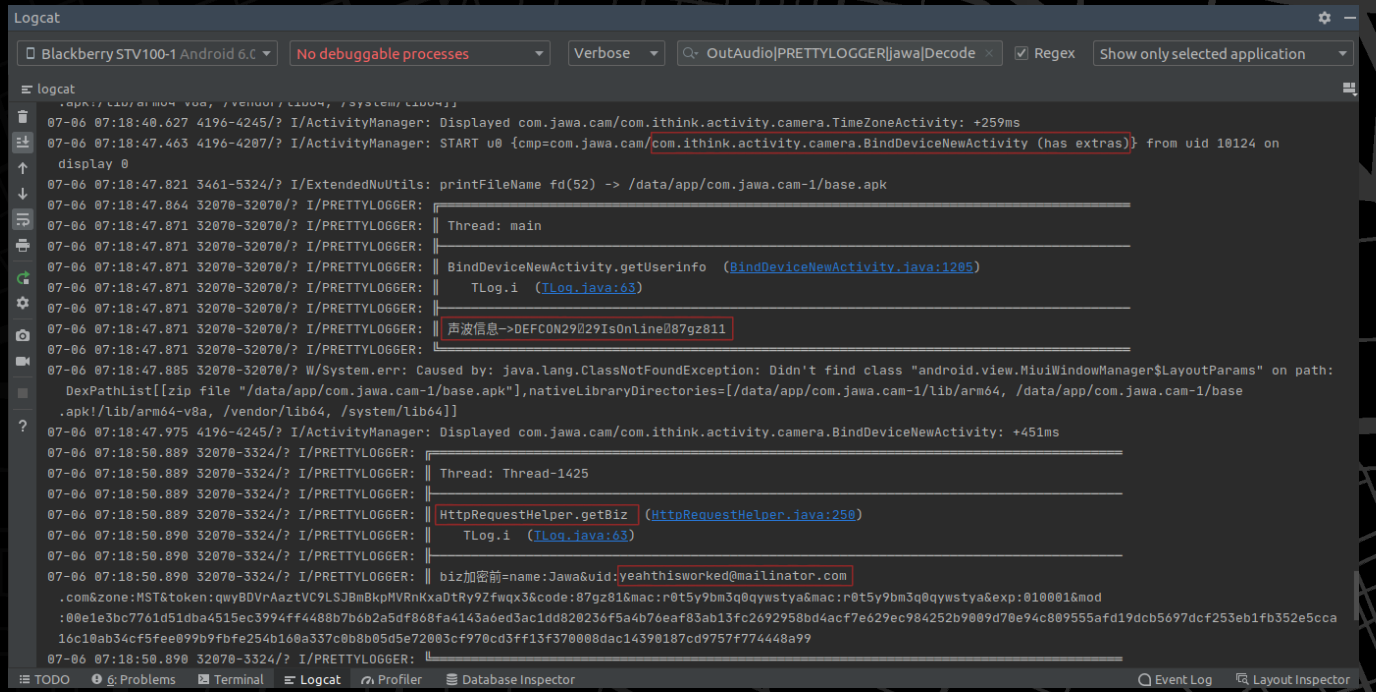
Live Log Analysis

Let’s see what we see!



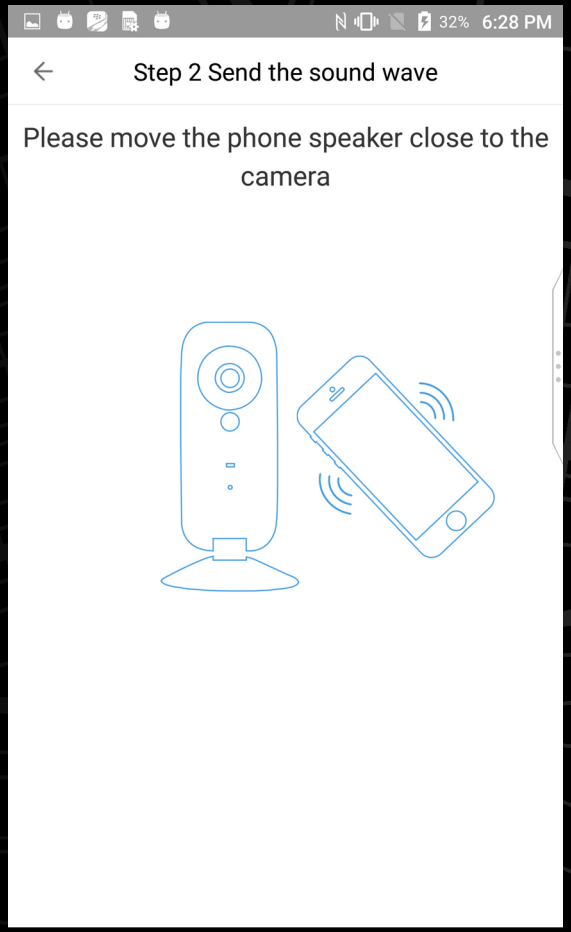
Live Log Analysis: Pairing

A few areas standout



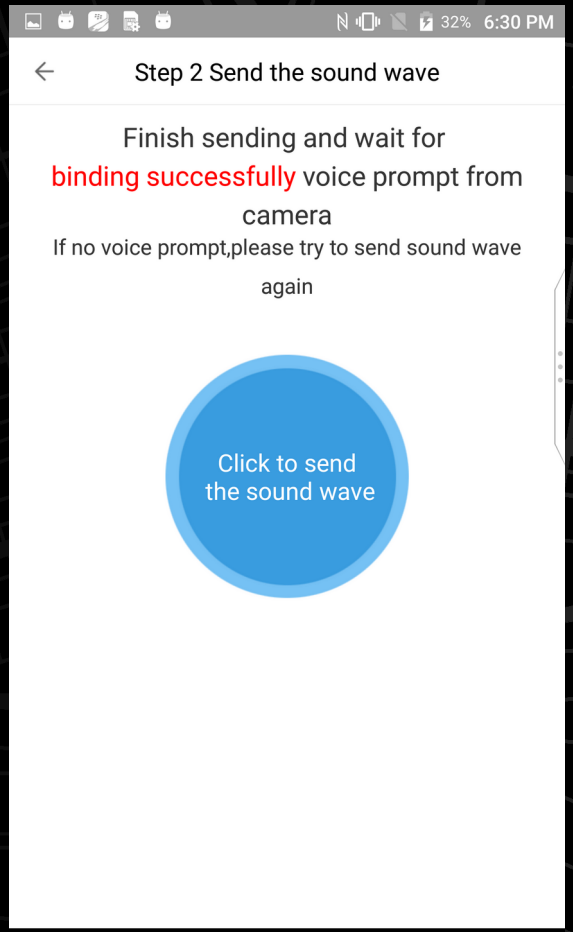
Live Log Analysis

Let’s see what we see!



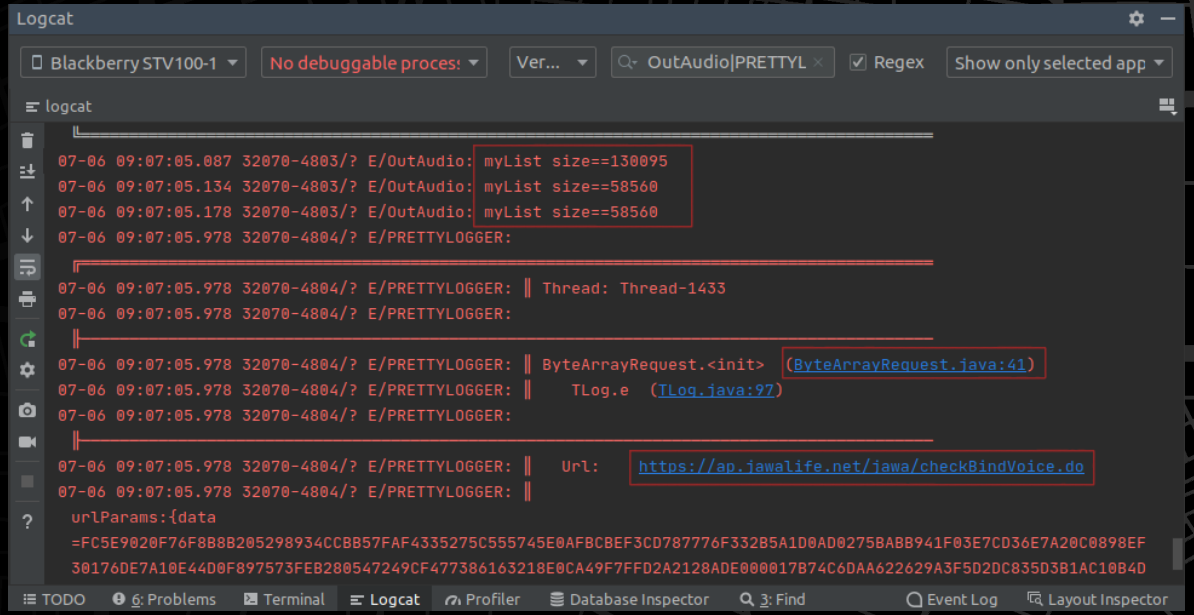
Live Log Analysis

Let’s see what we see!



