Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Bluetooth Jamming

Bachelor's Thesis

Steven Köppel

koeppels@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Michael König
Prof. Dr. Roger Wattenhofer

May 10, 2013

Abstract

Bluetooth uses a technology called *Frequency Hopping Spread Spectrum* (FHSS) to rapidly switch among its 79 channels in a pseudorandom fashion. This reduces interference between multiple networks that operate in the same area, because each network uses a different hopping sequence. An attacker with the intention of jamming a Bluetooth network would need to know its particular hopping sequence to follow it and transmit at the right frequency at the right time. In this thesis we will investigate ways for a jammer to learn this sequence by observing some of the network's traffic.

Contents

Abstract i

1 Introduction 1

2 Background 2

2.1 Bluetooth . . . 2

2.1.1 Adaptive Frequency Hopping . . . 3

2.2 Hopping Sequence Prediction . . . 3

3 Theory of the Attack 5

3.1 Determining the Master's UAP/LAP . . . 5

3.2 Determining the Piconet Clock . . . 6

3.3 Following the Piconet . . . 6

3.4 Jamming the Piconet . . . 7

4 Materials and Methods 8

4.1 Hardware . . . 8

4.1.1 Ubertooth . . . 8

4.1.2 Laptops . . . 9

4.1.3 Other Bluetooth Devices . . . 9

4.2 Software . . . 9

4.2.1 Ubertooth Firmware . . . 10

4.2.2 Ubertooth Host Utilities . . . 10

4.2.3 Bluetooth Baseband Library . . . 11

4.2.4 HCI Tools . . . 12

4.2.5 Test Automation . . . 12

4.2.6 Log Analysis . . . 12

4.3 Test Setup . . . 12

5 Results **14**
5.1 Encountered Problems 14
5.1.1 Frequency Spread 14
5.1.2 Adaptive Frequency Hopping 15
5.1.3 Drift and Jitter 15
5.2 Achieved Goals 16
5.2.1 Determining the Master's UAP/LAP 16
5.2.2 Determining the Piconet Clock 16
5.2.3 Following the Piconet 16
5.2.4 Jamming the Piconet 17

6 Discussion **18**
6.1 Summary 18
6.2 Challenges 18
6.3 Possible Improvements for Bluetooth 19
6.4 Future Work 19

Acknowledgements **21**

References **22**

Acronyms **23**

A Screenshots **A-1**

B Graphs **B-1**

Chapter 1

Introduction

More and more electronic devices today offer some form of radio connectivity. Many of them use the unlicensed ISM[1] bands. Of these, the 2.4 GHz band is by far the most commonly used, with applications such as WiFi, Bluetooth, ZigBee, cordless phones, remote controls, and toys. Because of the large number of devices and the unlicensed nature of the 2.4 GHz band, there is usually a lot of interference, especially in urban areas. It is therefore important for such devices to adequately handle interference.

Since there are no guarantees about the availability of specific frequencies, it makes sense to use a larger portion of the applicable spectrum, instead of relying on a narrow frequency band being available. There are two main ways to spread the transmission energy over some spectrum, *Direct Sequence Spread Spectrum* (DSSS) and *Frequency Hopping Spread Spectrum* (FHSS).

The first one, DSSS, uses a fixed transmission frequency, but a wider bandwidth. This is achieved by encoding each bit with a sequence of so-called chips obtained from pseudonoise. This pseudorandomness is shared between the sender and receiver. To other devices, this kind of signal looks very similar to noise, because the spreading also reduces the signal-to-noise ratio; but the intended receiver can use the pseudonoise to reconstruct the original information. This technology is used by WiFi and allows several networks in the same area to use the same channel.

By contrast, FHSS achieves the spreading by rapidly switching among a set of channels. This is done in a pseudorandom fashion, with a hopping sequence known to both the sender and the receiver. The goal is to minimize the time that a source of interference has an effect on the signal and therefore prevent long interruptions in communication. Small errors in the bitstream can usually be corrected by using a suitable *Error-Correcting Code* (ECC).

FHSS is used by Bluetooth to reduce it susceptibility to interference and jamming. In this thesis we will look at ways to learn the hopping sequence of a Bluetooth network with the goal of jamming its communication.

[1] *Industrial, Scientific and Medical*

Chapter 2

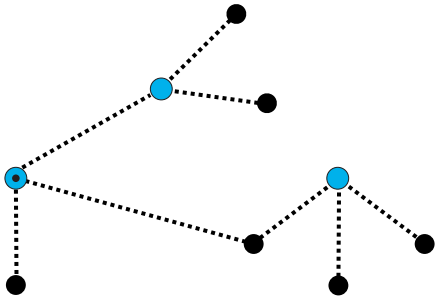
Background

2.1 Bluetooth

Bluetooth [3] communication is organized in so-called piconets. It consists of one master node and up to seven slave nodes. A node can be the master of at most one piconet and (additionally) act as slave in any number of piconets. Larger networks, called scatternets, can be created when a node participates in several piconets, as shown in figure 2.1c.

(a) A single-slave piconet

(b) A multi-slave piconet



(c) A scatternet

Figure 2.1: Bluetooth network types, with master nodes in blue [3, p. 65]

Bluetooth uses FHSS to alternate among 79 channels between 2402 and 2480 MHz. They are each 1 MHz wide and unlike with WiFi they don't overlap. There are generally 1600 hops per second, i.e. the frequency is changed after

each period of 625 µs, called a time slot. A Bluetooth packet covers either one, three, or five time slots. The master always starts transmitting packets on even-numbered slots and the slaves on odd-numbered slots.

