Towards Device Communities with User-Friendly Pairing for Modern Android Platforms

Pavel Shramau
Master's Thesis Spring 2019
Towards Device Communities with User-Friendly Pairing for Modern Android Platforms

Pavel Shramau

May 2, 2019
Towards Device Communities with User-Friendly Pairing for Modern Android Platforms

Pavel Shramau

http://www.duo.uio.no

Printed: Reprosentralen, University of Oslo
Abstract

Information and communication technologies, such as remote health monitoring and mobile Internet, have played a key role on a major improvement on patients’ lives. The problem is that patient monitoring requires higher degree of security and robustness than those provided by the typical Internet technologies. One viable solution to address these issues is DevCom (Device Communities), a network system that supports persistent, secure and user-friendly communications between doctor’s and patient’s devices.

This thesis investigates the use of DevCom to secure device communications in the domain of remote health monitoring systems. To address limitations of DevCom on modern mobile devices, we design the system called DevComApp. DevComApp aims to bring DevCom to mobile devices and to provide a user-friendly way to pair them.

An attempt to port DevCom inside an Android application in native C language reveals a major issue of failing ioctl() function call that is critical for tunneling functionality in DevCom. Conducted systematic analysis of the ioctl() issue shows the inability of processes started by the native C part of the Android application to obtain certain permissions. In order to address this issue, other developers should consider six possible solutions that we present in the thesis. Five of the solutions require root access to the device, while the sixth one allows to avoid rooting by reimplementing certain functionality in Java/Kotlin instead of C.
Acknowledgement

First of all, I would like to express my gratitude to Thomas Plagemann, Morten Lindeberg and Hans Vatne Hansen, my supervisors, for all the guidance and encouragement provided to me during the last year. Without your support and assistance, I would have not made it until the end. Thank you.

I wish to thank my family for making my dream to study in Norway come true.

I also want to thank my dear friends, Eugene, Igor K., Igor B., Kate, and Polina, for all of the support and encouragement I got from them. You have always believed in me more than I did.
Table of contents

Chapter 1  Introduction .................................................................................................................. 1
1.1  Background and motivation ................................................................................................. 1
1.2  Problem statement ................................................................................................................ 3
1.3  Outline .................................................................................................................................. 4

Chapter 2  Application Domain .................................................................................................... 6
2.1  Application scenario ............................................................................................................. 6
2.2  Overall requirements ........................................................................................................... 8
2.3  Extended application scenario ............................................................................................ 10

Chapter 3  Background .................................................................................................................. 15
3.1  DevCom ................................................................................................................................ 15
  3.1.1  Description of DevCom .................................................................................................... 16
  3.1.2  Use of DevCom as a network technology to secure communications between the devices .................................................. 22
3.2  Secure Device Pairing .......................................................................................................... 28
3.3  QR codes ............................................................................................................................. 33
3.4  TUN/TAP ............................................................................................................................. 35
  3.4.1  Description of TUN/TAP ................................................................................................. 35
  3.4.2  TUN/TAP in DevCom .................................................................................................... 37

Chapter 4  Design .......................................................................................................................... 39
4.1  Design assumptions .............................................................................................................. 39
4.2  Design requirements ............................................................................................................. 40
  4.2.1  General requirements ...................................................................................................... 40
  4.2.2  DevComApp requirements ............................................................................................. 41
4.3  Design decisions .................................................................................................................. 42
  4.3.1  DevCom on mobile operating system .............................................................................. 42
  4.3.2  Use of hostnames instead of physical addresses for join operation ............................ 43
  4.3.3  QR codes to provide a user-friendly SDP in DevCom .................................................. 45
4.4  System architecture .............................................................................................................. 47

Chapter 5  Implementation and Systematic Analysis .................................................................... 51
5.1  Studies prior to implementation .......................................................................................... 51
  5.1.1  Evaluation of screen and camera requirement .............................................................. 51
  5.1.2  DevCom study .............................................................................................................. 55
5.2  Technical decisions .............................................................................................................. 59
  5.2.1  Mobile operating system: Android ................................................................................. 59
  5.2.2  DevCom as part of Android APK as a preferred option to cross-compiling .................. 60
  5.2.3  Technical decisions regarding hardware ...................................................................... 64
5.3  Languages and tools ............................................................................................................ 65
5.4  Implementation process ....................................................................................................... 66
  5.4.1  Implementing use of SHA-2 hash function .................................................................... 67
List of Figures

Figure 2.1: Illustration of application scenario ................................................................. 7
Figure 2.2: Illustration of the extended application scenario where the data is transferred to multiple receivers ......................................................................................... 11
Figure 2.3: Health monitoring system suite ....................................................................... 13
Figure 3.1: Layers of the DevCom stack (Hansen et al. 2013) ........................................... 16
Figure 3.2: Two example device community addresses (Hansen et al. 2013) ................. 18
Figure 3.3: DevCom architecture (Hansen et al. 2013) .................................................... 19
Figure 3.4: DevCom in the health monitoring system suite ............................................... 22
Figure 3.5: One device community per patient but multiple for doctor ......................... 25
Figure 3.6: Extended application scenario in DevCom ...................................................... 26
Figure 3.7: Generalized pairing procedure (Fomichev et al. 2018) ................................... 29
Figure 3.8: Version 7 QR code symbol (ISO/IES 18004:2015) ......................................... 34
Figure 4.1: Multi-tier architecture of DevComApp ............................................................ 48
Figure 4.2: “Generate QR code” sequence diagram ......................................................... 49
Figure 5.1: Maximum amount of pairing information encoded in QR code with ECC level M. ...................................................................................................................... 53
Figure 5.2: Confirmation that the TUN module is loaded (Screenshot from TUN.ko driver installer app) ........................................................................................................ 74
Figure 5.3: Analysis structure ............................................................................................ 77
Figure 5.4: Possible solutions flow chart .......................................................................... 84
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMS</td>
<td>Health Monitoring System</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>RCV</td>
<td>Receiver</td>
</tr>
<tr>
<td>TRN</td>
<td>Transmitter</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPv4</td>
<td>Internet Protocol Version 4</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol Version 6</td>
</tr>
<tr>
<td>CEP</td>
<td>Complex Event Processing System</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest–Shamir–Adleman public-key cryptosystem</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>SDP</td>
<td>Secure Device Pairing</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical (channel)</td>
</tr>
<tr>
<td>HCI</td>
<td>Human-computer Interaction (channel)</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Capability</td>
</tr>
<tr>
<td>TUN</td>
<td>Virtual Point-to-Point Network Device</td>
</tr>
<tr>
<td>FQDN</td>
<td>Fully-Qualified Domain Name</td>
</tr>
</tbody>
</table>
APK  Android Application Package
MP   Megapixels
px   Pixels
PNG  Portable Network Graphics
DNS  Domain Name System
mDNS Multicast DNS
DNS-SD DNS Service Discovery
NDK  Native Development Kit
CPU  Central Processing Unit
OS   Operating System
SDK  Software Development Kit
JNI  Java Native Interface
IDE  Integrated Development Environment
adb  Android Debug Bridge
ABI  Application Binary Interface
API  Application Programming Interface
MAC  Mandatory Access Control
DAC  Discretionary Access Control
SELinux Security Enhanced Linux
Android ROM Android Read Only Memory (known as system image, firmware)
UID/GID User Identifier/Group Identifier
EUID/EGID Effective User Identifier/Effective Group Identifier
Chapter 1
Introduction

1.1 Background and motivation

During the last ten years, the number of personal computing devices has grown dramatically. We can easily observe this trend: a laptop has become a must-have device for work and leisure, and almost “everyone” in highly developed countries has a smartphone, sometimes even more than one. In addition to that, a tablet has become something common in our lives that is widely used in education and by professionals. Furthermore, we see the growing sales of personal fitness trackers, smart watches, smart TVs, smart home devices. These devices constitute a large group of devices known as Internet of Things (IoT). IoT is believed to continue expanding reaching to 75.44 billion devices in 2025 [1]. It leaves us with a lot of new great possibilities in the field of computer communications, but also new challenges.

One domain, which has inevitably benefited from the technologies and innovations in the field of mobile personal computing devices, is the health sector. Furthermore, innovations in the field of mobile health solutions, according to a review by Silva et al. [2], are highly attractive both to business and academic communities. During the last decade, these innovations have already played a key role on a major improvement on patients’ lives, especially in elderly, disabled, and chronically ill. Mobile health solutions already have a strong impact on typical healthcare monitoring and alerting systems, clinical and administrative data collection, medical information awareness, as well as detection and prevention systems [2]. Health monitoring systems (HMS) evolve rapidly and become an important tool in modern healthcare and is expected to continue growth in use and sales in the nearest future. According to a report by Reuters [3], all of that is the result of multiple factors, namely the growing population, growing prevalence of vascular disorders and heart diseases, decreasing cost of health monitoring devices, etc.

However, the communication technologies still cannot be characterized as optimal for these new types of mobile devices and, particularly, the health monitoring application domain. Normally, some close-distance communication between devices, e.g., smartphone and health sensor, happens via technologies like Wi-Fi or Bluetooth. However, almost any long-distance communication will go through the Internet. But Internet was designed almost half a century ago without too much thought about the mobility of people and devices. Most of the
machines back then were desktop computers physically attached to the network by cable. Today, the landscape of the devices has changed dramatically. Roaming between Wi-Fi and 3G/4G networks happens without our direct involvement and implies a change of network address. It makes the process of maintaining persistent connections hard, which is extremely important for real-time health monitoring. In addition to that, the presence of NATs and firewalls makes the process of establishing communications much harder.

Another important challenge today is security. According to papers [4][5], multiple privacy and security issues in health applications exist due to a lack of secure communications. For example, risks include monitoring and eavesdropping of patients’ vital signs, interception of health data during transmission in the network or via Internet, and modifications of the transmitted data. Deploying new mobile health solutions without considering security often makes patients’ sensitive data vulnerable. Lack of secure communications may lead to the situations, when sensitive data, e.g., mental health condition, is being eavesdropped, and later misused. In order to prevent these situations and make mobile health solutions more attractive to the users, communication technologies have to provide a certain level of privacy and security. Furthermore, latest regulations on data protection, i.e., General Data Protection Regulation, set additional requirements on how personal data should be proceeded and protected by companies.

DevCom addresses these issues, such as security and mobility in modern Internet communications. It is a network system that provides users with a trustworthy and user-friendly way to communicate, share and collaborate among distinct groups of devices simultaneously [6]. We analyze the remote health monitoring scenario, identify the key requirements, and elaborate on how DevCom can be used in this application scenario. In particular, DevCom can provide secure communication channels to share sensitive health data between patient’s and doctor’s devices.

There are two major contributions of this thesis work. The first one is to design a system, called DevComApp. Based on the analysis of DevCom and its application in the remote health monitoring scenario, we identify an important limitation in the current DevCom, i.e., limited pairing capabilities. Furthermore, we identified one minor issue, i.e., use of unsecure SHA-1 hash function that we later address with the implementation. DevComApp is a system that can be considered as an extension to DevCom, and is designed to address the pairing limitation. Its main goal is to provide a new user-friendly pairing mechanism for mobile device users in the remote health monitoring application scenario. Initially, we assumed that
DevCom was available on mobile operating system, i.e., Android, but during our thesis work, we discovered that this assumption was wrong. Therefore, DevComApp design was extended with the requirement to make DevCom available on modern mobile operating systems, i.e., Android, which we chose as a target platform prior to the implementation process. As we discovered during the implementation, bringing DevCom on modern Android is a major task. We analyzed two different options, and decided to port DevCom C code inside Android application.

Our second major contribution is the identification and analysis of the challenges related to porting DevCom C code to the Android software platform. During the implementation process, we discovered numerous issues with the chosen approach. We investigate the issues and explain how we address them. Due to one particular issue, i.e., failing ioctl() function call required for tunneling functionality of DevCom, we could not proceed further with the implementation, and decided to focus on the systematic analysis of this issue. The analysis revealed that the ioctl() issue is caused by the inability of processes, which are started by DevCom native C part of the Android application, to obtain certain permissions to control a network device. We present six different solutions to address this issue, which we did not have time to implement during the thesis work. The description of the implementation process, the systematic analysis of the permissions issue and the presented solutions are important parts of our second major contribution. Our findings should help other researchers and developers during the development of their Android projects partly in C programming language.

1.2 Problem statement

We argue that in the application domain of remote health monitoring systems, use of DevCom is a viable solution to secure communication between patient’s and doctor’s devices. Unlike VPN technologies that are commonly used to provide secure communication over the Internet, DevCom allows one device to be a part of multiple communities. This can be very useful in certain use cases that we discuss in the thesis. However, DevCom is a system that was implemented by research group for laboratory experiments back in 2013. It lacks a certain level of user-friendliness, especially for initial set up and creating a trustworthy group of devices. The current solution requires sending a command to the shell for execution and works only on Linux. In order to improve user-friendliness of DevCom on mobile devices, a new pairing solution has to be implemented.
In this thesis, we present design of DevComApp, DevComApp is designed as an extension that runs on top of DevCom and addresses the existing pairing limitations by implementing a user-friendly secure device pairing solution based on QR code exchange. The introduced secure device pairing solution focuses primarily on mobile device users. However, DevComApp cannot be implemented because DevCom is not available on modern Android versions. In order to solve this issue, DevCom should be built into an Android application. To port DevCom C code inside the Android application is a major task with multiple challenges. During the thesis work, this task could not be solved due to the discovered issue of inability of processes started inside the native C part of Android application to obtain critical permissions. We present the systematic analysis of this issue together with six possible solutions to address it.

1.3 Outline

In this master thesis, we start by describing the application scenario, i.e., remote patient monitoring in Chapter 2. Section 2.1 contains general description of the application scenario and its key components. We identify the key requirement of the application scenario in Section 2.2 and present possible complications of it in Section 2.3.

In Chapter 3, we describe the solutions and concepts that are relevant for our thesis work. Section 3.1 contains description of DevCom and analysis of how DevCom can be used in our application scenario. Section 3.2 describes concepts and a possible methodology of secure device pairing and how they are applicable in our scenario. In Section 3.3, we provide specification of QR codes followed by Section 3.4 that contains the specification of TUN/TAP kernel driver, which is central in systems like VPN and DevCom.

Chapter 4 describes the design of the solution we originally aimed to implement. We discuss assumptions made prior to design in Section 4.1. Section 4.2 revisits the general requirements from the application scenario and presents the design requirements. In Section 4.3, we discuss the design decisions made in order to address the design requirements. Section 4.4 present the system architecture of the designed system.

Chapter 5 describes the implementation process and challenges related to it. In Section 5.1, we present two studies that we conducted prior to the development process to evaluate technical requirements and to identify challenges that need to be addressed during the implementation. Section 5.2 describes the technical decisions made and the motivation behind them, and Section 5.3 describes the tools and programming languages we use during
the implementation. Section 5.4 revisits the challenges of the implementation process and describes how we address these challenges. In Section 5.5, we analyze a major issue that we discovered during the development process and propose possible solutions to the issue in Section 5.6. We summarize and describe what we have achieved with the implementation and analysis in Section 5.7.

Our conclusions from the thesis are given in Chapter 6. We discuss our contributions in Section 6.1, followed by critical assessment of our work in Section 6.2. Finally, we discuss possible future work in Section 6.3.
Chapter 2
Application Domain

In this chapter, we introduce our application scenario within the domain of health monitoring systems. Section 2.1 describes the application scenario. In Section 2.2, we discuss the key requirements of the given use case and discuss specific characteristic of the use case. Finally, Section 2.3 discusses possible extensions to our application scenario.

2.1 Application scenario

In this section, we discuss an application scenario that we regard as realistic and key components of this scenario.

In many cases a patient requires constant monitoring of physiological parameters, e.g., after-surgical state or a high risk of deterioration of health condition. Furthermore, health monitoring systems are not bounded to in-patient care anymore. Successful monitoring can be done by providing patient with all necessary equipment. It makes health monitoring possible at any time and any place. Later or real-time transmission of the health data to the hospital is possible, when the connection to the Internet is available. It does not only allow patients to do their usual activities, but also reduces the pressure on the healthcare system and hospitals. After setting up a health monitoring system (HMS) on a patient, a doctor can remotely collect and analyze the data collected by the different sensors to give the patient the best treatment. In addition to that, HMS can be extended to share data with multiple parties, not just a MD physician and a patient. It allows relatives and/or friends to access the health condition of the patient under monitoring. This could for example be used in providing a better care for elderly people, because relatives observing a critical health state might react in this situation faster than hospital workers.

Let us consider a situation, when a doctor decides to do remote patient monitoring. According to position paper by Center for Technology and Aging, this monitoring is extremely useful for chronical disease management, post-acute care management, etc. [7]. The patient is provided with necessary health sensors that are connected with patient’s phone/PC, which is then used to transmit the data collected by the sensors to a doctor via Internet. In this scenario, we can distinguish three types of devices:
1. **Receiver** (RCV), usually a doctor’s machine in a hospital that is the final recipient of the gathered data and is used by the doctor for further analysis of the patient’s health data

2. **Transmitter** (TRN), a patient’s device used to transmit the data to the receiver, and

3. **Health monitoring system** (HMS), the sensors that are collecting data and transmitting it to the TRN for further transmission to the RCV.

The use case we investigate is the one when a doctor receives data from a number of patients. The medical data is gathered by HMS sensors, transferred to the TRN device using wireless technologies, e.g., Bluetooth, and the forwarded by the TRN to the doctor’s RCV through the Internet. Figure 2.1 illustrates the application scenario.

![Illustration of application scenario](image_url)
2.2 Overall requirements

In this section, we look at the key requirements of the described application scenario.

In the described use case, we aim to fulfill seven main requirements, namely security, multiple patients, heterogeneity, dynamically changing number of devices, mobility of a patient, robustness and user-friendliness.

1. *Security*: In this scenario, we are working with sensitive health data that typically requires high level of security. It sets stricter requirements for data collection and transmission of this type of data. Thus, different cryptographical mechanisms have to be used to ensure confidentiality and integrity of data. In addition to that, we need to deliver data from health monitoring system to a transmitter device, e.g., a smartphone, securely. There are sensors that implement their own mechanisms for encryption at this stage, e.g., Cooking Hacks\(^1\), but others do not.

2. *User-friendliness*. This is a general requirement to any computer system that aims to be adopted by a specified group of users. Furthermore, the system should always provide efficiency and effectiveness in the specified context of use. In our case, we are describing a scenario where people with different background, technical skills, age and roles are involved. We have patients, a universal group that we cannot really make a lot of assumptions about. The system that aims to satisfy this group of people has to be universal and maybe as general as possible. Doctors is another group of users. For them we can make a number of assumptions, e.g., their age and education. But if the system is too complicated in technical terms and requires a lot of training to be used by doctors, it is unlikely to be adopted by them either.

3. *Heterogeneity*: In this application scenario, we have three types of devices as described above, each of them has its own purpose and role. It is important to mention that we cannot delegate the function of one device to another, in other words we cannot make a simple health sensor to become a TRN because it does not necessarily have enough transmitting and computational capabilities to do it. Each type of the device has a special set of characteristics that we want to underline:

   - HMS consists of multiple small sensors with low computational power. The communication interfaces for data transmission used by

\(^1\)Sensors developed by a company Cooking Hacks [https://www.cooking-hacks.com]
the sensors are usually wireless, e.g., Bluetooth, Bluetooth Low Energy, Wi-Fi. The sensors do not require a lot of energy to operate and usually powered by small electric batteries.

- **TRN** is a device with significant computational power, e.g., a smartphone or a tablet. These devices usually use wireless channels for communication, e.g., Bluetooth, Bluetooth Low Energy, Wi-Fi and Cellular. Due to mobility, TRN switches between different networks and interfaces, e.g., from Wi-Fi to Cellular. This leads to often changing IP address. Due to significant computational power, constant transmission of data via wireless channels and its small size, TRN tends to run out of energy.