Live Log Analysis: Pairing – Sound Waves

Doesn’t look like much? Think again!



Notable string: myList size

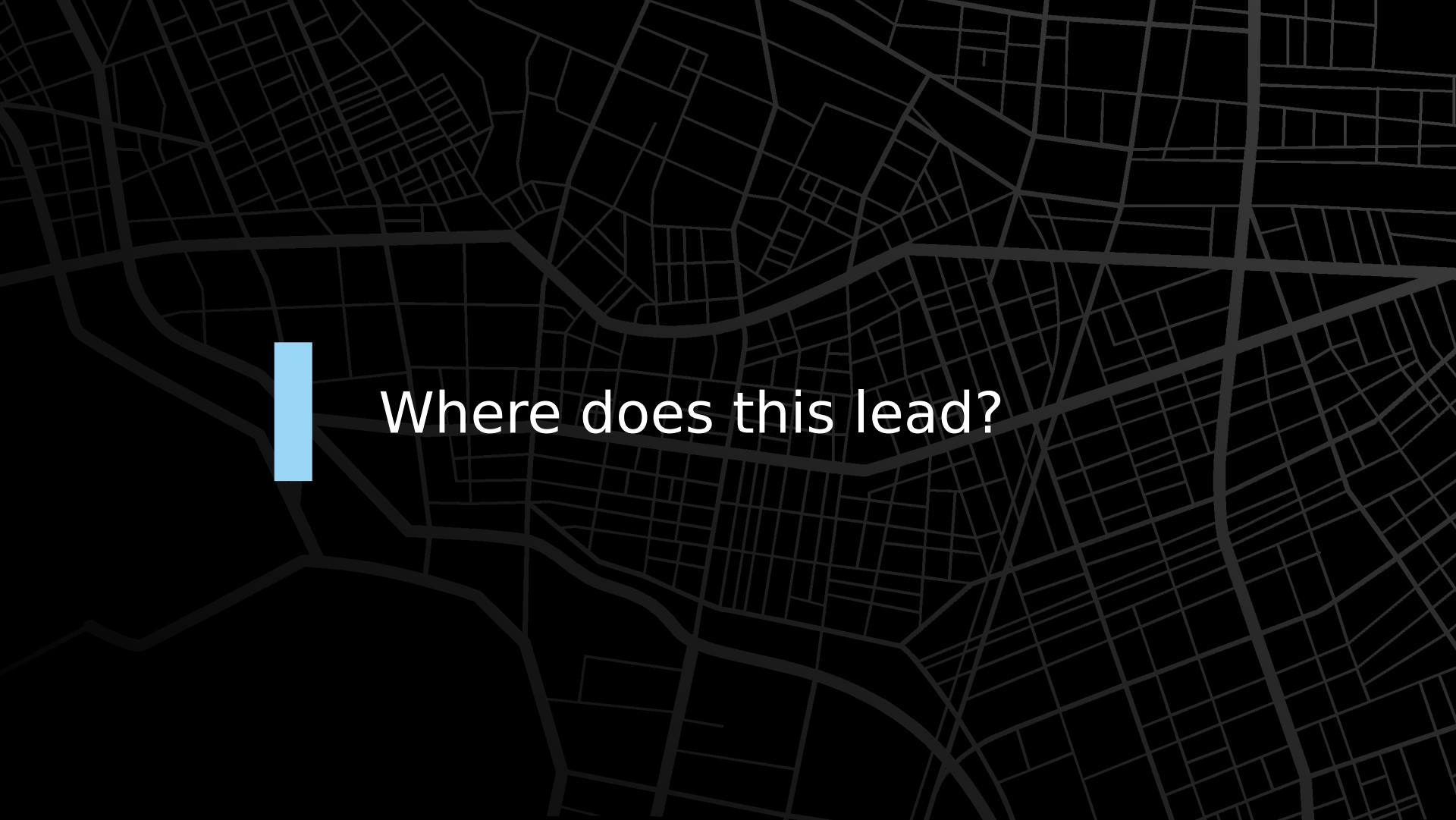
A couple of other items, probably more related to phoning home:

- ByteArrayRequest
- New URL endpoint ‘checkBindVoice.do’

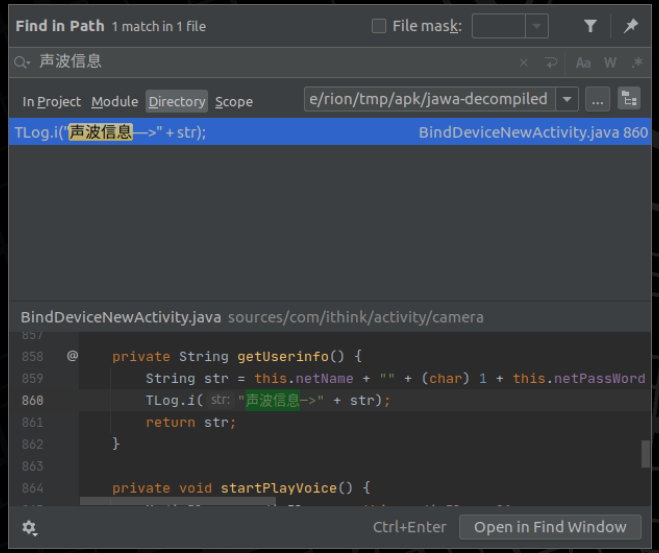
Live Log Analysis: Pairing

During the pairing process we discovered

- More distinctive characters, followed by the SSID, Password and… something else?:
  - 声波信息— >DEFCON29-29IsOnline-87gz811
- A class to investiate:
  - com.ithink.activity.camera.BindDeviceNewActivity
- An HttpRequestHelper used to phone-home with details about the pairing session
  - The request includes the email associated with my ‘cloud account’


Where does this lead?

Search for Strings



com.ithink.activity.camera.BindDeviceNewActivity

- Searching within the project for 声波信息 uncovered a method
- Aside from the expected (SSID/Password), we see:
  - 0x01 used to delimit fields
  - A smartCode (?)
  - The string “1” appended at the end (MSG_DB_NOTIFY_REACHED = “1”)

    My guess is this is a decompilation artifact

Search for Strings: What is a SmartCode?

It appears to be randomly-generated for each pairing attempt in getBundleExtras(Bundle bundle)

BindDeviceNewActivity.java | MessageService.java | HttpRequestHelper.java

```
363     /* access modifiers changed from: protected */
364     @Override // com.ithink.lib.base.BaseAppCompatActivity
365     public void getBundleExtras(Bundle bundle) {
366         this.uid = BaseApplication.getInstance().getUserId();
367         this.type = bundle.getString( key: "type");
368         this.netName = bundle.getString( key: "name");
369         this.netPassWord = bundle.getString( key: "pass");
370         this.smartCode = getCharAndNumr( i: 6);
371         this.provingNumber = getRandom( i: 0,  i2: 32767);
372         this.nickName = getString(R.string.camere_default_name);
373         this.timeZoneId = bundle.getString( key: "timeZone");
374         String str = this.timeZoneId;
375         if (str == null || str.equals("")) {
376             this.timeZoneId = TimeZone.getDefault().getID();
377         }
378     }
```

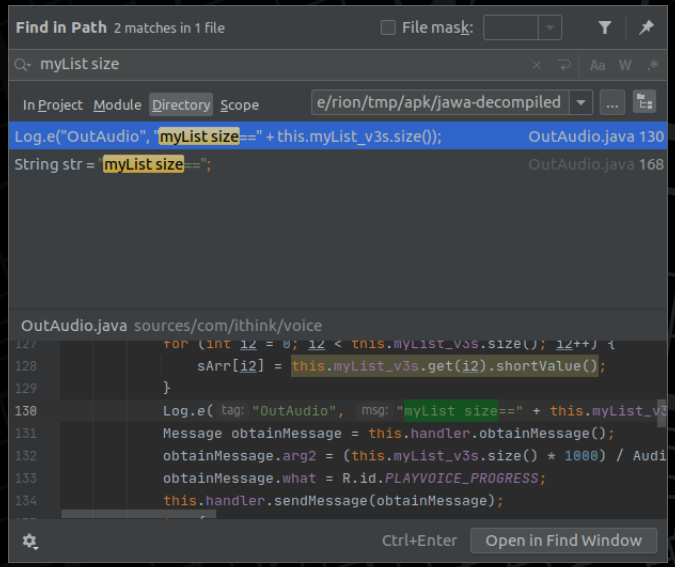
Search for Strings: What is a SmartCode?

