TECHNISCHE
UNIVERSITÄT
DARMSTADT

SECURITY ANALYSIS OF SAMSUNG'S ULTRA-WIDEBAND ECOSYSTEM AND THE USAGE OF NXP ULTRA-WIDEBAND CHIPS

MARTIN REZA HEYDEN

Master's Thesis

February 7, 2022

Secure Mobile Networking Lab
Department of Computer Science
Technische Universität Darmstadt

SECURE MOBILE NETWORKING

Martin Reza Heyden, *Security Analysis of Samsung's Ultra-Wideband Ecosystem and the Usage of NXP Ultra-Wideband Chips*, Master's Thesis, Technische Universität Darmstadt, 2022.

SEEMOO-MSC-0238
Date of submission: February 7, 2022

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Dr.-Ing. Jiska Classen

Secure Mobile Networking Lab
Department of Computer Science
Technische Universität Darmstadt

ABSTRACT

Ultra-Wideband (UWB) is a radio technology that uses a high bandwidth and enables use-cases for precise position estimation in close ranges. In recent years, UWB functionality found its way into many smartphones and Internet of Things (IoT) products, including devices from Samsung that use UWB chips by NXP. However, neither the security of Samsung's UWB ecosystem entities nor the usage and communication of the integrated NXP UWB chips were publicly explored yet. Since UWB integration into smartphones and UWB chips for smartphone-related use-cases are new, only a few directly related works exist. These works analyze the chips' physical-layer security and the integration of UWB into Apple devices, but no work addresses the firmware security of NXP's UWB chips and the UWB integration into Samsung's devices.

Therefore, in our thesis, we analyze the security of Samsung's UWB ecosystem entities, including NXP's *SR100T* UWB chip featured on the Samsung Galaxy S21 Ultra, which we use as our test phone. We further assess the security of Samsung's SmartTag+ that features NXP's *SR040* UWB chip and is part of the ecosystem. Our goal is to identify attack vectors and evaluate a selection of them. Furthermore, to aid our analysis and create attacks in our evaluation, we implement several utilities that help us decode the communication with NXP's UWB chips, attack the *SR100T* on our Samsung phone independently of the user space, and simulate attacks against the ecosystem's entities.

In our evaluation, we find several vulnerabilities in different ecosystem entities. In addition, our findings about NXP's UWB chips and their communication protocols provide a foundation for future research that evaluates the security of UWB chips addressable over Ultra-Wideband Command Interface (UCI) as well as the security of their integration.

ZUSAMMENFASSUNG

Ultra-Wideband (UWB) ist eine Funktechnologie, die eine hohe Bandbreite nutzt und Anwendungsfälle für eine präzise Positionsbestimmung im Nahbereich ermöglicht. In den letzten Jahren wird UWB häufig in Smartphones und Internet of Things (IoT)-Produkten integriert, einschließlich in Samsung-Geräten, welche UWB-Chips von NXP nutzen. Jedoch wurde bisher weder die Sicherheit von Samsungs UWB-Ökosystem, noch die Benutzung und Kommunikation von NXPs UWB-Chips öffentlich untersucht. Da die UWB-Integration in Smartphones und die UWB-Chips für Anwendungsfälle mit Smartphones neu sind, gibt es nur wenige einschlägige Arbeiten. Diese analysieren die Sicherheit der Bitübertragungsschicht von UWB-Chips und die Integration von UWB in Apple-Geräten, aber nicht die Sicherheit der Firmware von NXPs UWB-Chips und die Integration von UWB in Samsungs Geräten.

Deshalb analysieren wir in unserer Thesis die Sicherheit der Entitäten von Samsungs UWB-Ökosystem, mitsamt NXPs *SR100T* UWB-Chip, welcher in unserem Test-Gerät, dem Samsung Galaxy S21 Ultra, integriert ist. Wir überprüfen auch die Sicherheit von Samsungs SmartTag+, welcher NXPs *SR040* UWB-Chip nutzt und Teil von Samsungs UWB-Ökosystem ist. Unser Ziel ist die Identifizierung von Angriffsvektoren und die Evaluation einer Auswahl dieser. Zusätzlich implementieren wir Hilfsmittel für unsere Analyse und Attacken in unserer Evaluation, mit welchen wir die Kommunikation mit NXPs UWB-Chips dekodieren, den *SR100T* in unserem Samsung-Smartphone unabhängig vom Userspace angreifen, und Angriffe gegen Entitäten vom Ökosystem simulieren.

Wir finden in unserer Evaluation mehrere Schwachstellen in verschiedenen Entitäten des Ökosystems. Außerderm bieten unsere Ergebnisse über NXPs UWB-Chips und deren Kommunikationsprotokolle eine Grundlage für zukünftige Arbeiten, welche die Sicherheit von mit über Ultra-Wideband Command Interface (UCI) ansprechbaren UWB-Chips untersuchen oder die Sicherheit der Integration dieser analyisieren.

ACKNOWLEDGMENTS

I would like to thank Dr.-Ing. Jiska Classen for being a very helpful supervisor and for proofreading my thesis.

I would also like to thank my father Matthias Heyden and my mother Fereshteh Heyden. I am grateful that they placed a high value on my education. Both of my parents were always there for me and provided for me during my studies. Additionally, I thank my father for proofreading my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AoA	Angle of Arrival
AOSP	Android Open Source Project
AP	Application Processor
API	Application Programming Interface
BLE	Bluetooth Low Energy
CA	Certificate Authority
CE	Chip Enable
CIR	Channel Impulse Response
CRC	Cyclic Redundancy Check
DOM	Document Object Model
DPIDR	Debug Port Identification Register
DSP	Digitial Signal Processor
ECC	Elliptic-Curve Cryptography
EIRP	Equivalent Isotropic Radiated Power
EXT	Extended
FiRa	Fine Ranging
GID	Group Identifier
GPIO	General-Purpose Input/Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HBCI	Host-Based Command/Control Interface
HRP	High-Rate Pulse Repetition
IDE	Integrated Development Environment
IoT	Internet of Things
IPC	Inter-Process Communication
IR-UWB	Impulse-Radio Ultra-Wideband

LoC	Lines Of Code
LRC	Longitudinal Redundancy Check
LRP	Low-Rate Pulse Repetition
MK	Mobile Knowledge
MT	Message Type
nLOS	Non-Line Of Sight
NTF	Notification
OID	Opcode Identifier
OTA	Over-The-Air
PBF	Packet Boundary Flag
PCB	Printed Circuit Board
PSDU	PHY Service Data Unit
ROM	Read-Only Memory
RSSI	Received Signal Strength Indication
SDK	Software Development Kit
SDR	Software Defined Radio
SPI	Serial Peripheral Interface
SW-DP	Serial Wire Debug Port
SWD	Serial Wire Debug
SWUP	Software Update
TDoA	Time Difference of Arrival
TLV	Type-Length-Value
ToF	Time-of-Flight
TWR	Two Way Ranging
UART	Universal Asynchronous Receiver-Transmitter
UCI	Ultra-Wideband Command Interface
UWB	Ultra-Wideband
WiFi	Wireless Fidelity

XSS	Cross-Site Scripting

1 INTRODUCTION

The radio technology Ultra-Wideband (UWB) is integrated into recent smartphones like the iPhone 13 or the Samsung Galaxy S21 Ultra. It can be used for secure distance and direction measurements between devices, which creates, for example, the opportunity to use a UWB-enabled smartphone as a key in keyless car entry systems [3, 6, 47]. In these phones, UWB is integrated on separate chips, which implement operations such as the distance measurement.

In every UWB-enabled Apple device, the Apple U1 chip implements the UWB stack. On the other side, Samsung uses the NXP *SR100T* for their smartphones, which is a UWB chip made especially for mobile devices [26]. For their Internet of Things (IoT) tracking tag named *SmartTag+*, Samsung uses the NXP *SR040*. The firmware of NXP's UWB chips come in a signed and encrypted form. They are not decrypted in any instance before the chips receive the firmware. Moreover, the standard for UWB-related messaging with NXP's UWB chips is Ultra-Wideband Command Interface (UCI), which is a standard by the Fine Ranging (FiRa) Consortium [31] and is only accessible to its members. For example, a UWB ranging session can be established on the chips using UCI messages. Furthermore, the Host-Based Command/Control Interface (HBCI) protocol is used to manage the *SR100T*, while Software Update (SWUP) is used for the *SR040*. Both protocols are not publicly accessible and proprietary protocols by NXP.

Additional integration of UWB into IoT devices such as Apple's AirTag or Samsung's SmartTag+ broaden the UWB ecosystem's scope but also the attack surface. Apart from finding another UWB tag with a phone, as of now, it is possible to use a UWB-enabled smartphone as a car key [3, 6, 47] or to prioritize contacts in nearby sharing services based on the distance and direction calculated using UWB [4, 11, 45].

1.1 MOTIVATION

The UWB integration into smartphones is new, and the same applies to chips used in UWB use-cases with smartphones. This makes new features available for end-users. However, a new integration of this technology also brings new significant attack vectors. Vulnerabilities in different locations of the UWB ecosystem allow a variety of attacks and can be critical. For example, malicious apps can attack the UWB services and the integrated UWB chip on a phone. In addition, remote attacks against UWB chips or entities processing messages are possible. Also, attacks from a compromised UWB chip can be achievable after locally or remotely compromising the chip. Moreover, with additional devices like the SmartTag+, possible attacks against or from these devices emerge. Therefore, it is important to ensure the security of every entity part of the integration.

Previous work evaluates the physical-layer security of relevant UWB chips [23, 51] and makes an analysis of Apple's UWB ecosystem and the usage of Apple's U1 chip [10, 11, 44]. However, there is no similar research known to us regarding Samsung's UWB

ecosystem as well as the usage of NXP's UWB chips. In this thesis, we close the gap. We analyze Samsung's UWB ecosystem and the usage of NXP's UWB chips with a focus on the NXP *SR100T*. Furthermore, we identify attack vectors, attack selected ones, and point out essential information for future research.

1.2 CONTRIBUTIONS

Our goal is to evaluate the security of entities in Samsung's UWB ecosystem, including NXP's UWB chips and the SmartTag+. Furthermore, we aim to provide a foundation for future work that analyzes the UWB integration with UCI-addressable UWB chips. With respect to these goals, our main contributions are:

- Reverse engineering of the not publicly accessible protocols UCI and HBCI, which are used to communicate with NXP's UWB chips. In addition, a Wireshark dissector is implemented to decode these protocols' messages.
- Analysis of the firmware transfer and building of a state machine for NXP's *SR100T* UWB chip.
- Analysis of Samsung's UWB ecosystem entities and the role of each entity.
- Implementation of scripts that build on a pre-existing tool to directly attack the *SR100T* on Android phones. In addition, Frida scripts are implemented that are used for simulating attacks against Samsung's UWB services.
- Identification of attack vectors in Samsung's UWB ecosystem and an evaluation for a selection of them. Thereby, several vulnerabilities are found that are reported to Samsung.

1.3 OUTLINE

In Chapter 2, we introduce UWB and briefly summarize the adoption of UWB in systems related to our work. We further introduce NXP's UWB chips and Samsung's UWB ecosystem. Afterwards, we give an overview of related work in Chapter 3.

Subsequently, we analyze the communication with and usage of NXP's UWB chips in Chapter 4. Then, we analyze the entities of Samsung's UWB ecosystem in Chapter 5. Next, in Chapter 6, we identify attack vectors based on our previous findings, and we create a selection of them for our evaluation.

In Chapter 7, we present implemented utilities that we use for our evaluation and provide for future work. Then, in Chapter 8, we evaluate the security of entities in Samsung's UWB ecosystem with a focus on assessing our selected attack vectors. Afterwards, we discuss our results in Chapter 9. Finally, we conclude our thesis and give an outlook for future work in Chapter 10.

2 BACKGROUND

2.1 ULTRA-WIDEBAND

Ultra-Wideband (UWB) is a radio technology that enables high bandwidth communication over short distances with low energy consumption [56]. It is part of the 802.15.4 standard and is not a new technology [37, 56]. It gained proper recognition in the last few years, and manufacturers slowly started integrating it into their products. Moreover, in Impulse-Radio Ultra-Wideband (IR-UWB), which is a subtype of UWB, very short pulses are used [37]. In this thesis, UWB is used interchangeably with IR-UWB since all the systems analyzed use IR-UWB.