- **RCV** is a device with high computational power, e.g., a desktop computer. It has both interfaces for wireless communication, e.g., Bluetooth, Bluetooth Low Energy, Wi-Fi, and wired communication, e.g., Ethernet. Due to a more robust transmission of data via wire, RCV is usually connected to the network this way and does not tend to change its IP address very often. Energy is not an issue for RCV, because it is constantly powered via cord (note, RCV can also be a mobile device, we discuss this case in Section 2.3).

4. **Multiple patients**: Since we want to describe a use case that is as close to real life as possible we must underline that a doctor usually has multiple patients, which means that health data collected by HMS from different patients, needs to be delivered to one receiver from different transmitters. It is very important to guarantee secure transmission of data for each patient as well as to receive information from different sources by a doctor. Thus, the solution should scale to handle multiple patients by single doctor. Furthermore, the number of patient changes dynamically, i.e., new patients require remote health monitoring and current patients finish it.

5. **Dynamically changing number of devices**: When a patient requires monitoring of extra physiological parameters that where not considered in the beginning when HMS was initially deployed, more sensors should be easily added into the existing HMS. Furthermore, it is not limited only to sensors, due to the heterogeneity of the described system it should be possible to add new TRNs, in case of original TRN is
out of order, and RCVs, so the collected data can be sent to multiple parties. Number of devices should not only increase, but decrease as well. Sensors fail, break, being lost. In these cases we need to remove them from the system.

6. **Mobility**: Use of wireless interfaces as the main channels for data transmission. Use of wires to transfer data between HMS devices is very unwanted today. Having wires all over a patient’s body is simply uncomfortable and limits patient’s mobility, according to an article by Senseware [8]. The use case when we physically connect TRN to each sensor, in order to gather data from them, makes the use of HMS extremely annoying for patients. But the most significant drawback for our application scenario is Requirement #5, i.e., dynamically changing number of devices. Sometimes sensors have to be replaced and new sensors have to be added to HMS to monitor additional health parameters. In case of wired system, this procedure is much harder to do. Furthermore, since TRN in our use case is a phone that most likely has Bluetooth, Wi-Fi and Cellular, all of data transmission will likely happen via these interfaces. The scenario when a patient uses cable to transmit data from TRN to RCV is considered by us as very unlikely to happen nowadays. Use of wireless channels for data transmission secures mobility of a patient as well as user-friendliness of the whole HMS.

7. **Robustness**: Due to the last two requirements, #5 and #6, this requirement has to be taking into consideration as well. First, dynamically changing number of devices requires that the normal operation of the system should remain in case new devices are added to the system or existing devices fail. In other words, the system should be independent on number of devices. To provide additional robustness we should consider the scenario when a failed device returns to its usual operational flow. In this case, the device should still be a part of the system, being able to transmit or receive gathered data. Second, mobility and use of different wireless channels of communication by TRN introduces the problem that network changes from time to time. As the result, TRN’s IP address changes. This should not have influence on communication neither with the HMS nor with the RCV.

### 2.3 Extended application scenario

It is important to mention that the described use case is just a single solution for doctor-and-patient model within HMS domain. However, we want to mention possible complications of
the use case. Of course, one of the ways to make the model more complex is to add more devices. However, in order to do this there should be motivation behind it. For example, more sensors can be added to HMS to monitor more physiological parameters. We consider this scenario likely to happen. We recall that HMS communicates directly only with TRN.

Another practical extension of the use case is adding multiple parties, thus, not limiting communication between only one doctor and one patient. Adding more parties to the communication group should be allowed. For example, adding one more doctor to share medical data with or a relative (or relatives) to provide a better monitoring for elderly people (Figure 2.2).

![Diagram](https://via.placeholder.com/150)

**Figure 2.2**: Illustration of the extended application scenario where the data is transferred to multiple receivers

In the Figure 2.2, Patient 1 is sharing her data from HMS both with Doctor 1 and one of her relatives. This is a possible scenario in elderly care or cases with some special health
conditions. In these cases, relatives of the patient might want to have the information about patient’s health as soon as possible. A good example can be when HMS spots a critical health conditions of the patient, the relatives might be the first to provide some measures according to the situation.

Another scenario is illustrated in the Figure 2.2 for Patient 3. She shares her health data with two doctors, namely Doctor 1 and Doctor 2. This is the scenario where more than one doctor needs to monitor and analyze data collected on sensors. For example, HMS monitors multiple parameters of vascular systems and sends the data both to patient’s general practitioner and to blood specialist for a deeper analysis.

As we introduced earlier in Section 2.1, there are three types of devices in our application scenario, namely receiver, transmitter and health monitoring system. So far we have discussed possibilities to extend our application scenario with multiple devices in HMS by adding more health sensors to the existing HMS. We also consider the scenario where multiple RCVs are added to the system. We want to emphasize that in the example where we add relatives to the communication group, they are not necessarily desktop users. If we consider a scenario where we want to notify the relatives about patient’s critical heath condition, we should do it such that they receive the notification as soon as the critical situation occurs. In this case, desktop computer might not be the best option, simply because it is not possible to use it while moving around. If the person is moving, or is not in front of the desk, or has turned the computer off, she will not be able to receive the notification. However, the ubiquity of smartphones today allows us to suggest that optimal way for relative to monitor and get notifications about patient’s health condition is to receive the data on a smartphone. It means that RCVs can also be a mobile unit which extends our original application scenario where we considered RCV to be a desktop machine.

So far we have not discussed the scenario where we extend our system with multiple transmitters per patient. We consider this scenario as a possible one. Most of the sensors in HMS utilize Bluetooth in order to communicate with the device they are paired with. The current version of Bluetooth, at the time we are working on the thesis, is Version 5.1 that is adopted Jan 21st, 2019. It already supports connection to multiple devices simultaneously, as stated in specification [9], thus, we do not want to eliminate the scenario where multiple TRNs per patient are present in our system. For example, multiple TRNs might provide robustness to the data collection from HMS and transmission of the data to RCVs. However, we consider this out of scope for our work due to the multiple different systems present on
TRN. As we see it for now, the way the information is gathered on HMS and the behavior of data transmission in our application scenario, requires a presence of multiple systems. We illustrate it in the Figure 2.3.

![Diagram of Health Monitoring System Suite](image)

Figure 2.3: Health monitoring system suite

The way HMS collects and transfers data to TRN and the way TRN later handles this data is not decided by network technology of the system in our application scenario. Instead, some Complex Event Processing systems (CEP) that run on top of the network layer systems and configurations should carry out this type of behavior. The medical application in the Figure 2.3 represents an application layer system or a program that defines the way the data is presented to end-users and manipulated by them. In the Figure 2.3, we also have two HMS sensors that monitor some health parameters, e.g., blood pressure and sugar level. The bold red line illustrates how this data flows and is handled by different modules in TRN and RCV. When it arrives on TRN from HMS, e.g., via Bluetooth, the data is passed from the network layer to the CEP running on the transmitter that decides what to do with the gathered data. In this case, the CEP makes a decision to simply pass this data to the RCV. The CEP sends the data back to the network layer which, then, transmits it to the RCV. When the data arrives to the RCV, it is passed from network layer to receiver’s CEP. This CEP makes the decision to show this data in the Medical application on the RCV, e.g., a program for health data analysis on doctor’s machine.
Note, that HMS structure, CEP and medical applications are included to describe a full-case study by our research group. However, the focus of this thesis is the technology for secure and user-friendly communications. The solution we aim to achieve has to be agnostic to the systems that will utilize it. We emphasize that the scenario of multiple TRNs per patient is possible and already supported by existing technologies. Thus, the system to be used in our application scenario should not be limited to only one TRN and could be used on multiple number of TRN devices.
Chapter 3

Background

This chapter introduces key terms and technologies referred to throughout this master thesis. We start by describing DevCom, the system that will secure network communication between doctor’s and patient’s devices, and that we aim to extend, in Section 3.1. Secure device pairing concepts are described in Section 3.2, and we outline Quick Response code technology in Section 3.3 as a suggested approach to share the needed information during pairing. In Section 3.4, we describe the TUN/TAP kernel driver that is used by different VPN solutions and DevCom to provide tunneling functionality. The description of TUN/TAP is important to understanding the issue we analyze in Chapter 5.

3.1 DevCom

This section describes DevCom. First, in Subsection 3.1.1, we describe DevCom in details, its design and technical characteristics, what issues it solves and its architecture. Then, in Subsection 3.1.2, we describe our motivation to use DevCom as a system for secure communication between different devices in our application scenario from Chapter 2.

One of the important concerns with the increasing number of devices and overall computerization of our lives is ubiquitous computing. The term was proposed by Mark Weiser back in 1988 [10]. Ubiquitous computing, the idea that “many computers serve each person” is put in opposition to personal computing, where one person owns one computer, and mainframe, where one computer is shared by multiple number if people. Furthermore, we can consider ubiquitous computing within two aspects: quantitative, people own and interact with multiple number of devices, and qualitative, the interaction with devices is unconscious and is based on seamless communication and collaboration between them. The trend in increasing number of devices fulfills the quantitative aspect of ubiquitous computing. However, according to Hansen et al., the qualitative part of Weiser’s vision is still unfulfilled:

"Files (e.g., music libraries and text documents), peripherals (e.g., printers and web-cams) and services (e.g., remote desktop and secure shell) are not always accessible on the devices currently available to users. Access to such remote resources is even more cumbersome if proper security and privacy protection mechanisms are in place [6]."
In order to address the qualitative aspect of ubiquitous computing a system called DevCom (Device communities) was designed, implemented and evaluated by researchers in University of Oslo, Department of Informatics. Since our thesis work will be based on this system, we want to underline the main characteristics of DevCom.

### 3.1.1 Description of DevCom

DevCom is a network system that provides user with a trustworthy way to communicate among distinct groups of devices simultaneously. DevCom is designed with the goal that the existing network applications and typical internet protocols should be supported. The users should not see any significant difference in performance of applications either with or without DevCom running in the system. This principle makes it easier for developers to create new applications as well as integrate DevCom into the existing applications. In fact, applications are not aware of DevCom at all, since DevCom runs on a lower level of network protocol stack. Network protocols are not modified to work with DevCom, it is DevCom that takes care of all the necessary procedures required for device associations and communications. Figure 3.1 illustrates how DevCom operates within the TCP/IP protocol suite [11]:

![DevCom Stack Diagram](https://example.com/devcom_stack.png)

**Figure 3.1: Layers of the DevCom stack (Hansen et al. 2013)**
DevCom allows user-friendly and trustworthy communication and collaboration among the devices. Even though other solutions with the same purpose exist, DevCom combines all the mentioned principles in one system. For example, VPN solutions help to provide trustworthy communication, but they are usually limited to one VPN at a time. It means that if a person would like to associate one device with more than one group, it might me a problem since only one VPN connection is supported. DevCom solves this issue and lets devices be present in more than one group at a time.

DevCom does not have a significant influence on performance of the whole operating system as well as different applications in a grade noticeable by the end users. This influence is evaluated by Hansen et al. in [6]. In the evaluation, DevCom was compared with other existing solutions in terms of many different technical parameters, e.g., CPU utilization, latency etc. In addition to this, the authors did evaluation of quality of experience as well as critical analysis of user-friendliness. The quality of experience evaluation shows that the users do not observe any significant difference in the performance of applications, even critical ones like real-time shooter games. The critical analysis gives an overview of how user-friendly DevCom is for a novice, an intermediate user, and application developers. It is apparent that DevCom requires few steps to start functioning, and that its use is completely transparent [6].

Another important issue that DevCom solves is addressing in the Internet. In DevCom it is done in a user-friendly way. A huge number of devices today is mobile. That means that devices change their location leading to roaming. By joining a new network, a device gets a new address in this network, however, it is important to maintain communication channels created by the user with other devices, namely having persistent connections. Dynamic addresses make it difficult for users to locate their devices to access files, peripherals and services, and more importantly, address changes break all open connections. Persistent connections with DevCom are possible and automatically maintained. When IP address (which is also referred by Hansen et al. as a physical address or a location of the device) changes, the device must connect again to the device community. To do this, DevCom uses permanent virtual addresses within device communities instead of changing physical addresses, thus, decoupling the identification from the location. According to Hansen et al. [6], the permanent virtual addresses are created with a self-configuration technique. This self-configuration technique is less prone to address collisions and misconfiguration than the manual effort, and is also more user-friendly. DevCom combines a known prefix together with device community names and device’s public key fingerprint to create one static unique
address for each community on the device. Link-local IPv6 addresses are chosen over the IPv4 addresses due to exhausted IPv4 address space. Furthermore, use of IPv6 addresses guarantees that the traffic for a device is routed to the DevCom virtual interface instead of physical network interface, which might have the same address.

Figure 3.2 represents how DevCom separates the device community addresses into three parts: the link-local prefix, the device community name, and the device-user identification:

<table>
<thead>
<tr>
<th>Link-Local Prefix</th>
<th>Device Community</th>
<th>Device-user ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe80:</td>
<td>6661:6d69:6c79</td>
<td>d685:64ff:fe18:5542</td>
</tr>
<tr>
<td>fe80:</td>
<td>6d6f:6269:6c65</td>
<td>d685:64ff:fe18:5542</td>
</tr>
</tbody>
</table>

Figure 3.2: Two example device community addresses (Hansen et al. 2013)

The first 16 bits are the IPv6 link-local prefix. This prefix, **fe80**, is defined by the IPv6 standard and is intended for communication within the segment of a local network or a point-to-point connection. It enables DevCom to provide a LAN equivalent network over any underlying network. The following 48 bits comprise the device community name represented in hexadecimal. Device community names might be anything a user decides them to be and do not need to be globally unique. As the result, all of the devices in the community have the same 64 (16 + 48) bits network identifier prefix followed by the unique 64 bits device identifier. This device identifier is calculated by DevCom using self-configuration mechanism to compute the fingerprint of the device’s public key. The unique 128 (16 + 48 + 64) bit addresses ensure that DevCom delivers the data only to the device community members it is intended for.

Persistent connections is a challenge in modern Internet, where devices that need to be reached, are hidden behind NAT and firewalls. DevCom intends to address this challenge by using different techniques that should allow communication between trusted devices in presence of NAT and firewalls.

The DevCom system’s architecture is illustrated in Figure 3.3 with its four main components: Overlay Manager, Data Manager, Virtual Interface Manager and Key Manager.
The Overlay Manager is the component that maintains the device communities by managing control channels between members. Control channels are dedicated TCP connections between community members. Via these channels control messages related to device community functions are sent and received. The Overlay Manager maintains control channels to all device community members, but only one control channel is established for each physical address. It means that in the case when two devices trust each other in more than one device community only one control channel exists. If the change of physical network occurs, it does not affect the self-configured virtual device community addresses.

The Overlay Manager maintains mappings between the current physical IP addresses (known as locations) of the other devices and their virtual device community addresses (known as identities). Each time a device connects to a device community and authenticates itself, e.g.,
in roaming scenarios when a device switches from Wi-Fi to Cellular connectivity, the mapping is updated. Historical records of physical IP addresses are stored in the Overlay Manager to assist creating the control channels. When a device wants to connect to a device community, the Overlay Manager attempts to reach other devices in the community at the different physical addresses that have been used in the past.

The control message payloads are encrypted with the receivers public key to prevent eavesdropping and ensure privacy. Furthermore, the control messages are signed with the senders private key to authenticate the sender and to verify the integrity of the message. Standard asymmetric cryptography is used and the 4096 bit RSA keys are provided by the Key Manager component.

Control channels are used only for control messages. The application data is sent via data channels established by the Data Manager. This separation of control and data traffic is done because, according to Hansen et al. [6], early measurements showed that using UDP instead of TCP tunnels increases throughput by 27% and decreases average latency by 23%. Data packets are therefore tunneled by the Data Manager using UDP, with a fallback to TCP if UDP is blocked. The reasoning behind the fallback to TCP is that a suboptimal connection is better than no connection.

The Data Manager maintains data channels to all devices in a device community, which are separate from the control channels. This separation is illustrated in Figure 3.3. The Data Manager must be efficient, but without compromising privacy. In order to achieve this, the Data manager encrypts and decrypts data traffic with 256 bit Advanced Encryption Standard (AES) symmetric keys that are generated by the Key Manager. Cipher Block Chaining (CBC) mode is used. The symmetric AES encryption preserves privacy but is faster than the asymmetric RSA encryption. The AES keys are exchanged over the control channel to guarantee security and reliability with the help of RSA encryption used by the Overlay Manager.

The Virtual Interface Manager creates, configures and maintains the virtual network interface. The Virtual Interface Manager separates the identification (virtual interface) and location (physical interface) of a device. It allow DevCom to enable persistent connections when device changes the physical address. It is achieved by creating permanent addresses using the self-configuration algorithm and associating the addresses to a virtual network
interface. The applications shall then use this virtual address to communicate with other
devices, while DevCom will handle data transmission via the physical network interfaces.

The **Key Manager** is responsible for creating and maintaining the asymmetric RSA keys
used by the Overlay manager and the symmetric AES keys used by the Data Manager [6].

DevCom is designed in a way that most of the functionality, such as generating keys, keeping
persistent connections, creating of the community addresses etc., is handled by DevCom
automatically. For example, users do not need to define IP ranges, manage cryptographic
keys. Furthermore, application developers can continue to create network application without
any concern for DevCom. It provides user-friendliness to DevCom, however, we discuss the
user-friendliness of DevCom in more details further in the thesis.

One of the things users of DevCom need to perform manually is pairing procedure. In order
to join a community, one device has to be paired with another device that is already a
member of the community. Since in this thesis we will focus on designing a solution to
optimize this process, we describe how it is done in the current DevCom implementation.
When a user wants a device to initially join a device community, two tasks need to be
performed: **First**, a two-way, public key exchange with the device that is already in the
community. **Second**, provision of the physical address of the device, with which the keys
have been exchanged. These two steps can be performed by sending two commands to a
DevCom daemon that runs on the device. The two commands are -t, or trust and -j, or join.
DevCom discovers other hosts in the network that run DevCom, and creates a list of
neighbors. By executing trust command and providing community name and neighbor’s
fingerprint device gets a copy of neighbors public key. Note, this step has to be performed by
both parties in order to proceed to the next step. Next step, join command, implies that both
parties that want to establish a secure channel of communication have already exchanged
their public keys. If they did this, there are now considered as peers. Now, by executing join
command, we provide a community name within which we want to communicate with our
peer, peer’s fingerprint and peer’s physical address. After this step, control channels between
two peers exist and the communication between them is now possible.
3.1.2 Use of DevCom as a network technology to secure communications between the devices

We consider our application scenario, discussed in Chapter 2, to be a well suited application of DevCom. DevCom can be used as a network system to allow secure communication between multiple devices. We revisit the Figure 2.3 from Section 2.3, in which we illustrate the system suite in our health monitoring application scenario and illustrate how DevCom can be considered within this system suite in Figure 3.4.

![Diagram of DevCom in the health monitoring system suite](image)

Figure 3.4: DevCom in the health monitoring system suite

In Figure 3.4, we place DevCom at the bottom as a network technology to handle the organization of the devices into the network and to secure communication between the devices in the network.

DevCom partly addresses the requirements from the application scenario, but additional extensions and optimizations have to be made to DevCom, so it can fulfill the requirements from the application scenario. We discuss how DevCom addresses the requirements from the application scenario, as well as what drawbacks need to be addressed by us in the system we design and implement in the scope of the thesis.

We revisit the requirements from the application scenario: security, user-friendliness, heterogeneity, multiple number of patients, dynamically changing number of devices, mobility and robustness. In the following paragraphs, we discuss how DevCom addresses
these requirements and make a conclusion about its use as a network technology for secure
device communications in the application scenario.

The first requirement is the **secure communications** between the devices. The initial step to
secure communications is pairing. By creating a new device community and adding devices
into it, we build a trusted environment of the devices where communication is encrypted, and
devices outside the community are not able to decrypt the data sent between community
members. The users need to pair the devices, therefore, ensuring a high level of security by
allowing secure communications only between the devices that are mutually paired with each
other.