Each device has its own *native clock* CLKN, which is a 28-bit counter running at 3.2 kHz. The master's native clock is used as the piconet's clock CLK and each slave maintains an offset between CLK and its own CLKN. The hopping sequence of a piconet is essentially determined by the Bluetooth address of the piconet's master. The upper 27 bits of CLK are then used as an index into this sequence.

Every device is identified by its 48-bit Bluetooth address, which are very similar to MAC addresses in Ethernet. They are composed of a *Lower Address Part* (LAP), an *Upper Address Part* (UAP), and a *Non-significant Address Part* (NAP), as shown below.

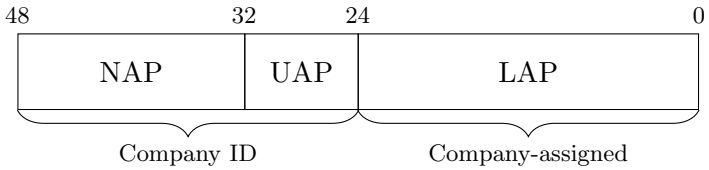


Figure 2.2: Format of a Bluetooth device address [3, p. 68]

2.1.1 Adaptive Frequency Hopping

Since version 1.2, Bluetooth includes a feature called *Adaptive Frequency Hopping* (AFH). It allows the master node to mark channels as either *used* or *unused*. When the master, or one of its slaves, detects repeated interference on a channel, the master can disable it and remap it to a *used* channel.

AFH also includes what is called the *same channel mechanism*. It causes the slaves to respond to the master on the same channel the master used to address the slave. This effectively cuts the hopping rate in half, reducing it to 800 hops per second.

2.2 Hopping Sequence Prediction

As we have seen, FHSS offers some protection against interference. While it is primarily meant for countering *accidental* interference, it can also help against *intentional* interference, also called jamming.

Let us assume a narrowband jammer that can only transmit on one channel at a time. If such a jammer does not know the pseudorandom hopping sequence,

he has no way of knowing on which channel to transmit at a given time and has little chance of actually causing any interference.

It is of course also possible to jam all channels at once, but such a wideband jammer needs more sophisticated equipment or numerous transmitters to cover the wider bandwidth. In addition, the total required transmission energy grows proportionally with the number of channels. He is thus at a disadvantage compared to a legitimate sender or a smart narrowband jammer. The latter also has the advantage of being able to selectively jam a network without affecting his own or other "friendly" communication.

In the Bluetooth setting, a potential narrowband jammer needs to know the relevant part of the master's address and its CLKN. Both of these are not immediately available to an observer. We will investigate ways of reconstructing these values from observed traffic. We then use this information to follow a piconet's hopping sequence and ultimately disrupt an active Bluetooth connection. See the next chapter for more details about the required steps to reach this goal.

Chapter 3

Theory of the Attack

Several steps are necessary to be able to jam a Bluetooth connection. First we need to find out some piconet specific parameters. We might try to intercept an FHS[1] packet sent by the master to the slaves, which contains all the required information. But these packets are only sent when the connection is initially established, so an attacker would already need to be present at that time. Furthermore, the payload of Bluetooth packets is usually encrypted, meaning an attacker would also need to somehow determine the encryption parameters. We therefore focus on alternatives that don't involve FHS packets.

The algorithm[2] that calculates the hopping sequence for a piconet takes various inputs. Relevant for distinguishing between different piconets is the *UAP/LAP* input. It is 28 bits wide and consists of the lower 4 bits of the *Upper Address Part* (UAP) and the 3 byte *Lower Address Part* (LAP) of the piconet's master.

3.1 Determining the Master's UAP/LAP

First we need to find out the LAP of the piconet's master. While this is not sent over the air in cleartext, it is quite easy to recover. The access code [3, p. 110ff.] which forms the beginning of each packet is derived from the LAP. First a 64-bit sync word is generated from the 24-bit of the LAP. The access code then consists of a 4-bit preamble, the sync word, and a 4-bit trailer. By reversing the sync word calculation, which is based on Barker codes, the LAP can be obtained.

Next we also need the UAP of the master, or rather the lower half of it. Each packet header ends with a *Header Error Check* (HEC). Before computing it, the HEC generator (essentially a LFSR[3]) is initialized with the UAP of the master. By using the received HEC and reversing this computation, one can obtain the UAP.

[1] *Frequency Hop Synchronization*
[2] described in [3, p. 83ff.]
[3] *Linear Feedback Shift Register*

These calculations depend on the error-free reception of the packet to yield the correct LAP and UAP. We should therefore capture several packets and compare the resulting values. If most of them agree, we have likely found the correct ones.

Since these values are part of the master's Bluetooth address, which doesn't change, we only have to determine them once. We can then manually specify them in future attacks on the same piconet.

3.2 Determining the Piconet Clock

Once we know the UAP and the LAP, all that we still need to determine the channel for a specific hop, is the value of the piconet clock; but we can already compute the entire hopping sequence and store it in an array of channels. The clock is then simply an index into this array.

We can now remember the pattern of observed patterns and search the precomputed sequence for a match. Since the algorithm for the hopping sequence is designed to prevent repetitive patterns, we can expect to find a unique match, once enough packets have been observed.