UWB uses large frequency ranges with a minimum bandwidth of 500 MHz or 20 % of the center frequency [37, 56], which can be considered a high value. From a list of commercially available Software Defined Radios (SDRs) [1], we know that even most of the high-end SDRs do not support this bandwidth. Moreover, the frequency range in which UWB is allowed to operate is between 3.1 and 10.6 GHz [13], which is also not supported by most SDRs [1]. Furthermore, the Equivalent Isotropic Radiated Power (EIRP) is limited to a low value of -41.3 dBm/MHz[1] [13]. Other radio standards such as Wireless Fidelity (WiFi) allow a higher EIRP. For example, in Germany, a device is allowed to emit a WiFi signal in the 2.4000 - 2.4835 GHz frequency range that does not exceed an EIRP value of 20 dBm for its complete signal [7]. When we consider a 500 MHz wide UWB signal with the maximal EIRP per MHz, we only have an EIRP of around -14.3 dBm for the 500 MHz wide signal, which is more than 2500 times less than the previous mentioned WiFi signal with an EIRP of 20 dBm. In conclusion, the large bandwidth, the limited power usage, and the high frequency range make UWB applicable for close range and high data rate use-cases.

There are two types of UWB physical interfaces defined in the IEEE 802.15.4 standard, which are Low-Rate Pulse Repetition (LRP) and High-Rate Pulse Repetition (HRP) [13], whereby HRP is used by most UWB chips integrated in devices for smartphone-related use-cases [51]. The rate impacts the energy per pulse: The lower rate, the higher the energy per pulse. Furthermore, LRP enables more energy-efficient implementations, while HRP allows higher precision as well as easier limiting of interference [18].

Usually, there is an out-of-band establishment of UWB sessions between devices, for example, over Bluetooth Low Energy (BLE) [37]. By making use of an already existing technology, steps like UWB device discovery or exchange of the UWB session parameters like the channel are facilitated. Additionally, it also can be used for out-of-band authentication before even trying to establish a UWB session [37].

Many use-cases exist to which UWB can be adapted. For example, using UWB, it is possible to implement high-rate data exchange services [56]. Also, it is possible to implement high-granularity distance and direction calculations over short distances between two or more devices such that further applications can be built, which rely

1 The frequency range and the EIRP are both declared by the Federal Communications Commission (FCC).

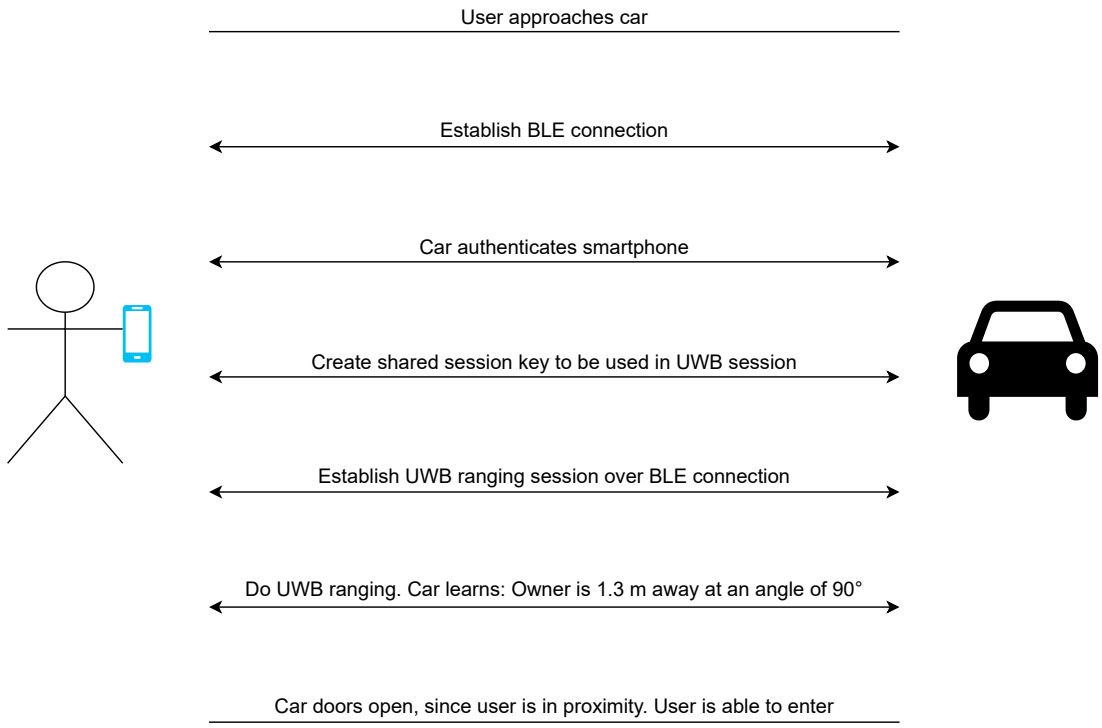

Figure 1: Overview of a ranging session example.

on precise position estimation between devices [37, 56]. This plays a significant role for UWB-equipped smartphones and cars, for example, to allow using the smartphone as a car key [3, 6] as well as to use it as a key for general physical access control systems [37]. A list of further use-cases exists in [17]. These use-cases include using a smartphone for indoor navigation, mobile payment, and social distancing.

Figure 1 shows the summarized process of how a smartphone can be utilized to open a car and drive away. We derive this figure based on the information provided in [3, 6, 37]. After two devices (smartphone and car) come close enough to establish a BLE connection, they can authenticate each other, while it is at least necessary to authenticate the smartphone. Afterwards, both parties can create a shared session key, which is used for *secure ranging*. For example, these two devices can then with Two Way Ranging (TWR) calculate their distance between each other by doing Time-of-Flight (ToF) measurements [37]. When the access controller (the car) knows that the user with its smartphone is in proximity, it will open the doors, and the user can enter [6]. Also part of ranging are Angle of Arrival (AoA) measurements, whereby multiple methods exist to calculate the direction of incoming signals of another device [12]. For example, the car can estimate with multiple antennas the owner's relative position to the car by calculating the time difference of arrival of the same smartphone's signal [12]. If the owner is behind the car, then the car can open the trunk.

Moreover, ranging can be used to find lost UWB tags: A device may learn the tag's distance and direction, and the user can actively move towards the tag to find it, while getting live updates of the distance and direction [46].

2.2 ADOPTION OF ULTRA-WIDEBAND

Many companies discovered the beneficial characteristics of UWB and started to integrate the radio technology into their devices. A crucial role for the adaptation of UWB in the mass market plays the organization *Fine Ranging (FiRa) Consortium*. The consortium's members include companies like Apple, Google, Samsung, Facebook, Cisco, NXP and Bosch. A complete list of the members can be found on the official website [16]. One may become a member for an annual membership fee starting from 5000 USD or a one-time fee of 2500 USD for educational purposes [14].

The FiRa Consortium's official goals are the development of UWB use-cases, ensuring a general seamless interoperability of UWB, and the promotion of UWB ecosystems [15]. Developed specifications and documents are not accessible to the public. Currently, only the consortium's members have access to these. One of these not publicly accessible standards developed by the consortium is the Ultra-Wideband Command Interface (UCI) [31]. This standard defines the interface to establish UWB sessions between an application and a chip implementing UWB.

The *Car Connectivity Consortium* also exists, which is similar to the FiRa Consortium but focuses on smartphone-to-car related use-cases only [8, 47]. It has smartphone vendors as members like Samsung and Apple as well as major car companies like BMW and Volkswagen [8]. This consortium is responsible for establishing interoperability standards between smartphones and cars for harnessing smartphones as car keys using UWB [9].

Apple was the first manufacturer to integrate UWB technology into smartphones. iPhones beginning with the iPhone 11 contain a dedicated chip — named U1 — for UWB. Additional to the iPhones, Apple includes UWB technology into other devices like the Apple Watch Series 6 and the AirTag. All of these devices have the dedicated U1 chip for enabling UWB [36].

After Apple, Samsung started to integrate UWB technology into their flagship smartphones, beginning with the Samsung Galaxy Note 20 Ultra, which contains a dedicated UWB chip from NXP, the *SR100T*. Samsung also introduced a UWB-enabled Internet of Things (IoT) device named *SmartTag+*, which is only compatible with recent Samsung phones, and the UWB functionality can only be used with UWB-enabled Samsung phones [46]. The SmartTag+ is equipped with a different UWB chip of NXP, which is the *SR040*. It is further similar to Apple's AirTag, and it also has the same intention, namely to find objects that are attached with the SmartTag+ over BLE and UWB, by calculating the direction and distance between smartphone and SmartTag+ [46]. As of the mid of December 2021, Samsung uses their own software in combination with NXP's UWB Software Development Kit (SDK) to implement a UWB Application Programming Interface (API) and the communication with the *SR100T*, which is independent of the Android Open Source Project (AOSP).

In 2020, Android started to integrate a UWB API into the AOSP, to later enable apps to make use of UWB functionality [53]. As of the mid of October 2021, the Hardware Abstraction Layer (HAL) interface currently is not fully implemented [48, 52]. The fully functional UWB API software stack will be likely included in the release of Android 13 in 2022 [48]. Furthermore, Google, the owner of Android, released the Google Pixel 6 Pro in October 2021, which also includes UWB and makes use of the UWB API. However, it uses custom software to implement the HAL interface [52]. Moreover, based on a teardown,

we know that the Pixel 6 Pro uses a UWB chip by Qorvo [54], which we also verify ourselves by downloading a Pixel 6 Pro image from Google's website [19]. However, we do not know the exact chip.

In August 2021, Xiaomi released the phone Xiaomi Mi Mix 4 in China, which also integrates UWB. Like Samsung phones, the Xiaomi Mi Mix 4 uses the NXP *SR100T* as its UWB chip [28].

Apart from Samsung, Google, Apple, and Xiaomi, many other companies work on UWB solutions. For example, Bosch promotes its keyless car access system based on UWB, which enables users of UWB-enabled smartphones to use the phone as the car key, without even bringing it out of the pocket [6]. BMW has already integrated a similar UWB keyless car system into their car BMW iX, released 2021 in Germany [3, 55].

2.3 SAMSUNG'S ULTRA-WIDEBAND ECOSYSTEM

Samsung integrates UWB functionality into their Galaxy flagship smartphones and the Galaxy SmartTag+. All of these UWB-enabled devices have a dedicated UWB chip from NXP. The firmware of these chips is signed and encrypted. It is not available as a decrypted version before it is sent to the corresponding chip, and only the chips can decrypt it. Moreover, the firmware itself is also developed by NXP.

Samsung's smartphones use NXP's *SR100T* UWB chip. The SmartTag+ uses the *SR040*, which can be seen as a low-energy version with less capabilities in comparison to the *SR100T* and the *SR150*, which is also a UWB chip developed by NXP [26, 31, 33, 34].

To manage the *SR100T* and the *SR150*, the communication standard Host-Based Command/Control Interface (HBCI) is used. On the other side, the *SR040* presumably does not support HBCI but only Software Update (SWUP), which is a different protocol only used for transferring the firmware. Furthermore, the communication standard to establish UWB sessions with all NXP chips is UCI, which is a standard by the FiRa Consortium [31]. All previously named standards are not publicly accessible, and the protocols UCI and HBCI will be reverse-engineered and presented by us in Chapter 4. Additionally, the transport of messages between each UWB chip and host processor is done over Serial Peripheral Interface (SPI) [31].

Currently, there are three official UWB use-cases, which are available to a user of a UWB-enabled Samsung smartphone [45–47]. Moreover, we find in our thesis an additional undocumented use-case of UWB, which we describe in Section 5.4.3. The first use-case is searching the SmartTag+ [46]. We show the high-level workflow of finding the SmartTag+ using UWB in Figure 2. First, the SmartTag+ distance and direction estimation process is started over a plugin in the Samsung SmartThings app[2], which is an app by Samsung to manage a variety of IoT devices. Then, the following distance and direction measurements between the phone and the SmartTag+ are set up over BLE. Afterwards, the UWB measurements are done directly between the UWB chips. The results of each measurement, which include the distance and values such as the AoA, are sent back from the UWB chip to the UWB processes on the phone for further post-processing.