Providing confidentiality and integrity is an important aspect of security in our application
domain. Due to cryptographic algorithms used in DevCom, we can address confidentiality
and integrity of sensitive health data flows between the devices. These algorithms (AES-256
and the RSA with 4096 bit key) comply with modern standards for encryption by
Datatilsynet\(^2\) [12]. SHA-1 hash function algorithm used in DevCom is not considered secure
anymore. Therefore, it should be replaced with SHA-2 (at least 256 bit), which is
recommended by Datatilsynet. We have addressed this issue during the thesis work and
describe the process in Section 5.4.1.

The next requirement is **user-friendliness**. User-friendliness of DevCom is achieved partly
and was evaluated primarily towards technical parameters, e.g., CPU utilization, latency etc.
It showed that users do not observe any significant difference in the application performance
with and without DevCom running [6].

We argue that DevCom does not address user-friendliness requirement from the application
scenario when it comes to the device pairing process. We argue that the current
implementation of DevCom lacks user-friendliness because the following steps cannot be
considered optimal. Both, starting DevCom and pairing devices is done through shell
command execution. Technical experience and pre-knowledge about commands is required
to perform these actions. Furthermore, to pair two devices pairing information has to be
provided to the command shell, e.g., device community address, physical address. This
information is hard to remember and to type into a command shell for regular users.
Therefore, the current implementation for pairing cannot be considered optimal.

\(^2\) Datatilsynet (*Norwegian*) – The Norwegian Data Protection Authority
Our next requirement from application scenario is **heterogeneity**. DevCom provides a LAN equivalent network over any underlying TCP/IP network. In theory, it means that DevCom is not device dependent and should run on device that supports TCP/IP network stack protocols, and that is almost any modern device. Therefore, DevCom can be used as an underlying network technology to associate different types of devices that are present in our application scenario into groups.

In practice, heterogeneity is not fully addressed by DevCom, since the only working solution exists for Linux devices. We aim to reduce this drawback by designing and implementing a solution that should bring DevCom on mobile devices.

The system to be used in our application scenario has to support the requirement of **multiple patients**. In Subsection 3.1.1, we described the limitations of VPN technology for the case, when a device has to be present simultaneously in multiple groups. DevCom was designed to solve the limitations of VPN technology. It provides a service for a device to be a member of multiple device communities, while keeping trustworthy communication between the devices within the community. DevCom allows to create one community per patient, while doctor’s device can be simultaneously present in multiple communities, thus, supporting the requirement for multiple patients. For each patient, who needs health monitoring, a new device community should be created. All of the required sensors (HMS) should be paired with patient’s phone (TRN) and with doctor’s computer (RCV). One device community per patient and multiple device communities for a doctor (Figure 3.5). As the result, the doctor receives data from multiple patients simultaneously, without them being able to receive and decrypt data of each other.
The extended scenario from Section 2.3 is illustrated in Figure 3.6. In this scenario, the relative of Patient 1 is added to the Device Community #1 and can receive data from Patient 1. For the Patient 3, another doctor is added to the Device Community #3 and can securely receive the data from the patient’s TRN device.
The **dynamically changing number of devices** requirement is addressed in DevCom by allowing users to add new devices into community with the pairing procedure. Another important case in this requirement is unavailability of the devices and their removal from the community if needed. In order to remove the device from the community, we need to delete a key pair associated with this community, thus, eliminating the ability to decrypt traffic sent by other members of the community. This functionality is not implemented in the current DevCom and should be addressed by designing and implementing a user-friendly solution to delete the corresponding public keys and to notify other community members about the distrust. Note, in this thesis, we do not address this issue, and leave it for future work.

**Mobility** is achieved by DevCom being fully agnostic to the underlying technology used at Data Link layer. DevCom is agnostic to the interface used to transmit data from a device. It means that both wired and wireless channels, i.e., Wi-Fi, Cellular, for data transmission are supported. Therefore, DevCom can be considered as a working solution in our application scenario, where most of the communications between the devices happen via wireless channels.
Note, however, that mobility requirement is not yet fully addressed in DevCom. This is because the heterogeneity requirement is also not fully addressed in DevCom, and no solution that would support use of DevCom on mobile devices exist.

**Robustness** is provided by DevCom with the help of multiple mechanisms that are implemented in the system. By adding new devices into the community, we improve robustness in DevCom. When only one device is available it becomes a single point of failure, but when more members are available, the likelihood of locating a member increases. This is due to more devices storing the configuration information vital for fast and reliable reconnection with the community for the devices on which network failures occurs. However, the drawback here is the growing amount of control traffic that is used by DevCom. DevCom handles reconnection to the community in case of device and network failures, so that devices are able to transfer and receive data after returning back to operation. This is possible because DevCom stores physical addresses of other community members, thus, the device is able to reach at least one of the community members to notify about its status. Furthermore, DevCom intends to maintain connections between the trusted devices placed behind NAT and firewalls, which play significant role in hospital network administration.

All issues caused by patient’s mobility and frequently changing IP address due to switch from Wi-Fi to Cellular are solved by DevCom automatically. DevCom uses virtual addresses instead of changing physical addresses, thus, it decouples identification from location. We find it particularly helpful and important in our application scenario, because it provides persistent connections between patient’s and doctor’s devices. We argue that persistent connections are extremely important for health monitoring, especially when we consider real-time information gathering, transmission and analysis.

**Summary**

As discussed in this section, DevCom can be considered as a network system to secure device communications between patient’s and doctor’s devices in our application scenario. With DevCom one doctor is able to communicate with multiple patients simultaneously. It works on top of existing network technologies and allows secure communications between devices of different types and roles.
3.2 Secure Device Pairing

Device pairing is an important topic, because it is not only the question about user friendliness, but also security. In order to introduce security into the ubiquitous computer environment in which devices pair in an ad-hoc manner, an approach called Secure Device Pairing (SDP) was proposed. In order to establish a secure communication channel between two devices that have never met each other before, a key exchange and authentication procedures first have to be performed. During our work on the thesis, we have read and analyzed a paper by Fomichev et al. [13]. The authors provide a survey and systematization of common SDP schemes. We consider this study important to perform a structured and systematic analysis of the current SDP scheme in DevCom taking into consideration the characteristics of the application scenario. It helped us to understand the limitations of the current DevCom specification and to design a solution, according to the principles defined by Fomichev et al. in [13]. In this section, we describe the key ideas of the paper and discuss the how this ideas are relevant for our work.

The authors of the paper [13] gave the following contributions. Firstly, they presented a system model and terminology, which facilitates precise description and reasoning about different SDP schemes based on three components: physical (PHY) channels, human-computer interaction (HCI) channels and application classes. Then, Fomichev et al. used this model to classify the existing SDP schemes followed by identification and analysis of the security weaknesses found in these schemes. Finally, they defined principles for designing robust SDP schemes.

Generalized pairing procedure can be considered as depicted in Figure 3.7. The scenario illustrated in Figure 3.7 consists of two device, D1 and D2, which do not share any prior knowledge and would like to pair. To do it, the devices need to exchange a secret information and ensure that it came from the correct party. Furthermore, no third-party should obtain this information during paring. In order to achieve pairing the following three steps need to be performed: 1) discovery, 2) secret exchange and 3) verification.
Figure 3.7: Generalized pairing procedure (Fomichev et al. 2018)

This scenario reflects also the current implementation of pairing in DevCom. Step 1 is performed by LAN discovery of the devices in the network. Then, the secret (key) exchange is performed by DevCom as the result of executed trust command, followed by execution of join command to establish control and data channels for communication (Step 2). Finally, the user, who started the pairing procedure, would be able to observe (verify, Step 3) successful pairing by seeing that the device has (or has not in case of error) joined the group of other devices.

Fomichev et al. define a terminology that is important in the scope of SDP. We provide the description of this terminology and discuss how it is relevant to the current DevCom specification and the application scenario from Chapter 2.

- **Pairing** is the establishment of a secure communication channel between two or more devices.

- An **application class** is a representation of a particular pairing scenario. It is determined by the degree of involvement and level of control that a user has over the pairing devices. An application class covers the use-cases that share similar security threats and objectives.
• **An SDP scheme** is composed by the procedures, cryptographic protocols and the motivating application class needed to securely pair devices.

• **An SDP method or SDP procedure** is the sequence of actions required to be performed to execute an SDP scheme. While considering method and procedure interchangeable, the authors avoid the synonym protocol, due to its strong association with cryptographic protocols.

• **A party** is someone or something who controls one or more devices participating in an SDP procedure.

• **A security domain** is the set of devices, data, policies and intentions that a single party controls. Every device belongs to a security domain, however multiple security domains might be involved in a given application class.

• **A channel** is a means by which communications occur in an SDP scheme, e.g., a physical medium or human-computer interaction.

• **A PHY channel** is a communication channel that allows data transmission or acquisition over a physical medium and can be described by their objective physical characteristics.

• **An HCI channel** is a means of communication where a user acts as the channel. In this case, the communications happens by user interaction with the devices involved. For example, a user might read the information from the display of two devices, and then enter confirmation that they match into one of those devices [13].

**Pairing** in DevCom is a set of steps, a user has to go through in order to establish a secure channel of communication, i.e., trust operation and join operation. Every time when a user wants to add a new device into the group, pairing needs to be done. **Application class** is defined by a pairing between particular devices in our application scenario. For example, a patient wants to pair her smartphone with doctor’s computer – that is one application class. When a patient wants to associate a new sensor in health monitoring system (HMS) with her smartphone that acts like transmitter (TRN) – that is another application class. **SDP scheme** in the current DevCom implementation consists of 1) set of procedures that the user must do, as well 2) steps that DevCom does prior to pairing and 3) the motivating application class. **SDP procedure** is performed by a patient (or doctor) to execute a corresponding SDP scheme. The
current SDP procedure in DevCom is the following one: the user who has control of two devices, connects them into one network, starts DevCom on each of them, provides group name for association on one device, exchanges public keys by performing trust operation and establishes control channels by performing join operation. A party in DevCom in our application scenario is either a patient, or a doctor, or a relative. A security domain is represented by a set of devices, policies and intentions. For example, the patient, who has control over HMS and TRN devices with intention to collect the health parameters for further transmission of this data to the doctor.

Based on the analysis by Fomichev et al. [13], we can identify important characteristics of our application scenario. The variety of the devices reflects in the variety of different PHY channels that can be used to pair different devices with each other. It means that there is no universal SDP scheme that can be implemented to be used for pairing in our scenario. For example, HMS devices do not have screen or microphone, therefore, an SDP scheme via visual and audio PHY channels cannot be used for their pairing. On the other hand, these PHY channels can be considered user-friendly to pair a TRN device, a smartphone, with a RCV, a computer or another smartphone. When implementing an SDP scheme via PHY channels, we should also analyze possible threats. Threat analysis is possible only when understanding the security domain and the application class, which represents a particular pairing scenario. When designing an SDP scheme based on HCI channels, we should take into consideration the users of the devices, who will perform the pairing procedures. HCI channels should be analyzed in terms of security and usability properties.

We stated that no universal SDP scheme exists in our scenario. Furthermore, Fomichev et al. state that it is impossible to find one. Instead of doing it, we should adapt another approach that is based on the analysis of multiple factors, described in the previous paragraph, namely application classes, the environment and social context, potential attacks, the data to be exchanged and availability of the channel. In other words, we should focus on implementing multiple SDP schemes and let the user choose the one that is best suited in the particular pairing scenario. Furthermore, in order to make a pairing scheme secure and user-friendly a continuous feedback loop with the user should be provided in the system. It can help to mitigate many aspects of user misbehavior which is in many cases the security breach of the SPD scheme.

Taking into consideration the principles discussed by Fomichev et al., we argue that the SDP scheme in DevCom cannot be considered user-friendly for the application scenario from
Chapter 2. The current DevCom SDP scheme was not designed with the thought about application classes and implements only one SDP procedure that requires deep understanding of DevCom and high technical skills.

As Fomichev et al. propose [13] that design of SDP scheme should be application class driven. An application class covers a set of similar SDP use cases, each of which involves a similar degree of involvement and level of user control over the pairing process. In our application scenario, application class can be defined as social. Social application class represents a case where two different users would like perform pairing between their personal devices. In our application scenario, we consider these two users to be a patient and a doctor, or another example can be the patient and her relative. In social application class pairing typically involves user interaction. Furthermore, this user interaction likely to happen in a safe environment, like doctor’s office or patient’s house. However, a public class can also be considered applicable in our application scenario. Public class corresponds to the case where a user possesses one device but has no control over another device to be paired with. For example, this can be a case when a patient pairs a new phone (TRN) with doctor’s computer in the hospital from home.

The selection of PHY and HCI channels is the next important step in designing a suitable SDP scheme for the application scenario. Different devices have different technical characteristics, therefore, different SDP schemes that utilize these channels should exist. For example, pairing a patient’s smartphone with doctor’s computer can rely on exchange of visual information, since both devices have displays to show this information on. But the same channel is not possible to use for pairing health sensors due to lack of displays. SDP for sensors should rely on PHY channels that are available to them, e.g., Bluetooth, Wi-Fi.

We also need to carefully consider HCI channels when developing an SPD scheme. In our application scenario we have different people with different roles and technical skills. A patient that does not have any a priori knowledge of DevCom is not able to perform pairing of two devices. Furthermore, additional parameters should be taken into consideration, e.g., age and health condition. For example, an SDP scheme that will require user to read and verify small text on the screen. In the elderly care application scenario, where a lot of patients struggle with reading small text without glasses, this SDP scheme cannot be considered user-friendly.
To conclude, user-friendliness of DevCom can be significantly improved for the use in our application scenario. It is possible to implement a secure and user-friendly device pairing scheme by taking into consideration multiple factors, such as application class, types of devices and PHY channels, as well as understanding of the parties and their capabilities.

### 3.3 QR codes

QR code (abbreviated from Quick Response code) is a type of matrix barcode (or two-dimensional barcode). QR code specification is standardized and described in ISO/IES 18004:2015 “Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification”. According to this standard [14], QR code is a matric symbology, in which the symbols consist of an array of nominally square modules arranged in an overall square pattern. A unique finder pattern is located at three corners of the symbol and provide easy location of the symbols position, size and inclination.

We provide a description of the basic characteristics of QR codes. All of the technical details can be found in [14].

QR Code is a matrix symbology with the following characteristics:

1. There are two different symbol **formats** allowed:
   
   a. QR code, with full range of capabilities and maximum data capacity;
   
   b. Micro QR code, with reduced overhead and restrictions on capabilities and reduced data capacity (compared with QR code symbols). Note, the Micro QR code format is not relevant for our work. All of the specifications and characteristics we will provide further in the text refer only to full QR code format.

2. **Encodable character set**, which defines the encoded data symbols in the QR code:
   
   a. numeric data (digits 0 - 9);
   
   b. alphanumeric data (digits 0 - 9; upper case letters A - Z; nine other characters: space, $ % * + - ./ : );
   
   c. byte data;
   
3. The size of the symbol calculated in square modules: from $21 \times 21$ modules to $177 \times 177$ modules (Versions 1 to 40, increasing in steps of four modules per side).

4. Selectable error correction (ECC) allow four levels of Reed-Solomon error correction, referred to as L, M, Q and H in increasing order of capacity, that provide recovery of up to 7%, 15%, 25% and 30% of the symbol codewords respectively.

5. Data characters per symbol, defined by the encodable character set and symbol’s size. For example, for a maximum QR code symbol size, Version 40 with ECC level L a maximum amount of encoded data is limited to 7089, 4296, 2953 and 1817 characters for numeric, alphanumeric, byte and Kanji encodable sets respectively.

Each QR code symbol is constructed of nominally square modules, the smallest elements in QR code that can be either black (nominally, a binary 1) or white (nominally, a binary 0). Modules are arranged in a square array, or a symbol. The symbol consist of an encoding region and function patterns that do not encode any data. These function patterns are a finder, a separator, timing patterns, and alignment patterns. The symbol also needs to be surrounded on all four sides by a quiet zone border that has to be free of any markings. Its width for QR code symbols should be at least 4 modules. Figure 3.8 illustrates the structure of a Version 7 symbol. QR codes are orientation independent, which means that they are readable in reflection and any rotation.

![Figure 3.8: Version 7 QR code symbol (ISO/IES 18004:2015)](image)
There are forty sizes of QR code symbol referred to as Version 1, Version 2 and up to Version 40. Version 1 measures $21 \times 21$ modules, Version 2 measures $25 \times 25$ modules and so on increasing in steps of 4 modules per side up to Version 40 which measures $177 \times 177$ modules.

QR code has also error correction capability to restore data. In case when QR code is dirty or damaged, parts of the encoded information might not be possible to read. ECC allows successful reading of the QR code in such cases. There are four error correction levels available for users to choose according to the operating environment. Raising this level improves error correction capability but also increases the QR code size. To select error correction level, various factors such as the operating environment and QR code size need to be considered. Level Q or H may be selected for factory environment where QR codes get dirty, whereas Level L may be selected for clean environment with the large amount of data. According to QRCode.com [15], typically, Level M (15%) is most frequently selected.

3.4 TUN/TAP

In this section, we describe the universal TUN/TAP device driver that provides packet reception and transmission for user space programs. This description is relevant to the issue that is analyzed in Section 5.5. The full specification and guidelines can be found in [16].

3.4.1 Description of TUN/TAP

TUN/TAP is a virtual network kernel device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program. In order to use the driver a program has to open a device special file `/dev/net/tun` and issue a corresponding `ioctl()` call to register a network device with the kernel. Then, depending on the chosen options, a network device will appear as `tunXX` or `tapXX`. When the program closes the file descriptor, the network device and all corresponding routes will disappear.

Depending on the type of device chosen, the user space application has to read/write either IP packets (in case of TUN), or ethernet frames (in case of TAP). The TUN is Virtual Point-to-Point network device. TUN driver was designed as low level kernel support for IP tunneling. The TAP is a Virtual Ethernet network device, which was designed as low level kernel support for Ethernet tunneling.
The user space applications are provided with two interfaces:

- Character device: /dev/tunX (for TUN) or /dev/tapX (for TAP).
- Virtual interface: tunX (for TUN) or tapX (for TAP).

User space application can write IP or Ethernet frame to the corresponding character device using the file descriptor returned by the open() function call. Kernel, then, will receive this frame from the virtual interface. In the same time every frame that kernel writes to the virtual interface will be read by the user space application from the character device. The user space application choose the configuration, either TUN or TAP, by setting flags in the ioctl() call [16].

In order to create a virtual interface the following sequence of steps needs to be done [16]:

1. **Create device node:**

   ```
   $: mkdir /dev/net (if it doesn't exist already)
   $: mknod /dev/net/tun c 10 200
   ```

2. **Set permissions:**

   ```
   $: chmod 0666 /dev/net/tun
   ```

3. **Load TUN/TAP driver module** (either autoloading every time the device is opened or manual loading every time it is required).

4. **Allocate network device** (char *dev is the name of the device with a format string, e.g., “tun%d” that will be overwritten with real device name, e.g., “tun0”):
```c
#include <linux/if.h>
#include <linux/if_tun.h>

int tun_alloc(char *dev)
{
    struct ifreq ifr;
    int fd, err;

    if ( (fd = open("/dev/net/tun", O_RDWR)) < 0 )
        return tun_alloc_old(dev);

    memset(&ifr, 0, sizeof(ifr));
    /* Flags: IFF_TUN - TUN device (no Ethernet headers)
     *        IFF_TAP - TAP device
     *        IFF_NO_PI - Do not provide packet information
     */
    ifr.ifr_flags = IFF_TUN;
    if ( *dev )
        strncpy(ifr.ifr_name, dev, IFNAMSIZ);

    if ( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
        close(fd);
        return err;
    }
    strcpy(dev, ifr.ifr_name);
    return fd;
}
```

After these steps, the network interface is allocated and the program has control over the
device associated with this virtual interface. The virtual interface needs to be configured (set
IP addresses, MTU length, etc.) and brought up. Then, the other user space applications can
use this configured network interface, and the kernel will forward all the data send to this
interface to the program that controls the associated character device.