The lower six bits of the clock (CLK6) are also used for the so-called whitening process, which is applied to the data before transmission to prevent long runs of equal bits. One can now brute force search the 64 possible values for each packet header to find likely candidates[4] for the lower part of the clock. This allows the hopping sequence search to be accelerated by only checking every 64th clock. Additionally, we are much less likely to find matches in the sequence, which are false positives, because those with a wrong CLK6 are never considered.

Both this and the next step are further complicated by Bluetooth's *Adaptive Frequency Hopping* (AFH) feature, explained in section 2.1.1. We have decided to try disabling AFH, as fully supporting it is quite difficult. See section 6.4 for some ideas of how this might be improved.

The `ubertooth-follow` program offers an alternative to the passive approach above. It uses the assistance of the computer's local Bluetooth adapter (via the `bluez` libraries). It can either query the adapter's own clock or ask it to try and connect to a piconet, which will then reveal its clock.

3.3 Following the Piconet

Once we have determined the UAP/LAP and the clock, we can start following the piconet. We need to synchronize our own clock to that of the piconet and start switching channels according to the hopping sequence. If all the parameters

[4] when the unwhitened bits look like valid *Forward Error Correction* (FEC) encoded data

are correct and the clocks are properly synchronized, we should now be able to observe all of this piconet's traffic. If not, we are either using a wrong sequence or the right one but shifted.

Getting either the sequence or the clock wrong, will have the same effect, that we only see roughly $\frac{1}{79}$ of the traffic. This is because the sequence is pseudorandom, and we "randomly" choose one out of 79 channels for each time slot.

3.4 Jamming the Piconet

Once we are correctly following the target piconet, we can start jamming it while continuing to hop. The actual jamming is usually accomplished by transmitting something that looks like noise, e.g. pseudorandom bits, at full power.

If our transmission power is high enough and we are close enough to the piconet nodes, our signal will distort the legitimate traffic to the point where it's indistinguishable from noise. Since no more traffic is getting through, the connection should fail at this point. Normally, when a Bluetooth master node detects interference, it will try to apply AFH to avoid the jammed channels, but we are actually jamming *all* the channels. Even if it still decided to remap some channels, it couldn't inform the slaves, as the update message would also be drowned by our jamming.

Chapter 4

Materials and Methods

In the following, we describe the hardware and software that we used for this thesis, and explain how we conducted the tests and analyzed the resulting log files.

4.1 Hardware

A normal off-the-shelf Bluetooth dongle does not offer the kind of low level access to the radio that was required for this project. One option would have been to use a *Software-Defined Radio* (SDR) like the USRP [2], but luckily there was a more economical solution available. We decided to use a small USB device called Ubertooth, which only costs a fraction of even the cheapest USRP.

4.1.1 Ubertooth

Project Ubertooth [1] was started by Michael Ossmann around 2010 to develop an affordable 2.4 GHz device capable of sending and receiving the kind of signals used by Bluetooth. It is completely open source, both the hardware design and the provided example firmware and host software. For this thesis we used an Ubertooth One, which is the latest hardware revision and successor of the Ubertooth Zero. It consists mainly of a CC2400 wireless transceiver, a CC2591 radio front end, and an LPC175x microcontroller with integrated USB 2.0 support. It is used like a USB dongle and requires a host computer to operate.

(a) Front

(b) Back

Figure 4.1: Ubertooth One

The transceiver has an integrated modem supporting data rates up to 1 Mbps and handling GFSK[1] modulation. This is a perfect match for Bluetooth's *Basic Rate* (BR) packets. Unfortunately it does not support the 2 or 3 Mbps data rate and DPSK[2] modulation used for the payload of *Enhanced Data Rate* (EDR) packets; but the Ubertooth can still decode the access code and header of EDR packets, as those are always sent using the BR modulation mode.

A fully assembled Ubertooth One can be bought from various retailers[3] for about 120 USD, at the time of writing this. We had a total of three units available, but one is enough for following and jamming a piconet. Having several Ubertooth devices was helpful to get familiar with the provided firmware code. We could e.g. use two of them to send raw data back and forth, or use one as a spectrum analyzer[4] to see if the other one is actually transmitting as intended.

4.1.2 Laptops

All development was done on a ThinkPad T400 laptop running Ubuntu Linux. The same computer was also used with an additional ThinkPad T430 for generating some test traffic via their built-in Bluetooth adapters (see section 4.3). The T400 was also used as the host computer for operating the Ubertooth.

4.1.3 Other Bluetooth Devices

Before settling with the laptops, we also used various other Bluetooth devices to generate some traffic for testing. This included two Android smartphones, a GPS logger, and a pair of wireless headphones; but using the computers provided more control over the connection parameters and allowed the testing to be automated (see section 4.2.5).

4.2 Software

The creators of the Ubertooth also provide firmware, some host utilities, and a Bluetooth baseband library to demonstrate various capabilities of the Ubertooth. We used and extended these and also developed various scripts to help create and analyze the numerous log files of our tests.

[1] *Gaussian Frequency-Shift Keying*
[2] *Differential Phase-Shift Keying*
[3] see `http://greatscottgadgets.com/ubertoothone/` for a list
[4] see figure A.1 in the appendix for screenshots

4.2.1 Ubertooth Firmware

There are several firmwares included with the Ubertooth software. Some of them are quite simple and meant to test individual features or to confirm the correct assembly of the hardware. The main firmware for the Ubertooth is called `bluetooth_rxtx`. Upon booting, it performs some basic initialization and configuration of the various peripherals on the microcontroller and the transceiver. Afterwards it accepts commands over USB from the host computer. It operates in different modes, one of which is particularly interesting, as it sends a stream of demodulated bits back to the computer as they are received on the radio.