The second use-case is the integration of UWB as part of the *Nearby Share* process [45]. *Nearby Share* lets users share files with other users in proximity. When using *Nearby Share* to share a file with a UWB-enabled Samsung phone, the user who wants to share the file

2 Package ID: com.samsung.android.oneconnect

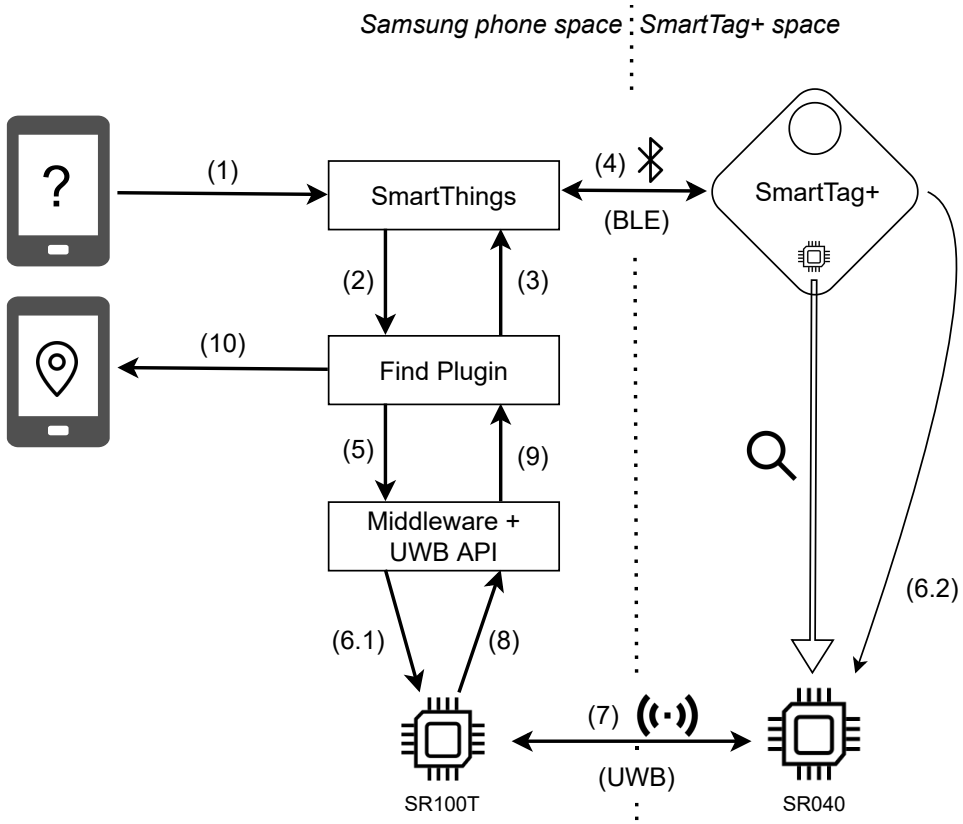

Figure 2: High-level overview of the ranging session establishment with a SmartTag+. The numbers indicate the steps.

can see the live direction of other users that also have a UWB-enabled Samsung phone and have *Nearby Share* enabled. The displayed direction changes in real-time when one of the phones in the process is moved. Even if the data could be exchanged over UWB, here UWB is only used for the real-time direction measurements, and the data transfer is done over Bluetooth, BLE, WebRTC, or peer-to-peer WiFi [50].

In the third use-case, a user can harness its UWB-enabled Samsung smartphone as a car key [47]. Because UWB is used, the car can learn that the owner or members are in proximity, and if enabled, the car opens automatically and can be started. Thereby, the user does not need to put the smartphone out of the pocket and unlock the car because the authentication and proximity detection happens in the background [47]. In 2021, this will be only available for buyers of a Genesis GV60[3]. However, one can expect soon that users of UWB-enabled Samsung smartphones can use their phone as a UWB car key of other major car companies like BMW, since Samsung is part of the Car Connectivity Consortium [47].

2.4 MK UWB KITS

For companies and other interested parties, there is a possibility to develop and test products with integrated UWB technology by using development kits. Mobile Knowledge

3 Genesis is a car manufacturer and is part of the Hyundai Motor Group.

CHARACTERISTIC	SR040	SR100T	SR150
Firmware Encryption	Yes	Yes	Yes
Secure Ranging	No	Yes	Yes
# TX	1	1	1
# RX	1	2	2
Communication	SWUP & UCI	HBCI & UCI	HBCI & UCI

Table 1: Characteristics of NXP's UWB chips.

(MK) provides a small variety of such UWB development kits, which come with boards that have integrated NXP UWB chips and to which one can push their self developed application that communicates with the UWB chip [25]. From the kits' attributive contents, we know that they also contain precompiled applications, which can be pushed to the board's host chip. They further contain the applications' source code, which can fully communicate with the NXP *SR040*, *SR100T*, and *SR150*. The source code that is responsible for the communication with NXP UWB chips is provided by NXP. Moreover, the kits contain the firmware of NXP's UWB chips, which are however encrypted and signed. These firmware also are uploaded in encrypted form to the chips. Apart from the source code, some documentation files explain the general contents and setup of the kits.

2.5 NXP ULTRA-WIDEBAND CHIPS

As of December 2021, there are five chips released by NXP that implement UWB, which all run under the code name *Trimension*. The official names and the use-case of these chips are *Trimension SR040* for Tags, *Trimension SR100T* for mobile devices, *Trimension SR150* for common devices and anchors, *Trimension QL23D0* for industrial devices, and *Trimension NCJ29D5* for cars [30]. While the communication protocols with the *SR040*, *SR100T*, and *SR150* are UCI, HBCI, and SWUP as well as the communication is done over SPI on the Printed Circuit Board (PCB), it is not clear which protocol and hardware communication interface is used with the two other chips. However, it is likely that these two at least also support UCI. Because of the missing information, we will not consider the *Trimension QL23D0* and *Trimension NCJ29D5* in our thesis.

To learn the different capabilities of the *SR100T*, *SR040*, and *SR150*, we examine the chips' documents and descriptions, which are provided on NXP's official website [26, 31, 33, 34]. We further use the MK UWB kits' source code to learn the chips' essential differences. We show an overview in Table 1.

For this thesis, particular the *SR040* and *SR100T* are interesting, because the *SR040* is integrated in the SmartTag+, and the *SR100T* is integrated in all UWB-supporting Samsung smartphones. In this thesis, the greatest focus lies on the *SR100T*. Both of the chips and the *SR150* are similar to a certain degree with different capabilities. They have a similar description on the official sites of NXP [26, 31, 33, 34] and share the main communication protocol UCI. Also, the *SR100T* and *SR150* support HBCI. Furthermore, the firmware of each of these chips comes encrypted and signed. A decrypted version is not publicly available. The encrypted firmware is transferred in encrypted form to the

chips and is not decrypted in any instance before. Neither searching in the MK UWB kits' source code nor analyzing the communication with the *SR040* or *SR100T* as well as disassembling or decompiling UWB-related code on the Samsung phone, let us get the decrypted firmware to analyze it further.

From [31, 33, 34] we learn much information about the *SR040* and *SR150*, which we describe in this paragraph. We learn that the *SR040* and *SR150* are Arm Cortex-M33 based chips and implement UWB in HRP mode. In addition, the *SR150* comes with a separate Digitial Signal Processor (DSP) on board that implements algorithms for measurement of ToF, Time Difference of Arrival (TDoA), and AoA. We assume the same for the *SR040* but do not find a clear statement. The DSP firmware is part of the encrypted chip firmware. Also, both chips have a ranging accuracy of ±10 cm for Non-Line Of Sight (nLOS). Furthermore, the *SR040* is suitable for battery-operated devices like UWB Tags. It comes with only one antenna that can used as a transmitter and receiver, and between transmitting and receiving mode is switched. The *SR150* comes with one transmitting antenna and two receiving antennas for enabling two-dimensional AoA measurements. We find no indication that the transmitting antenna might be used as a third receiving antenna to enable three-dimensional AoA measurements. Additionally, we find the indication that the *SR150* has software support for handling three antennas, and with these three-dimensional AoA measurements can be done. To enable this, we assume an external third receiving antenna can be connected to the *SR150*'s General-Purpose Input/Output (GPIO) pins.

There is no publicly available information by NXP, which elucidates specifics about the *SR100T*, except a press release of NXP. Based on this press release in [26] and additional teardown photographies of the Samsung Galaxy S21 Ultra in [20], we can tell that the *SR100T* comes in combination with an external secure element from NXP, which is the *SN110U*. It further has a ±10 cm range for nLOS and ±3° angle accuracy. It also has two receiving antennas for two-dimensional AoA measurements like the *SR150* [31, 34]. For other characteristics, we assume a high similarity to the *SR150*'s characteristics. Additionally, the contents of the MK UWB kits and UWB-related files on our Samsung phone indicate that the *SR100T*'s codename is *Helios*.

Based on documentation files from the kits, we know that the *SR150* additionally integrates Arm Trustzone[4], which can be connected to a secure element by NXP (*EdgeLock SE051W*) to generate and store cryptographic keys. The *SR150* uses the secure element to enable *secure ranging* measurements [31, 34].

We further find certain UCI opcodes for the *SR150* in the UWB kits' source code, which are related to a secure element connection. We also find these for the *SR100T*. Additionally, we assume that the *SR100T* also integrates Arm Trustzone and that the secure element *SN110U* is also used to enable *secure ranging*. Futhermore, for the *SR040* there exists no UCI opcode in the source that indicates the support for a secure element connection. We also do not find statements in the documentation files related to the *SR040*, which indicate a secure element connection support [31, 33]. In addition, the *SR040*'s official information website states only a "Reliable UWB Solution" [35]. Thus, we conclude that *secure ranging* might not be possible with the *SR040*. However, the *SR040* still integrates Arm Trustzone [33].

4 Trustzone is a family of Trusted Execution Environment (TEE) implementations by Arm.

We do not find UCI opcodes or configuration values indicating that it is possible to directly enable *secure ranging* with the *SR100T* and *SR150*. However, it is possible to set certain configuration values, which are related to a secure distance evaluation. We assume by setting these configuration values, *secure ranging* will be implicitly used.

2.6 OTHER ULTRA-WIDEBAND CHIPS

Apple's U1 UWB chip is included in various Apple devices [36]. It implements UWB in HRP mode [51], and the firmware is not encrypted [11]. Additonally, the chip is separated into an Application Processor (AP) and a DSP [11]. We further do not find information that declares if UCI is used to communicate with the chip from the user space.

The Google Pixel 6 Pro integrates a UWB chip by Qorvo [54]. We do not find any resource that names the exact chip. We can only tell that it is a UWB of Qorvo's DW3000 series since the product websites state these are FiRa-compliant [39–42], which we assume means interoperability with other current UWB chips as these from NXP. In addition, the product websites state interoperability with Apple's U1 chip [39–42].

Qorvo's chips implement UWB in HRP mode [38]. They come with a separate AP and DSP [23]. Furthermore, we do not find any information with web searches, on product websites, or in documentation files that tell us if the communication protocol with the Qorvo chips is UCI or not. However, since the product websites declare the chips are FiRa-compliant, we assume it is possible [39–42].

Currently, there also exist UWB chips from many other vendors [43]. These are intended for different use-cases and are often not compatible with the previously mentioned chips, but they often also do not target the same use-cases and therefore do not aim for interoperability with FiRa-compliant devices. On the other side, NXP's and Qorvo's chips are FiRa-compliant, meaning they can operate mutually [31, 39–42]. In addition, chips from both vendors can operate with Apple's U1 UWB chip [32, 39–42]. For general interoperability between devices featuring a UWB chip, it is also essential that they are interoperable on a different layer for out-of-band session establishment. This means an entity must exchange the UWB session parameters out-of-band, and the same or a different entity hands the chip the UWB session parameters. Furthermore, one can expect that FiRa-compliant UWB chips will follow in the future, and general interoperability between these is ensured, also between devices featuring the chips and that exchange out-of-band session parameters.

3 RELATED WORK

In this chapter, we give a general overview of related work. We divide the related work into two parts. The first part concentrates on the physical-layer security of recent Ultra-Wideband (UWB) chips including selected NXP UWB chips. Moreover, the second part concentrates on Samsung's normal SmartTag and selected entities of Apple's UWB ecosystem.

Next, we describe related work. Afterwards, we briefly explain the difference of related work to our contributions.

3.1 PHYSICAL-LAYER SECURITY OF UWB CHIPS

In [51], Singh et al. study the physical-layer security of UWB in High-Rate Pulse Repetition (HRP) mode. The work evaluates possible attacks and attack strategies. In simulations, Singh et al. derive two attacks from the Cicada attack, which builds on the hardness of differentiating a received signal from interference [51]. The evaluation indicates distance reduction attacks in HRP mode might be possible. Furthermore, Singh et al. come to the conclusion that UWB ranging in HRP mode is hard to configure both performant and secure simultaneously.

Leu et al. practically evaluate in [23] distance reduction attacks againt recent chips that implement UWB in HRP mode. The work successfully demonstrates a reduced measured distance on an iPhone 11 Pro, which contains Apple's U1 UWB chip and does UWB ranging with another UWB device. The other device can have a UWB chip from a different manufacturer like NXP, and at least one of the ranging partners need to have Apple's U1 chip. The attack works by overshadowing parts of the transmitter's signal at the receiver, which tricks the receiver to detect an early copy of the signal in the attacker created noise. In result, the receiver calculates a shorter distance of up to 12 meters, and the attack success rate is up to 4 %. In addition, an attacker does not need to know cryptographic secrets for the attack. Furthermore, multiple devices with different UWB chips are used as the iPhone's ranging partner, which are two tests devices using NXP's *SR040* or *SR150* UWB chip.

3.2 TRACKING TAGS AND APPLE'S UWB ECOSYSTEM

In [11], Classen and Heinrich analyze the UWB integration into iOS. The work presents which entities use and provide UWB in iOS. For example, when discovering devices in proximity for sharing a file, Apple's AirDrop[1] service sends extra UWB beacons over Bluetooth Low Energy (BLE). Devices in proxmity that are the sender's contacts and receive these UWB beacons do UWB ranging with the sender when they have a UWB-capable iPhone. Then, the sender's phone gets information about the distance and

1 https://support.apple.com/en-us/HT204144

direction of contacts. This information is used to display the closest contact to which the sender's phone points.