TUN/TAP driver is used mainly for tunneling by virtual private network applications
(OpenVPN, VTun), NATs (TAYGA), for virtual-machine networking (VirtualBox, coLinux)
and to connect real machines with network simulators (ns-3).

### 3.4.2 TUN/TAP in DevCom

TUN/TAP plays a critical role in DevCom. Use of TUN/TAP and virtual interface
configuration is the task of the Virtual Interface Manager of DevCom. Virtual Interface
Manager uses TUN/TAP driver to create a virtual Point-to-Point interface (TUN). When
started, DevCom configures the name of the virtual interface, `trampX`, and assigns the
device community addresses for this device. For example, on the device with two
communities, **work** and **family**, DevCom configures the virtual interface **tramp0** and assigns two IPv6 community addresses to it:

inet6 addr: fe80:776f:726b:0:9d88:e19d:f514:d42/64 Scope:Link  

Consider an application that wants to use DevCom. In order to communicate with other devices in the community, it needs to send its packets through this virtual interface, **tramp0**. All these packets that kernel receives at this virtual interface are sent to a character device /dev/net/tun. Since DevCom controls the network device and has a file descriptor attached to /dev/net/tun, it receives all the data that is sent to it by the kernel, and processes it, e.g., encrypts it. After that, DevCom sends the encrypted data to the other member in the community over UDP or TCP. On the other side, the receiving party passes the received packets to DevCom, which decrypts them and writes to the TUN device. The kernel handles the packets like they came from a physical interface.
Chapter 4
Design

In this chapter, we present the design of our system called DevComApp. DevComApp is the extension of DevCom for mobile operating system with remote health monitoring in mind. Its purpose is to make DevCom available on mobile devices that act as transmitters (TRNs) or receivers (RCVs) in our application scenario, and to implement a new user-friendly secure device pairing mechanism. The design of the app is motivated by the requirements of the application domain.

We start by discussing design assumptions we make in Section 4.1. Section 4.2 revisits the requirements of the application scenario from Chapter 2. In Section 4.3, we discuss our design decisions and how they address the requirements from the application scenario. Section 4.4 presents a design solution and its system architecture.

4.1 Design assumptions
DevComApp is the extension of DevCom targeted for use on mobile devices, e.g., smartphones, tablets. After the installation it should allow users to start DevCom on the device, create new communities and allow to pair the device with other devices with the help of user-friendly secure device pairing procedure.

Prior to the process of designing DevComApp, we make a number of assumptions. The application scenario from Chapter 2 includes a range of devices, roles and systems. The scenario has different types of devices, namely HMS, TRN and RCV, and different roles: patients, doctors and third-parties, e.g., relatives or friends. As discussed in Section 2.3, there are different systems in place, e.g., CEP, user applications and network technologies. All of the systems that will be designed, implemented and studied in the future, should address the overall requirements of the application domain. We make following assumptions:

1. This thesis focuses only on a technology to allow secure communications in our application scenario.

2. This thesis focuses on a solution to be used on mobile devices that act as TRN or RCV in the application scenario, e.g., smartphones.
Assumption #1 means that we do not focus in this thesis on systems, such as CEP, user medical applications. Our primary topic of research is a network technology to allow secure communication between devices. A possible solution to achieve this already exists. In Subsection 3.1.2, we elaborated on why an existing system, DevCom, can be used in our application scenario to provide a secure communications between the devices.

With the second assumption we aim to focus on the extension of DevCom for the use on mobile operating systems, thus, leaving out of scope devices in health monitoring system (health sensors) and desktop receivers. Our goal is to design a system for use on smartphones that can act like transmitters (or receivers in the extended application scenario from Section 2.3). The current code runs only on Linux. Hansen et al. [6] state that it is possible to run DevCom on Android under certain conditions, but early testing shows that DevCom is not possible to run on modern Android.

### 4.2 Design requirements

We develop DevComApp to make DevCom available on mobile devices and to provide a more user-friendly pairing scheme. DevCom partly fulfills the requirements from the application scenario. In this section, we list the requirements of DevComApp from a design viewpoint and the requirements from the application scenario.

#### 4.2.1 General requirements

In this section, we revisit the requirements of the application scenario from Chapter 2, and provide a short explanation to each of them. We cover all the requirements in the same order as they are presented in the Section 2.

1. **Security**, because in the application domain we operate the sensitive health data.
   
   - Confidentiality and integrity of health data upon transmission.
   
   - Devices shall be paired in a secure way.

2. **User-friendliness**, because patients and doctors are people with different academic background, technical skills, age and habits and use different devices.
   
   - A user-friendly secure device pairing scheme is in place.
   
   - The system is transparent to the applications running on top of it, e.g., CEP.
• The system is optimized for different types of devices and operating systems.

3. **Heterogeneity**, because the system needs to address devices with different roles and technical characteristics.

• Different devices with different characteristics, i.e., HMS, TRN, RCV.

4. **Multiple patients**, because in a real-world scenario one doctor has multiple patients.

• One doctor communicates with multiple patients.

5. **Dynamically changing number of devices**, the system should not be dependent on a constant number of devices.

• Additional devices can be added to the system.

• Existing devices can be removed from the system.

6. **Mobility**, because a number of devices are mobile (HMS, TRN) and use primarily wireless channels for communication.

• Support for wireless channels of communication, e.g., Wi-Fi, Cellular, Bluetooth.

• Multihoming.

7. **Robustness**, because it guaranties persistent connections and reliable communication between the devices.

• The system is able to recover from the failure of particular devices.

• Communication channels can are able to recover from roaming.

These are the requirements that are partly addressed by DevCom already (see Section 3.1.2), and should not be compromised by DevComApp.

### 4.2.2 DevComApp requirements

In this thesis, we design a solution that can be considered as an extension of DevCom, which is made and optimized for the use on mobile devices, i.e., smartphones and tablets. Initially, we made an assumption that DevCom was available on mobile devices. Later, we discovered that this assumption was wrong. Therefore, we added the requirement that DevCom should
be available on mobile devices via DevComApp. Design requirements for our solution aim to fix the drawbacks of the current DevCom implementation. These drawbacks result in DevCom not being able to fully address a number of the requirements from the application scenario. We identified these issues in Section 3.1.2. In order to eliminate these drawbacks we consider the following design requirements for the system we implement in the thesis:

1. DevCom should be available on mobile devices in order to address heterogeneity and mobility requirements.

2. Hostnames should be used for pairing in order to address robustness and user-friendliness requirements.

3. A user-friendly secure device pairing mechanisms should be in place in order to address security and user-friendliness requirements.

These are the requirements that we set for the solution we design and implement in the scope of the thesis to make DevCom available for user-friendly use in our application scenario of health monitoring systems.

4.3 Design decisions

In this section, we discuss different design decisions and choices that we make to address the requirements from the application scenario and discuss our motivation behind each of the design decisions. Our design requirements, discussed in Section 4.2.2, aim to address the drawbacks of the current DevCom implementation. In the following subsections, we discuss design decisions made to address the design requirements of DevComApp, and the motivation behind them. Section 4.3.1 describes the choice to make DevCom available on mobile operating system. In Section 4.3.2, we discuss how to provide a better user-friendliness and robustness by allowing use of hostnames for pairing. Lastly, Section 4.3.3 contains motivation behind using QR codes exchange to provide new user-friendly secure device pairing scheme for DevCom.

4.3.1 DevCom on mobile operating system

In Section 3.1.2, we discussed that mobility and heterogeneity are not fully addressed by DevCom in the current implementation. The reason to that is the unavailability of DevCom on mobile devices. So far, DevCom supports only Linux, but in our application scenario we have different types of devices, e.g., sensors, smartphones, PCs. In order to use DevCom for
secure communication between different types of devices, DevCom has to be available to run on these different types of devices.

We revisit Assumption #2 from Section 4.1, that the scope of this thesis is mobile devices that act like transmitters/receivers, e.g., smartphones. Therefore, we decide to design and implement a system that will support use of DevCom on a smartphone. By making DevCom available on a mobile operating system used in modern smartphones, we will improve the way DevCom addresses mobility and heterogeneity requirements from the application scenario.

### 4.3.2 Use of hostnames instead of physical addresses for join operation

Use of only IP addresses for device pairing is considered as a limitation of the current DevCom implementation in our application scenario. The IP address of the device may change over time. Notifying about it is handled by DevCom automatically. The device that changes its physical address will send its new physical address to other devices in the device community via control channels. Pairing in DevCom is based on the use of IP addresses of the pairing devices. But if the device that we want to pair with, changes its physical address prior to the pairing itself, then no channels exist to notify about this change. Therefore, pairing will not be successful, since it relies on the real-time IP addresses. If we want to implement a secure device pairing scheme that relies on a distribution of static data, such as QR codes, key files etc., we should not limit its use to in real time only. In this case, a better solution would be to rely on distribution of information that is less likely to change over time, e.g., a hostname and a domain name.

Section 3.1.1 discusses that there are two essential stages in creating a communication channel between two devices in DevCom:

1. A **trust operation** between two hosts in the network to exchange their public keys and

2. a **join operation** between two peers (devices that have exchanged public keys) to establish a control channel to support secure communications.

The current implementation of DevCom uses the physical addresses of devices to perform the join operation. During the join operation, device A needs to specify a physical address of its peer B, which A wants to establish a secure channel of communication with. After that, device A writes this address into a cache file so it can connect to the peer in the future. If one
of the peers changes its address, it will notify the other party about its new address via control channel and this new address will be written to the cache file by other peers.

We argue that use of only physical addresses for the join procedure is not optimal in our application scenario. In order to pair the patient’s phone with the doctor’s device, e.g., a desktop computer in the hospital, we should take into consideration that the doctor’s machine is unlikely to have a static public IP address. A common practice from industry to solve this issue is to use Fully-Qualified Domain Names (FQDN), a combination of a hostname and domain name written in a specific form, in order to identify a remote machine to connect to [17]. The use of FQDN for the join operation will provide a better robustness which is one of the requirements from the application scenario.

If the IP address is the only way to identify a remote peer to establish a communication channel with, we might experience a situation when the peer is unavailable on the given address. However, if use of FQDN is supported, we can access the remote machine even if its IP address has changed. It is extremely useful when we consider an SDP scheme that relies on a distribution of static data, e.g., QR codes. If we identify the peer we want to connect to with a FQDN, we can encode this information and use it anytime. The requirement in this case is a configured DNS server that resolves the hostname into the corresponding IP address. It also introduces a drawback of possible DNS spoofing attack scenario, but significantly improves distribution of QR codes and key files over time.

A practical example to illustrate a benefit of using FQDN over IP addresses is the following scenario. Assume we use QR codes to encode all the necessary information for pairing. Now consider a scenario where patient’s smartphone (TRN in our application scenario) is out of order. The patient wants to add a new smartphone to the device community to replace the old one. To do that the new smartphone has to be paired with the doctor’s computer. If we used the IP address of the doctor’s computer when we generated a QR code for the first smartphone, the chances that we could reuse the same QR code to pair the new smartphone with the doctor’s computer are low. However, if we use the hostname of the doctor’s computer and encode the hostname into the QR code, we can use the same QR code on the new smartphone again, even if the IP address of the doctor’s machine is different now.

Note, however, that the use of IP address for pairing is not an issue in the current DevCom implementation, since the current SDP procedure is based on LAN discovery of other hosts that run DevCom. In this case, the use of hostnames is not relevant, since we operate with the
physical addresses in the network in real time. If one of the hosts suddenly changes its physical address, DevCom will announce the new physical address and other hosts in the network will see this new address.

We want to emphasize that we do not propose to eliminate the use of physical addresses from DevCom completely. First of all, we discuss the use of FQDN only in a context of the join operation. DevCom still uses physical addresses to maintain connections inside a community. Second, we still want to use physical addresses for join operation when use of hostnames is not possible. For example, a patient’s smartphone that is used for personal needs, is unlikely to be assigned a hostname. Therefore, we make the design decision to extend a functionality of DevCom to perform a join operation not by a physical address only, but by a hostname too. The device owners, then, can choose a scheme suitable for them.

4.3.3 QR codes to provide a user-friendly SDP in DevCom

In Section 3.1.2, we point out that DevCom current pairing procedure lacks user-friendliness. In order to address the user-friendliness requirement from the application scenario, a new secure device pairing scheme should be implemented. Since the focus of this thesis is on mobile devices, we design an SDP scheme tailored to mobile devices. We propose an SDP scheme that uses a visual PHY channel through QR code exchange.

Fomichev et al. [13] state that there is a tradeoff between security and user-friendliness of a pairing scheme. We propose an SDP scheme that relies on exchange of QR codes that encode all the necessary information for pairing. In Section 3.2, we discussed how to design a new SDP scheme in DevCom for the use in our application scenario. We stated that application class of the SDP scheme in our scenario is social. It means that a certain degree of security is already achieved, since eavesdropping is less likely to happen in this type of environment. The initial pairing happens in the doctor’s office, where a patient reads the doctor’s device QR code with her smartphone. In addition to that, pairing in DevCom requires a mutual trust operation: both parties have to trust each other before the communication channel is established. It means that the attackers need access to one of the devices in the community in order to approve their membership. And even if this happens, the attacker will not be able to decrypt the traffic from patient’s TRN to doctor’s RCV, since it is encrypted by a symmetric session key.
QR code exchange is easy-to-perform on mobile devices. This procedure is fast and can be considered universal due to ubiquitous presence of cameras in modern smartphones. All of the phones also have a screen to display a QR code.

QR code exchange does not require the party, who is performing pairing, to have technical skills. Thus, this solution can be considered user-friendly in our application scenario, where patients and doctors are people with different background and knowledge. It does not rely on HCI channels that might not be user-friendly for patients in elderly care. If any assistance to the patient to perform pairing is needed, it can be easily provided by a third-party, e.g., a doctor.

To perform pairing, a QR code should contain the information about the device that is sufficient to do the trust and join operation by another party. In the current DevCom implementation this information is a community name, a fingerprint, a physical address and a public key.

We propose to use the following information to encode into the QR code:

1. Device community address (contains community name and fingerprint)
2. IP address or FQDN (depends on network configuration)
3. Public key

In order to pair a patient’s mobile device and a doctor’s device in DevCom with the help of QR codes, a following SDP procedure can be performed:

1. The doctor creates a new community on her device.
2. The doctor generates a QR code that encodes device’s community address, FQDN and public key.
3. The patient scans the generated QR code from the display of the doctor’s device.
4. The new community is generated on the patient’s device and all the information from the scanned QR code is stored.
5. The patient generates a QR code that encodes her device’s community address, IP address and public key.
6. The generated QR code is transferred to the doctor’s device by reading it from patient’s device’s display.

7. The information about patient’s device is stored on doctor’s device now.

8. Any party performs join operation by the IP address/FQDN of the other device.

After these steps, pairing is over and communication channels between the devices are established.

We emphasize that this is just one SDP scheme that we design to be used in our application scenario in order to address the missing user-friendliness requirement of pairing procedure in DevCom. However, as stated by Fomichev et al. [13], more SDP schemes can be developed based on selection of PHY and HCI channels and application classes. QR codes exchange SDP scheme is considered user-friendly for the use on mobile devices that have a camera and a screen, but cannot be applicable to pair health sensors.

### 4.4 System architecture

This section presents a system architecture for the design of DevComApp, the extension of DevCom that aims to bring DevCom on mobile devices and to provide user-friendliness to pairing with the help of QR code exchange.

Taking into consideration all of the design requirements and design decisions, we propose a following design solution: a mobile application that runs DevCom on mobile device and has functionality to perform pairing in a secure and user-friendly way with the help of QR codes. DevComApp should rely on the existing DevCom codebase as part of its own codebase so DevCom can be run on mobile devices via the app. QR codes should encode all the necessary information to perform pairing of two devices, namely global IPv6 device community address, physical address or a hostname of the device and its public key.

The current DevCom codebase should be reused to a large extent on mobile operating system. DevComApp will run on top of DevCom and should be able to send commands to DevCom, so it can perform pairing according to the parameters it gets from DevComApp. DevComApp should provide a graphical interface to interact with DevCom instead of command shell that is used in DevCom now. It should allow users:
1. To generate a QR code that can be used for pairing with the device by other devices.

2. To open camera to scan another device’s QR code to pair with.

3. To create a new key pair.

4. To create a new community.

5. To give an overview of the communities that the device is a member of, and the other devices that are present in the communities.

6. To start and to stop DevCom.

These are the points we consider important for making DevCom available on mobile devices in the scope of this thesis. In the future, DevComApp can be extended by adding more functionality, e.g., new secure device pairing schemes can be implemented in addition to QR code exchange.

In terms of architectural pattern, DevComApp represents a simple multitier pattern with four tiers (or layers): Presentation, Application, DevCom and Datastore (Figure 4.1).

The **Presentation tier** is represented by a set of GUI components that the user interacts with. The **Application tier** is responsible for interactions with the DevCom and the Datastore and...
updates of the presentation layer. The **DevCom** layer is DevCom itself that is responsible for secure communication between the devices in the device communities. The **Datastore** stores all of the required data necessary for DevCom and the Application layer to operate, e.g., public and private keys, QR codes, cache files etc.

To better illustrate the architectural pattern of DevComApp, consider the following scenario: a patient generates a QR code on her device (Figure 4.2). To do this, the patient presses the button “Generate QR code” which is a component in the Presentation layer. The Presentation layer sends a signal to the Application layer. The Application layer component gathers all the required information from the DevCom and the Datastore, namely device community address, hostname/IP address and the public key, and generates a new QR code. The generated QR code is then saved in the Datastore, and the Application tier can now update the Presentation tier so it displays the generated QR code.

![Figure 4.2: “Generate QR code” sequence diagram](image)

Note, that DevComApp is the extension of DevCom that should not necessarily target mobile devices only. We revisit the design assumption #2 made in Section 4.1, which we made to focus only on mobile devices in the thesis. However, the proposed design and system architecture are independent on the operating system, and the same pattern can be applied to the desktops. In this case, the first design requirement of DevComApp, which is availability of DevCom on mobile devices, is not relevant. This requirement should be changed to reflect the devices that DevComApp is considered to be used on. For example, if we consider Windows desktop machines, then the first design requirement of DevComApp should be
changed to, e.g., “DevCom should be available on Windows desktop computers in order to address heterogeneity requirement”. Note, however, that the use of the QR code exchange SDP scheme on desktops might be limited due to the lack of camera module, required for reading the QR code. On the other hand, other SDP mechanisms can be implemented in DevComApp Application layer to target different types of devices, but DevCom module should remain untouched.
Chapter 5
Implementation and Systematic Analysis

In this chapter, we discuss the implementation process of DevComApp according to design we presented in the previous section. During the implementation process we faced unforeseen conditions, which have made the implementation of our system impossible. The unforeseen conditions is the result of the technical choices we made during the implementation process. These choices were made to provide, what we argued, a best possible implementation of the solution, designed in Chapter 4. We decided not to change the technical decisions to less optimal that could have helped us to finish the implementation. Instead, we shift the focus of the thesis to the structured systematic analysis of why the selected implementation method did not work out in the end. We believe that other researchers and developers can benefit from this analysis, when designing and implementing a similar project for Android mobile operating system. We also suggest possible solutions and describe the conditions, on which these solutions can be adopted.

In Section 5.1, we discuss the evaluations we did prior to the development process, namely DevCom codebase analysis and QR code analysis. Section 5.2 discusses closely technical decisions we make prior to implementation. Section 5.3 provides an overview of the programming languages and tools that are used in the development. Section 5.4 describes the implementation process. In Section 5.5, we analyze the failed attempt to run DevCom inside an Android application (also referred as APK) using native C code. Section 5.6 presents a possible solutions to the issue we have faced. Finally, Section 5.7 concludes the implementation chapter.

5.1 Studies prior to implementation

5.1.1 Evaluation of screen and camera requirement

Central part of DevComApp is secure device pairing using QR codes. In this section, we provide a pre-study that we conducted prior to the implementation process in order to define the camera and screen requirements for reading and displaying a QR code.