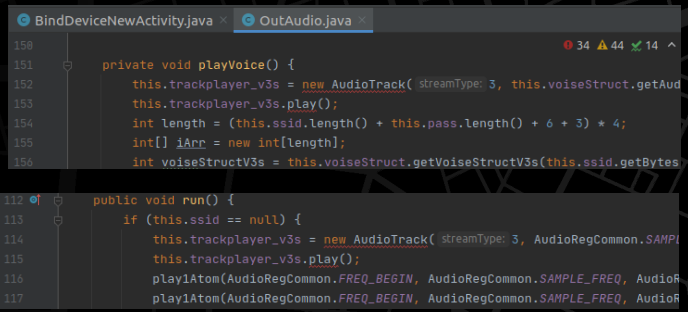
Yes, but what _is_ a smartCode?

- It’s a random code, **used by the camera when it phones home to link up to our cloud profile**
  - `this.smartCode = getCharAndNumr(6);`
  - Definition of getCharAndNumr:

```
private String getCharAndNumr(int i) {
    Random random = new Random();
    String str = "";
    for (int i2 = 0; i2 < i; i2++) {
        String str2 = random.nextInt( bound: 2) % 2 == 0 ? "char" : "num";
        if ("char".equalsIgnoreCase(str2)) {
            str = str + ((char) (random.nextInt( bound: 26) + 97));
        } else if ("num".equalsIgnoreCase(str2)) {
            str = str + String.valueOf(random.nextInt( bound: 10));
        }
    }
    return str;
}
```

Search for Strings: What Else?





Check another string from log analysis: myList size

It looks like the magic happens in these 2 methods:

- run()
- playVoice()

Extractive Analysis

Extractive Analysis

At this point we've learned a lot about the codebase

We can extract the code we need to automate our analysis and place it into a clean project

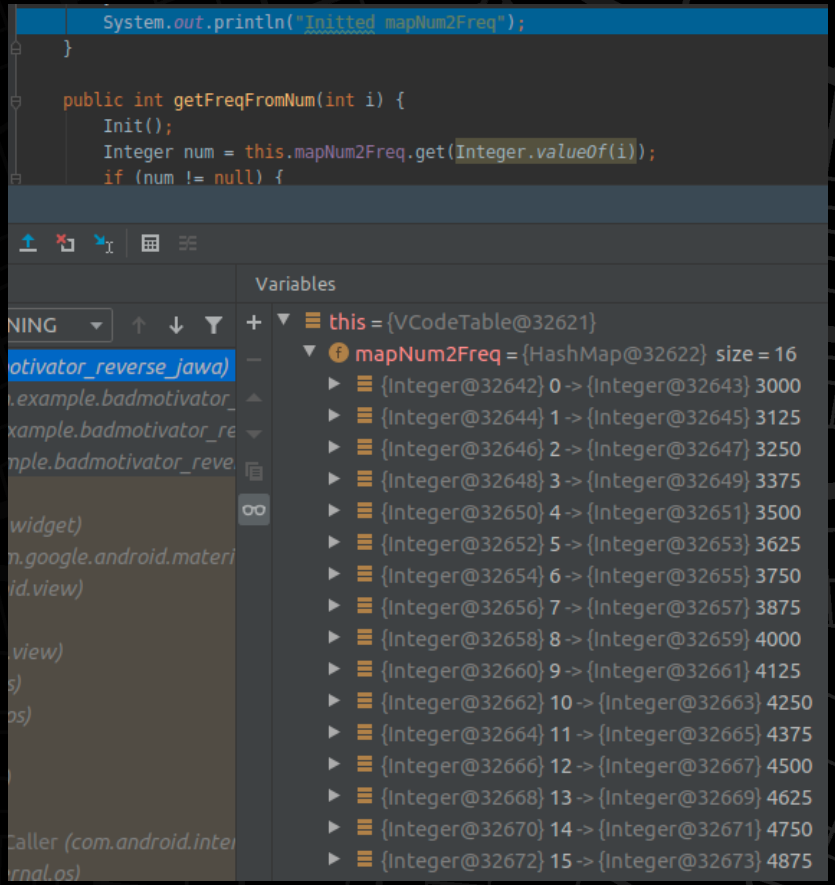
Extractive Analysis

A couple of notes on project setup

- You may need to manually create your **jniLibs** folder
  - Inside you'll need to place all the contents of the donor project's lib/jniLibs directory
- The package/class name has to match what you pull from the donor project, or it won't load

Once configured, you'll be able to automate a black-box analysis of the compiled library

A Question Answered



VCodeTable holds the key to digit/frequency mapping

We found this by extracting the class into a test project and executing it.

A Question Answered

0=3000Hz
1=3125Hz
2=3250Hz
3=3375Hz
4=3500Hz
5=3625Hz
6=3750Hz
7=3875Hz
8=4000Hz
9=4125Hz
A=4250Hz
B=4375Hz
C=4500Hz
D=4625Hz
E=4750Hz
F=4875Hz

VCodeTable holds the key to digit/frequency mapping

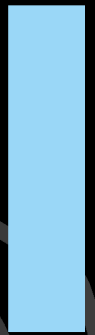
We found this by extracting the class into a test project and executing it.

Findings

Findings

Starting in com.ithink.voice.OutAudio and tracing, we find:

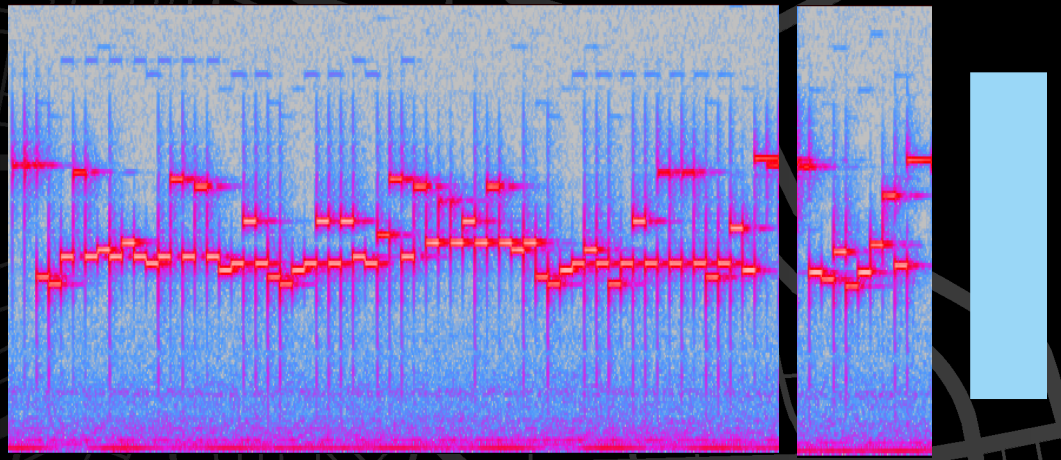
1. AudioTrack to play the signal
   - Appears to be 3 blocks to a single transmission
2. strInput == getUserinfo(), which contains:
   - SSID, Password, Random ‘smartcode’
3. Control tones:
   - AudioRegCommon.FREQ_BEGIN and FREQ_END
4. Space Tone is used to split up identical tones
   - Ensures the receiver can discern between 2 distinct tones and not one ‘long’ tone
5. play1Char which points us to:
   - VcodeTable class which maps digits to frequency tones. Looks like Hex encoding!
6. The use of CRC values
   - GetCrcVal(String str) uses a shifted CRC32 for error correction
7. An opaque VoiseStruct class that loads a library by the same name
   - If we need to drill-down into this library we will require a binary decompiler



Findings

Using what we know we can:

- Reconstruct Section 1
- Reconstruct Section 2



Section 1

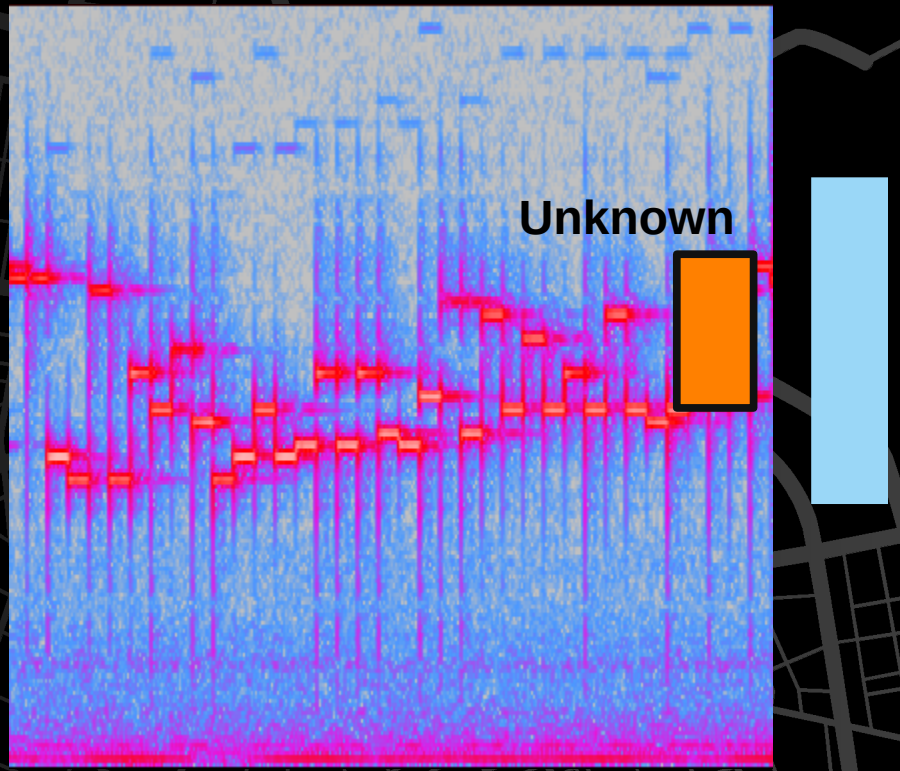
Section 2

Findings

Section 3 is incomplete

1. We know everything except for one signal in section 3
   - This appears to be an error correction code of some kind(?)