Moreover, the work in [11] also presents information about the *Nearby Interaction Framework*. Beginning with iOS 15, third-party apps can use the framework to initialize UWB ranging sessions with certified third-party UWB devices. Independently of the apps using UWB on an iPhone, the *nearbyd* daemon handles the communication with the chip using Apple's *IOKit*, which provides an interface for communicating with drivers [11].

In [11], Classen and Heinrich also give additional information about the communication and usage of Apple's U1 chip in iOS. The work further provides information about the chip itself.

Furthermore, the work in [10] analyzes the communication between Apple's iPhone and AirTag. Additionally, the Over-The-Air (OTA) firmware update process is analyzed. Two findings are that the firmware of Apple's U1 UWB chip on the AirTag and the AirTag's main firmware can be downgraded.

Chips of the nRF52 series have an integrated protection that disables Serial Wire Debug (SWD) access [24]. This protection can be bypassed using a fault injection attack to reenable the SWD interface [24].

In [44], further work builds on [24] and attacks Apple's AirTag that features a chip of the nRF52 series. The work shows that the firmware of Apple's AirTag can be dumped and modified over SWD after exploiting the chip's vulnerability.

Samsung's normal SmartTag, which does not integrate UWB functionality, also uses a chip of the nRF52 series like the AirTag [5]. In a GitHub repository [5], we find information that the chip's vulnerability also can be exploited on a SmartTag. A firmware dump is provided and further information for replicating the attack. We conclude that a firmware manipulation is also possible since full debug capabilities exist on the chip after exploiting it [24].

3.3 DIFFERENCES TO OUR CONTRIBUTIONS

The physical-layer security of UWB chips is not a part of our thesis. Instead, we focus on the implementation of entities on a Samsung phone that provide UWB functionality to apps. We also have a focus on the communication of these entities with NXP's UWB chips, and we further want to make conclusions about the chips' firmware security. To the best of our knowlegdge, there is no such research yet.

Learning from [10, 11] about the integration of UWB functionality into Apple's UWB ecosystem initally helps us understanding how such a system can be implemented. Yet, Samsung's UWB ecosystem is differently implemented and runs on a different mobile operating system. Furthermore, Samsung uses a different UWB chip that has a different communication protocol, which is not publicy available and was not analyzed by other work. Therefore, we have to start from the beginning to learn about Samsung's UWB ecosystem and the communication with NXP's UWB chips.

While research about the AirTag's OTA firmware update meachnism [10] helps us to get an understanding of a pracical implementation for tracking tags, we have no information about the normal SmartTag's and SmartTag+'s OTA firmware update meachnism. We have to analyze the mechanism by ourselves.

Apple's AirTag and Samsung's normal SmartTag are vulnerable to hardware exploits due to the integrated chips of the nRF52 series [5, 24, 44]. On the other side, Samsung's SmartTag+ has a complete different Printed Circuit Board (PCB) with a different main CPU, which is the NXP *QN9090*. We do not know of research about the SmartTag+'s hardware, and we also do not know of attacks against the CPU. Therefore, we have to do a hardware analysis from scratch.

4 COMMUNICATION WITH NXP ULTRA-WIDEBAND CHIPS

Before we identify attack vectors and create attacks, we need to understand Samsung's Ultra-Wideband (UWB) ecosystem and learn more about NXP's UWB chips. When we describe our analysis and findings of Samsung's UWB ecosystem, we follow a bottom-up approach since it is easier for the reader to understand how the entities in the UWB ecosystem work together. Hence, we first show how the communication between the user space of the operating system (Android) and the NXP *SR100T* works on a Samsung phone. Thereby, we focus on the *SR100T*'s communication protocols. These protocols are Ultra-Wideband Command Interface (UCI) for establishing UWB sessions and Host-Based Command/Control Interface (HBCI) for managing NXP UWB chips. Neither in the web nor the UWB kits' source code, we can find the full name or an explanation of HBCI, and we only assume the given full name. Additionally, we use NXP's *SR100T* UWB chip as a representative for the *SR040* and *SR150* since it is integrated on our phone and the chips are similar [26, 31, 33, 34].

Furthermore, the *SR040* does support UCI messages but presumably does not support HBCI messages. In this chaper, we find in our UWB kit content analysis that a protocol named *Software Update (SWUP)* is used for transferring the firmware. This protocol is comparable to the HBCI protocol but has a different message set. We do not find comparable protocols for SWUP and conclude it also is a not publicly available standard. We have a focus on the *SR100T* and skip an analysis of SWUP since it is only used for transferring the *SR040*'s firmware, and UCI and HBCI are the main protocols we target.

To understand the usage of and the communication with NXP UWB chips, we reverse engineer the source code of the standard *Mobile Knowledge (MK) UWB kit SR150/SR040*[1] as well as of the *MK UWB kit mobile edition*[2], which can be bought on MK's official website [25]. Moreover, we enable *Verbose vendor logging* and *Samsung verbose debug logging* in the developer options of our test phone, which is the Samsung Galaxy S21 Ultra. This results in additional log messages. Then, when we trigger UWB functionality on our phone, the exchanged UCI and HBCI messages with the *SR100T* are logged. We use the logged messages to help our understanding of the communication with the *SR100T*.

Next, we analyze the MK UWB kits' contents. Subsequently, we examine the communication protocols UCI and HBCI. Then, we briefly analyze the *SR100T*'s local firmware download process. Afterwards, we briefly describe the *SR100T*'s driver and build the *SR100T*'s state machine based on the knowledge of the protocols and the driver.

4.1 MK UWB KITS

We analyze the UWB kits' contents in order to achieve several goals. We want to extend our understanding of UCI and learn how a UWB application works from the application level down to the communication level of communicating with the UWB chips. Addi-

1 https://www.themobileknowledge.com/product/mk-uwb-kit-sr150-sr040/
2 https://www.themobileknowledge.com/product/mk-uwb-kit-mobile-edition/

tionally, we want to get a better general understanding of NXP's UWB chips, and we also want learn to which extent Samsung has inherited the provided source code in their ecosystem, which we cover in Chapter 5.

4.1.1 *Content Analysis*

There is a great number of files that come with the kits. After automatically extracting every zip file, we count more than 12000 files using the Linux *find* and *wc* commands. The main folder's documentation files do not help us to learn the information we need to achieve our main goals. There is no description of which files and folders handle the communication with the chips or how the communication works. In addition, the general usage of NXP's UWB chips is not explained.

Therefore, to achieve the goals, we first need to analyze the contents. First, we manually click through folders to get a general understanding of the folder structures. Afterwards, we automatically search for helpful documentation and source code files. Thereby, we filter out the most important folders and files, including documentation files in lower folders and UWB-related source code files. We further load essential source code files into an Integrated Development Environment (IDE) and analyze the meaning and interconnection of the source code files' methods.

4.1.2 *Overview of Important Findings*

We find several interesting documentation files, which are distributed across the MK UWB kits. Some are handy and include information about the inner workings of the source code files as well as general mechanisms like an abstract description of the timeout handling with the UWB chips.

We further find the encrypted and signed firmware for the *SR040*, *SR100T*, and *SR150*. For both latter chips we find multiple firmware versions.

Based on our previous analysis of the contents, we can derive that only a fraction is intended to be modified. Most source code files are ready to import and provide a fully working Application Programming Interface (API) to build quick applications, which we describe next.

4.1.2.1 *NXP's UWB API*

In Appendix A.2 we show the paths where the UWB-related files can be found, whereby each path targets another chip. The gist is in the paths' subfolder *libs*, and it contains three more subfolders. These subfolders build the UWB API itself and are intended to be imported without modifications. Additionally, all of the code in *libs* is provided by NXP.

The first subfolder is *halimpl*, which name likely stands for "hardware abstraction layer implementation". It contains many files that are related to the low-level operations with the chips, such as the firmware transfer to the *SR100T*. Here we also can extract all HBCI messages by evaluating headers and methods.

The second subfolder of *libs*, which is named *uci-core*, contains methods related to UCI message creation and processing. It further contains all UCI-related opcodes and parameters declared in C define statements. In Figure 3, we show the naming scheme of

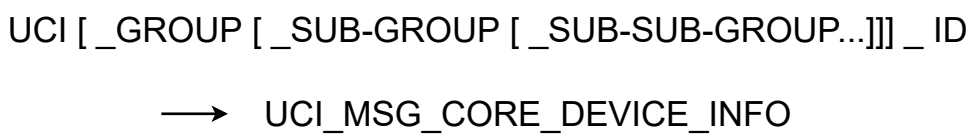

Figure 3: Naming of UCI parameters in define statements of the UWB kits. "[...]" indicates optionality.

these define statements. The naming scheme is expressive and helps us understanding the opcode's or parameter's meaning as well as the referencing method's fuctionality.

The last subfolder, *uwb-iot*, contains the actual UWB API, which is the main starting point from which the rest of the classes in *libs* are addressed and which one can use to build their application.

We conclude that the UCI communication is not fully equal between the host and all chips, because of minor differences in the *libs* folder's contents for each targeted chip. For example, there are differences for definitions and interpretations of proprietary UCI messages that extend the standard set of UCI messages.

Additionally, we conclude that the knowledge of the source code is helpful for us when we attack the chip in the future. The probability is high that some of the code is also used in the encrypted firmware of NXP's UWB chips. To save time and costs, it would make sense for NXP to reuse as much code as possible.

4.1.2.2 *Ucitool*

Apart from findings in the source code, we make another beneficial discovery that helps understanding UCI. The standard MK UWB kit ships with an undocumented tool called *ucitool*. This tool is written in Python and can be used on a local PC to communicate with the UWB kit's Plug and Play Boards, which can be connected to a PC. It serves as an alternative to the API source code written in C, which is intended to be installed on the UWB kit boards and used by the boards' host CPU. Moreover, one can use the *ucitool* with an additional helper binary to communicate with the *SR100T* on Android phones.

We find out that the *ucitool* contains in YAML files the UCI opcodes for almost every UCI message and their corresponding payload structure. There exist multiple YAML files with presumably different UCI specifications. However, we are only interested in the latest version, which is *1.10*, and therefore, we ignore the other versions. The YAML file with the latest UCI version is very useful for learning the UCI specification in combination with the source code files found in the *uwb-iot* subfolder. Before discovering this file, we did reverse engineering of the UWB kits' source code files in order to learn the UCI and HBCI specifications. The discovery makes some of our previous work obsolete. However, we find in the YAML file not every UCI opcode for the communication with every chip and no opcode for HBCI, although we found that information in the source code files. Furthermore, only by reverse engineering the kits' source code files, we can understand the UCI header's evaluation, which is not possible by examining the YAML file. In order to learn the UCI header's evaluation and the HBCI specification without these source code files, we would need to analyze the source code of the *ucitool* itself. This means that only parts of our previous effort are obsolete.

To identify the *ucitool*'s targeted chip, we compare some of the UCI payload structures in the YAML file with the creation or decoding of UCI payloads in the C source code files. We find that specifically UCI messages for the communication with the *SR100T* are declared in the YAML file.

Another beneficial aspect is that one can use the *ucitool* to communicate with the *SR100T* on Android phones, without using any service or library provided by Samsung in the user space. To use the *uctitool* to communicate with the *SR100T*, one needs to use a provided helper binary named *akash*. When we load this binary into Ghidra, we identify many functions of the UWB kits' source code, which also are included in the *halimpl* subfolder. This helps us understanding *akash* more quickly.

The binary *akash* needs to be pushed onto the phone and executed as a superuser (i.e. on a rooted phone). When executed, *akash* acts as a server to which the *ucitool* connects, which itself runs on the host PC. In addition, *akash* generates all chip management messages (HBCI) and establishes a connection to the driver. Furthermore, *akash* reads the firmware locally from a certain path on the Samsung phone, and this firmware is used by *akash* in the firmware transfer process. Also, *akash* reads local configuration files from the device. On the other side, the *ucitool* creates and processes UCI messages, which are forwarded by *akash* to/from the driver. The predefined UCI messages are generated by using the *ucitool*'s API. Additionally, it is possible to control multiple phones (or boards) with the *ucitool* at once. Hence it is possible for us to establish UWB session between phones by just using the *ucitool*.

We make a few modifications to the *ucitool* and *akash*, such that we can send our own messages as well as any byte we want, which we will later use for attacking the *SR100T*. We describe the modifications in Section 7.3.1.