In Section 4.3.3, we proposed the information that needs to be encoded into a QR code, to pair two devices with each other, namely:
1. Device community address (contains community name and fingerprint)
2. IP address or FQDN (depends on network configuration)
3. Public key

In order to understand the minimum requirements for camera and screen, we investigate a worst case scenario and calculate the largest amount of data to be encoded into a QR code.

First of all, we need to understand the character set we should use for encoding. The only option for us is to use a byte encoding, because the public key can contain lower case and symbols that are not allowed by alpha-numeric encoding. During a byte data encoding scheme we use 8 bits to encode one character.

Then, we need to define a maximum length of the pairing information to be encoded. A maximum size of the device community address is 39 characters (8 groups of 4 digits and 7 semi-colons). The maximum size of the IP address/FQDN part is limited to the max length of FQDN, which is 253 characters. The public key in DevCom can be shortened by not encoding the standard fields “-----BEGIN RSA PUBLIC KEY-----” and “-----END RSA PUBLIC KEY-----”, which can later be appended by DevComApp after scanning the QR code. In this case, 714 characters have to be encoded. Finally, we need a delimiter to separate different blocks of information within one QR code. For this purpose we decide to use space character that is not present in any other information that will be encoded. The result is 39 + 253 + 714 + 2 = 1008 characters to be encoded. Thus, the required amount of bits to encode this data is 8064.

The next step is to choose the error correction level of the QR code. As discussed in Section 3.3, there are four error correction levels to choose from, i.e., L, M, Q and H that respectively allow to recover up to 7, 15, 25 and 30 percent of the unreadably data. We choose level M, since it is the most common one, however, for the use in a clean hospital environment level L can also be considered.

Using a table “Number of symbol characters and input data capacity for QR code”, provided in ISO/IES 18004:2015 [14], we can define a minimum version of QR code that fits our requirements. In our case, it is Version 26 with 121 x 121 modules. Together with a 4 module wide quiet zone, we end up with 129 x 129 modules totally that have to be displayed and read for pairing. In our example, we generated the QR code illustrated in Figure 5.1. The illustrated QR code contains IPv6 device community address of maximum length, an
example FQDN of maximum length and 714 characters of public key. Two space characters separate the data components of the QR code. The data is encoded by using byte encoding set and ECC level M.

The outcome of scanning QR codes depends on multiple factors, e.g., scanning distance, module size (in sq. millimeters or other units), camera resolution. In order to read a QR code, a camera has to distinguish single modules in the QR code. If the size of the modules fall below the resolution limit of the camera, then the QR code cannot be read by the device.

If we consider a patient and a doctor who pair their devices in the doctor’s office, we can make an assumption, that the distance does not play a critical role in this scenario. A form of natural recovery exists, because the devices can be moved closer to each other when scanning of the QR code happens. If the patient’s device cannot read the QR code on the doctor’s computer screen from one meter distance, a patient can move closer to the screen and try to read the code at a closer distance.

In order to understand what are the screen and camera requirements to show and to read the QR code, we conduct the following experiment. We read the QR code in Figure 5.1 with devices that have 12MP (megapixels), 8MP and 5MP cameras. These camera resolutions aim to represent a variety of camera modules in smartphones. According to the GSMArena article [18], among the top 20 most popular phones in 2018, there is not a single one with camera
resolution lower than 12MP. The 8MP and 5MP cameras in our experiment aim to address the older phones.

The QR code is shown for reading on a computer screen (1920x1080 pixels, 21 inches diagonal) and on a smartphone screen (720x1280, 4.8 inches diagonal). We choose these screen sizes to represent a doctor’s computer and a patient’s smartphone in our pairing scenario. FullHD (1920x1080 pixels) resolution of the desktop screen can be considered as a standard desktop monitor resolution in 2018. The phone screen resolution (720x1280) is chosen, because it is the lowest most common resolution in 2018, according to the report by DeviceAtlas [19].

Since we make an assumption that scanning distance is not relevant in our application scenario, our goal is to find out if the given QR code can be read by the given devices. For each test, we make 10 attempts to read the QR code to understand the successful rate. The QR code is shown as an image 500x500 pixels in PNG file-format.

The results of the experiment in the Table 5.1 show whether the device with the given camera resolution read the QR code displayed on the computer or the phone screen. The numbers in parentheses show the amount of successful readings. We do not take into consideration the time the camera needs to focus on a picture, but in all the cases this time was less then 5 seconds.

<table>
<thead>
<tr>
<th>QR code shown on</th>
<th>12 MP camera</th>
<th>8MP camera</th>
<th>5MP camera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer screen</td>
<td>YES (10/10)</td>
<td>YES (10/10)</td>
<td>YES (10/10)</td>
</tr>
<tr>
<td>(1920x1080 px, 21 inches)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone screen</td>
<td>YES (10/10)</td>
<td>YES (10/10)</td>
<td>YES (7/10)</td>
</tr>
<tr>
<td>(720x1280 px, 4.8 inches)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Results of QR code evaluation study

We observe that a phone with 5MP camera fails to read the QR code from the phone screen 3 out of 10 times, but performs 7 successful readings. We argue that it still can be considered as a successful reading rate. However, by lowering camera resolution, the chances that the QR
code is read successfully drop too. We propose to set the hardware requirement for a camera to be at least 5 megapixels in order to provide a high successful reading rate of QR codes for pairing in our application scenario.

The minimum screen requirements can be set to the phone size that we used in our experiment, namely 720x1280 px. The majority of the smartphones in the world in 2018 have a higher resolution and a larger screen diagonal than the one used in our experiment, therefore, they comply with this minimum requirement.

We propose to set the hardware requirement for a camera to be at least 5 megapixels in order to provide a high successful reading rate of QR codes for pairing in our application scenario.

Note, that we can set the screen requirement to be even lower than 720x1280, but we must always take into consideration the minimum camera resolution requirement. For example, we can read the QR code in Figure 5.1 from the screen of a smartwatch (312x390, 2,17 inches) with the help of 12MP camera, but 5MP fails to do it. One should always consider this tradeoff between camera and screen resolution, when setting the minimum hardware requirements for the devices to be paired with QR code pairing scheme.

This evaluation was based on a worst case scenario, in which pairing information with maximum possible length is encoded in the QR code. In a real-world pairing scenario, FQDN (or IP address) of the device is shorter than 253 characters, therefore, the QR codes will be of a lower version and, thus, easier for the devices to recognize and read. In addition to that, a lower ECC level L can be used, so the resulting size of the QR code is lower than the one produced by applying ECC level M.

5.1.2 DevCom study

Prior to implementation of DevComApp, we conduct a study to understand DevCom code and practical use of it. The conducted DevCom code study opens for us the following challenges:

1. SHA-2 hash function should replace SHA-1 hash function used in DevCom.
2. A function that resolves FQDN to IP address needs to be added in DevCom.
3. DevCom is not available on mobile devices.

We describe the observations we make in this section.

When started for the first time, DevCom generates a key pair on the device. Each device community is represented by a separate directory. The name of the directory is the
community name. For example, **family/** is the directory for community with the name “family”. In this directory, DevCom stores the public keys of other devices in the community that this device trusts. DevCom computes the device’s fingerprint, creates a virtual interface and assigns device community addresses to this interface. When a new connection between devices is established, DevCom writes the physical address of the other device into the file “devices.cache”. This file is used to map the globally unique community address and the physical address of the other devices.

As stated by Hansen *et al.* [6], DevCom requires OpenSSL and Avahi libraries. These libraries can be installed on a Linux machine via **apt-get install** (or other package manager), and then DevCom can be compiled and linked on Linux. OpenSSL provides all the cryptographic functionality to DevCom. We discovered that DevCom uses OpenSSL library version 1.0.2, which is an old version and is no longer available on latest Linux distributions. For the time we work on the thesis, the current OpenSSL Long Term Support version is 1.1.1 and it is not backwards compatible with OpenSSL 1.0.2. It means that in order to support OpenSSL 1.1.1 APIs, DevCom code has to be changed in a large extent. We decide to not rework DevCom to support OpenSSL 1.1.1, because it requires time for making significant changes in the code and time for testing and debugging that we do not have during master thesis. Code analysis of OpenSSL cryptographic functions used by DevCom revealed that DevCom uses cryptographic algorithms, which comply with the modern requirements by Datatilsynet [12]. The only exception is the SHA-1 hash function that is not considered secure anymore for signing messages [20]. We address this issue by implementing use of SHA-2 hash function in DevCom and describe the process in Section 5.4.1.

Avahi is a system which facilitates service discovery on a local network via the mDNS/DNS-SD protocol suite [21]. DevCom uses this library to discover other devices in the local network that are running DevCom. These devices are called neighbors. If we perform trust operation with neighbors, they become peers. Peers are the devices that have exchanged their public keys, and can join the community to establish control and data channels for communication. The Avahi library is used only for discovering devices in the local network, which means that it has no practical use, if two devices in different networks need to be paired with each other. Avahi service discovery feature allows process of device pairing within the network. DevCom uses Avahi functions to exchange the keys and to discover physical address when the user executes the trust and join command. It might lead to misunderstanding that DevCom operates only within local network.
However, DevCom is not limited to local area network use only. The devices can be paired and communication channels can be established when the devices are present in different networks. In this case, the user needs to do public keys exchange manually prior to join operation. In addition to that, the user has to know the physical address of one of the devices to do join operation. Furthermore, this address has to be routable so the packets sent from one device to establish control and data channels will reach the other device.

In order to verify that it is possible to pair two devices in different networks, we conduct the following experiment. We create an Ubuntu server at DigitalOcean\(^3\) with a public IP address and install DevCom on it. The other machine we use is the Ubuntu machine in our lab. We start DevCom on both machines, and create a community directory on each of them. When DevCom starts for the first time, it generates public and private key. We transfer the public key from the lab machine into the community directory on the remote Ubuntu server, and the public key of the Ubuntu server to the lab machine. Then, we execute a join command on the lab machine, providing the IP address of the Ubuntu server. We observe that communication channels are established, therefore, proving our assumption that pairing in DevCom can happen between the devices in different networks.

Note, that in DevComApp we aim to implement pairing with QR codes. Therefore, the public key will be transferred to the pairing devices by reading the QR codes. It is a different SDP scheme, and this SDP scheme via visual PHY channel is completely independent on network discovery feature. Furthermore, QR code pairing aims to address the limitations of the current SDP scheme in DevCom that allows to pair devices only within the network.

Our original idea was to use Avahi for network discovery, so the users can choose the SDP scheme that is optimal for them. It could be either QR codes exchange, or pairing in the networks the way it is done in the current DevCom, or other SDP schemes that might be implemented in DevComApp in the future. Fomichev et al. [13] call this approach *Adaptable Secure Device Pairing*. In this case, the best security-usability trade-off can be achieved for a given situation, where the user chooses the SDP scheme by taking into account different factors, e.g., application classes, the environment and social context, potential attacks etc. However, cross-compiling Avahi library on Android is a challenge, and we discuss it closer in Section 5.4.3.

---

\(^3\) Cloud service provider [available on https://www.digitalocean.com/]
One of the requirements of our design is to allow the use of FQDN for pairing. In the current DevCom, the join operation (establishing of control and data channels between two devices) is available only by providing physical address of one device to another. We address this limitation and implement the function to translate FQDN to physical addresses in Section 5.4.2.

Our goal was to implement DevComApp that would provide a more user-friendly SDP scheme for mobile device users of DevCom. Initially, we built our design based on the assumption that DevCom was available on mobile devices that run Android. According to Hansen et al. [6], the only condition to run DevCom on Android is that the TUN/TAP driver [16] is installed in kernel.

Hansen et al. [6] do not describe how to build DevCom for Android, and introduce only one requirement that is prior to running DevCom on the Android device. This requirement is an installed TUN/TAP driver in the kernel. By investigating the Makefile, we discovered that Hansen et al. [6] cross-compiled DevCom for Android 4.3 using Android NDK\(^4\) version r8. The make instructions in the Makefile are not relevant for today’s use, since it uses toolchains that are not present in the latest versions of NDK [22]. Furthermore, Android gcc cross-compiler used by Hansen et al. is also deprecated in the NDK. In order to cross-compile DevCom for Android in the latest NDK versions, e.g., r18/r19, we would need to use clang cross-compiler. NDK provides us with multiple clang cross-compilers that should be used to compile for different CPU architectures and Android versions. In addition to that, we need to link DevCom with OpenSSL and Avahi libraries that also need to be cross-compiled for the same architecture and Android version, that DevCom will be used on.

We make an attempt to cross-compile DevCom for Android, however, we argue that cross-compiling of DevCom should not be considered as a good approach, and elaborate on this in Section 5.2.2. We managed to cross-compile DevCom using the ARM clang compiler from NDK that targets Android version 28, and statically linking OpenSSL library (reasons why Avahi library is not included are discussed in 5.4.3), but DevCom crashes on the Android device with TUN/TAP driver installed in the kernel. Therefore, we state that DevCom is not available on modern Android devices, and in order to implement DevComApp, we need to address this issue. As the result, we added the requirement that DevCom should be available on mobile operating system to the design of DevComApp in Section 4.2.2.

\(^4\) Android Native Development Kit (NDK) is a set of tools that allow developers to implement parts of Android applications in native C/C++ code.
5.2 Technical decisions

This section presents the technical choices we make to implement DevComApp according to the design requirements and decisions made in Chapter 4. In Section 5.2.1, we discuss why we choose Android as a target mobile operating system for DevComApp. Section 5.2.2 presents two options to make DevCom available on mobile devices, namely a stand-alone mobile app and a cross-compiled binary executable file. We discuss both options and argue why the first one is more preferable. Section 5.2.3 describes the hardware requirements.

5.2.1 Mobile operating system: Android

The two major mobile operating systems in 2019, are Android and iOS. According to IDC, worldwide market share in 2018 for these two mobile operating systems combined is nearly 100% [23]. We choose to bring DevCom on Android in this thesis, since Android SDK (and particularly part of it called Android NDK) allows developers to create native Android mobile applications using C/C++ programming languages. Since DevCom is written in pure C, we expect to be able to reuse most of the original codebase in our Android solution. Furthermore, the presence of a modified Linux-kernel in Android allows us to interact with the kernel via the same system calls as we use in DevCom on Linux. In addition to that, according to Hansen et al [6], they managed to run DevCom on Android 4.3 for their tests. In order to do that, they needed root access rights on the Android device and additional kernel module installed for tunneling functionality [6]. This approach is not well suited in our application scenario, and we elaborate on this in the next subsection.

We decide to implement DevComApp for the latest Android version available, which is by the time we are working in thesis Android 9 Pie. This is highly motivated by the fact that the APIs available in the latest version will have a longer support from Google and device manufacturers in the future. In addition to that, we want to investigate the development process for the latest Android API level, in order to understand what kind of features and restrictions it has already. We consider development for the latest API to be more practical and more important to research, because the older API levels might have features that are removed or no longer supported in the current Android version. Therefore, it makes less practical value to investigate them. Last, but not least, the latest version of Android implements the newest privacy and security features of the platform. Since privacy and security is also a requirement in our application scenario, we do not want to focus on designing a system for the older versions, which might have security breaches that are fixed in the newest version.
For now, we decide to set apart idea of bringing DevCom to iOS. Implementing DevCom for iOS requires a deeper understanding of Apple’s mobile operating system, its APIs and its software development kit, that we currently do not possess. It would take us a lot of time to learn iOS SDK, which we do not have during the master thesis work.

5.2.2 DevCom as part of Android APK as a preferred option to cross-compiling

There are two different ways to run DevCom on Android, cross-compiling a binary executable file and inbuilding DevCom into the regular Android application with NDK. The first one is to cross-compile the existing DevCom codebase for Android on a regular Linux desktop machine to create an executable that can be run on Android devices. According to Hansen et al. [6], there is one requirement to do it, which is an installed TUN/TAP tunneling driver in the kernel. In case it is not pre-installed on the device, a root access to the device is needed to install it. The second option is to create a stand-alone Android application using Android NDK. Android NDK allows developers to implement parts of Android applications in C/C++ programming languages. Therefore, the current DevCom codebase (or at least part of it) can be ported into an Android application that is written in Java/Kotlin.

We consider the second option, i.e., a stand-alone Android application, to be more preferable than a cross-compiled application. This decision is based on the analysis of the following factors:

1. Support from OS developer
2. Distribution and installation
3. Execution
4. Building
5. Linking
6. Debugging

We discuss these factors, and show why each of them favors the stand-alone application option.
#1 Support from Google

Typically, Android applications are written in Java, or Kotlin that has been officially supported for Android development since 2017 [24]. However, an Android application can be partly written in C/C++. Developers can implement parts of their application in C/C++ in Android SDK with a module called Android NDK. With it, developers can add C and C++ code to their Android project [25]. In order to add C code into the project, developers can place it into a cpp directory in their project module, which will be compiled into a native library that Gradle build system will package into the app. Java or Kotlin code can then call the functions from the native library through the Java Native Interface (JNI) [25], [26]. Android NDK is also supported in Android Studio that is the official Integrated Development Environment (IDE) for Android app development. Since DevCom is written in pure C, we initially consider migrating full DevCom codebase into our Android project to be a viable approach, because Google officially allows developers to do this and supports this by providing tools and manuals for the development.

Cross-compiling is supported by Google too, however, Google states that the NDK does not contain official support for cross-compiling. The purpose of cross-compiling is to build existing code that uses other build systems. For example, this is the case for OpenSSL library that is not Android-specific and has to be cross-compiled for Android [27].

#2 Distribution and installation

The second reason to choose a stand-alone app over a cross-compiled one is the distribution and installation model. Since using C/C++ code inside Android apps is allowed and supported by Google, there is no restriction on distribution of these apps. After the project is build, we get an APK file. APK stands for Android Package Kit, it is a package file format used by the Android operating system for distribution and installation of mobile apps. Just like Windows (PC) systems use .exe and macOS uses .dmg files for installing software, the APK does the same for Android. After we make an APK file, we are ready to distribute it across the devices. There are multiple options to do this:

1. We might publish it in the Google Play store, in fact APK is the only way to distribute apps via Google Play.

2. We can transfer the APK file with help of adb (Android Debug Bridge), a versatile command-line tool that lets us communicate with a device. The adb command
facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. In order to do this we need to use `adb install <apk>` command.

3. We can also simply share the APK file to be downloaded via a link.

All of the three options mentioned above will result into an app being installed on the Android device and being visible in the app list.

On the other hand, the cross-compiling option will result into a binary executable file (after compiling and linking). Compared to the APK files, distribution of this type of files is less user-friendly, especially in a hospital setting that is relevant for our application scenario. Binary executable files are not allowed to be distributed via Google Play. Furthermore, the `adb install` command will not install binary executable file as an app. In order to transfer the binary executable file to the device we can use the command `adb push <file>`, which simply copies the file to the device. A user will not be able to see this file in the list of installed apps, because it is not an app. It is just a binary executable file which exists somewhere in the file system. We can also download the binary executable file via link. In this case we can see the file in the Downloads folder.

Furthermore, copying a binary file to the device and running it can be considered as a breach of the security requirement of the application scenario. We should not rely on a solution that requires users to download a binary executable file and run it. In case binary executable file is compromised and acts maliciously, it is hard for users to discover this during installation process and after the file is executed. Lastly, Android requires that all APKs are signed with developers digital signature [28]. Only a signed app is allowed for distribution in the Google Play (other app stores like Amazon App Store require signing too). Having a digital signature of the developer helps us to verify the source of the APK and provides a better security.

#3 Execution

After the installation, Android recognizes the APK file as an application file and installs it as an app. The app will appear in the list of installed apps on the user’s device, and the user will be able to open and run the installed app. This process can be considered as a standard approach that mobile device users are familiar with.

The process of executing a binary executable file is different, and cannot be considered as user-friendly and familiar to regular users. We install the binary executable file either by
using **adb push** or directly downloading the binary executable file via a link, however, we need access to the command shell to run it. One of the options can be to use **adb** command shell to manually run the binary executable file by typing commands. Another option can be to install an app from Google Play store on the mobile device that emulates an Unix shell and run the cross-compiled binary executable file via it. We consider both of these options to lack user-friendliness that we aim to achieve with our design. One of our goals is to implement a solution that avoids interaction with command shell by end-users to do device pairing. We argued that command shell interactions are not user-friendly in our application scenario, where patients are not expected to have high technical skills.