To uncover the remaining secrets of section 3, we need binary analysis

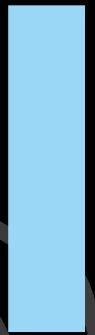


Sgnal blocks 2 and 3 are generated by VoiseStruct. Reversing compiled code requires a binary analysis tool. Ghidra is the tool I’ll be using

Get Setup with Ghidra

Get Setup with Ghidra

- Visit https://github.com/NationalSecurityAgency/ghidra/releases
  - Pull the latest version
- Follow the installation guidance for your platform:
  - https://github.com/NationalSecurityAgency/ghidra#install



Get Setup with Ghidra

- Open a new project
  - File -> New Project -> Fill out wizard
  - Click the 
- Import the library:
  - File -> Import File
  - jawa-decompiled/resources/lib/x86_64/libvoisestruct.so
  - Import with default options
  - Select Yes to analyze the file

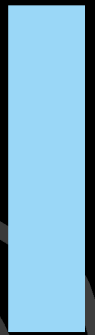
RE With Ghidra

RE With Ghidra

Identify which Library functions are used

- From within Android Studio we find these functions are used

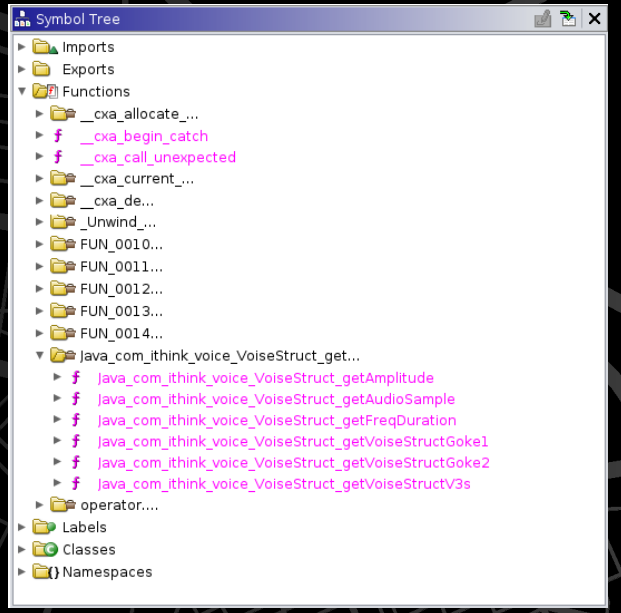
```
int voiseStructV3s = this.voiseStruct.getVoiseStructV3s
int voiseStructGoke1 = this.voiseStruct.getVoiseStructGoke1
int voiseStructGoke2 = this.voiseStruct.getVoiseStructGoke2
```



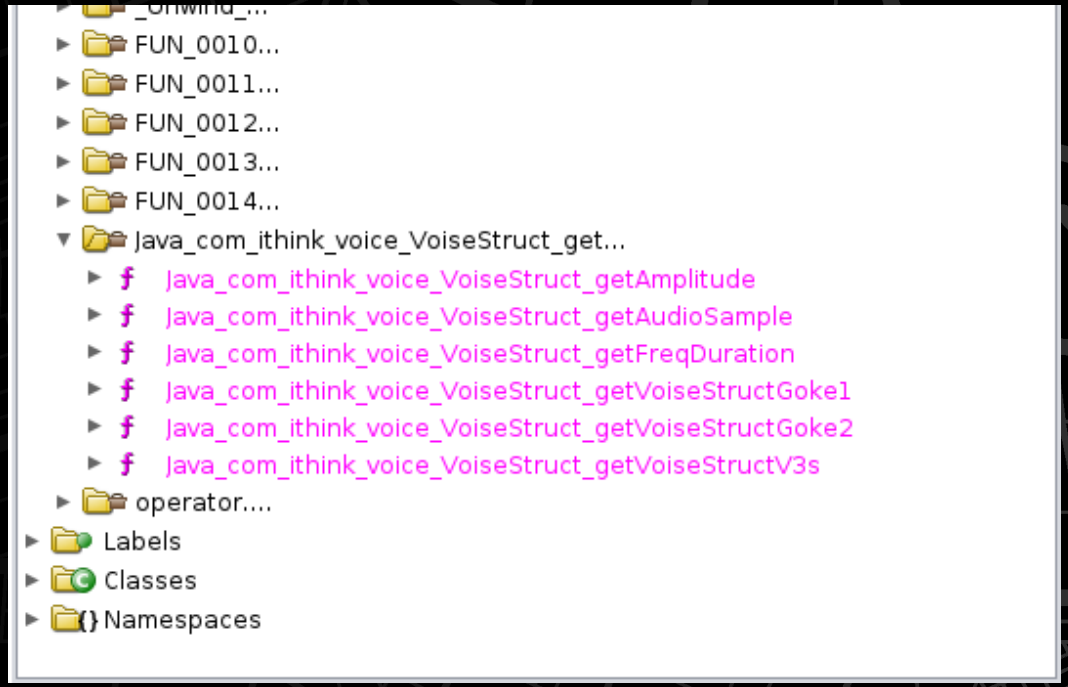
RE With Ghidra

Investigate the Library

- Examine the Symbol Tree to find the JNI functions
  - Java_com_ithink_voice_VoiseStruct_getAmplitude
  - Java_com_ithink_voice_VoiseStruct_getAudioSample
  - Java_com_ithink_voice_VoiseStruct_getFreqDuration
  - Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke1
  - Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2
  - Java_com_ithink_voice_VoiseStruct_getVoiseStructV3s



RE With Ghidra



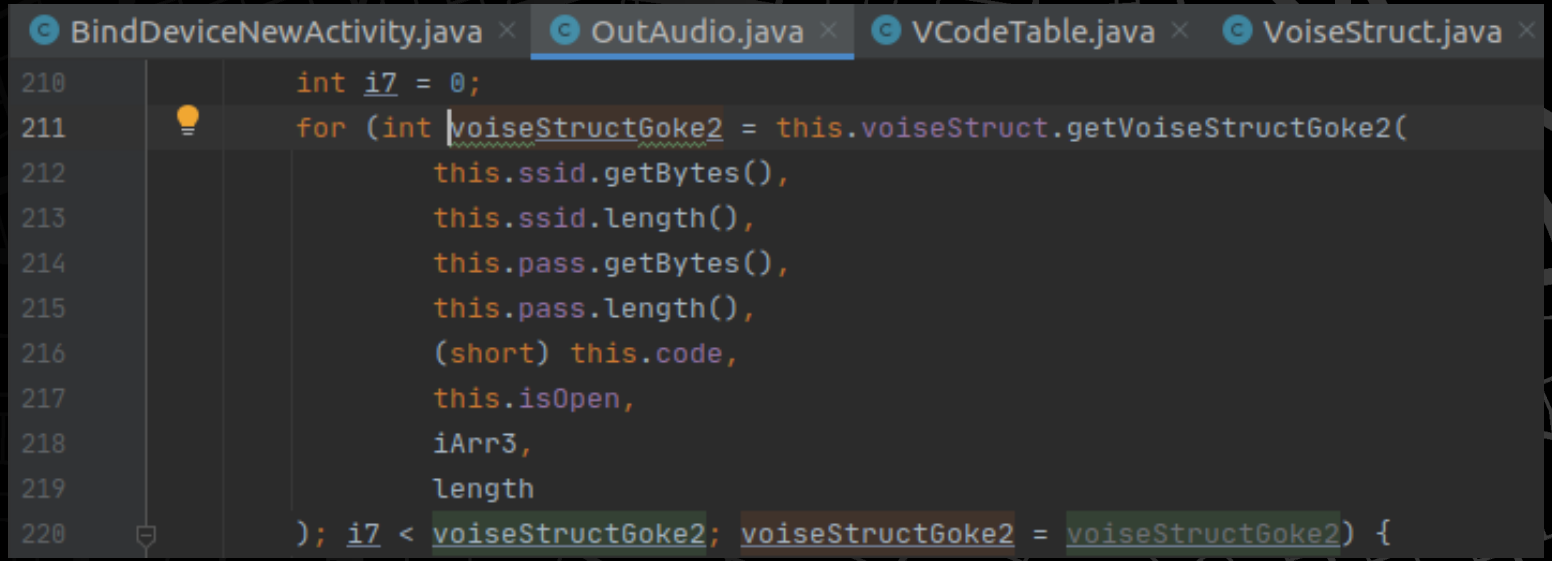
A Closer Look at what we see in Ghidra

Pick a Function and Dig In

Ghidra: Analyze a Function

Key function: getVoiseStructGoke2

- Generates Signal sections 2 and 3
- Here's what we see in Android Studio:
  - 8 arguments passed to getVoiseStructGoke2