The *ucitool* is fortunate for the rest of our work on multiple occasions. With our modifications, we can send any message we want to the *SR100T* and can build applications quickly by ourselves. Thereby, the *ucitool* and *akash* handle the connection with the chip. The *ucitool* further provides an API for selected messages, which we also can use in some cases. So on our Samsung phone, we can establish UWB sessions, send arbitrary bytes, and more, without using and manipulating Samsung's UWB services and libraries on the phone since the *ucitool* and akash are independent of these. Furthermore, the *ucitool* also can parse given bytes of a UCI message and print the decoded message, which is helpful for debugging.

4.2 COMMUNICATION PROTOCOLS UCI AND HBCI

The UWB entities in Samsung's user space authenticate other devices, exchange UWB session parameters with other devices, post-process UWB measurements, and more. The integrated *SR100T* does the UWB measurements. However, the firmware of NXP's UWB chip is encrypted. Hence, we cannot analyze the firmware implementation details. But there are multiple steps we can take to learn as much as possible about the *SR100T* and NXP's other UWB chips, which can help us when we analyze the chips' security. One significant step is to understand the communication with the chips.

One of our primary goals is to reverse engineer the not publicly accessible protocols UCI and HBCI to note them down. We can use our findings for our further work and also can provide these other researches for future work. There are multiple reasons why

understanding these protocols is important for us. The first major reason is that we need to understand them in order to get a better understanding of NXP's UWB chips and their states. Furthermore, knowing these protocols helps us attacking the chips. For example, we can try an attack by sending a UCI message with a 100-byte sized payload and declare at the same time in the same message's header a payload size of 50. If the chip allocates memory for a buffer based on the declared payload size but writes the whole payload into the buffer, we achieve a buffer overflow. We also can reverse the direction of attacks. Thereby, we can simulate attacks from the chip towards the user space of Samsung's UWB ecosystem by manipulating the chip's returned messages. Moreover, with knowledge about these protocols, we may learn messages that can be used to request information from the chip like logs.

We also can learn more about the firmware transfer process. This process of sending the firmware to NXP's UWB chips is called *firmware download* in different entities of Samsung's UWB ecosystem. However, the process is completely independent from a Internet download and is confusing. Beginning from here, we use the term *local firwmare download* for this firmware transfer process.

For understanding the protocols, we analyze our retrieved source code files from the kits precisely to learn how UCI and HBCI headers and payloads are created. We further analyze how these are processed as well as which opcodes exist. Additionally, we take the *ucitool*'s UCI declaration as a utility to help our understanding.

Furthermore, we establish UWB ranging sessions between our Samsung smartphone and the SmartTag+, which results in logs of UCI and HBCI messages (hex strings) that are exchanged with the *SR100T*. We use these logged messages as examples to help our understanding of UCI and HBCI. In the beginning, we notice that HBCI messages are not logged if they relate to the local firmware download process. A service that implements the Hardware Abstraction Layer (HAL) for UWB functionality is responsible for logging all UCI and HBCI messages. However, the service logs only the additional HBCI messages if a certain configuration value flag is set. The service reads this flag from a configuration file before downloading the firmware locally to the chip. Hence, we create a simple Frida script that manipulates the configuration value everytime it is read. Beginning from now, we always use this script and get all messages logged.

We find out that UCI is used to exchange UWB-related messages with the *SR100T*, and HBCI is used to manage the chip. Furthermore, HBCI may be related to UCI and may be even a part of the Fine Ranging (FiRa) Consortium's standard. However, we conclude that this is not the case because of the clear separation of the both protocols' usage. In addition, HBCI messages are not related to UWB functionality in any form. Moreover, UCI messages are used for operations like setting up a UWB ranging session on the chip or receiving the ranging data from the chip. In contrast, HBCI messages are used to manage the chip itself with operations such as sending the firmware to the chip or receiving the chip ID from the chip. Our conclusion will be further justified for the reader, when we describe UCI and HBCI in the following sections.

Next, we explain the protocol UCI, and afterwards we explain HBCI. For easier reading, we give a detailed overview for both protocols by defining every HBCI message in Appendix A.10 and every UCI message in Appendix A.11. If we can extract the corresponding message's payload, we include the payload structure as well. Furthermore,

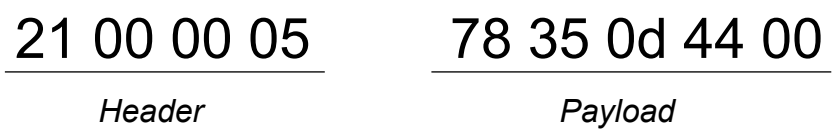

Figure 4: Hex string of an example UCI message.

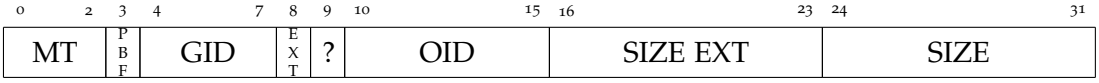

Figure 5: UCI header.

we implement a UCI and HBCI Wireshark[3] dissector based on the extracted protocol definitions. We introduce the dissector in Chapter 7.

4.2.1 *UCI*

UCI has a defined set of messages, consisting of a four-byte header and an optional variable-sized payload. The headers are handled the same by all of NXP's UWB chips and most payload structures as well. We encounter one payload structure that differs between the UWB chips and also between the firmware version of the same UWB chip, which is the returned ranging data. The ranging data is formatted in the *SR100T*'s new firmware versions like the ranging data returned by the *SR150*, while it is formatted differently in older versions.

To facilitate the understanding of the UCI protocol, we refer while explaining to an example UCI message we retrieved from the logged messages, which we show in Figure 4. The example message is a command to the chip with a five-byte sized payload. It commands the chip to initialize a session with the session ID 0x440d3578 (because of little endianness). Additionally, the session type is zero, which stands for a ranging session. Note that the ranging process starts only after an additional *start ranging* command.

4.2.1.1 *UCI Header*

We show the header's structure in Figure 5. It is always four bytes long, and each bit always has the same meaning. The first byte defines three values at once: *Message Type (MT)*, *Packet Boundary Flag (PBF)*, and *Group Identifier (GID)*. Note that we read from left to right if we point out the X-th bit of a byte.

MT makes up the first three bits of the header's first byte. It defines the message's type, which can be either *COMMAND*, *RESPONSE*, *Notification (NTF)*, or *DATA*. If *MT* is from type *COMMAND*, meaning *MT* = 1, then the UCI packet's sender commands the receiver to perform an action such as starting a UWB session or applying the configuration included in the payload. In our example UCI message, the first byte is 0x21. Therefore, the header declares a command since the first three bits are equal to one. Moreover, for each UCI packet a receiver gets that is a *COMMAND*, it will return a direct *RESPONSE* UCI packet (*MT* = 2), which at the same time is an acknowledgment. In the payload,

3 https://www.wireshark.org/

this packet includes either a response code notifying the commander of the command's status, or it contains the requested information such as the chip's information. Another responding packet type is a *NTF* packet (*MT* = 3), which abbreviation's meaning is not resolvable like HBCI, and we assume it likely means "Notification". This UCI packet type is an asynchronous response/notification and is not the related response to a specific command, but is sent as a notification to events that happen in the chip. For example, after a UWB ranging session is set up on the chip, the ranging data of continuous UWB measurements is sent as an *NTF* message periodically by the chip to the host. Furthermore, the last message type is *DATA* (*MT* = 0). Messages of this type are used to transfer raw data between UWB chips after a data exchange session is established.

The second value that can be retrieved from the header's first byte is the *PBF*, which is declared by the fourth bit. The *PBF* is a flag and is defined by one bit. If it is set (bit value = 1), then the flag indicates that the whole UCI packet is fragmented and the current packet is a fragment. Moreover, a count of fragments is not defined in the UCI packet or a previous message. However, each fragment that is sent to the receiver has this flag set, and for the last fragment, this flag is not set. The receiver can then reassemble the whole packet after receiving the last fragment. Furthermore, UCI packets are generally fragmented in the UWB kits if the payload size exceeds 255 bytes (excluding header), which may also be the maximum allowed payload size of UCI messages. Since the *PBF* flag is not set in our example UCI message, we know without looking at previous UCI messages that this is either an unfragmented message or the last message from a set of fragmented messages.

The *GID* is the third value that the header's first byte holds, and it makes up the byte's second nibble. This value defines the UCI message's group. Each group contains a selected set of messages, which are defined by the header's second byte that is the *Opcode Identifier (OID)*, which we explain later in this section. Additionally with *MT* and *OID*, the *GID* defines uniquely a UCI packet. Furthermore, there exist multiple groups indicated by the *GID*, which names also explain the abstract meaning of messages from the group. These are *CORE* = 0, *SESSION_MANAGE* = 1, *RANGE_MANAGE* = 2, *DATA_CONTROL* = 3 = 9, *PROPRIETARY_SE* = 10, *TEST* = 13, *PROPRIETARY* = 14, and *INTERNAL* = 15. In our example, we have *SESSION_MANAGE* as the *GID*, since the second nibble is one. We further point out that we encounter the value three or nine for *DATA_CONTROL* at different points in our analysis. Moreover, we assume that both proprietary groups and the *INTERNAL* group contain a set of messages, which are proprietary extensions to the official UCI standard. This allows chip developers like NXP to extend the UCI standard with additional messages. However, since we have not the specification of UCI and cannot find a clarification in any of our reversed targets, it stays an assumption.

We point out that the first byte's values can be read out quickly without calculation. The first nibble defines *MT* and *PBF*. So one can retrieve the value of *MT* by using the first nibble, whereby 0-1 = *DATA*, 2-3 = *COMMAND*, 4-5 = *RESPONSE*, and 6-7 = *NTF*. Furthermore, if *MT* is odd, then the *PBF* flag is set. Additionally, the second nibble defines the *GID*.

The first bit of the header's second byte is *Extended (EXT)*, which is a flag that signalizes if the payload has an extended length, meaning if it is greater than 255 bytes. This is contrary to our previous finding that the maximum allowed payload size of UWB may be 255 bytes. Our example UCI message has the flag not set. Therefore, it has no extended

length. Moreover, for the second byte's second bit, we do not find the meaning anywhere. We assume it is reserved for future use.

The last six bits of the header's second byte are the *OID*, which defines the corresponding group's message, whereby the group is identified by the *GID*. In our example message, we have a *OID* of zero. By looking either into *uci_defs.h* of the UWB kits' source code files or into the *ucitool*'s YAML file, we can resolve the meaning. Since the *GID* is *SESSION_MANAGE* = 1, the *OID* with the value zero stands for *session initialization*.

The header's third and fourth byte display the UCI packet's payload size. If the *EXT* flag is not set, then only the fourth byte displays the payload size. This is the case in our example, where we have a payload size of five. For a payload size less than or equal to 255, this flag will be not set and the third byte is ignored. Otherwise, if the *EXT* flag is set, then the third byte displays the payload size together with the header's fourth byte, whereby the fourth byte contains the most significant bits (i.e. little-endianness is used).

4.2.1.2 *UCI Payload*

While the UCI header is always four bytes sized and each byte in the header has the same meaning for each different UCI message, this is not the case for the payload of UCI messages.

The identification of a UCI message and its payload structure is defined uniquely by the *MT*, *GID*, and *OID*. The identification is simple: With the *GID* we can retrieve the list of its corresponding *OIDs*, in which we find the entry with the *OID*'s value. Then, with the *MT* we know the type (e.g. command). Moreover, even if *GID* and *OID* are the same, for different *MTs* the payload structures are most often different as well. Also, the values' endianness in the payload are little endian.

While each payload has a defined structure, it does not mean that it always has the same size. Some payloads allow a variable length, which is indicated by specific bytes in the payload. For example, the command for applying the app configuration of a ranging session (*MT* = 1, *GID* = 1, *OID* = 3) defines a variable number of different configuration values to apply in the fifth byte. Hereby, each of these configuration values has a structure also known as Type-Length-Value (TLV). The structure consists of a configuration identifier byte, followed by a byte indicating the configuration value's variable length, and the configuration value itself. As a result, different sized payloads are possible. In addition, decoding these values is only possible by starting from the payload's beginning and looping through the configuration values.

In our example, we know based on the values *MT* = 1, *GID* = 1, and *OID* = 0, which UCI message is used and what the payload structure looks like. Based on our extracted UCI specification (see Appendix A.11), we know that the first four bytes display the session ID, which is 0x440d3578. Furthermore, we know that the fifth byte indicates the session type, and the byte is zero, which indicates the session type of a ranging session.