#4 Building

The reason why integrating DevCom into a mobile application is preferable to cross-compiling is the build system. Android Studio, which is the main IDE for Android development, has a flexible Gradle-based build system. One of the most important things the build system provides is multiple APK support. In other words, we can create different versions of our app for a broad variety of Android devices. Android handsets have different CPUs, which in turn support different instruction sets. Each combination of CPU and instruction set has its own Application Binary Interface (ABI). The ABI defines how application’s machine code interacts with the system, therefore, developers must specify an ABI for each CPU architecture they want the app to work with [29]. Different Android versions reflect in different API levels that applications have to support. For example, Android 9 Pie defines an API level 28. In practice, it means that the app with a minimum target API level 28 will not be able to run on the versions of Android lower than Android 9 Pie. Since the Gradle build system in Android studio allows us to configure the target ABIs and API levels, and it has to be done only once, our building process becomes more automated and easy.

In case of cross-compiling, building the program requires knowledge of the target device we cross-compile binary executable file for. In practice, it means that we have to cross-compile a number of binary executable files target for different ABIs and API levels. The larger number of target devices we want to cover, the more binary executable files we have to cross-compile. It also makes the distribution of the program much harder because in order to distribute a correct binary executable file to a particular device we need to know certain characteristics of the device, i.e., Android version it runs and its CPU architecture.
5 Linking

Another aspect, which is a part of building process, is linking. However, we separate linking from building because it is a major task in itself. Android OS contains a modified Linux kernel. When we develop application in C/C++ for Android we most likely use standard libraries that are used in Linux, e.g., standard C library libc, dynamic link library libdl, etc. These libraries are part of NDK bundle, and are compiled for different Android ABIs. When we develop an app in Android studio, the IDE automatically handles linking toward these required libraries based on the build configuration. For example, if one of the C-files includes <stdio.h>, the build system will include required header-file and link toward the library that corresponds to the ABI (or ABIs) specified in the build-configuration file.

When it comes to linking, in case of cross-compiled option, we have to manually specify directories to include during compiling and libraries to link towards to, based on the target ABI of the binary executable file.

6 Debugging

Finally, important reason to not use cross-compiling option is debugging. In Android Studio we have an in-built debugger that allows us to debug our code relatively easy. Furthermore, if we find a bug in our application during testing, we can fix it and then build the app, according to the build configuration that we set. If we have a support for multiple APKs, fixing a bug and building new APKs for different targets is an automated process that will be handled by Android Studio itself. In case of cross-compiled program, each bugfix will require to recompile all the required binary executable files for different targets.

Summary

Considering all of the issues mentioned above, we argue that a stand-alone Android application with DevCom codebase in C as one of its modules should be considered as a preferable design solution compared to a cross-compiled for Android binary executable file.

5.2.3 Technical decisions regarding hardware

We decide to not set any specific hardware requirements, apart from the screen and camera. These requirements are put by:

1. Size of QR codes that are used for pairing
2. Android 9 Pie hardware requirements

In section 5.1.1, we conducted an experiment, after which we propose to set minimum camera resolution requirements to 5 megapixels and minimum screen resolution requirement to 720x1280 pixels.

Android 9 Pie has the following requirements [30]:

- **Screen** (requirements 2.2.1 and 7.1.1.1):
  - **MUST** have a screen at least 2.5 inches in physical diagonal size.
  - **MUST** have at least 320 dp x 426 dp.

- **Camera** (requirement 7.5.1)
  - **SHOULD** include a rear-facing camera.
  - **MUST** have a resolution of at least 2 megapixels (if device include rear-facing camera implementation).

Note, that Google uses **dp, density-independent pixel**, which is equivalent to one physical pixel on a 160 dpi screen. We conducted our tests on a 4.8 inch screen with 306 dpi. The resolution of this screen represented in dp is 376 x 669. The screen requirements we set after the experiment in Section 5.1.1 are stricter, than those that are set by Android 9 Pie. Therefore, screen requirements that we set after the experiment, namely 720x1280 px (376 x 669 dp) should be used as minimum screen requirements for our system, since they provide a more reliable pairing with QR codes.

Camera requirements set by Android 9 Pie are less strict than these we set after the experiment in Section 5.1.1. We observe errors in scanning QR code from the phone screen already with 5 megapixel camera and we do not consider lower resolutions to provide a reliable pairing in our application scenario. Therefore, we set minimum camera requirements for DevComApp to be 5 megapixels as concluded in the experiment in Section 5.1.1.

5.3 Languages and tools

In this section, we describe languages and tools we use during the implementation process. We decide to implement DevComApp in Java and use Java Native Interface (JNI) interface to control DevCom that is written in C.
In order to call C functions on DevCom layer from the Application layer of DevComApp, we need to use JNI. JNI is written in Java. Kotlin still can be used with JNI due to full interoperability, but the amount of documentation about how to do it and examples is significantly less than for Java. Thus, Java is our preferable choice.

Each Android mobile application installed on a device gets a special directory in the device’s file system, where it can store files. This directory can be accessed by ‘/data/data/<NAME OF THE APK>’. We will use this directory as a Datastore layer of DevComApp.

For the development of DevComApp we use Android Studio, the official IDE provided by Google. It supports all of the necessary tools and systems that we need for the development of Android project. These tools and system that we use during development in Android Studio are:

1. Android SDK provided with Android Studio and Android NDK, the latest available version r19.
2. Gradle and CMake for building the app.
3. Debugger in Android Studio.

Android Studio also allows us to run the app on physical devices connected via cable and on different emulators that can be downloaded in Android Studio.

During the implementation process we run and test our application on a Google Pixel XL phone, with stock Android 9 Pie (API Level 28) and ARM CPU architecture. Additional tests are done on Nexus 5X emulator with Android 9 Pie and x86 processor. The Pixel phone is rooted [31] for testing and deeper analysis of the failed attempt in Section 5.5.

5.4 Implementation process

Prior to the implementation process, we revisit the limitations of DevCom from Section 3.1 and Section 5.1.2, that we aim to address:

1. Security of DevCom is compromised due to the use of SHA-1 hash function. In 2019, SHA-1 is not considered safe due to a possible collision attack [20].
2. DevCom does not support pairing by a hostname. The only way to pair one device with another is to provide a physical address of the device we want to establish secure communication channel with.

3. Early testing showed that DevCom is not available on any modern mobile operating system.

4. DevCom does not have a user-friendly secure device pairing scheme for mobile devices.

The following subsections describe what we did to address the issues named above, except the last one. We could not proceed to the implementation of new pairing mechanisms in DevComApp, since we have not succeeded to port DevCom into the Android application. We provide the analysis of the failed attempt in Section 5.5.

5.4.1 Implementing use of SHA-2 hash function

We improve the security of DevCom by changing SHA-1 algorithms used by DevCom to SHA-2 algorithms. Note, SHA-2 is a family of hash functions, namely SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, which generate 224, 256, 384, 512, 224 and 256 bits respectively of output (digest) from a given input. We choose to use SHA-256 in DevCom, because it operates on the same message block size, 512 bit, as SHA-1 that is already in use in DevCom, and divides it to 32-bit words just like SHA-1.

DevCom uses SHA-1 for two purposes:

1. to hash the supplied key material, when DevCom creates the key and initialization vector for AES encryption, and

2. to sign encrypted control packets with the RSA algorithm.

In the first case, the message digest function used to sign the supplied key material in the function int aes_init() in the file security.c is EVP_sha1(). This function returns an EVP_MD structure that contains the implementation of the symmetric cipher. We change the use of EVP_sha1() function to EVP_sha256() function. As the result, we get the 256 bit AES symmetric key, derived with SHA-256 message digest.

In the second case, SHA-1 is used when DevCom signs the encrypted control packet. To do that, DevCom signs not the encrypted packet, but a hash of the packet (both header and
payload) computed with SHA-1 hash function. This is implemented in the function `int control_packet_sign()` in the file `control_traffic.c`. In order to change SHA-1 to SHA-256 in this case, we need to call the `SHA256()` function on the packet, and write the computed hash into the digest array. Since the SHA-1 digest is 20 bytes long, and the SHA-2 digest is 32 bytes long, we need to allocate different amounts of memory for the digest. We change the constant `SHA_DIGEST_LENGTH` to `SHA256_DIGEST_LENGTH` to allocate the correct amount of memory in the stack for the computed digest. The same steps has to be done in the function `control_packet_verify()` in the file `control_traffic.c`, which verifies the signature of the received packet.

Implemented changes are included in Appendix A. We have compiled and deployed the code with the implemented changes on three different machines and verify that DevCom works as intended.

### 5.4.2 Implementing hostname functionality for join operation

We extend the existing DevCom implementation of join operation to support the use of FQDN in addition to the IP addresses. In order to do it, we implement a new function `int hostname_to_ip_addr()` that will translate the FQDN of the device to the IP address that has to be called prior to `connect_tcp_control_server()` function, which creates a TCP connection to the device by the given IP address. In this case, DevCom still uses only physical address of the device to establish communication channels with, while `hostname_to_ip_addr()` function is responsible for eventual translation of FQDNs to IP addresses. If the user provides IP address of the device to connect to, instead of FQDN, the `hostname_to_ip_addr()` function will return this address.

We implement the `hostname_to_ip_addr()` function with the help of Linux function `getaddrinfo()` that resolves a hostname into the IP address by making a DNS lookup. In case of FQDN provided to the `hostname_to_ip_addr()` function, it will resolve it to IP address and write this IP address in the memory block accessible by the second pointer argument. If the user provides the IP address for a lookup, the function will write this IP address in the memory block. Then DevCom uses the IP address written in the memory block to do the join operation with the device available by this address and to update the information about the device in the cache file.
The code of \texttt{hostname\_to\_ip\_addr}() is included in Appendix B. We have compiled and deployed the code with the implemented changes on three different machines and verify that DevCom works as intended.

5.4.3 Integrating DevCom as C module into Android project

Our original goal was to port DevCom into DevComApp project structure. The motivation behind it was to make the installation process of DevCom on mobile devices more user-friendly for end-users, i.e., patients, their relatives and doctors, as discussed in the Section 5.2.2. After installing DevComApp from a Google Play store (or another application store on Android), the user would get DevCom installed on her device. Then, the user could start and stop DevCom from within the DevComApp as well as do pairing in a user-friendly way with the help of QR codes.

Migrating DevCom to the APK is a complex task with the following challenges:

1. The OpenSSL library, used for cryptography, is not present in Android.

2. Avahi library, used for announcing devices in the network, is not present in Android.

3. Other C/C++ standard libraries in Linux might not be available in Android, e.g., POSIX threads (referred as Pthreads), tunneling, sockets, etc.

During the development process we did not manage to solve one critical issue, namely tunneling functionality of DevCom. We investigate this issue closer and describe it in Section 5.5.

The first step in the implementation process is to configure build system and project structure. We create a new Android project in Android Studio with C/C++ support. By doing this Android Studio automatically adds \texttt{cpp} directory where we should put all our C code. Android Studio also creates a \texttt{CMakeLists.txt} file, where we will specify the native libraries we add to our project. This file is then used by CMake and Gradle build systems to include the libraries into the APK.

Originally, we added the whole DevCom codebase into our project. Soon we discovered that this was a bad approach, since we got multiple errors and warnings during compilation due to missing header files. We got these errors because Android does not have OpenSSL library by default, therefore, NDK does not include OpenSSL header files and there are no OpenSSL libraries to link our code against. In addition to OpenSSL, there are header files that exist in
Lin\text{}ux, but not in our target version of Android, i.e., API level 28. An example of such file is \texttt{sys/_types.h}, that was available in NDK for API level 20 [32], but is not present in the latest Android versions.

Therefore, we decided to migrate DevCom into our Android project file by file. With this approach we could fix the arising issues one by one, and describe the whole process in a more structured way.

**Challenge #1: OpenSSL library is not present in Android.**

One of the most important steps to start with is to include OpenSSL library into our project. In order to do this, we have to cross-compile the OpenSSL library for Android devices. This process is described on the OpenSSL wiki page [33]. This guide provides us with the script, in which we can specify the NDK version to use for cross-compiling, CPU architecture, and ABI and API level of the device we cross-compile OpenSSL library for. Since the script uses gcc cross-compiler that is not present in Android NDK r19 that we use in our development, we need to download an older version of the NDK with the gcc cross-compiler. We use NDK r10e. In the script we specify that OpenSSL should be compiled with NDK r10, for ARM architecture, for latest ABI supported by NDK r10e, which is 4.9 and latest API level that is allowed in the script, namely API level 14. We choose these parameters due to our main testing device, namely Google Pixel XL. For other devices, e.g., Android devices with Intel’s x86 CPU, we should run the script with corresponding configuration. The result of the script execution is an \texttt{openssl} directory. In it we find \texttt{include} directory, which contains header files to be included during compiling, and \texttt{lib} directory with .\texttt{so} files that can be linked dynamically and .\texttt{a} files for static linking.

We add the compiled OpenSSL libraries (libcrypto and libssl) as dynamic imported libraries in \texttt{CMakeLists.txt}, so our code will be linked to \texttt{libcrypto.so} and \texttt{libssl.so} in a runtime. An attempt to test the linking leads to the following error:

```
java.lang.UnsatisfiedLinkError: dlopen failed: library "/system/lib/libcrypto.so" needed or dlopened by "/system/lib/libnativeloader.so" is not accessible for the namespace "classloader-namespace".
```

This issue is described in NDK documentation [34]. Starting in Android 7.0, the system prevents apps from dynamically linking against non-NDK libraries, which may cause the app to crash. In our case we cannot link against OpenSSL libraries, because they are not part of Android NDK. We tested this issue on earlier versions of Android, namely Android 5.0, and can confirm that there is no linking error on it. In order to solve this issue, the NDK
documentation [34] presents possible solutions. Accordingly, we decide to link our app against `libssl.a` and `libcrypto.a` static libraries. In this case, they will be automatically packed in the APK during building stage.

**Challenge #2: Avahi library is not present in Android.**

Avahi is a system which facilitates service discovery on a local network via the mDNS/DNS-SD protocol suite [21]. As discussed in Section 5.1.2, in order to keep the existing pairing scheme, Avahi library needs to be compiled and linked against DevComApp.

In the beginning, we assumed this challenge to be similar to the first challenge, namely cross-compiling OpenSSL for Android. After doing a research and multiple number of attempts, we discovered that our assumption was wrong, and it is a more complex process than compiling and linking in Android against OpenSSL library. We decide to not port the Avahi dependent module of DevCom, because unlike OpenSSL library, Avahi library does not have official support for Android, according to the developers of Avahi [35]. Building OpenSSL for Android is well-documented on the OpenSSL wiki page [33], while documentation for cross-compiling Avahi for Android is represented only by blogposts of different programmers. We found one guide in a blog that was posted in 2015 [36], and followed it, but did not succeed in configuring and building Avahi for Android in 2019.

We did not find a single working tutorial on building Avahi for Android, and therefore decided to not keep the mDNS discovery feature of DevCom. Alternatively, an SDP scheme that is based on network service discovery can be implemented according to Android developers official documentation [37].

**Challenge #3: Other C/C++ standard libraries in Linux might not be available in Android, e.g., POSIX threads, tunneling, sockets, etc**

Our last challenge is to move DevCom module by module, file by file to the `cpp` directory of our Android project, DevComApp, and to verify that it functions the way it is intended. Since we managed to compile and link against OpenSSL library, and decided to set apart bringing Avahi library to Android, the only libraries that DevCom requires to function are standard C libraries, e.g., `libc`, `libdl`. Android NDK provides these libraries compiled for a given API level and ABI, therefore, DevCom can use these libraries as it would in regular Linux. The build system includes the correct header files from the NDK and links correspondingly against the requested libraries. However, Android’s version of C library is not the same as
used in desktop Linux (typically GNU C Library). Android uses its own standard C library that is called Bionic. Bionic is targeted for use on the devices with limited storage and lower CPU speeds, therefore, it is smaller than glibc. Early research shows, that there are a number of limitations in Bionic, which can be critical for DevCom, e.g., POSIX threads [25]. As shown by Devos in the master thesis [38], Bionic does not allow Pthread cancellation\(^5\) and use of `pthread_atfork()` function. DevCom is not dependent on any of the named issues, therefore, DevCom can use Phreads in Android the same way it does in Linux.

Another important functionality provided by glibc in Linux that is critical in DevCom is sockets. We did not find any specific restrictions to use sockets in Android, apart from adding one permission to the application manifest, `AndroidManifest.xml` [39]. The permission that needs to be added to the manifest is:

```
<uses-permission android:name="android.permission INTERNET" />
```

Adding this permission allows us to open and configure sockets both in native C/C++ code and in Java/Kotlin code.

One of the serious issues regarding the existing DevCom code that was discovered by us during the development is unavailability of implemented logging procedures in Android. In verbose mode, DevCom sends log messages using standard output communication channels, namely `stdout`, standard output stream for logging its normal activity, and `stderr`, the stream to output standard error messages. In Android, the standard logging tool is called `logcat`. By default, it does not output messages from `stdout` and `stderr` streams, which are written to `/dev/null` and discarded. To see and analyze DevCom log messages is critical during the development. Therefore, we need to addressed the issue of missing DevCom log messages in Android. There are three possible solutions that we considered:

1. Refactor DevCom code to use `__android_log_print()` function implemented in Android NDK instead of standard C `fprintf()` function that is currently used in DevCom.

2. Redirect `stdout` and `stderr` output to `logcat`.

3. Redirect `stdout` and `stderr` output to a file.

\(^5\) Pthread cancellation is a mechanism used by one thread to terminate the execution of another thread in a specified manner.
We argue that the first option should be used, because it is the only way described in official NDK documentation [40]. The second option, according to one user at StackOverflow [41], is available only on rooted Android devices and redirects only output generated by Java/Kotlin code. The last option would require reading the output file every time when the program is over and when writing to the file is over too, which does not give us the runtime debugging possibilities.

We port DevCom in cpp/directory of DevComApp file by file starting with Key Manager module that consist of two files: security.h/.c. These files contain implementation of cryptographic functions in DevCom that call functions from OpenSSL library. Since OpenSSL library is cross-compiled and added to our project for static linking, we include all the necessary header files that have declaration of the required functions used by DevCom. In order to generate key files and to store them on a device, we need to add two extra permissions to AndroidManifest.xml that will allow DevComApp to read and write the keys into the storage. These permissions are:

android.permission.WRITE_EXTERNAL_STORAGE, and
android.permission.READ_EXTERNAL_STORAGE.

No additional changes to the code are required, apart from the change of logging functions. The Key Manager works as intended and generates a Key Pair that is written in the Datastore of DevComApp. We discovered that generate_key_pair() function call that is called the first time when key pair does not exist, takes up to 10 seconds to execute. It leads to the behavior, when Choreographer\(^6\) skips frames, because the application may be doing too much work on its main thread. This issue is described in the blog post from 2013 [42]. Generating a key pair is computationally intensive task on a smartphone, therefore, Choreographer skips frames to not block the application’s main thread. One of the possible solutions to this problem is to call generate_key_pair() from another thread, but then, there is no guarantee that the keys are generated prior to any other activity in DevCom. If we implement a thread blocking mechanisms to ensure that the keys are generated prior to any other activity, Choreographer will skip the frames. In order to address this issue, we decided to call generate_key_pair() function during the app launch. In this case, it takes more time for the app to start for the first time because it will generate the keys, but solves the problem of skipped frames and guarantees that the keys are always generated prior to any other activity within the app.

\(^6\) Android component, scheduler, which is responsible for time coordination and performance improvements of Android applications.
The next DevCom module we add to the Android project is the Virtual Interface Manager that is responsible for maintaining the virtual network interface. Its functionality is implemented in `tun.h/.c` files.