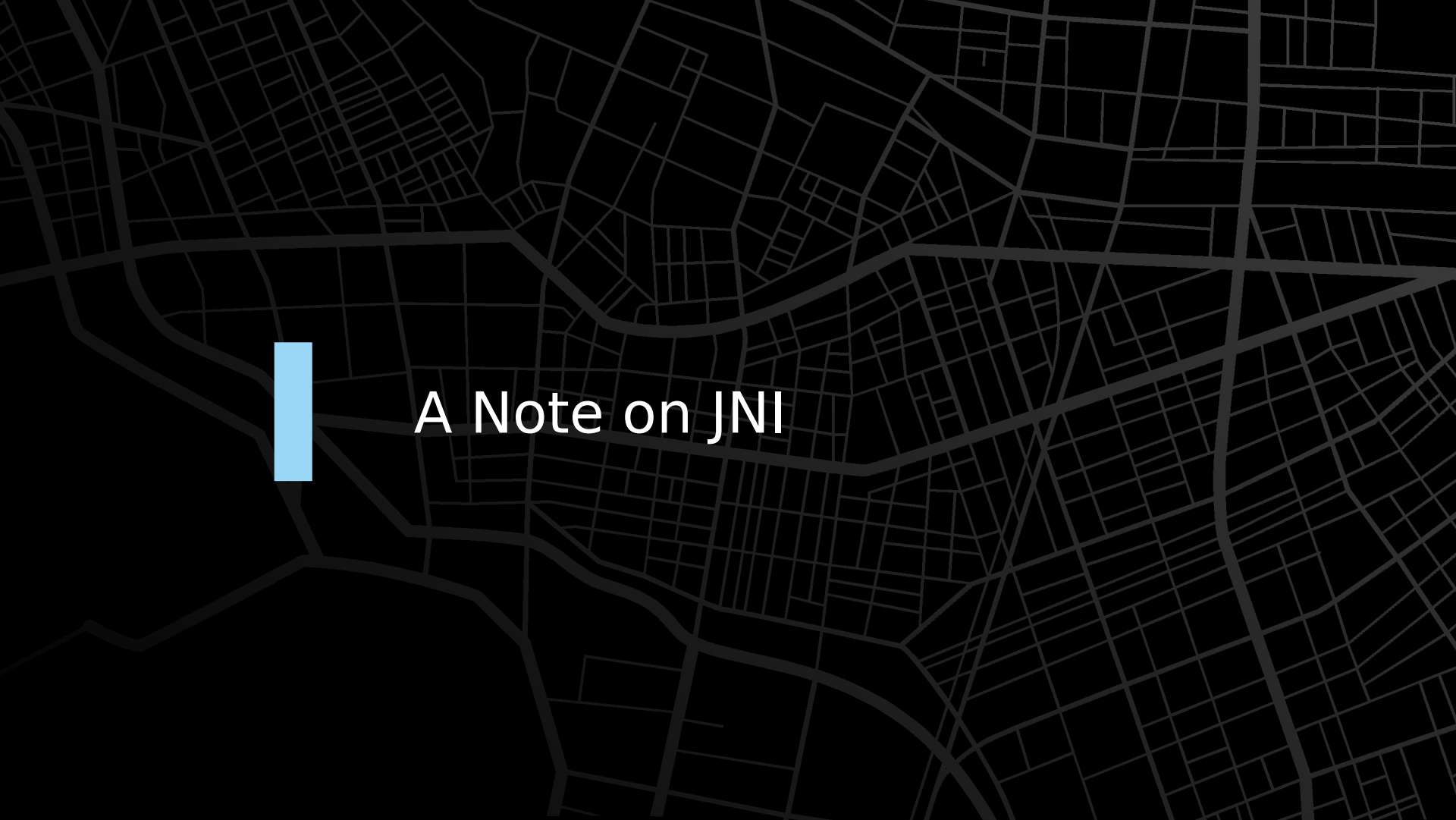
Ghidra: Analyze a Function

Ghidra: getVoiseStructGoke2

- Appears to be a parameter count mis-match. This is related to JNI
  - 8 params passed from Java
  - 10 params in Ghidra

Decompile: Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2 - (libvoisestruct...

```
1
2  undefined4
3  Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2
4            (long *param_1,undefined8 param_2,undefined8 param_3,undefined4 param_4,undefined8 param_5
5            ,undefined4 param_6,short param_7,undefined4 param_8,undefined8 param_9,
6            undefined4 param_10)
7
8  {
9    undefined4 uVar1;
10   undefined8 uVar2;
11   undefined8 uVar3;
12   undefined8 uVar4;
13   long in_FS_OFFSET;
14   undefined local_100 [200];
15   long local_38;
16
17   local_38 = *(long *)(in_FS_OFFSET + 0x28);
18   uVar2 = (**(code **)(*param_1 + 0x5c0))(param_1,param_3,0);
19   uVar3 = (**(code **)(*param_1 + 0x5c0))(param_1,param_5,0);
20   uVar4 = (**(code **)(*param_1 + 0x5d8))(param_1,param_9,0);
21   FUN_001092a0(local_100);
22                     /* try { // try from 0010bd7c to 0010be05 has its CatchHandler @ 0010be37 */
23   uVar1 = FUN_00109930(local_100,uVar2,param_4,uVar3,param_6,(int)param_7,param_8,uVar4,param_10);
24   (**(code **)(*param_1 + 0x600))(param_1,param_3,uVar2,0);
25   (**(code **)(*param_1 + 0x600))(param_1,param_5,uVar3,0);
26   (**(code **)(*param_1 + 0x618))(param_1,param_9,uVar4,0);
27   FUN_00109360(local_100);
28   if (*(long *)(in_FS_OFFSET + 0x28) == local_38) {
29     return uVar1;
30   }
31                     /* WARNING: Subroutine does not return */
32   __stack_chk_fail();
33 }
34
```



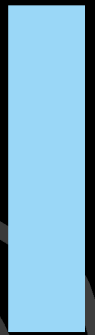
A Note on JNI

Digging Deeper: A Word on JNI

The argument-count did not line up between Android and the JNI library call

This is due to JNI Calling Conventions

- There are 2 additional arguments as part of the specification:
  - JNIEnv pointer
  - Jobject
- These 2 params are front-loaded to the function signature