4.2.2 *HBCI*

Unfortunately, we cannot use the *ucitool*'s YAML file that declares only UCI messages for understanding HBCI. Nevertheless, by analyzing the *ucitool*'s source code, and the source code of NXP's API, we learn in a great measure how HBCI works. HBCI is purely used for chip management such as sending the encrypted chip firmware to the *SR100T*

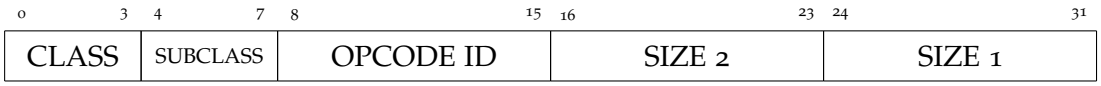

Figure 6: HBCI header.

or *SR150*. The message set also is much smaller in comparison to UCI. Additionally, in contrast to UCI, the header and the payload are sent in separate messages. Furthermore, for some messages are acknowledgements returned from the receiver. Despite access to the source code, we cannot tell with certainty what conditions must be fulfilled, such that an acknowledgement is sent back from the receiver. Yet, by analyzing logged messages between the host and chip, we observe that an acknowledgement is sent back from the receiver in three cases. In the first case, the receiver sends back an acknowledgement when the header is received, and a payload size greater than zero is defined. When the header defines a payload size greater than zero, the receiver knows that the sender will send a message with the payload data afterwards. We assume that the receiver acknowledges the header to tell the sender that the receiver is ready to receive the payload. In addition to headers that indicate a payload size greater than zero, the receiver acknowledges every payload message in the second case. In the third case, a receiver sends back an acknowledgement when a command declared in a header is received. This is independent of the header's defined payload size and if the sender sends a payload afterwards. We assume the receiver tells the sender with the acknowledgement that the command was executed successfully.

Based on the purpose of HBCI messages, we assume that HBCI also is used to manage other chips from NXP, which are not related to the UWB chips. The whole specification is not related to UWB and in theory, each message also can be used to communicate with other chips.

4.2.2.1 *HBCI Header*

In Figure 6, we show the structure of an HBCI header. Like in UCI, the header is also four bytes long, but the structure is simpler. Here, the first byte's first nibble declares the message's class. Five classes exist, whereby the first class *General* is intended for all messages that do not fit into the other three classes. This class is displayed with the value zero. The second class is indicated by the value one. The classe is named *Test*, and it contains debug-related opcodes. By slightly modifying the communication with the *SR100T*, we detect that this class is not supported by the *SR100T*. Next, the third class is *Patch_ROM*, and it is displayed with the value two. We saw no message of this type used in real communication. Moreover, we assume that messages of this type are used to update the chip's Read-Only Memory (ROM), which includes the bootloader and certificates. *HIF_IMAGE* is the third class and is displayed with the value five. Its messages are used to locally download the firmware to the chip. The fourth class is *IM4_Image* and is displayed with the value six. *IM4* is a feature that controls applet migration and operating system update processes [27]. We assume it is unrelated to the firmware running on the UWB chips. This class is also not supported by the *SR100T*.

Furthermore, the first byte's second nibble defines the subclasses with self-explanatory names. Thereby, we have *Query* = 1, *Answer* = 2, *Command* = 3, and *Ack* = 4.

The second byte indicates the opcode identifier and is different from the ones of UCI. For some messages, the opcode identifier is used as a direct status indicator in response, making a payload superfluous.

The third and fourth bytes always signalize the payload size, whereby the bytes are represented in little-endian. So the fourth byte makes up the most significant bits. For headers with a payload size greater than zero, the payload with the defined size follows in an additional message, which we describe next.

4.2.2.2 *HBCI Payload*

We find no defined structure for HBCI payloads. The payloads we retrieved from the real communication consist of data and an additional byte, which is the data's Longitudinal Redundancy Check (LRC). Furthermore, there exist only a few messages with a payload. We try to identify how payloads are interpreted in the NXP UWB API's and *ucitool*'s source code. We further analyze Samsung's UWB libraries. However, we do not find decoders for most payloads and can only guess the data meaning based on the opcode identifier names. Moreover, we only find for two HBCI payloads the decoding of selected bytes, which is done in one of Samsung's UWB libraries. However, not being able to decode these payloads is not severe since it probably will not hinder us in any form in the duration of the thesis. This makes only the presented HBCI specification not entirely complete. In Appendix A.10, we include this specification.

Additionally, we find that there exist two additional HBCI messages, which are not referenced anywhere in the API source code or the *ucitool*, but only in one of Samsung's UWB-related libraries. These messages are the query and response of the device life-cycle data. In conclusion, this means there may be additional unknown HBCI messages, even if we have not found them in our logs or in the source code.

4.3 SR100T'S LOCAL FIRMWARE DOWNLOAD PROCESS

By learning the HBCI specification and using logs as examples, we also can understand the local firmware download process for transferring the encrypted and signed firmware. In Figure 7, we show the steps of the *SR100T*'s local firmware download process. For the *SR150* we assume a very similar or equal order of messages. However, the *SR040*'s local firmware download process is different. From a documentation file in the UWB kits, we learn that is named *SWUP*, and it has a different message set. We delegate an analysis of this firmware transfer process to future work.

The firmware is downloaded locally to the *SR100T* in the *HBCI mode*, which we describe in Section 4.5. Additionally, it is already stored locally on the device, and nothing is downloaded from a web server. The firmware comes pre-installed on our Samsung phone and only updates when the phone's operating system is updated.

To start the local firmware download process and to make the *SR100T* ready, the service that handles the HBCI communication with the chip sends a command to start the *HIF image* mode. In this mode, the firmware is locally downloaded. When the chip accepts the command and goes into this mode, it sends back an acknowledgement containing a positive status code. Now, the service can optionally ask the chip again for the chip's status, which should be the *HIF image* mode.

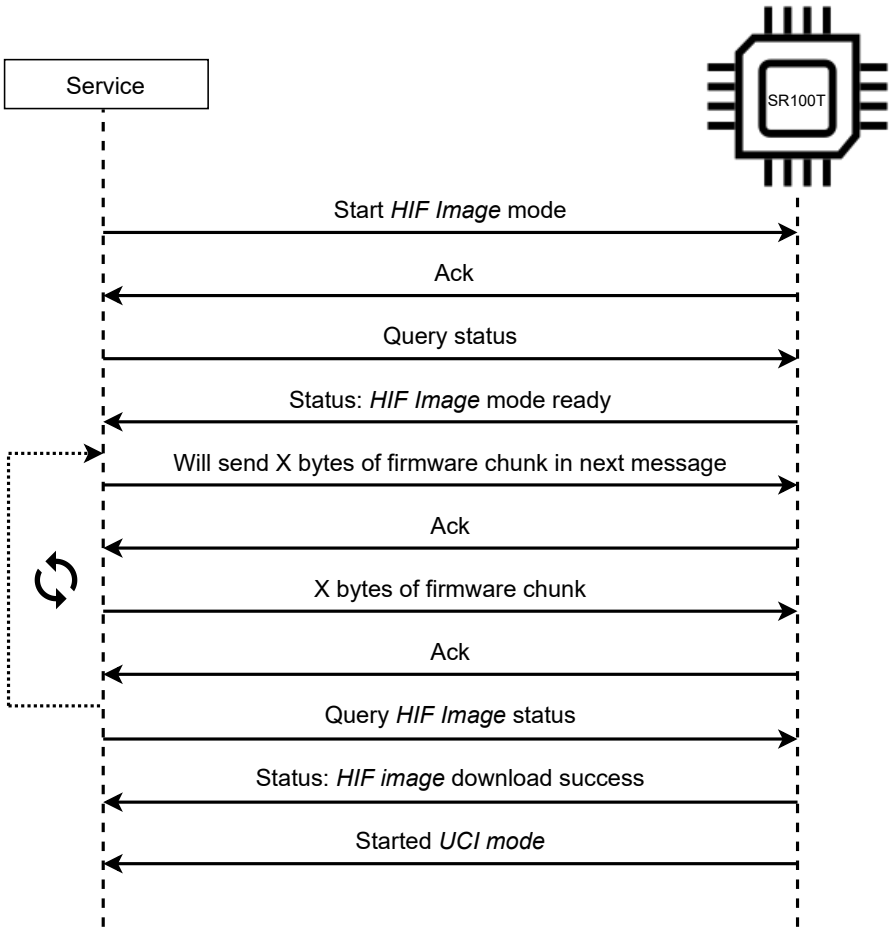

Figure 7: SR100T's local firmware download process.

After these steps, the service sends the firmware in chunks to the chip using two messages, whereby the first message declares the chunk's size in an HBCI header and the second message contains the raw chunk data itself. For each message, the chip sends back an acknowledgement. In the end, the service asks the chip if the firmware was downloaded successfully. We assume that this query simultaneously signals that the transfer is finished since no message declares the firmware's whole size before. If the firmware was downloaded successfully, the chip responds with a positive status message and directly goes into *UCI mode*, which we describe in Section 4.5.

4.4 SR100T'S DRIVER

The *SR100T*'s driver is open-source, and we find the current version for our phone model on Samsung's open-source kernel release website[4]. After downloading the open-source kernel, we locate four source code files that implement the driver. We analyze these files and present our key findings next.

4 https://opensource.samsung.com/uploadList?menuItem=mobile&classification1=mobile_phone

The driver is developed by NXP, written in C, and is a character device driver[5]. We learn from the source code that a device handle will be created under */dev/sr100* after the driver is started. This is also the case on our Samsung phone. Additionally, we find on our phone that the Linux user *uwb* is the owner of this file. Moreover, only the Linux user and group *uwb* have read-write permissions on the file. Other users have no permission for any operation. So on our phone, only processes can operate on the device handle that run under the same Linux user or group ID or that run as *root*.

When the driver is initialized, it also creates a Linux *proc* entry under */proc/uwblog*, which is owned by the Linux user *root* but can be read by any user. This entry enables and handles read access to debug and error messages, which the driver writes if a debug flag is set. Fortunately, this flag is set in the driver's source code, and by reading the contents of the *proc* entry on our smartphone, we can see that Samsung did not change it. If we attack the driver, these logs can be useful for us.

We learn from the source code that the driver implements four file operations: *open*, *write*, *read*, and *ioctl*. After a user space process opens with *open* a handle to */dev/sr100*, it can use the other three file operations to send data to the driver, receive data from the driver, and execute specific commands.

There are several commands that the driver accepts over *ioctl*. The most interesting command enables and disables the local firmware download mode by setting a flag. We analyze the flag's evaluation with the source code and logs of the driver. Thereby, we learn the flag is set when the Serial Peripheral Interface (SPI) connection to the chip is (re-) enabled. Furthermore, the driver interprets all messages as HBCI messages in the local firmware download mode. When the flag is unset, the driver interprets all messages as UCI messages. The different interpretations of messages based on a flag may indicate two states for the *SR100T*, which we further analyze when we build the state machine in the next section. The other commands that can be passed to *ioctl* are not essential for us.

In addition, the driver's methods *write* and *read* do a check for a maximum allowed length of the data to write or read. The limit is 4200 bytes. Larger writes or reads are discarded. This limitation is essential to know when we try to attack the *SR100T*, since the payload of UCI and HBCI messages can be sized up to 65535 bytes in theory. Moreover, if the messages are not discarded, these methods only forward HBCI and UCI messages between host and chip.

4.5 SR100T'S STATE MACHINE

To better understand the *SR100T*, it is helpful to build its state machine. The state machine helps us identifying the chip's states and what conditions cause a transfer between states. We can use this knowledge when we attack the chip. Furthermore, we do not want to build a complete detailled state machine but focus on the essential states relevant to us.

We build the *SR100T*'s state machine based on our previous analysis and findings of UCI, HBCI, the source code in the UWB kits, and the *SR100T*'s driver. In addition, we do extra tests, which we describe next. In Figure 8, we show the resulting state machine.