Prior to the implementation process, we need to make sure that TUN/TAP device driver is loaded in the kernel, which is the requirement to run DevCom on Android devices introduced by Hansen et al. [6]. In case it is not present in the kernel, we need to install it manually. In our tests we use Google Pixel XL with Android 9 Pie. The kernel version that is installed on this unit is 3.18.100. According to TUN.ko driver installer\(^7\), the TUN module is loaded in the kernel (Figure 5.2). To assure this, we install a VPN client that uses TUN/TAP driver, namely OpenVPN for Android [43], and verify that it works.

![Figure 5.2: Confirmation that the TUN module is loaded (Screenshot from TUN.ko driver installer app)](https://play.google.com/store/apps/details?id=com.aed.tun.installer)

---

\(^7\) TUN.ko Installer, app in Google Play Store by DroidVPN Inc. that allows to download and install missing TUN/TAP drivers (available on [https://play.google.com/store/apps/details?id=com.aed.tun.installer](https://play.google.com/store/apps/details?id=com.aed.tun.installer))
The way DevCom opens and controls the TUN device in Linux is described in Section 3.4. We add the `tun.h/.c` files to our Android project and call the function `create_virtual_interface()`, the successful execution of which results in the creation of new virtual interface. The interface is not up yet, but should be visible in the list of all network devices, e.g., with the command `$: ip link show`.

With the first attempt, we get an error that tun device special file that should be located in `/dev/net` (further in text we refer to it as `/dev/net/tun`) does not exist. By default, the `/dev/net/tun` is not present in Android OS, despite the Linux kernel. We create this device and configure it the same way it is described in Section 3.4.

Our next attempt leads to a new error. This time, the device special file `/dev/net/tun` exists, but Android does not allow us to open a file descriptor to it, and the following error message is logged in the debugger:

```
W/admin.devcomapp: type=1400 audit(0.0:119636): avc: denied { open } for path="/dev/net/tun" dev="tmpfs" ino=2671150 scontext=u:r:untrusted_app:s0:c140,c256,c512,c768 tcontext=u:object_r:device:s0 tclass=chr_file permissive=0
```

This error appears due to the mandatory access control (MAC) system for the Linux operating system, i.e., Security Enhanced Linux (SELinux), which is described in Android documentation [44] and by Shabtai _et al._ in [45]. Compared to Linux’s familiar discretionary access control (DAC) system, where a concept of ownership exists, MAC system used in Android consults a central authority for a decision on all access attempts. The ownership concept in DAC implies that the owner of a particular resource controls access permissions associated with it. Unlike the DAC system, SELinux recognizes various kernel objects and sensitive actions performed on them. To enforce security policy, SELinux tags each process and object (file, package) in the system. Each time when the sensitive actions are performed, SELinux uses these tags to determine whether or not the action is authorized. Tag assignment and authorization rules are stored in the SELinux policy file.

Since our application does not appear in the list of the allowed applications and processes that can access the TUN network device, SELinux does not allow it to perform any actions on this object. There are two options to overcome this issue:
1. Set SELinux from “enforcing” to “permissive” mode.

2. Add a rule to the SELinux policy to allow performance of certain function of kernel objects.

Both of these options require superuser access rights. The first option can be considered as a temporary solution during the development process, but cannot be considered as a viable solution for end-users, i.e., patients and doctors. Disabling SELinux introduces a security breach in the device. Since one of the requirements for DevComApp and DevCom is security, we cannot compromise it by turning off one of the security mechanisms by the operating system. The second option is to create a new policy. The process and rules of how to implement a new SELinux policy is described in Android documentation [44] and tutorial [46]. We add the rule to allow our application to call `open`, `read`, `write` and `ioctl` functions on the TUN character device.

In order to create new rule, we needed root access. Furthermore, according to official Android SELinux documentation [47], the customization of SELinux policy should not be made by third-party application developers, but by developers that do customizations to the Android operating system (create custom Android ROMs), or by device manufacturers. The open question is how to allow use of DevComApp with DevCom module that calls the functions, i.e., `open()`, `read()`, `write()` and `ioctl()` on character device, on a regular patient’s Android device with no root access. At this point, we can already state that the use of DevComApp in this way, or any other third-party application that requires change in SELinux policy, is not possible, unless the change is made by the manufacturers or creators of Android ROM, which is installed on the device.

We proceed further with the development process and call `create_virtual_interface()` function in DevCom from DevComApp. Since we added the rule to allow `open()` call from our app on the `/dev/net/tun` to create a file descriptor to it, SELinux does not stop the application from accessing the character device, but still logs this activity. However, the next call, namely `ioctl()` that gives DevCom a control over the TUN network device, fails. We did not manage to address this issue, and, therefore, cannot open a virtual interface and control the associated network device in the Android application. Since this functionality is critical in DevCom, we cannot proceed further with implementation process. We shift our focus from the implementation to the analysis of this issue and describe it in the next section.
5.5 Systematic analysis of the failed attempt

In this section, we describe the analysis of the failed attempt to open a virtual interface with TUN device in Android. From the beginning, we have assumed two types of possible causes, and possible solutions. We illustrate the systematic analysis process that we describe in this section with the help of decision tree in Figure 5.3. Solutions shown in the figure represent different options investigated during the analysis to address the corresponding cause of the failing `ioctl()` system call. We analyze all the potential causes and discuss the solutions to them. At the end, we discovered that the actual cause was Cause #4 “Missing process permissions”. We have made multiple attempts to address the issue with different solutions, and discuss the process and our findings in this section.

![Figure 5.3: Analysis structure](image)

The reason why the native Android application cannot open the virtual interface and control the associated with it network device is the failing `ioctl()` call within DevCom native C code. Without the `ioctl()` call, DevCom cannot register TUN network device with the kernel and
configure it correctly. It means that network device is not allocated. Therefore, kernel will not forward the data from/to the virtual interface to/from the character device (/dev/net/tun), and DevCom cannot read/write the data from/to it. As the result, DevCom cannot manipulate the traffic on the virtual interface and do what it is supposed to do, namely encrypt the data and forward it to the receiver of the data. Note, that it is possible to create and configure the virtual interface in a different way, e.g., by executing shell commands either manually, or implement the command sequence or script that is run within the code. For example, the command $: ip tuntap add tramp0 mode tun will create the virtual interface that we can further configure with the shell commands the same way as we do it in the DevCom code. However, it does not solve the issue of the failing ioctl() call, because DevCom still needs to get control over the network device associated with the virtual interface in order to receive/send data from/to the kernel.

According to ioctl() Linux man page [48], upon an error resulting from the function call, the return value is -1, and errno is set appropriately. We output the return value of ioctl() call in the logcat, and investigate the error. For this case -1 is the return value and errno is used to output the error message, which is “Operation not permitted”. In order to analyze why the error occurs, we define the possible reasons that might lead to it. Since the operation is not permitted, we expect that the restrictions are set by the operating system on our application to allocate the network device and access the character device. We consider two possible types of reasons to the error:

1. **Configuration related**, i.e., wrong configuration of /dev/net/tun and TUN/TAP driver issues in kernel.

2. **Permission related**, i.e., either missing app permissions in the AndroidManifest.xml file, or missing permissions of the application process that does ioctl() call.

Note, that we do not consider SELinux as the cause that restricts access to the /dev/net/tun, because we added the rule to allow this call. In our case, we can see that SELinux logs the ioctl() call, but sets no restrictions on it. Even if we set the SELinux mode to permissive globally, we still get the same error, i.e., “Operation not permitted”. Therefore, the issue is not caused by SELinux.

In the following subsections, we investigate the possible reasons to the ioctl() issue.

---

8 Error number in Linux, which can be used to identify the error that set it.
5.5.1 Configuration related causes

Cause #1: TUN device misconfiguration

During the implementation process, we had to create /dev/net/tun device special file, and set the permissions on it. Since Android does not have the device installed by default, we created it by ourselves. However, superuser rights are needed in order to do it. The question is how do other programs, like VPN clients, solve the issue of missing /dev/net/tun. We investigated the code of OpenVPN for Android [43], and found out that instead of /dev/net/tun it uses another device, namely /dev/tun. We did not find the documentation that explains why Android uses another TUN device, compared to a regular Linux distribution. Furthermore, if we try to create the virtual interface in the shell, via `ip tuntap` command, we get an error “open: No such file or directory”, because `ip tuntap` looks for the character TUN device in /dev/net, and not in /dev. We did not find the explanation why Android uses /dev/tun and why /dev/net/tun is not present in Android by default, but we consider that this behavior is important to document in this thesis. We change the DevCom code to open /dev/tun instead of /dev/net/tun, but it does not solve the issue of failed `ioctl()` call. Now, our configuration of the TUN device is the same as used by other apps, therefore, it cannot be the cause of the failing `ioctl()` call.

Cause #2: Kernel driver unavailability

We also verify that the kernel driver module is loaded and recognizes the /dev/tun device. In order to do it, we cross-compile a program that calls `create_virtual_interface()` function from the tun.c file, install it on the Google Pixel XL device as binary executable file and run it. In order to do it, we need a superuser access in the shell. After running the binary executable file, we see a new link in the `ip link show` list and the `ioctl()` call does not fail. After this observation, we can state that the error in `ioctl()` call from the native C code in the APK is caused by the restrictions set on the APK from Android, and is not caused by the missing kernel driver support.

5.5.2 Permission related causes

Cause #3: Missing manifest permissions

We start by checking the permissions in the manifest of our APK and compare them to the permissions of OpenVPN for Android, because missing permissions in the manifest might
lead to the issue that our application is not granted the rights to access the TUN device. We do not observe any specific permissions that are granted to OpenVPN compared to DevComApp. The VPN functionality of OpenVPN is granted via "android.permission.BIND_VPN_SERVICE", but OpenVPN is implemented in Java. This permission is required for OpenVPN to use Java API provided in the SDK and, therefore, not applicable in our case, where the TUN device is accessed directly in the C code with ioctl() system call. By inspecting the full list of permissions that can be granted to third-party apps [49], we make a conclusion that the error we receive is not due to the missing special permissions in the AndroidManifest.xml file.

**Cause #4: Missing process permissions**

As we revealed, the reason is not the missing APK permissions in the AndroidManifest.xml, and we managed to successfully call ioctl() function in the binary executable file in the previous Section 5.5.1. However, our application does not have the same level privileges as superuser shell, which executes the same code without any restrictions. Therefore, we expect that in order to make ioctl() call that does not return an error from within the APK, we need to grant the application the same level privileges as we grant to the shell process, namely superuser access. Note, that superuser access is not required by the permissions that are set from the /dev/tun (or /dev/net/tun) device. When we set the permissions, we used the command: S: chmod 0666 /dev/net/tun. According to Krasnyansky [16], there is no harm in allowing the TUN device to be accessible by non-root users, since CAP_NET_ADMIN capability is required for creating network devices or for connecting to network devices, which are not owned by the user. Since we want to create persistent devices and give ownership of them to unprivileged users, we need the /dev/net/tun device to be usable by these users.

CAP_NET_ADMIN is one of Linux capabilities. According to Linux manual pages [50], for the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: 1) **privileged**, whose effective user ID (EUID) is 0, referred to as superuser or root, and 2) **unprivileged**, whose EUID is non-zero. Superuser processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking. This checking is based on the process’ credentials, e.g., EUID and EGID. Starting with kernel version 2.2, Linux implements another privilege model. According to paper by Hallyn and Morgan [51], this privilege model introduces a separation of root privilege into a set of distinct units, capabilities. Capabilities break the superuser’s privilege into a set of
meaningfully distinguishable privileges. For example, capability CAP_SETUID allows to switch UIDs, while capability CAP_CHOWN enables ability to change the ownership of an object [51]. Capabilities are a per-thread attributes and can be enabled and disabled independently. CAP_NET_ADMIN is the Linux capability that performs various network related operations [50]:

- interface configuration,
- administration of IP firewall, masquerading, and accounting,
- modifications of routing tables,
- binding to any address for transparent proxying,
- setting type-of-service (TOS),
- clearing driver statistics,
- setting promiscuous mode,
- enabling multicasting, and
- allowing use of `setsockopt(2)` to set certain socket options.

When we run the cross-compiled binary executable file in the superuser shell, we get all of the capabilities, and particularly CAP_NET_ADMIN. Therefore, `ioctl()` call is permitted in this case. We can also verify that the superuser shell has all of the permissions by checking its UID and GID, which are set to 0 in our case. We state that the “Operation not permitted” error occurs in the APK process, because the APK process, which calls `ioctl()` function, does not have the CAP_NET_ADMIN capability.

According to one StackOverflow user [52], Android modifies the kernel capability system to allow verification of specific capabilities based on group ID. It means that in order to be granted CAP_NET_ADMIN, the application process needs to have a group ID that is granted the CAP_NET_ADMIN. It can either be root (GID is 0), which is granted all the capabilities, or it can be another group, with group ID set to non-zero that is granted the specific capabilities. For example, if group ID 1234 is granted with the CAP_NET_ADMIN capability, then all the processes with this group ID also have the CAP_NET_ADMIN capability. In order to get these additional capabilities, Android applications have to declare
permissions, according to Android SDK documentation [53]. We have already inspected the list of permissions available for third-party applications in Android SDK documentation [49] and stated that no permission for granting CAP_NET_ADMIN is present in the list.

In another sources, i.e., Android Open Source project repository on GitHub [54] and StackOverflow [55], we found the use of permission "android.permission.NET_ADMIN", which allows access to configure network interfaces. This permission should grant the application processes a group ID with access to CAP_NET_ADMIN, but this permission is not available for third-party applications, and can be granted only to system apps. System apps are the apps that are pre-installed in the system partition with the firmware, also referred as ROM. Since installing DevComApp APK as a system app requires root access and ROM customizations, we do not consider converting DevComApp to a system app as a viable approach in our scenario, but we add it as a possible solution in Section 5.6.

As we discovered, it is not possible for a third-party application to be granted CAP_NET_ADMIN capability via declaring AndroidManifest.xml permissions. We assume that the last option left is to obtain CAP_NET_ADMIN together with other capabilities by granting the APK superuser privileges, as we do for the command shell. In order to grant the user application a superuser access, we need a rooted device [31]. We elaborate on the issues related to rooting further in Section 5.7. We grant our Android application superuser rights, which we consider a solution to the Cause #4 in Figure 5.3, and execute the code to call the ioctl() function. As the result, we observe the same error as before. Despite having superuser privileges, DevComApp is not permitted to perform ioctl() call.

The only reason why this happens, is that the process that calls the ioctl() function does not have the CAP_NET_ADMIN capability, despite granting superuser rights to the application. We can confirm this statement by logging process’ user ID and group ID before the ioctl() call and verifying that both are set to non-zero values. It means, that despite granting superuser rights to the APK, the processes that are started by the native C code within the application are not granted these rights.

The last option for us to test is to set UID and GID of the process that does ioctl() call to zero. In Linux, it is done with commands setuid() and setgid(). We expect that these calls might succeed because we have granted our APK superuser rights, which are required if we need to delegate superuser rights to other processes. However, we observe the error “Operation not permitted” for both calls, which confirms the statement from the previous
paragraph, i.e., despite granting superuser rights to the APK, the processes that are started by the native C code within the application are not granted these rights.

According to Dianne Hackborn [56], the native part of the application has the same permissions as the Java part. On the other hand, developers post on StackOverflow issues like [57], and also receive answers [55], which state that granting superuser rights to the Java application still leads permission denials from inside the native C code. The StackOverflow question [57] was posted in 2010 and there is still no answer that has the solution. We also found this question from 2015 [58] by another StackOverflow user, who reports the exact same issue with `ioctl()` call as we have in our project. We have also reported the issue to StackOverflow [59], but have not received any answers by the time we deliver the thesis.

We confirm that granting superuser rights to the Android application does not lead to native code processes within the application to obtain superuser rights too. Rooting of the device does not play any role in this case and is needed only for granting superuser privileges to the APK in the first place. Therefore, the `ioctl()` call cannot be performed to register the TUN network device with the kernel in the native C code within the third-party APK. It means that DevCom cannot be ported natively inside the Android application, because its processes are not granted the CAP_NET_ADMIN capability that is necessary to do tunneling. This is applicable for all the threads, started inside the native code, i.e., main-thread and POSIX threads.

### 5.6 Possible solutions

This section analyzes six possible solutions to the problem discussed in Section 5.5. The problem is that the processes, started by the DevCom native C part of the APK, cannot obtain the capability to register the TUN network device with kernel via `ioctl()` function call. We also recall that granting superuser rights to the APK does not solve the issue. Our analysis of the possible solutions focuses on two questions:

1. What could be a solution?
2. Can such a solution be implemented in our application scenario?

In this section, we present six different solutions that should be considered by DevCom developers in the future. The solutions are illustrated in the Figure 5.4 with the help of flow chart that we use to systematize the analysis of possible solutions. It should help developers to make a decision on which one of the presented possible solutions suits them best.
5.6.1 System app with NET_ADMIN permission

As described in Section 5.5, the CAP_NET_ADMIN capability can be granted to the system apps by declaring "android.permission.NET_ADMIN". In order to solve the ioctl() issue, our app can be converted to a system app. In this case, it is allowed to declare NET_ADMIN permission and the application processes should be granted the group ID with CAP_NET_ADMIN capability. In order to convert the application to a system app, we need to sign the APK with the system certificate as used for signing the firmware. On our device
we use stock Android 9 Pie, therefore, we do not have access to the required certificate. One of the ways to get the certificate is to make a new custom ROM, and use its certificate to sign our application with it. Another solution is to install the APK as a system app via adb as described by one StackOverflow user in [60]. In both cases, root access to the device is required.

In our application scenario this solution cannot be considered realistic, because it requires root access and critical modifications to the device. We cannot rely on the solution that requires rooting of the patient’s personal device. This solution might be considered realistic only in the scenarios, where a patient is provided with a rooted device that has all of the required software installed. In this case a transmitter, TRN, is not the patient’s smartphone, but a smartphone that the patient gets from the doctor. However, in this case we need to have qualified people in the hospital that will root each device and install the APK as the system app.

5.6.2 Patching the kernel

Android’s source code is open source, which means that any developer can edit the firmware code, recompile it and install it on the device. Thus, we can create our own version of Android in which the ioctl() call from a native C code inside the APK is permitted.

The check of the capabilities of the particular process or group occurs in the kernel. We can modify this check function so it does not return “Operation not permitted” (known as EPERM) if the process does the ioctl() call to register the TUN network device with the kernel. Alternatively, we can modify the AndroidManifest.xml file in the Android core to assign the group ID with CAP_NET_ADMIN for custom permissions and then declare these permissions in our application.

As the solution described in Section 5.6.1, this approach is also considered not realistic in our application scenario, unless the patient is provided with the modified TRN device. In this case, each device to be distributed, has to be rooted and patched, so the APK with the source code can be installed on it. Furthermore, custom patches to the kernel, particularly those that disable permission checks, can be considered as a security breach of the whole firmware. Since security is one of the major requirements from the application scenario, this approach should not be advised.
5.6.3 Installation of custom ROM (firmware)

Instead of making the changes to Android source code by themselves, developers might rely on already existing Android custom ROMs that have less restrictions, compared to the stock firmware that comes with the device. An example of such custom ROM can be LineageOS [61]. We have not conducted any tests within LineageOS to find out if the ioctl() call issue is addressed in this ROM, and only mention it as a possible solution.

Installation of custom firmware does not require recompiling of the kernel compared to the approach in Section 5.6.2. Therefore, this solution can be achieved with less effort. On the other hand, installation of custom ROM still requires root access to the device, which leads to the same conclusions as in the former two approaches: this solution can be considered realistic only in cases where the patient is provided with the TRN device. Furthermore, installation of custom ROMs that have less restrictions does not inevitably preserve the advanced security mechanisms that the stock Android firmware has. This can be considered as a compromise on the security requirement from the application scenario.