Digging Deeper: A Word on JNI

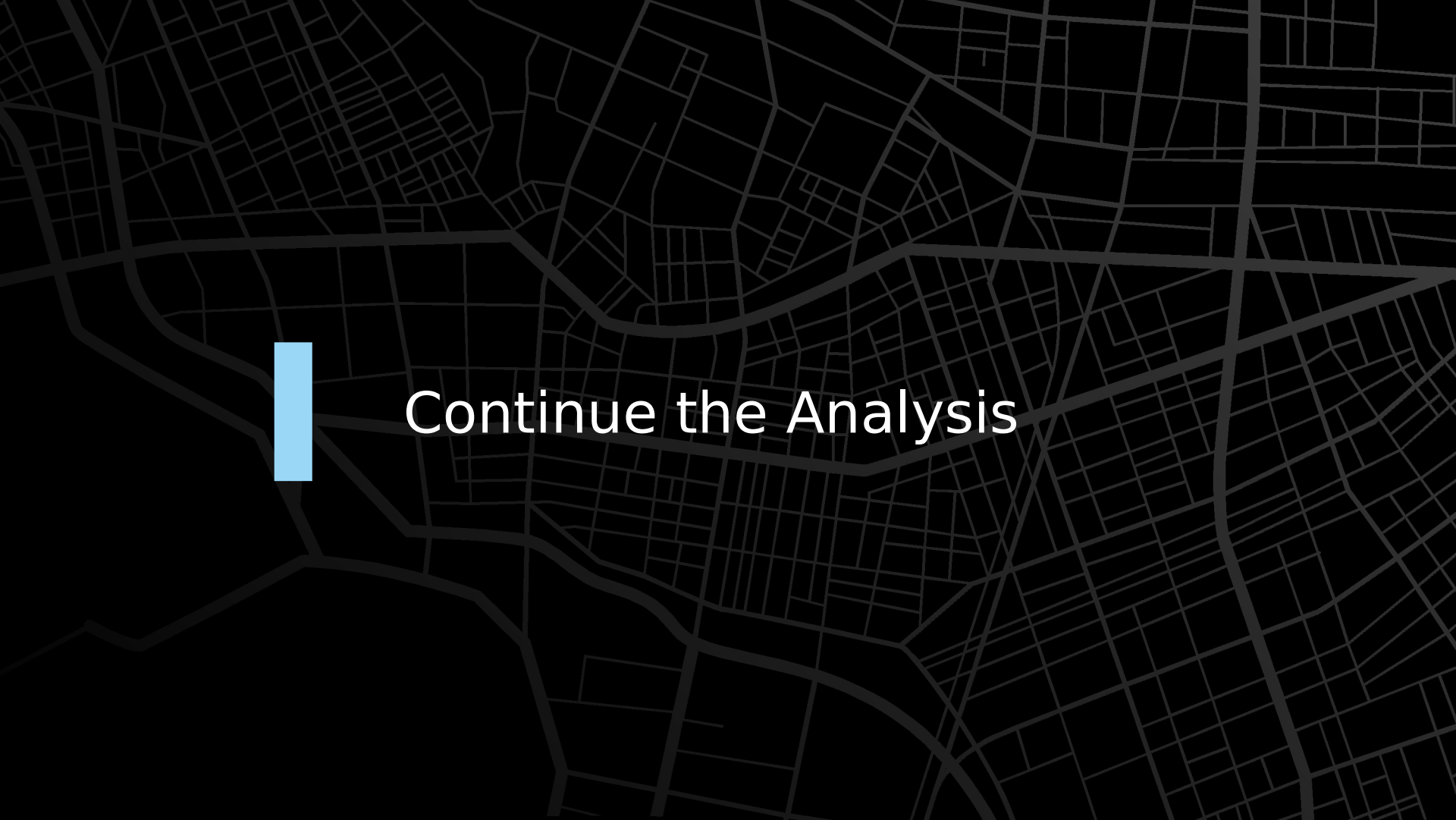
Raw decompiled view

```
undefined4
Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2
          (long *param_1,undefined8 param_2,undefined8 param_3,undefined4 param_4,undefined8 param_5
          ,undefined4 param_6,short param_7,undefined4 param_8,undefined8 param_9,
          undefined4 param_10)
```

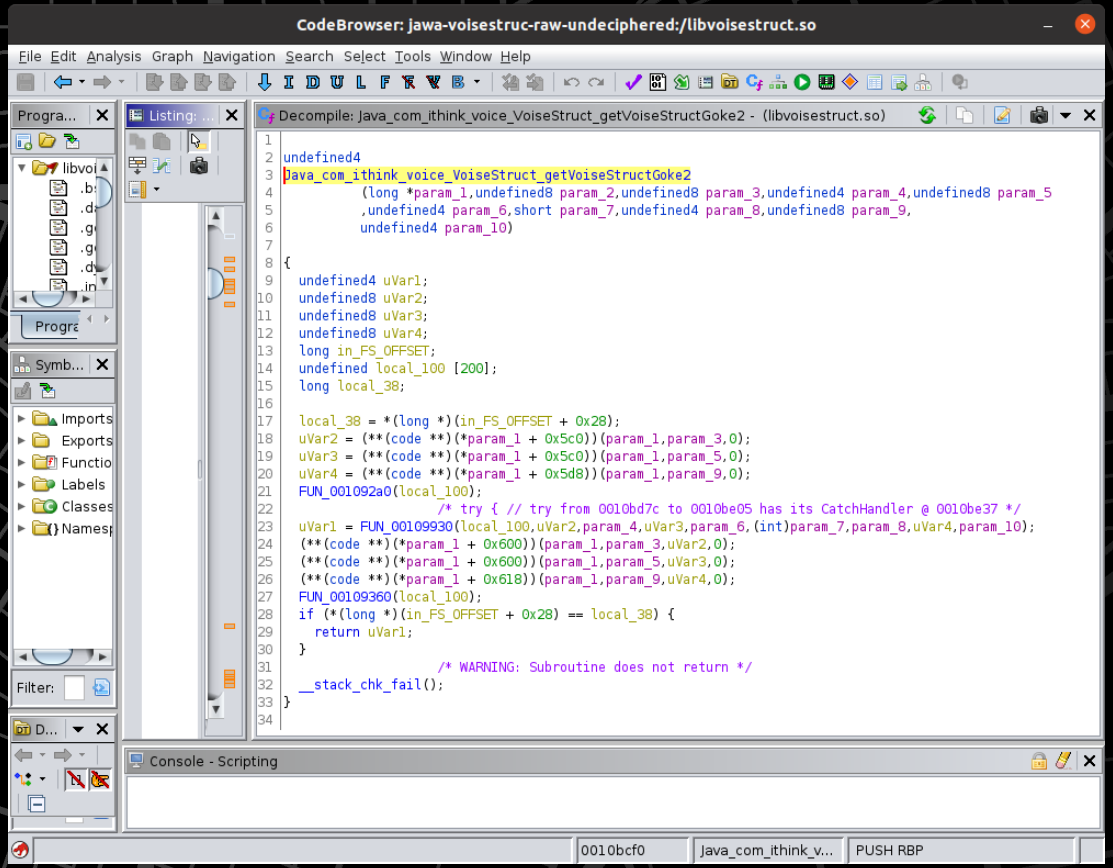
Refactored in Ghidra

```
undefined4
Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2
          [redacted]undefined8 ssidB,undefined4 ssidBLen,undefined8 passB,
          undefined4 passBLen,short smartCode,undefined4 isNetworkOpen,undefined8 returnArr,
          undefined4 returnArrLen)
```

* I couldn't rename param_2 to jobject as it's not actually used in the library call

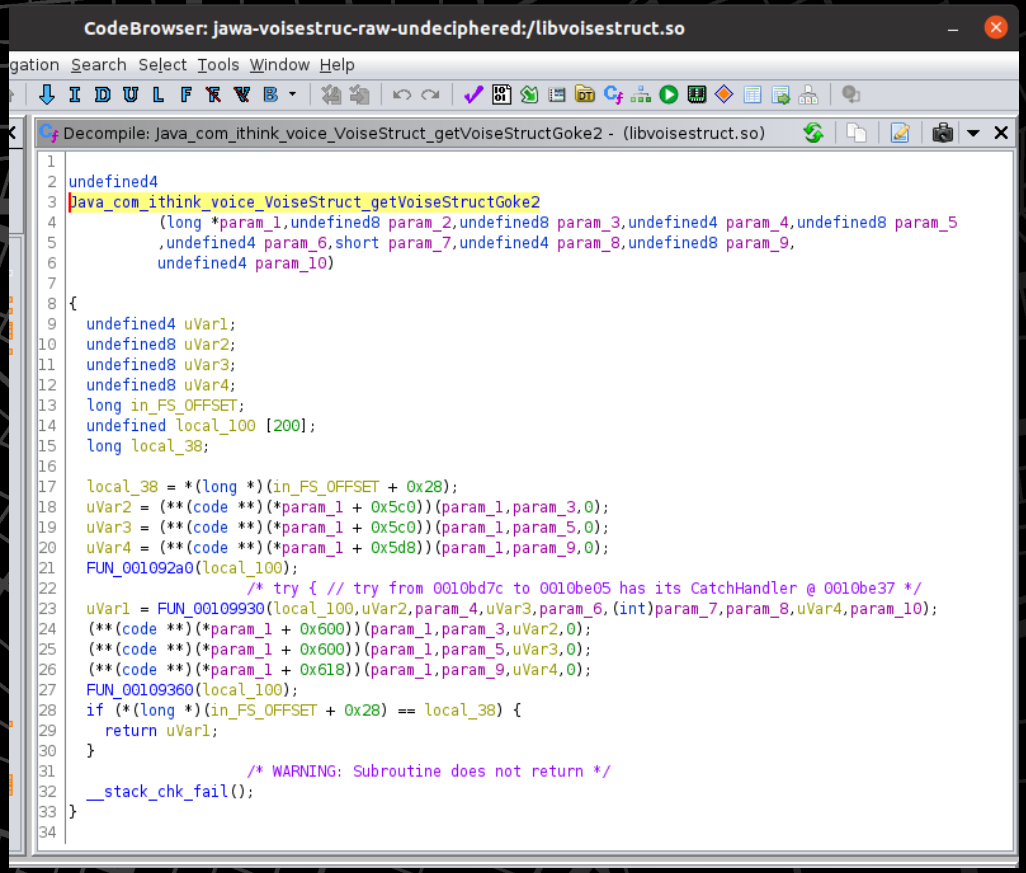

Continue the Analysis

Ghidra: Analyze a Function



Function Decompiler View

Ghidra: Analyze a Function



Continued analysis of getVoiseStructGoke2:

- FUN_00109930 leverages our inputs
- First, let's cleanup this view...