To build the state machine, we further analyze the driver's logs extracted from our phone and normal logs extracted with *Logcat*. Because we do not detect in the regular workflow crashes or timeouts and we want to ascertain our knowledge, we manually

5 A character device is a device to which single bytes can be sent to and read from.

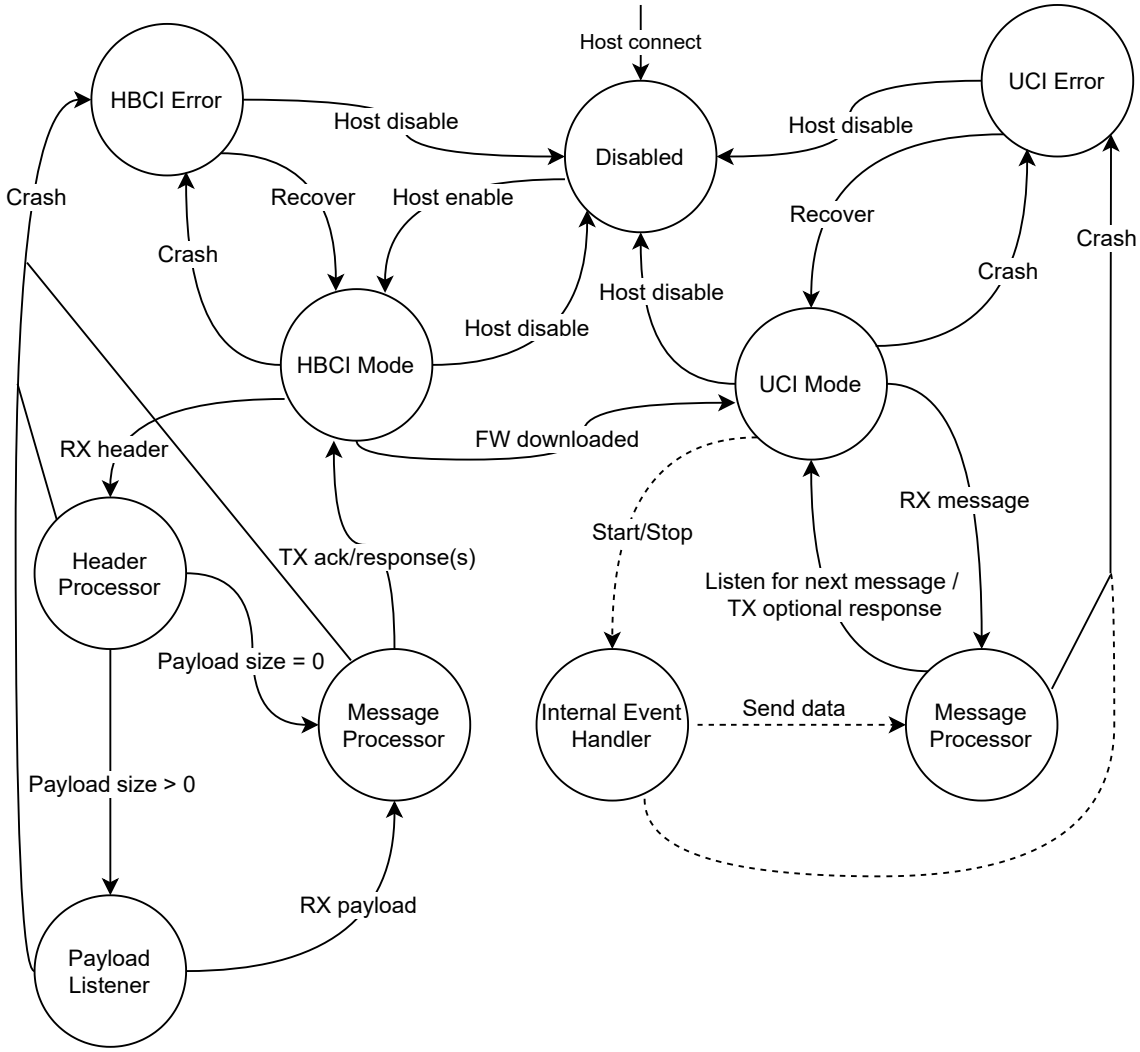

Figure 8: SR100T's state machine.

trigger simple crashes, which we describe in Section 8.3.2.2 in more detail. This helps us learn the state transitions in case of crashes and timeouts. Additionally, in the regular workflow the user space processes on our phone do not deviate from the order of messages and always send expected messages to the chip. Therefore, we want to verify that we build the state machine of the chip itself, and the user space processes do not determine the state machine. In order to do this, we manipulate HBCI and UCI messages with *Frida*. For example, we send UCI messages in the *HBCI mode* state by changing HBCI messages to valid UCI messages before they are sent. Then, we review the response from the chip. We also inject messages. For example, we inject messages after the chip has crashed in the *UCI mode* state and before the communication with the chip is reenabled. Thereby, we send UCI messages to test the chip's responsiveness and check if the only way out of the error state is the re-establishment of the communication with the chip.

Furthermore, when the host writes HBCI messages to the chip in the regular workflow, it always first sends the four-byte header, which the chip then processes. When a payload exists, the receiver acknowledges the header, and if no error happens after the sent header, then the host sends the payload. The same applies to the opposite direction, meaning

for messages that the chip sends. To test if it is possible to send the header and payload at once to the chip, we manipulate selected HBCI messages and generate messages consisting of header and payload without separation. We test successfully that the chip interprets only the header, and the rest of the message is ignored by the chip, even if the payload size greater than zero is defined in the header. This means that the HBCI payload needs always to be sent in a separate message after a received acknowledgement. Additionally, we test the case when the chip sends HBCI messages to the host. We test successfully that the chip always sends the header first. Only after an acknowledgement, it sends the payload. Next, we describe the state machine.

When the phone is started and the driver is initialized, an SPI connection is established with the *SR100T*. This is handled by the driver's function *sr100_dev_init*. During the connection setup, the driver sets the SPI connection's Chip Enable (CE) line to zero, which in effect temporarily disables the data exchange connection to the *SR100T*. This means that after the connection establishment, the chip is in the initial state of *Disabled*.

The CE line is set to one to start the communication with the chip. Then, the chip state transfers to *HBCI mode*. This mode is also named *firmware download mode* in the source code, which fits its primary cause, namely to send the firmware to the *SR100T*. Yet, there exist many HBCI messages that are unrelated to the local firmware download, so we name it *HBCI mode*. Moreover, only HBCI messages are accepted by the chip in this state, because when we inject a UCI in this state with a different header than any HBCI message could have, it is discarded. Thereby, the chip responds with a message that declares the error, which relates to unknown header bytes.

After the raw firmware data is transferred, a final HBCI command must be sent to the chip. This command requests the firmware download status from the chip, and we conclude it signalizes the end of the transfer simultaneously. The *SR100T* directly responds to this command, and the response signalizes a positive status code if the chip has accepted the transferred firmware. After the response, the chip immediately sends an additional *NTF* UCI message declaring the state transition to *UCI mode*. There is no additional command needed, and the state transition happens instantly. In this mode, all UWB-related messages are exchanged with the chip, and only here we can use any UWB functionality. Furthermore, just as UCI messages are not recognized in the *HBCI mode*, HBCI messages are not recognized in the *UCI mode*. Moreover, we assume that when the chip goes into the *UCI mode*, it starts an asynchronous event handler. When an event happens that requires notifying the host, the event handler passes the data to the *Message Processor*, which creates the corresponding UCI message from the data. For example, this happens for a finished UWB measurement.

When we trigger crashes, we encounter two types of crashes in the *UCI mode*. When the first crash type occurs, the *SR100T* does not respond anymore to our commands. The only way out of this mode is to disable and reenable the connection, which starts the *HBCI mode* again. When the other crash type happens, the *SR100T* sends a notification to the host and in addition, a 52 byte sized debug message. Afterwards, the chip stays responsive, and we can continue to send UCI messages and get the responses. Moreover, normally, the UWB services disable and reenable the communication for both types of crashes, even if the *SR100T* stays responsive.

Unfortunately, we cannot trigger crashes in the *HBCI mode*. However, we find in the UWB kits' source code an HBCI status opcode named *phHbci_General_Ans_HBCI_Fail*.

We assume the chip uses it to notify the host of an error state. Hence, we conclude that the chip stays responsive when this notification is sent like in the *UCI mode*. Furthermore, we also assume that another crash type can happen, whereby the chip does not respond anymore, and the communication needs to be disabled and reenabled.

5 ENTITIES OF SAMSUNG'S UWB ECOSYSTEM

In Chapter 4, we learned the *SR100T*'s states and the communication with it. We also learned the driver's role. However, at this point, we do not know how the *SR100T* is used in the user space of Samsung's Ultra-Wideband (UWB) ecosystem. We also mostly do not know which entities exist and how these are interconnected. Therefore, we want to find out which services communicate with the *SR100T* and expose UWB functionality to other entities, which we also want to learn. We further aim to understand all entities' roles and interconnection in Samsung's UWB ecosystem.

In this chapter, we explain how the UWB is integrated into the user space in Samsung phones by analyzing the apps, services, and libraries that provide or use the UWB functionality in Samsung phones. In addition, we examine in which way Samsung's SmartTag+ is integrated into the ecosystem, how it is controlled by the phone, and on a high level how the UWB distance and direction measurement (ranging) between phone and SmartTag+ is done.

5.1 LABORATORY

We use multiple tools for reverse engineering UWB-related apps and other binaries, which mainly are the three tools we name next. We use Frida[1] for dynamic instrumentation of functions. We also use Ghidra[2] for disassembling as well as decompiling binaries. Additionally, we use JADX[3] for decompiling Android (system) apps in .apk format and other .jar library files.

We further use two Samsung Galaxy S21 Ultra (SM-G998B) phones with different images and the Samsung Galaxy SmartTag+ (EI-T7300M) for our analysis. In Table 2, we show the basic information about the phones' images. In addition, we enable superuser access on our Samsung phones, meaning we "root" it. In Appendix A.1, we show the summarized steps we take to enable superuser access. We also enable *Verbose vendor logging* and *Samsung verbose debug logging* in the developer options. This results in additional log messages that help us in our understanding.

5.2 ANALYSIS PROCEDURE

We take several steps to learn the ecosystem's entities. First, we trigger UWB functionality on our Samsung phone by doing ranging with the SmartTag+, which can be started over a plugin in the SmartThings app. Additionally, we trigger it using *Nearby Share*. Meanwhile, we take logs and evaluate these after, which helps us learn unique strings that are most likely only contained in a specific app, executable, or library. Then, we semi-automated search for UWB-related files on our phone, which include apps, executables,

1 https://github.com/frida/frida
2 https://github.com/NationalSecurityAgency/ghidra
3 https://github.com/skylot/jadx

PHONE NUMBER	INFORMATION ID	VALUE
1	Phone model number	SM-G998B/DS
	Phone Android version	11
	Phone security patch level	2021-08-01
	Phone firmware version	G998BXXU3AUGM
2	Phone model number	SM-G998B/DS
	Phone Android version	11
	Phone security patch level	2021-11-01
	Phone firmware version	G998BXXS3AUJ7

Table 2: Information about the phones' images

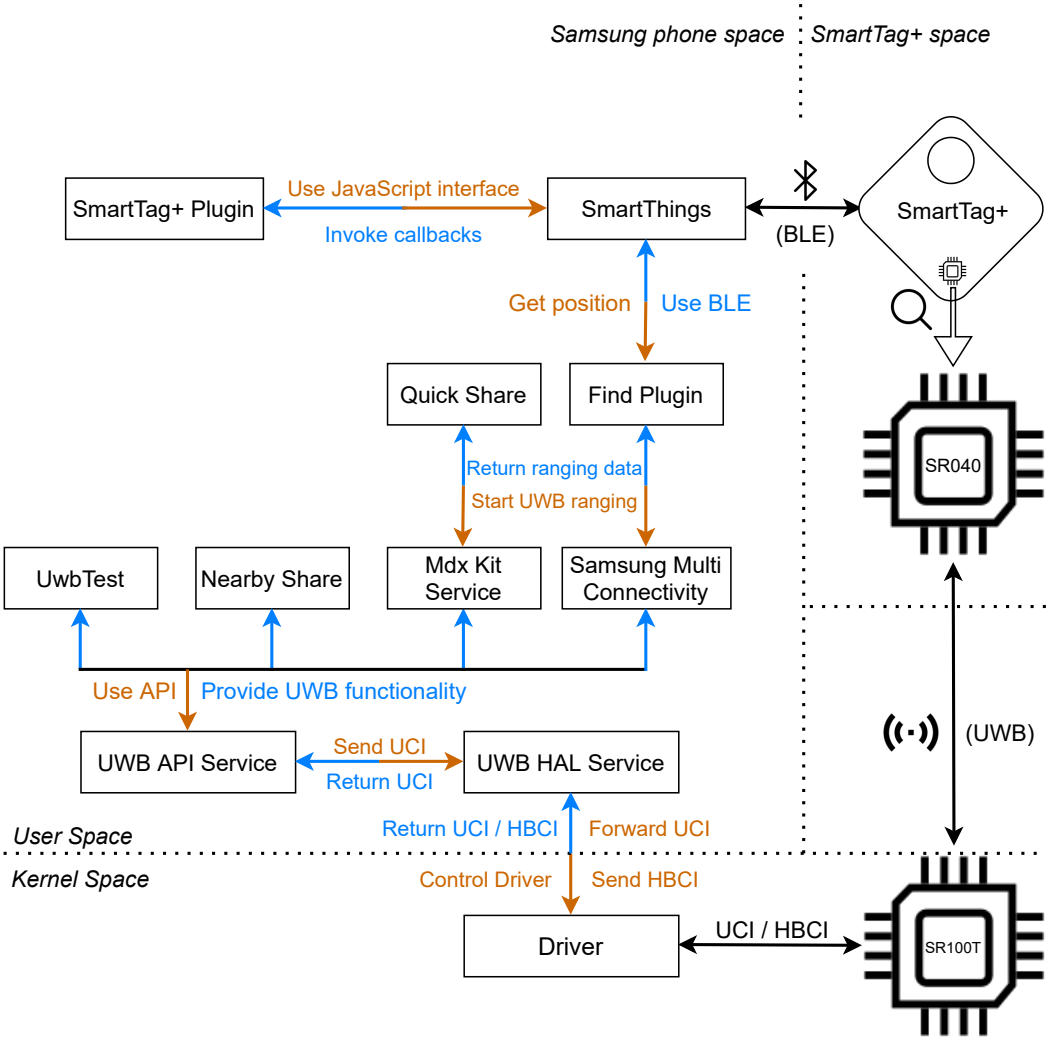

Figure 9: Overview of Samsung's UWB ecosystem.

and configuration files. In our search, we look for filenames containing a specific string in the name and files containing a specific string in the contents. Every found file is extracted from the phone.