5.6.4 Bind a socket to the interface

This is the solution that we initially considered as a workaround that does not require granting superuser privileges to the app, and implemented it. Instead of using file descriptor to the TUN device to intercept the data sent to it by kernel from the virtual interface, we decided to bind a socket to the virtual interface to listen to the traffic. In order to achieve this, we created and configured a virtual interface with the help of ip tuntap commands. Then, we opened a socket and used setsockopt() system call to set SO_BINDTODEVICE option on the socket, as suggested by StackOverflow user in [62], to bind it to the virtual interface. However, we received “Operation not permitted” error again. We analyzed the issue and found out that in order to set SO_BINDTODEVICE option, a process needs to have a CAP_NET_RAW capability. CAP_NET_RAW, just as CAP_NET_ADMIN, is impossible to obtain by a process started by the native code of the APK, even if the superuser privileges are granted to the app, and therefore, does not solve the issue. Just as CAP_NET_ADMIN, CAP_NET_RAW is also not possible to obtain by declaring any permissions available for third-party applications. In order to address the missing CAP_NET_RAW capability we can use the same solutions as described in Sections 5.6.1 – 5.6.3.

5.6.5 DevCom as a binary executable file

In Section 5.2.2, we elaborated that DevCom should be built into the APK, instead of cross-compiling it as binary executable file. During the development process, we managed to
successfully open a virtual interface, allocate a network device and to control a corresponding character device from within the binary executable file. It is possible because the file is executed by a superuser in the shell that has all of the capabilities.

Running DevCom as a binary executable file is a working solution, however, all the drawbacks discussed in Section 5.2.2, like cross-compiling for different ABIs, installation issues etc., exist. With this approach, we separate DevCom and DevComApp. DevCom runs independently as daemon with superuser privileges, while DevComApp is a separate Java or Kotlin Android application that interacts with DevCom via DevCom commands API. This approach, as all four solutions described above, also requires root access to the device.

5.6.6 Reimplementing DevCom in Java
The last option, which we consider as a possible solution, is to reimplement the DevCom codebase in Java/Kotlin. DevCom can be either fully refactored to Java/Kotlin, or only the tunneling module, the Virtual Interface Manager. In the latter case, all data from the virtual interface needs to be forwarded from Java/Kotlin part to the C module via JNI for encryption/decryption and vice versa. This approach requires most effort, compared with previous solutions, but it is the only possible approach to bring DevCom to non-rooted Android devices. In favor of this approach plays the fact that multiple number of VPN applications: OpenVPN [43], Easy-VPN [63], eduVPN [64], are implemented in Java and work as intended. All of these applications use the VpnService API provided by Google that allows to connect the application to the local TUN interface [65].

This approach complies with the security requirements of stock Android, compared to the previously introduced solutions. DevCom Java/Kotlin application does not need to declare any additional permissions, apart from the BIND_VPN_SERVICE permission, which is an allowed permission for third-party applications. No superuser privileges need to be granted to the app or to the end user, thus, limiting the capabilities to perform dangerous activity. It also addresses the issue that we discussed in Section 5.4.3 regarding SELinux. Since now the application interacts with the TUN interface via API provided by the operating system, SELinux policy does not have to be modified.

Lastly, no root access to the device is required to install the application and to run it. It means that the application can be distributed to the end users via Google Play Store. The users can install the application on their personal Android devices without any additional changes to
the device, e.g., rooting, installation of custom ROMs or patching the kernel. This approach is considered by us as the most realistic scenario among all of the discussed approaches.

5.7 Implementation summary

In this section, we summarize what was achieved in the development process and conclude this chapter.

Our original plan to implement the extension to DevCom for Android devices, DevComApp, has failed. The reason is that we cannot run the current DevCom on Android in such way that complies with the security and user-friendliness requirements of the application scenario. The DevCom pre-study in Section 5.1.2 helped us to identify the challenges that needed to be addressed during the implementation process. The major discovery, which was made during the pre-study, is that the assumption that the current DevCom can be compiled and run on modern Android versions was wrong. To understand this further, we identified challenges that the developers should consider when bringing DevCom on Android in the future, e.g., OpenSSL and Avahi cross-compiling, permissions management.

We have added new security and user-friendliness features to DevCom, namely the SHA-2 hash functions and support for pairing with FQDNs. When we discovered that DevCom cannot be run on Android, we decided to solve this issue with the approach that we elaborated on in Section 5.2.2, namely to port DevCom inside Android APK using Android NDK. We started to solve this issue, cross-compiled OpenSSL library, successfully imported the Key Manager module of DevCom, and verified that simplified POSIX-thread functionality in Android does not affect DevCom. When we worked on bringing the Virtual Interface manager module of DevCom to the APK, we discovered a major issue in Android NDK that leads to the error, when ioctl() call is made from within the native C code. Without successful ioctl() call, DevCom cannot register the network device with the kernel, and cannot read and write data from/to the virtual interface. The tunneling functionality is critical in DevCom. Unless the ioctl() issue is solved, DevCom cannot be run on Android. We shifted our focus on analyzing this issue and reasons that lead to it. We made a conclusion that our idea to bring DevCom to Android as a native C library within the APK is not possible unless compromises to the security are made.

We presented six possible solutions to the ioctl() issue that we discovered. The presented solutions to the problem require an answer to the following question: is rooting of the device considered as an option or not. Five of the six solutions require root access to the device. If
we consider these solutions in the scope of our application scenario, then the only realistic conditions is to provide each patient with the rooted Android device. We cannot expect patients, their relatives and doctors to perform rooting of the device by themselves because it is a procedure that requires technical knowledge. Furthermore, according to the journal article by Zhang et al. [66], many of the rooting methods operate by exploiting vulnerabilities in Android. It means that rooting of the device needs to be done by highly qualified personnel. The personnel need to understand the risks of different rooting methods, so they do not compromise on security during the rooting procedure. Meanwhile, the whole design of DevComApp is built on the premise that the parties in our application scenario do not have high technical skills. In case of one of the possible solutions that require rooting is chosen, then the understanding of who and how will perform the rooting and DevCom installation within the hospital application domain is required. Unless this issue is addressed, we do not consider the five solutions to be used in a real world health monitoring scenario.

We argue that the only possible solution that should be considered for our application scenario is the solution discussed in Section 5.6.6, namely reimplementing DevCom, fully or partly, in Java/Kotlin. In this case, we can provide a user-friendly and secure installation and patching of the DevComApp and DevCom via making it available in the Google Play Store (or other application stores). It means that patients can use their personal Android devices as transmitters, and hospital does not need to provide a device with special settings to the patient. Since no rooting of the device is required we do not introduce the situation where a user has unnecessary superuser privileges, thus, preserving the security of the device on the same level as set by the firmware developers. Lastly, rooting the device may lead to the warranty on the device being revoked by the seller/manufacturer.
Chapter 6
Conclusion

In this master thesis, we have designed and made an attempt to implement the system called DevComApp that aimed to improve user-friendliness of pairing mobile devices in DevCom. We have not implemented the solution we aimed in the beginning, but discovered important challenges and issues that we described in detail and analyzed during the development process. In this chapter, we conclude our work. First, we present our contributions in Section 6.1. In Section 6.2, we make a critical assessment of this thesis work. Section 6.3 presents ideas and topics for the future work.

6.1 Contributions

As described in the Introduction, we consider two major contributions in this thesis. The first one is the design of DevComApp proposed to improve user-friendliness of DevCom. The second one is the analysis of challenges and issues during the implementation process of porting DevCom C code inside the Android APK. In this section, we divide these two major contributions into six smaller contributions that were made during the thesis work, and discuss them closely.

Originally, the goal of the thesis was to design and implement the system that would improve pairing procedures in DevCom by introducing new secure and user-friendly pairing scheme, namely pairing with QR codes. Our contributions to this goal are: First, analysis of the health monitoring application scenario. Second, identification of the pairing limitations in the current DevCom, and design of a system to address these limitations on mobile devices, called DevComApp. Third, the conducted QR code study to identify the minimum camera and screen requirements for DevComApp. When we discovered that DevCom is not available on Android, we decided to focus on the implementation process of the solution that would make DevCom available on Android. It became our major goal, and our contributions to it are: Fourth, we conducted a study to understand DevCom code and its practical use, and identified challenges for the development process. Our fifth contribution was to address important limitations of the DevCom itself, namely unsecure use of the SHA-1 hash function and pairing by FQDN. Finally, our sixth contribution is the analysis of the reasons behind the ioctl() issue, and the analysis of six possible solutions to solve it. In the following paragraphs, we discuss the introduced six contributions in detail.
Our **first** contribution is to analyze the health monitoring application scenario. Through the analysis, we identify different roles, devices and their characteristics. Using this information, we identify key requirements that should be taken into consideration by the developers of the systems to be used in the application scenario.

Our **second** contribution is to study DevCom and analyze its use as a system to provide secure device communications between the devices in the application scenario. We analyze how it addresses the requirements from the application scenario and identify its limitations: lack of user-friendly pairing mechanisms, use of the unsecure SHA-1 hash function, unavailability of DevCom on mobile devices. We analyze the existing SDP scheme in DevCom taking into consideration the application scenario according to the principles described by Fomichev *et al.* in [13]. Based on the analysis, we designed a system, DevComApp, to improve the existing pairing mechanisms in DevCom. We decide to focus on mobile devices and propose a more user-friendly SDP scheme that is based on QR code exchange. DevComApp can be considered as an extension that runs on top of DevCom and provides regular users a graphical interface to interact with DevCom, and also allows to perform pairing of two device by exchanging QR codes. We also identify the information that should be encoded in the QR code and describe a possible pairing scenario between a patient and a doctor.

Our **third** contribution is to conduct a study to identify minimum screen and camera requirements for the devices that will run DevComApp and perform pairing with QR codes. We analyzed the worst case scenario, described the process behind QR code encoding and presented the results of our study. In order to provide high level of successful QR code readings of pairing information, DevComApp should be run on the devices that have at least 5MP camera module and 720x1280 px screen resolution.

Our **fourth** contribution is to analyze, describe and solve the challenges to make DevCom available on Android devices. We elaborated on why DevCom should be ported into the APK instead of cross-compiling option. We discussed the challenges to cross-compile external libraries used by DevCom, namely OpenSSL and Avahi. We described the process and all of the issues that we faced so the other developers can use this knowledge (about SELinux, linking, cross-compiling, permissions etc.) when implementing parts of their APK in native C/C++ with Android NDK, which is poorly documented.
Our fifth contribution is to improve the existing DevCom codebase by addressing two limitations: use of unsecure SHA-1 hash function and use of only physical addresses for pairing. We refactored the code to utilize secure SHA-256 hash function and implemented a function that allows pairing to happen not only by IP addresses, but also hostnames. We have evaluated the changes and verified that DevCom works as intended.

Finally, our sixth contribution is to analyze the unforeseen limitation of accessing the TUN network device within the DevCom native code inside the APK. We discovered that processes started by the native part of the APK cannot be granted certain capabilities to perform critical functionality, i.e., ioctl() call on the TUN device without compromise on security. We could not find the solution that would both comply with the requirements set by the application scenario and that would be possible to implement within the time limits of master thesis work. We analyze and describe six possible solutions that should be considered by the DevCom researchers and developers in the future. Five of the presented solutions require rooting of the device, while the sixth one implies refactoring of DevCom in Java/Kotlin.

### 6.2 Critical assessment

If we were to redo this master thesis with the knowledge that we have now, there are things we would consider doing differently. In this section, we discuss these things.

We started off with the goal to design and implement a solution that would provide a more user-friendly pairing mechanisms in DevCom. In order to do this, we have studied DevCom and SDP papers, to identify what should be done to DevCom and how it could be done, according to the principles discussed in the SDP survey. From the start we decided to target Android mobile devices and we knew that Hansen et al. had a solution that worked on Android back in 2013 [6]. We expected that this solution might not work on the modern Android devices, however, we did not expect that making DevCom available on Android would be such a complex task with major issues arising to be addressed. Even the cross-compiled version of DevCom does not run on modern Android devices. And in order to understand why, a significant effort was needed to research the code and Android OS specification itself. Thus, we understood that DevCom was no available on any mobile operating system. Therefore, we added the requirement #1 to the design of DevComApp to make DevCom available on mobile operating systems via DevComApp. Since the cross-compiled option was not considered optional by us from the beginning (we elaborated on it in Section 5.2.2), we decided that porting DevCom to the APK is more preferable. I did not
have much experience in Android development, neither SDK nor NDK, but I relied on my knowledge and experience from C and Java programming languages. I had enough time, around six month until thesis delivery deadline, to learn the basics of Android development so I could implement DevComApp and to port DevCom C code inside it. I started to port DevCom into DevComApp cpp/ directory file by file, solved the arising issues, e.g., POSIX threads, OpenSSL cross-compiling, adding new dependencies to the Gradle build system. When I faced the ioctl() issue for the first time, I expected it to be an issue that can be solved within couple of days. After weeks of research and analysis of the error I could state that this issue cannot be solved unless we compromise the security of the device, which is unacceptable due to the requirements of the application scenario. Together with my supervisors, we decided that this analysis might have larger value than continuing to work on the implementation of DevComApp and decided to shift focus of the thesis to systematic analysis of the failed attempt and analysis of possible solutions. Furthermore, there were no guaranties that we would not face another issue like the one we had with ioctl() call. This would again require a lot of time to solve that we did not have at the end.

Without DevCom running on the device, implementing DevComApp does not have much academic value, since we cannot evaluate the results of the new SDP schemes towards security and user-friendliness if we cannot perform pairing between two devices. Therefore, we put so much time and effort into making DevCom available on Android mobile devices. In hindsight, we should have started to investigate how to run DevCom on Android device much earlier, before starting to work on the design. In this case, we could have identified that bringing DevCom to Android is a major task in itself, and perhaps could have shifted the focus of the thesis into solving this single task. In this case we would have faced the ioctl() issue earlier, and would have had time to reimplement DevCom Virtual Interface Manager module in Java. By the time we started the implementation, we have already spent significant time on the design and research about QR codes. The study to evaluate screen and camera requirements from Section 5.1.1 have already been conducted and we had the understanding of what had to be done to implement DevComApp, apart from functioning DevCom on Android.

I should have also spent less time on studying Android SDK prior to the implementation because in fact most of the time I worked with Android NDK. I spent time to study Android NDK as well but this knowledge was not in a large degree applicable to solve the issues that were arising during the development. In practice, most of the solutions and tutorials to the
issues I found on different developer’s forums and StackOverflow. They were written by a few people and I can state that Android NDK is poorly documented. Taking into consideration this fact would have changed my initial idea to inbuild DevCom C code inside the APK with the NDK. This is also the reason why we described the implementation process in details, so the other developers might find the solution to their implementation problem that they might not find in the NDK documentation.

If we knew in the beginning that DevCom was not available on Android, we could have still achieve the initial idea to implement user-friendly pairing mechanisms in DevCom if we would focus only on desktop Linux computers. In this case, we could have implemented DevComApp as a third-party application that uses command API of the running DevCom daemon. As the result, we would have had a functioning solution implemented according to the design that also can be evaluated and shown as demo.

Maybe I should have been less optimistic when I assessed the ioctl() issue and should have expected it to be a serious issue at the first place. In this case, I would have had time to implement a workaround solution. If the goal would have been to overcome the issue by any means, I could have tested one of the solutions, e.g., patching the kernel. However, if I did this, I could have not presented the systematic analysis of the problem and it would remain unclear why it happened and how it could be solved.

I can say I am quite satisfied with my work, even though there are things that we did not achieve and things went not as originally planned. I still have learned a lot when working on the thesis and hope to find application to this knowledge further in my career.

6.3 Future work

During the thesis work, we have discovered several ideas and issues that might be worth investigating. Due to time restrictions, we could not explore or address them further. The ideas are as follows:

- Make DevCom available on Android devices: We propose to reimplement DevCom or at least the Virtual Interface Manager module in Java/Kotlin since this approach does not require rooting of the devices.

- Implement support for OpenSSL 1.1.1: As we discovered during the DevCom study in Section 5.1.2, DevCom uses the outdated version of OpenSSL library, namely 1.0.2. In 2019, the latest LTS (long term support) version of OpenSSL is 1.1.1.
Making DevCom compatible with the OpenSSL 1.1.1 API requires a major refactoring of the code, but provides a longer support and allows to use more advanced cryptographic functions.

- Implement the **distrust** functionality in DevCom: It should be possible to distrust the community member, delete the corresponding keys, terminate all the connections to the distrusted device, and notify other members about distrust. Furthermore, a user-friendly solution to do it should be designed and implemented in DevComApp.

- Implement support of Android keystore system [67]: The generated keys are stored in the Datastore as regular .pem files. In order to provide a better security of the keys, they should be stored so unauthorized use can be prevented. A possible solution is described in [67].

- Implement QR code exchange SDP scheme and add more SDP schemes that use different PHY channels: Fomichev et al. [13] present a survey of different SDP schemes that can be implemented to provide *Adaptable Secure Device Pairing*, e.g., NFC exchange, RFID, Wi-Fi integrity codes etc.

- Implement SDP scheme for pairing a transmitter device, e.g., a smartphone with sensor devices in health monitoring system: This is a major task for research and should address the characteristics of HMS: low power, embedded OS, limited number of PHY channels. Furthermore, possible use of DevCom on these devices should also be evaluated.

- It would be interesting to evaluate DevCom and implemented pairing schemes towards a real cyberattack scenario: This could help to evaluate the efficiency of the different SDP schemes as well as provide understanding of general security in DevCom.
Bibliography


Appendix A

SHA1 SHA2
$: diff oldsecurity.c security.c
186c186
<   i = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), salt, key_data,
key_data_len, nrounds, key, iv);
---
>   i = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha256(), salt, key_data,
key_data_len, nrounds, key, iv);

SHA1 SHA2
$: diff oldcontrol_traffic.c control_traffic.c
364,365c364,365
<   unsigned char digest[SHA_DIGEST_LENGTH];
<   SHAL(packet, 23 + 1536, digest);
---
>   unsigned char digest[SHA256_DIGEST_LENGTH];
>   SHA256(packet, 23 + 1536, digest);
367c367
<   int success = RSA_sign(NID_sha1, digest, SHA_DIGEST_LENGTH, packet
+ 23 + 1536, &signature_length, key_pair);
---
>   int success = RSA_sign(NID_sha256, digest, SHA256_DIGEST_LENGTH,
packet + 23 + 1536, &signature_length, key_pair);
379,381c379,381
<   unsigned char digest[SHA_DIGEST_LENGTH];
<   SHA1(packet, 23 + 1536, digest);
<   int success = RSA_verify(NID_sha1, digest, SHA_DIGEST_LENGTH,
packet + 23 + 1536, 512, key_pair);
---
>   unsigned char digest[SHA256_DIGEST_LENGTH];
>   SHA256(packet, 23 + 1536, digest);
>   int success = RSA_verify(NID_sha256, digest, SHA256_DIGEST_LENGTH,
packet + 23 + 1536, 512, key_pair);

Changes made to support SHA-256 in DevCom (using command line diff tool to compare with SHA-1 implementation) from Section 5.4.1
# Appendix B

```c
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <arpa/inet.h>

int hostname_to_ip_addr(char* hostname_or_ip, char* physical_address) {
    struct addrinfo *result, *res;

    if (getaddrinfo(hostname_or_ip, NULL, NULL, &result) == -1) {
        perror("getaddrinfo()");
        return -1;
    }

    for (res = result; res != NULL; res = res->ai_next) {
        struct sockaddr_in *address;

        if (res->ai_family == AF_INET) { // IPv4
            address = (struct sockaddr_in *) &(((struct sockaddr_in*) res->ai_addr)->sin_addr);
        } else if(res->ai_family == AF_INET6 ){ //IPv6
            address = (struct sockaddr_in *) &(((struct sockaddr_in6*) res->ai_addr)->sin6_addr);
        } else {
            continue;
        }

        if (inet_ntop(res->ai_family, address, physical_address, INET_ADDRSTRLEN) != NULL) { // resolved and written succesfully
            freeaddrinfo(result);
            return 0;
        }
    }
    freeaddrinfo(result);
    return -1;
}
```

Implementation of `hostname_to_ip_addr()` function from Section 5.4.2