During the course of this thesis, we added some new functionality to the `bluetooth_rxtx` firmware. This includes the ability to send debug output to the host computer and a method to start jamming. For the latter we used the PRNG[5] built into the transceiver to generate a noise-like signal which we then transmitted at the highest power amplifier setting.

4.2.2 Ubertooth Host Utilities

On the host side, there are several utilities providing different functionalities. Each of them communicates with the Ubertooth over USB, issues some commands, and processes the response. This allows some of the heavier processing to be offloaded from the microcontroller to the host CPU. The prime example is calculating the entire hopping sequence for a piconet. It would be impossible to do this on the Ubertooth. If we store each of the 2^{27} entries in a byte, the complete hopping sequence requires 128 MB of RAM, which is far beyond what the Ubertooth has.

An overview of the different utilities that we used:

- `ubertooth-util`: general utility to get the version and serial number, run some tests, reset an Ubertooth, enter DFU[6] mode, etc.
- `ubertooth-lap`: searches received traffic for anything that looks like an access code and calculates the LAP from it
- `ubertooth-uap`: for a given LAP, analyzes received packets and calculates the UAP from their headers
- `ubertooth-hop`: analyzes traffic for a given LAP and UAP and determines the piconet clock; we incorporated most of our changes into this utility and added appropriate command line options to activate them

[5] *Pseudo-Random Number Generator*
[6] *Device Firmware Upgrade*

- `ubertooth-follow`: uses the host's Bluetooth adapter to determine a clock value, either its own or that of a piconet by attempting to connect to it, then uses that clock to follow the piconet
- `ubertooth-dfu`: utility to flash new firmware unto the Ubertooth device
- `ubertooth-specan-ui`: a graphical Ubertooth spectrum analyzer; see figure A.1 in the appendix

We modified some of these utilities, especially `ubertooth-hop`, to add various new features and fix some bugs that we found in the existing code.

When the already implemented functionality of `ubertooth-hop` did not work as intended, we added our own implementation of searching the hopping sequence. For this we gathered packets from a single channel and recorded their arrival times. We then converted this to a bitmap (`observed_bitmap`) of configurable length, where each bit represents a time slot and is set to 1 if a packet arrived in that slot and to 0 if not. We normalized the arrival clock values so that the first packet received always corresponded to the first bit of the bitmap. By also representing the hopping sequence as such a bitmap (`sequence_bitmap`), we could compare the pattern of our observed packets with the hopping sequence. For this we took the n-bit `observed_bitmap` and XORed it with the first n bits of `sequence_bitmap`. After each comparison, we shifted the `sequence_bitmap` (or the portion we were looking at) one bit to the left and compared the bitmaps again. This was repeated until we found one or several matches with fewer bit errors than a configurable threshhold.

This approach allowed us to gain a deeper understanding of the process of determining the clock of a piconet. It also helped us to identify the problems listed in section 5.1. We could then go back and apply our newly gained knowledge to the original Ubertooth tools.

4.2.3 Bluetooth Baseband Library

The parts of the host code that are not specific to the Ubertooth hardware are distributed as a separate library called `libbtbb`. It covers the Bluetooth baseband, meaning it contains data structures for describing packets and piconets, methods for manipulating them, algorithms for calculating a piconet's hopping sequence, etc.

The Ubertooth host utilities mostly act as wrappers for the functionality in this library and the Ubertooth library `libubertooth`.

4.2.4 HCI Tools

The *Host/Controller Interface* (HCI) standardizes the communication between a host's Bluetooth stack and the actual Bluetooth chip. The most commonly used Bluetooth stack for Linux is called `bluez`. The corresponding package is installed on Ubuntu by default and includes various HCI-related utilities.

`hciconfig` is used to set global settings for a Bluetooth adapter, while `hcitool` is used to manage per-connection parameters. `hcidump` allows all HCI traffic to be captured and saved to a file, but unfortunately this gives you only a high-level view of the traffic. Baseband-level details like packet headers, physical channel used, and retransmissions are already abstracted away. For normal Bluetooth operation, they are not needed and are thus not sent via HCI.

Finally we have `l2test`, which is a simple tool to setup L2CAP[7] streams between two computers. Since the laptops both had built-in Bluetooth adapters, `l2test` was a convenient way to generate some test traffic for the Ubertooth to analyze.

4.2.5 Test Automation

In order to simplify the process of running our tests, we wrote some Python scripts to automatically launch the various tools and save their output. We usually ran two or three main programs in parallel. Firstly, we had to generate some test traffic. Then we could run one of the Ubertooth host utilities to analyze the traffic. We also found it helpful to dump the traffic that was sent, so we could e.g. compare actual and observed packet rates.

These scripts also allowed us to repeat the same test several times and vary the parameters that were passed to the tools in each run.

4.2.6 Log Analysis

Additionally, we also created some simple scripts to analyze the log files and calculate packet rates, etc. And finally, we wrote some more Python scripts to graph the live packet rate of our tests similar to the graphs seen in figure B.1 in the appendix.