Ghidra: Analyze a Function

```
undefined4
Java_com_ithink_voice_VoiseStruct_getVoiseStructGoke2
          (long *jni,undefined8 param_2,undefined8 ssidB,undefined4 ssidBLen,undefined8 passB,
          undefined4 passBLen,short smartCode,undefined4 isNetworkOpen,undefined8 returnArr,
          undefined4 returnArrLen)

{
  undefined4 uVar1;
  undefined8 ssidBLocal;
  undefined8 passBLocal;
  undefined8 returnArrLocal;
  long in_FS_OFFSET;
  undefined local_100 [200];
  long local_38;

  local_38 = *(long *)(in_FS_OFFSET + 0x28);
  ssidBLocal = (**(code **)(*jni + 0x5c0))(jni,ssidB,0);
  passBLocal = (**(code **)(*jni + 0x5c0))(jni,passB,0);
  returnArrLocal = (**(code **)(*jni + 0x5d8))(jni,returnArr,0);
  InitReturnValues(local_100);
                    /* try { // try from 0010bd7c to 0010be05 has its CatchHandler @ 0010be37 */
  uVar1 = Section2and3Generator
                    (local_100,ssidBLocal,ssidBLen,passBLocal,passBLen,(int)smartCode,isNetworkOpen,
                     returnArrLocal,returnArrLen);
  (**(code **)(*jni + 0x600))(jni,ssidB,ssidBLocal,0);
  (**(code **)(*jni + 0x600))(jni,passB,passBLocal,0);
  (**(code **)(*jni + 0x618))(jni,returnArr,returnArrLocal,0);
  ReturnVoid(local_100);
  if (*(long *)(in_FS_OFFSET + 0x28) == local_38) {
    return uVar1;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Cleaned-up view of getVoiseStructGoke2:

- FUN_00109930 renamed to `Section2and3Generator`
- Time to dig-deeper
- We want to identify the final code that is missing from our waveform

Ghidra: Analyze a Function

Raw view of Section2and3Generator

- Kind of terse
- We need more descriptive identifiers

Ghidra: Analyze a Function

Decompile: Section2and3Generator - (libvoisestruct.so)

```
int Section2and3Generator
              (undefined8 returnArr,undefined8 ssidB,undefined4 ssidBLen,undefined *passB,
              int passBLen,undefined4 smartCode,int isOpenNet,long returnArrLen)

{
  undefined iSection3StartValCrcd;
  undefined2 ssidCrc;
  int tonesRetVal;
  int iVar1;
  size_t __n;
  undefined *passBLocal;
  long in_FS_OFFSET;
  undefined section3StartVal;
  undefined iSsidCrc;
  undefined iSsidCrcMangled;
  undefined smartCodeLocal;
  undefined smartCodeLocalShifted;
  undefined iPassBLocal [507];
  undefined section2StartVal;
  undefined iSmartCodeLocal;
  undefined iSmartCodeLocalShifted;
  undefined iSection2StartValCrcd3;
  long local_38;

  local_38 = *(long *)(in_FS_OFFSET + 0x28);
  if (isOpenNet == 1) {
    passBLen = isOpenNet;
  }
  passBLocal = &DAT_00142940;
  if (isOpenNet != 1) {
    passBLocal = passB;
  }
  memset(&section2StartVal,0,0x200);
  section2StartVal = 0x12;
  smartCodeLocal = (undefined)smartCode;
  smartCodeLocalShifted = (undefined)((uint)smartCode >> 8);
  iSmartCodeLocal = smartCodeLocal;
  iSmartCodeLocalShifted = smartCodeLocalShifted;
  iSection2StartValCrcd3 = CRC(&section2StartVal,3);
  tonesRetVal = GenerateTones(returnArr,2,&section2StartVal,4,returnArrLen);
  __n = SEXT48(passBLen);
  memset(&section3StartVal,0,0x200);
  section3StartVal = 2;
  ssidCrc = CRC(ssidB,ssidBLen);
  iSsidCrc = (undefined)ssidCrc;
  iSsidCrcMangled = (undefined)((ushort)ssidCrc >> 8);
  memcpy(iPassBLocal,passBLocal,__n);
  iSection3StartValCrcd = CRC(&section3StartVal,__n + 5 & 0xffffffff);
  iPassBLocal[__n] = iSection3StartValCrcd;
  iVar1 = GenerateTones(returnArr,2,&section3StartVal,passBLen + 6,
                        returnArrLen + (long)tonesRetVal * 4);
  if (*(long *)(in_FS_OFFSET + 0x28) == local_38) {
    return iVar1 + tonesRetVal;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Section2and3Generator deciphered

- Start with what you know
- Once you find a good starting point, the rest of symbol names fall into place
- There can be value even if you don’t get the names 100% correct
- Critical section highlighted, which is what I need to reproduce the signal
  - To accurately reproduce the signal, I need to take the CRC value of the SSID and shift it

Ghidra: Analyze a Function

Zero-in on the section: **Needs another bitshift!**

issidCrcMangled = (undefined)((ushort)ssidCrc >> 8);

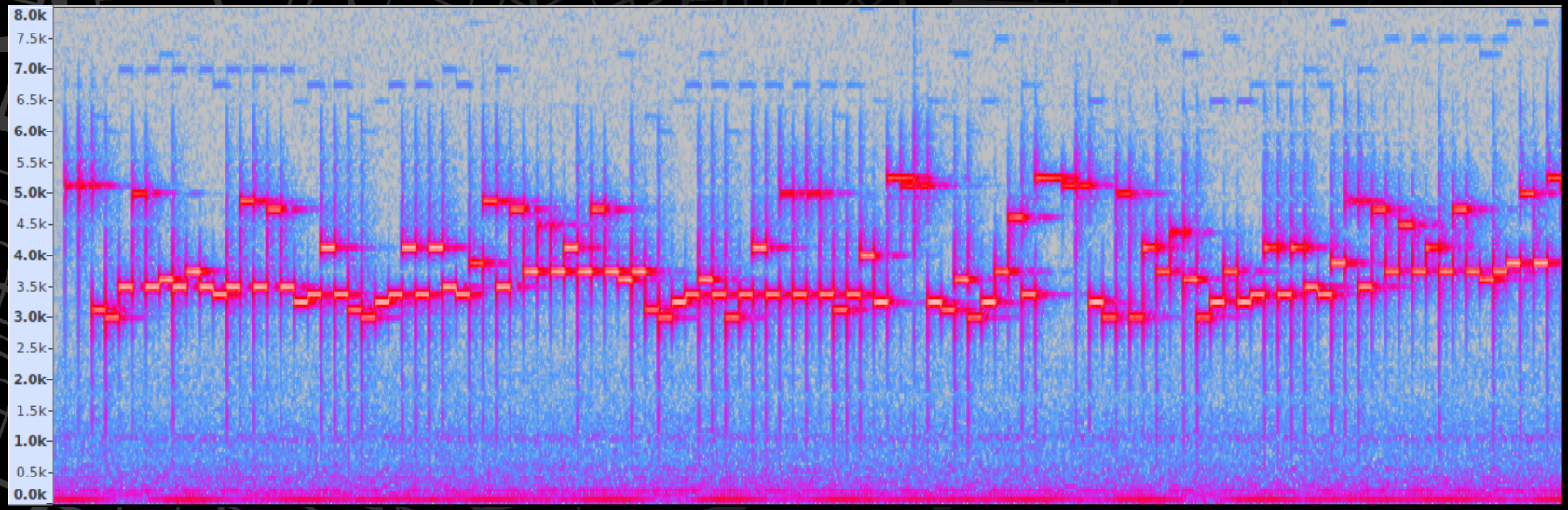
```
44    section3StartVal = 2;
45    ssidCrc = CRC(ssidB,ssidBLen);
46    iSsidCrc = (undefined)ssidCrc;
47    iSsidCrcMangled = (undefined)((ushort)ssidCrc >> 8);
48    memcpy(iPassBLocal,passBLocal,__n);
49    iSection3StartValCrcd = CRC(&section3StartVal,__n + 5 & 0xffffffff);
50    iPassBLocal[__n] = iSection3StartValCrcd;
51    iVar1 = GenerateTones(returnArr,2,&section3StartVal,passBLen + 6,
```

Hack the Signal

04

Can We Recreate It?