4.3 Test Setup

We initially experimented with various different Bluetooth connections; but to get meaningful results we had to create a stable test setup, which is shown in the following illustration. We used USB extension cables to allow for flexible

[7] *Logical Link Control and Adaptation Protocol*, a low-level data transport protocol

positioning of the Ubertooth devices. The initial position of the Ubertooth was too close to the laptops (see section 5.1.1), so we moved it further away and higher up. The USB cable was kept in place by taping it to a bookshelf.

We noticed in our early experiments that a person moving through the line-of-sight of a Bluetooth connection had a pretty significant effect on its signal strength and packet rate. We thus let the automated tests run unattended.

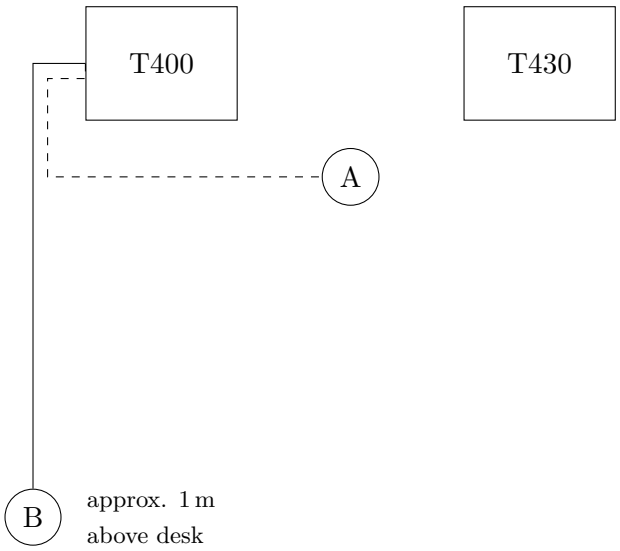


Figure 4.2: Overview of our test setup showing the relative position of the two laptops and the initial (A) and adjusted position (B) of the Ubertooth

On the software side, we primarily used the aforementioned `l2test` for the test traffic. We set it to continually send 16 byte payloads form the T430 to the T400. The small payload ensured we would mostly have small (i.e. single-slot) packets and thus many headers to analyze.

Chapter 5

Results

One of the intended purposes of the Ubertooth was sniffing Bluetooth traffic. So most of the steps outlined in chapter 3 were already implemented to some degree in the code provided with the Ubertooth. In theory we would only have had to add the actual jamming part. But as it turned out, there were still a number of obstacles to overcome to reach our goals.

5.1 Encountered Problems

Let us first revisit the biggest problems and obstacles that we identified.

5.1.1 Frequency Spread

After conducting some of our experiments to determine the clock we realized that we were receiving some packets from neighboring channels. In our original test setup (figure 4.2), the Ubertooth was positioned between the two laptops, but this was too close as it turned out.

The signal strength of a wireless signal typically looks like a bell-shaped curve, meaning that some portion of it is outside the nominal bandwidth. Normally, the absolute signal strength of that portion is very low, even below the noise-level. But at close proximity, it can be high enough so that this frequency spread causes packets to also be received on adjacent channels.

We verified this by listening on frequencies just outside the range used by Bluetooth. The channels actually used by Bluetooth are numbered from 0 to 78. When we listened on "channel 79", we still received a few percent of the traffic, which was presumably sent on the adjacent channel 78. The same also happened analogously on "channel -1". This warranted the assumption that we would also receive traffic from adjacent channels when listening on any other channel.

Unfortunately, the channel that a packet was sent on is not part of its header.

Some preliminary tests showed that one could use the RSSI[1] values to differentiate between packets sent on the expected channel and those sent on a different channel; but the easier solution was to just move the Ubertooth farther away from the transmitters. The frequency spreading effect was almost completely eliminated by moving the Ubertooth about a meter farther away (see figure 4.2).

5.1.2 Adaptive Frequency Hopping

If AFH is already actively used when we want to start our attack, we face several problems. If we don't know which channels have been remapped, the pattern matching that we use to search for the piconet clock is likely to fail. Even if we find the correct clock, the jamming part then uses the wrong frequency for any remapped channel.

We therefore decided to try and disable AFH, at least for our early experiments. For the Linux computers that we used for the test traffic, this could be accomplished by running `hciconfig hci0 afhmode 0`, but it was not immediately clear what disabling AFH meant. We initially expected such a device to behave as if it didn't support AFH at all; but it turned out that only the remapping of channels was disabled, while the *same channel mechanism*[2] was still used.

5.1.3 Drift and Jitter

When we decided to implement our own algorithm for finding the piconet clock, we noticed that the clock on the Ubertooth exhibited significant jitter and drift relative to the master's clock. The drift was roughly 100 time slots per hour or about 17 ppm, when we specifically tested for it. The bigger problem, however, was the jitter. The clock values of observed packets as obtained from the Ubertooth turned out to be offset from the actual clock by ± 1 time slot for about half the packets.

As explained in section 3.2, we were able to determine the lower 6 bits of the piconet clock (CLK6) from the whitening of packets. We could then calculate, for each packet, an offset between the lower 6 bits of the Ubertooth clock and CLK6. Using the relative change[3] of these offsets we were able to compensate for the jitter.

As it turned out we could actually use the drift to our advantage, as explained in section 5.2.4.

[1] *Received Signal Strength Indicator*
[2] see section 2.1.1
[3] ideally there would be no jitter and the offset would remain constant

5.2 Achieved Goals

5.2.1 Determining the Master's UAP/LAP

The Ubertooth host utilities already contained good support for finding the LAPs of any nearby piconets and finding the UAP for a specific LAP. So the first step (as outlined in chapter 3) was very easy to accomplish and didn't require any further work on our part.

5.2.2 Determining the Piconet Clock

From the description of the Ubertooth tools, it looked like this part *should* also have worked out of the box, but unfortunately that wasn't the case. We spent a lot of time investigating the possible causes for this, which are summarized in section 5.1. As explained earlier, we eliminated the frequency spread problem and compensated for the jitter. We had already disabled AFH, but didn't immediately realize that the *same channel mechanism* was still being used. The Ubertooth tools already had some limited experimental support for AFH, particularly to adjust the hopping sequence to take the *same channel mechanism* into account. But they failed to detect the usage of AFH, even though the detection was actually based on recognizing two consecutive packets arriving on the same channel. We added an option to manually set the AFH flag for a piconet and this dramatically improved the speed and accuracy of clock detection using `ubertooth-hop`.

We also tried the active[4] clock discovery offered by `ubertooth-follow`. This did get the correct clock value, but failed at the next step of following the piconet for two reasons. For one, there was a serious bug in the code, resulting in the clock delay being sent byte-swapped. Thus, the default delay[5] of 5 half-slots caused an offset of `0x 0500 0000` to be added to the clock on the Ubertooth. Secondly, we also had the same problem of not using the *same channel mechanism*, which resulted in us missing essentially all slave packets, even when the clock was set correctly.

5.2.3 Following the Piconet

After obtaining the correct clock, we could start hopping according to the hopping sequence which we calculated earlier. In theory this would have allowed us to sniff all the traffic of the piconet. While this wasn't our primary goal, it

[4] using `bluez`, see the end of section 3.2

[5] a value additionally added to the Ubertooth clock to compensate for the delay in reading the clock, transferring it to the Ubertooth, and applying it

would have made the jamming step easier, by confirming that we were on the right channel at the right time.

In practice we found it quite difficult to accurately follow a piconet. Synchronizing the Ubertooth clock to that of the piconet wasn't very reliable and even a properly synchronized clock drifted away from the piconet by more than a time slot after less than a minute.

5.2.4 Jamming the Piconet

For the purpose of jamming, we didn't absolutely need a perfect way of following a piconet for extended periods of time. If we could just jam a piconet for long enough so that the connection would break, we still reached our goal.

We realized that we could actually use the drift to our advantage. For our particular combination of Ubertooth and laptop, we found out that the Ubertooth's clock was slightly too fast relative to that of the laptop's Bluetooth adapter. We could now synchronize our clock and then manually rewind it by a few ticks, before starting to jam. The Ubertooth clock then slowly caught up with the piconet clock at which point we were jamming the piconet perfectly.

In our experiments using `l2test`, we were able to reproducibly jam the connection until it failed with a timeout. At the final presentation of this thesis we showed a demonstration where we jammed an audio stream between a smartphone and a laptop. One could directly hear the effect on the audio stream. First it just crackled a bit, then it stopped intermittently, and then it failed for several tens of seconds straight. Compared to `l2test`, the smartphone streaming the audio was more tenacious, continually trying to send packets, even in the presence of a jammer. This meant that stopping the jamming process immediately brought the audio back. Even if we continued to jam, the connection would eventually recover, as we drifted away from the piconet's clock. We didn't have a single audio stream fail completely and time out like in the case of `l2test`. The window during which we achieved a clean jamming was probably too short.

You can see some graphs of the packet rate during our `l2test` experiments in figure B.1 in the appendix. The first one is from a run where we are purposely using a wrong clock. As you can see, there are some periodic packet rate drops during the jamming, but the connection never dies. The second one is from a successful jamming run. The packet rate also shows some periodic drops before it goes to zero, recovers momentarily, and drops back to zero. It then stays at zero as the connection failed and timed out. We are not sure what exactly causes the periodic drops seen in the graph.

Chapter 6

Discussion

6.1 Summary

We have shown that it is possible to jam a specific Bluetooth connection without any prior knowledge about it. All the required parameters can be reconstructed by analyzing observed traffic of a piconet. They can then be used to follow this piconet and jam it.

It is not yet a universal solution, as we relied on two particular assumptions. Firstly, we always disabled AFH on our test laptops. Further work would be needed to also support piconets with AFH enabled. Secondly, we exploited the drift of our Ubertooth's clock to implement a jamming attack, which only worked for a relatively short time. Sustained jamming would require a more accurate clock or some other way of staying synchronized with the piconet clock.

6.2 Challenges

One of the biggest challenges of this work was the limited debugging capabilities of the Ubertooth. I didn't have any experience with embedded systems development and was used to debugging software on a PC by watching variables, using exceptions, printing debug information, etc. During firmware development for the Ubertooth we didn't have any of these luxuries. While the device has a JTAG port, we didn't have the necessary equipment available to use it. We ended up implementing a kind of debug output over USB. But due to issues like buffering, it wasn't completely reliable, so that we could never be sure why some expected output did not reach the host.

Another challenge was the lack of documentation for some of the tools we used. The manual pages for the HCI tools in particular were quite minimal. For some commands they mentioned parameters, but never explained what were valid arguments; neither did they include any examples. One had to hunt for information on the internet, which wasn't always accurate.