Let’s look at what we know

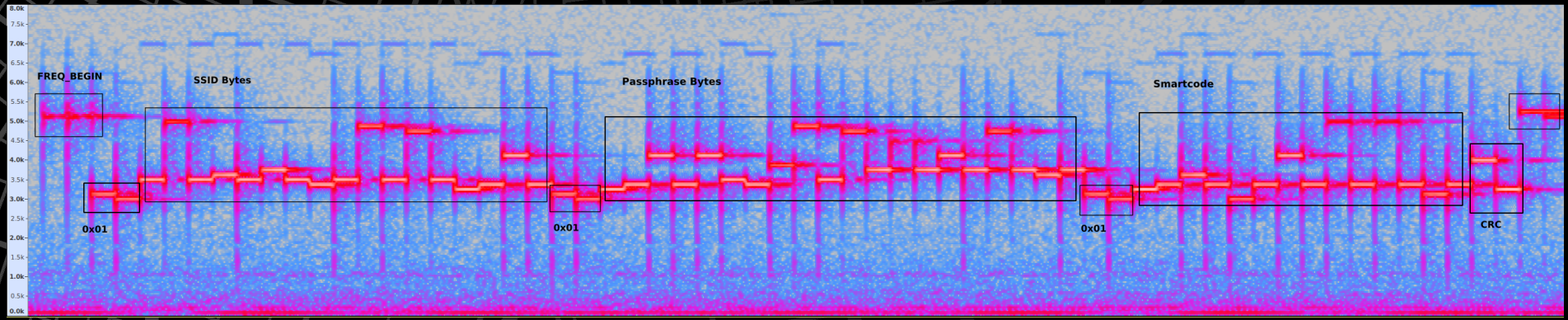


The Waveform: What we know

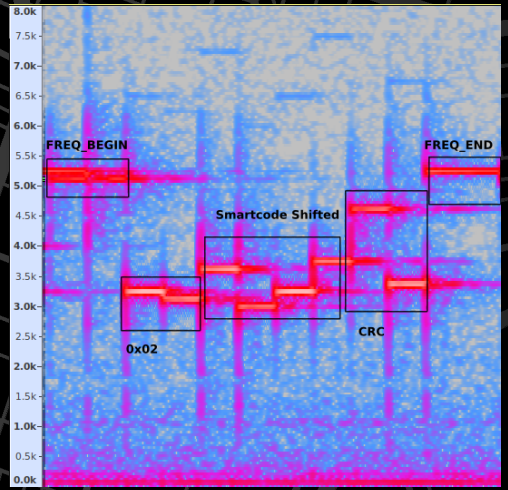
1. The waveform is comprised of 3 sections of hexified data
2. Each section is prefixed and suffixed by control codes and section identifiers
3. When 2 sequential tones are identical, a space tone is used between them
4. The duration of each tone is around 50 / 60 milliseconds
5. We know the structure of each waveform section

Section 1 Signal Structure

1. FREQ_BEGIN
2. FREQ_BEGIN
3. 0x01
4. SSID_BYTES
5. 0x01
6. PASSPHRASE_BYTES
7. 0x01
8. SMARTCODE_DIGITS
9. CRC(0x01+SSID_BYTES+0x01+PASSPHRASE_BYTES+0x01+SMARTCODE+”1”)
10. FREQ_END
11. FREQ_END



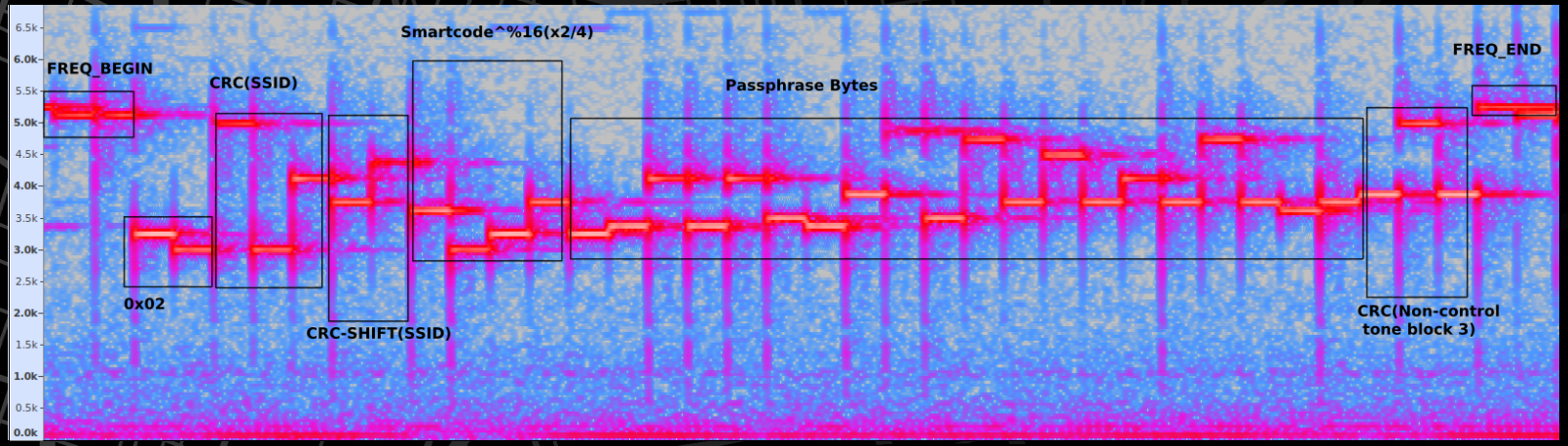
Section 2 Signal Structure



1. FREQ_BEGIN
2. FREQ_BEGIN
3. 0x12
4. SMARTCODE >> SHIFTED8
5. (SMARTCODE >> SHIFTED) >> SHIFTED8
6. CRC(0x12+(SMARTCODE>>SHIFTED8)+ (SMARTCODE>>SHIFTED8>>SHIFTED8))
7. FREQ_END
8. FREQ_END

Section 3 Signal Structure

1. FREQ_BEGIN
2. FREQ_BEGIN
3. 0x02
4. CRC(SSID_BYTES)
5. CRC-SHIFTED(SSID_BYTES)
6. SMARTCODE%16 (x2/4)
7. PASSPHRASE_BYTES
8. CRC(0x02+CRC(SSID_BYTES)+CRC-SHIFTED(SSID_BYTES)
   +SMARTCODE>>SHIFTED8+PASSPHRASE_BYTES)
9. FREQ_END
10. FREQ_END



Demo

05

Yes, we can reproduce the signal!

Demo

05

THAMYRIS: Replicate The Audio Protocol

Limitations

1. Can’t easily discover the camera’s admin password
   - It’s 6 HEX Characters
   - Password changes each time camera is reset, not tied to MAC or Serial#
2. Unable to decipher camera -> cloud communication
   - Camera has a local RSA keypair, that may change on each reset
   - Encrypted payloads are sent to the server
   - So, even though you can intercept the comm with a self-signed MITM, you can’t see or make sense of the payload
3. You get what you pay for
   - Even when you know the password, it won’t always connect (VLC is my ‘go to’)

THANKS

7thzero.com

github.com/7thzero

Technical References

- JADX: Dex to Java Decompiler - https://github.com/skylot/jadx
- Efficiency: Reverse Engineering with ghidra - http://wapiflapi.github.io/2019/10/10/efficiency-reverse-engineering-with-ghidra.html
- Guide to JNI (Java Native Interface) - https://www.baeldung.com/jni
- JDSP - Digital Signal Processing in Java - https://psambit9791.github.io/jDSP/transforms.html
- Understanding FFT output - https://stackoverflow.com/questions/6740545/understanding-fft-output
- Spectral Selection and Editing - Audacity Manual - https://manual.audacityteam.org/man/spectral_selection.html
- Edit>Labelled Audio>everything greyed out - https://forum.audacityteam.org/viewtopic.php?t=100856
- Get a spectrum of frequencies from WAV/RIFF using linux command line - https://stackoverflow.com/questions/21756237/get-a-spectrum-of-frequencies-from-wav-riff-using-linux-command-line
- How to interpret output of FFT and extract frequency information - https://stackoverflow.com/questions/21977748/how-to-interpret-output-of-fft-and-extract-frequency-information?rq=1
- Calculate Frequency from sound input using FFT - https://stackoverflow.com/questions/16060134/calculate-frequency-from-sound-input-using-fft?rq=1
- Intorduction - Window Size - https://support.ircam.fr/docs/AudioSculpt/3.0/co/Window%20Size.html
- Android: Sine Wave Generation - https://stackoverflow.com/questions/11436472/android-sine-wave-generation
- Android Generate tone of a specific frequency - https://riptutorial.com/android/example/28432/generate-tone-of-a-specific-frequency
- Android Tone Generator - https://gist.github.com/slightfoot/6330866
- Android: Audiotrack to play sine wave generates buzzing noise - https://stackoverflow.com/questions/23174228/android-audiotrack-to-play-sine-wave-generates-buzzing-noise

Credits

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**