Finally, the code provided by the creators of the Ubertooth was not a very polished piece of software. That's understandable, as it is mainly meant as an example or starting point for further development. After all, the Ubertooth is a *development* platform and not a consumer product. Still, the fact that we found a few concrete bugs in their code, meant that we couldn't take anything for granted and ended up double-checking many implementation details against the Bluetooth specification.

6.3 Possible Improvements for Bluetooth

The attack that we have shown relies on the parameters necessary for the hopping sequence calculation being reconstructable from observed traffic. These parameters basically just consist of the UAP/LAP and there isn't any notion of secrecy involved regarding these values, they might as well be transmitted in cleartext with each packet. The part of finding the clock can be thought of as basically a brute force search of a 2^{27} space. While this not a problem for modern computers, it is actually made even easier by the possibility to determine the lower 6 bits of the clock separately. The search is thus effectively reduced to a 2^{21} space (and a negligible 2^6 space).

One could now imagine alternative ways of calculating the hopping sequence which would make it much harder for an attacker to determine the sequence. The parameters might e.g. be based on some shared secret established at connection setup. For piconets of only two nodes, it could even happen during Bluetooth's pairing process. Furthermore, one could use a wider clock of say 64 bits instead of the 28 bits used now. This would increase the wrap-around time of the clock from about 23.3 hours to about 585 billion years. The search space for determining the clock by brute force would increase accordingly (provided the clock is properly initialized at device startup and not simply started from zero as is usually the case with Bluetooth).

Perhaps a future version of Bluetooth will include such improved jamming resistance. But as mentioned earlier, for Bluetooth the focus is on avoiding *accidental* interference and not offering military-grade jamming resistance. For the latter, 79 channels of 1 MHz each is hardly enough. A determined attacker can jam the full 79 MHz bandwidth without significant difficulty.

6.4 Future Work

There are a few areas with room for improvement. One significant simplification that we made was disabling AFH. In a real-world scenario, where AFH might already be active, one would first have to find a way to determine the current AFH channel map. It's conceivable to gather statistics about the received packets

on several channels and using it to categorize channels as *used* and *unused*. The clock could be determined similarly to the case without AFH, by listening on a channel that is in fact *used*. But we might get false positives, if the same channel is also used as the result of remapping an *unused* channel. The remapping function that determines this depends again on the clock. So for cases with significant remapping, we have a chicken-and-egg problem.

Acknowledgements

I would like to thank my supervisor Michael for his kind help with this thesis. He gave me a lot of valuable advice in the many discussions we had.

I would also like to thank my girlfriend Andrea. She not only gave me moral support while I was writing this thesis, but also offered some technical advice and helped me carry out some of the tests.

Finally, I would also like to thank the creators of the Ubertooth and other people on the Ubertooth mailing list for their help.

References

[1] Project Ubertooth. http://ubertooth.sourceforge.net/.

[2] Universal Software Radio Peripheral, by Ettus Research. https://www.ettus.com/product.

[3] Bluetooth SIG. Specification of the Bluetooth System, Core Version 4.0, Volume 2. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737, June 2010.

Acronyms

AFH *Adaptive Frequency Hopping*

DSSS *Direct Sequence Spread Spectrum*

ECC *Error-Correcting Code*

FHS *Frequency Hop Synchronization*

FHSS *Frequency Hopping Spread Spectrum*

HEC *Header Error Check*

LAP *Lower Address Part*

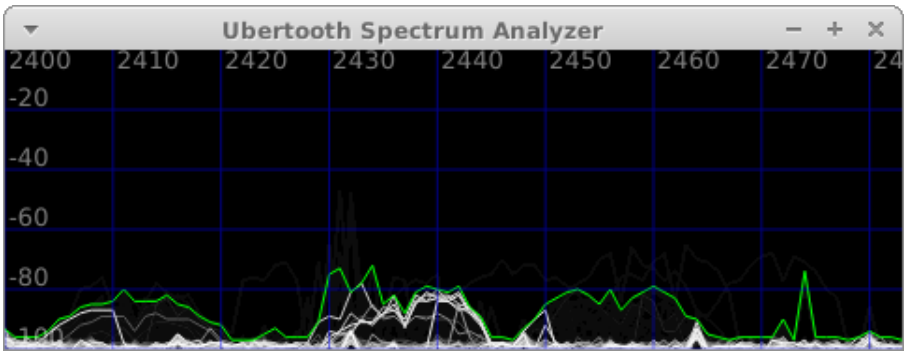
NAP *Non-significant Address Part*

SDR *Software-Defined Radio*

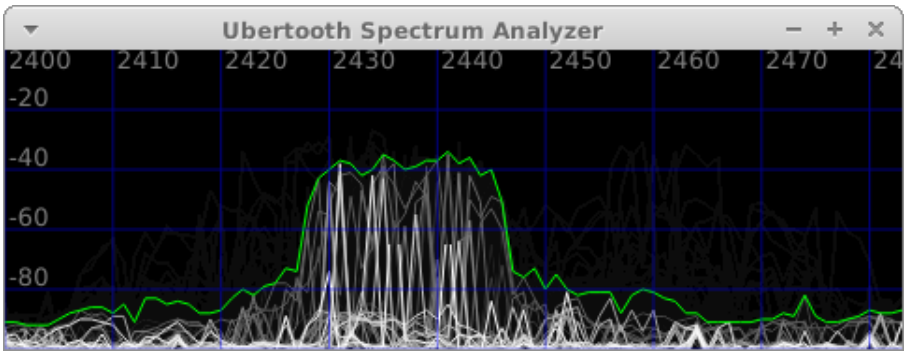
UAP *Upper Address Part*

Appendix A

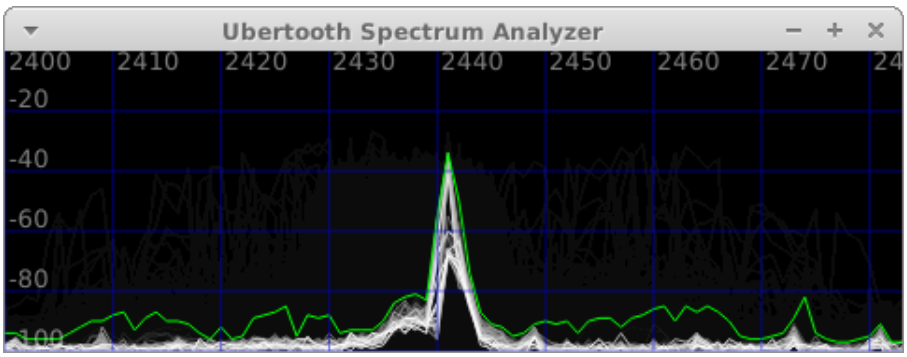
Screenshots



(a) Spectrum when idle (with some background WiFi activity)



(b) Spectrum with an active WiFi connection (on WiFi channel 6)

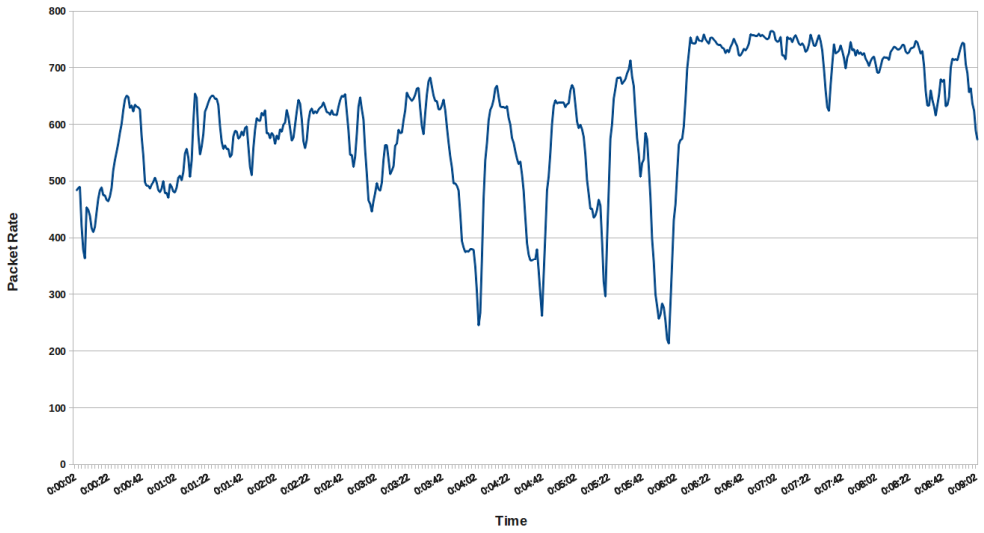


(c) Spectrum while actively jamming Bluetooth channel 39

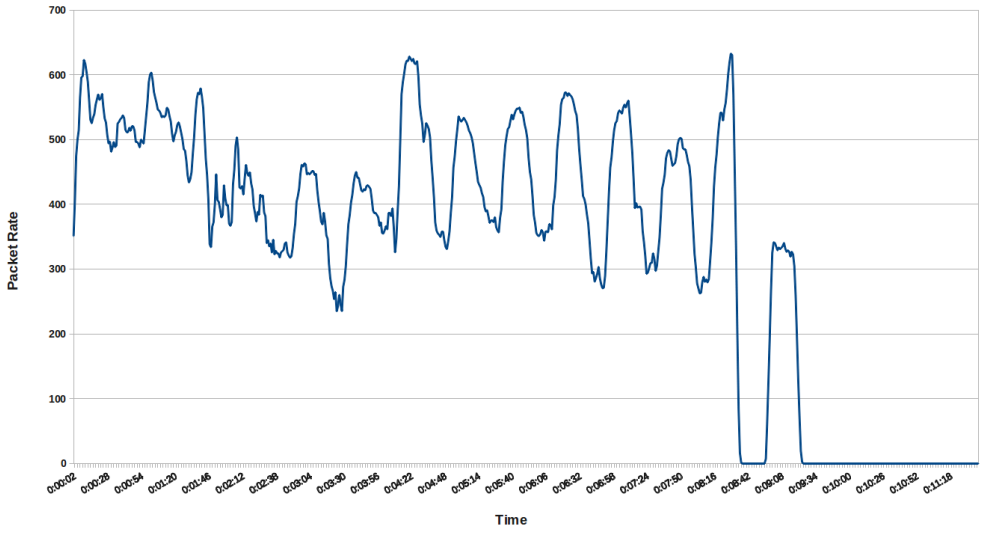
Figure A.1: Ubertooth spectrum analyzer with various activity

Appendix B

Graphs



(a) Jamming using the *wrong* hopping sequence or clock; jamming started around 3:00, stopped around 6:00



(b) Jamming using the *correct* hopping sequence and clock; jamming started around 3:00

Figure B.1: Graph showing packet rate drop while jamming