Next, we examine which files every process opens and look for files we have already found. Thereby, we determine the UWB-related processes and which files are related to these processes, including the executable name of the process itself and the imported libraries. Previously unknown files are also extracted from the phone if we do not know the file's meaning based on the name. Furthermore, we match log contents to the corresponding files or processes.

Subsequently, we begin to analyze the functionality of the UWB-related processes. Thereby, based on the order of the workflow that can be derived from the evaluated logs, we analyze or reverse engineer each extracted file with a focus on the process executables and their libraries. It is helpful for our analysis that most apps, executables, and libraries use no form of obfuscation.

In Figure 9, we show an overview of the ecosystem, which we created with our analysis results. In the following sections, we elaborate on our findings.

5.3 SAMSUNG'S UWB API

In Figure 10, we show the entities that build Samsung's UWB Application Programming Interface (API) and the apps that use it, which all are system apps. Two processes form the UWB API on our Samsung phone, which both run under the Linux user *uwb* like the driver. Both also run in the background after the device starts. The first process is *vendor.samsung.hardware.uwb@1.0-service*, which is started through the same-named ARM executable. It implements the Hardware Abstraction Layer (HAL) for UWB on our phone, and it communicates with the *SR100T*'s driver. Hence, we name it the *UWB HAL service*. Furthermore, it is responsible for the *SR100T*'s setup including the local firmware download to the chip. All Host-Based Command/Control Interface (HBCI) messages that are sent to and received from the chip are created and processed here. Additionally, it forwards all Ultra-Wideband Command Interface (UCI) messages, which the second process sends, which we describe after the next paragraph. In addition, the bytes of most HBCI and all UCI messages are logged by this service. Here, we make the same manipulation as described in Section 4.2 to log all messages in our following tests.

Using Ghidra, we identify that one library imported by the *UWB HAL service* contains many functions, which also are included in the *halimpl* subfolder of the Mobile Knowledge (MK) UWB kits' source code. Therefore, the library uses NXP's source code included in the MK UWB kits. We use this source code to analyze the library more quickly. The library is named *uwb_uci.helios.so*. We further compare this library to *akash* of the *ucitool*. Both have very high similarities. Examined functions and strings are either identical or have slight differences. However, *akash* contains the additional server component, which this library does not. Therefore, we conclude *uwb_uci.helios.so* is statically imported by *akash* as it is dynamically imported by the *UWB HAL service*, and both underlying executables are different. Additionally, we assume the slight differences in identified methods relate to different library versions.

The second process is *com.samsung.android.uwb* and implements the actual API that is accessible by other apps. Hence, we name it *UWB API service*. Behind this process

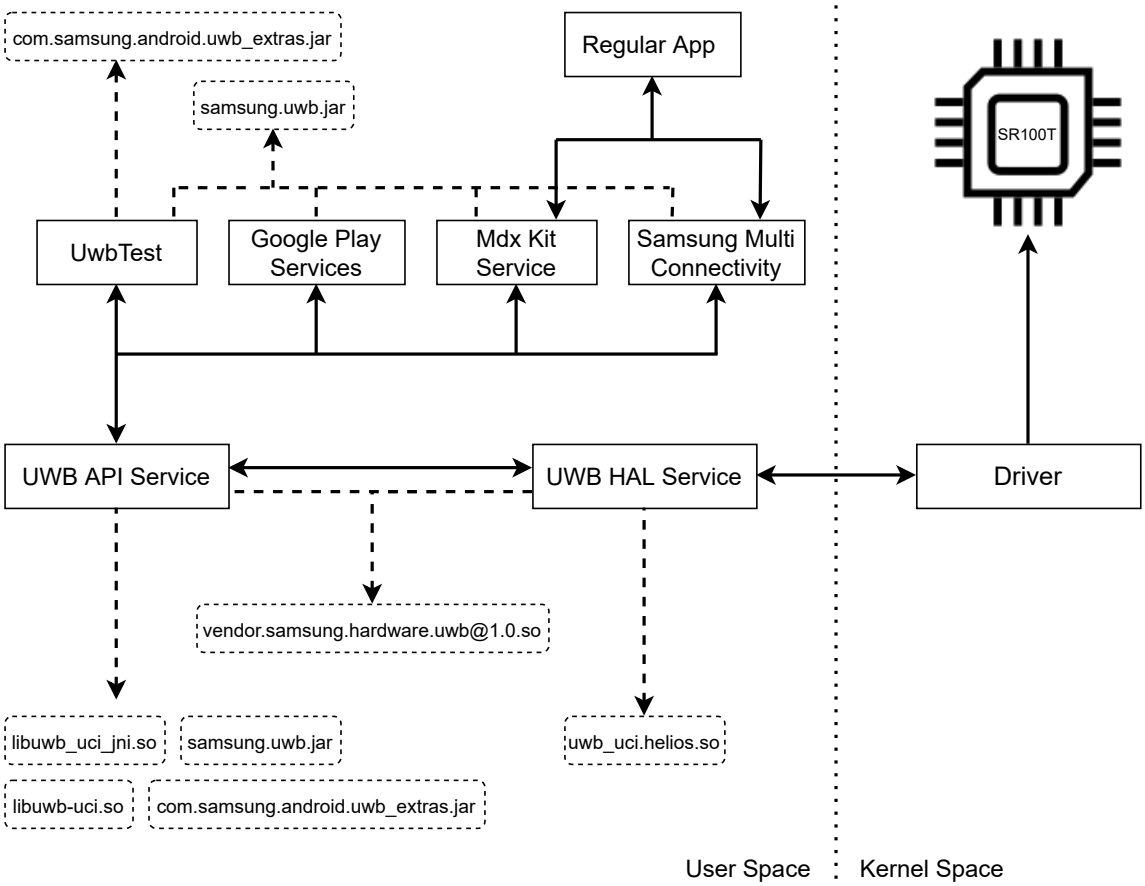

Figure 10: UWB API services and the system apps using it. Dashed line means import, and *Regular App* is a collective term for multiple apps.

is the app *UwbUci.apk*, which is a system app. The API checks for some method calls that implement privileged actions if the calling app declares the Android permission *WRITE_SECURE_SETTINGS*. This permission can only be granted to system apps, which implicates that only selected system apps can call them. However, we do not find permission checks for methods that are independent of the privileged methods. Thus, we conclude that the *UWB API service* can be used by third-party apps in the near future directly. Currently, we do not find third-party apps using it, and we also do not find an API for third-party apps. Furthermore, the communication with the *UWB HAL service* works with Binder[4] and is implemented in imported native libraries.

UwbUci.apk imports a library named *libuwb_uci_jni.so*, which implements the interface between the app and native code. This library uses another library named *libuwb-uci.so*, which provides over exports an API for creating predefined UCI messages and returning selected UCI responses. Here are all UCI messages created and processed that are used to communicate with the *SR100T*. Moreover, one additional export exists, which enables *UwbUci.apk* to send a raw UCI message through the library, whereby the own bytes can be chosen, and no validity checks are done for the bytes. Additionally, all UCI messages generated by the library *libuwb-uci.so* are forwarded to the *UWB HAL service* using Binder.

4 Android Binder is an Inter-Process Communication (IPC) mechanism provided by the Android operating system [49].

NAME	PACKAGE ID	IMPORTS
UwbTest	com.sec.android.app.uwbtest	samsung.uwb.jar & com.samsung.android.uwb_extras.jar
Samsung Multi Connectivity	com.samsung.android.mcfserver	samsung.uwb.jar
Mdx Kit Service	com.samsung.android.mdx.kit	samsung.uwb.jar
Google Play services	com.google.android.gms	samsung.uwb.jar

Table 3: Apps that use the *UWB API service*.

Using Ghidra again, we identify that *libuwb-uci.so* has a high similarity to parts of the MK UWB kits' source code. Many functions of this library are included in the *uci-core* subfolder of *libs*. Again, we conclude that NXP's source code is used, and this helps us understanding the library more quickly. Moreover, we conclude that between *libuwb_uci_jni.so* and *libuwb-uci.so* lies the separation of code without NXP connection and the usage of code developed by NXP, which includes the code of *uwb_uci.helios.so*. Beginning from this library, through the *UWB HAL service* and up to the chip, all functions directly involved for the communication with the *SR100T* are developed by NXP. Fortunately for us, this is the NXP source code from the UWB kits with slight variations, which likely come from different versions. Furthermore, Samsung still developed code after the separation. However, this code is not related to the communication with the chip but primarily provides the foundation of the *UWB HAL service*.

UwbUci.apk and thereby the *UWB API service* expose many functions, and an app can access these using Binder. These functions provide abstract UWB functionality to apps, which do not need to handle the specifics. Furthermore, the functions themselves shift all UCI message creation and processing to the library *libuwb-uci.so*. In the end, most of the *UwbUci.apk*'s functions just forward commands to the native libraries. Additionally, one provided function named *sendRawUci* enables sending raw bytes directly to the chip, which are not checked for validity, except of the length. However, this method checks the calling permission, and currently only system apps can use it.

5.4 APPS USING THE UWB API

We encounter multiple apps that use the *UWB API service*, which we list in Table 3. All of them are system apps and import *samsung.uwb.jar*, which is a framework library included on the phone. In addition, the app *UwbTest* also uses *com.samsung.android.uwb_extras.jar*, which is also a framework library. Both of these libraries implement the connection to the *UWB API service* and provide a simple API, which the system apps can use for UWB functionality. Furthermore, both libraries can only be used if the calling app has the Android permissions *ACCESS_FINE_LOCATION* and *ACCESS_BACKGROUND_LOCATION*.

5.4.1 *UWB Test App*

While our search on the phone for UWB-related files, we found the *UwbTest* system app, which is developed by Samsung. By decompiling the app and looking at the Android manifest file, we learn that we can start it by typing **#8928378#* into the phone app.

The app lets a user test the UWB functionality on the phone. Moreover, with the app a user can start UWB ranging sessions with other phones that use this app. It is also possible to establish a ranging session with other devices that do not use this app for ranging. For example, one phone uses the app and another phone uses a *ucitool* script, which works independently of Samsung's user space. We also developed a corresponding *ucitool* script, which we describe in Section 7.3.2.

This app is helpful for us because it sets up a connection with the *UWB API service* using both UWB jar libraries. Furthermore, it does not contain much code unrelated to UWB usage and is easy to reverse engineer. Moreover, we can use the setup methods with a simple Frida script instead of developing our own app. The result is a quick initialized service connection, and we can use this connection in our Frida script to call the *UWB API service*'s methods. This is fortunate for two reasons. First, we can send from here any byte we want to the *SR100T* by calling the function *sendRawUci* in our Frida script. However, with the *ucitool* we have already an easier and better way to communicate with the chip. Second and more important, starting from here we can attack the *UWB API service* and the *UWB HAL service*. Thereby, we can simulate attacks that come from system apps now, and attacks that come from third-party apps in the future.

5.4.2 *Samsung Multi Connectivity*

The next app that uses the *UWB API service* is *Samsung Multi Connectivity*. It is a system app and runs as a service in the background. UWB ranging sessions with the SmartTag+ are established through this service, which then interacts with the *UWB API service* using *samsung.uwb.jar*. The ranging data is then returned to the calling app of this service. We will elaborate on the SmartTag+ and the app that uses this service in Section 5.5.1.

5.4.3 *Apps for Sharing Files*

The app *Google Play services* includes the *Nearby Share* functionality and imports the library *samsung.uwb.jar*. When a user selects one or more files to share and afterwards presses "Nearby Share" in the sharing pop-up, the app searches for other devices nearby independently of UWB. Another device is found when it has *Nearby Share* enabled and is a sharer's contact. Now, if both devices are Samsung phones and are UWB-enabled devices, they do a UWB ranging session, which the *Google Play services* app starts using the *UWB API service*.

Mdx Kit Service is another system app that uses the *UWB API service* and is used by Samsung's *Quick Share* app, which is an app developed by Samsung with very similar functionality to *Nearby Share*. When selecting one or more files to share, *Quick Share* is always started. This is even the case when *Quick Share* is disabled. We think it gets temporarily enabled when sharing a file. Furthermore, *Quick Share* immediately prepares a UWB ranging session without starting it by using *Mdx Kit Service*. When nearby

Samsung devices are found that are UWB capable and "Quick Share" is pressed in the sharing pop-up, a ranging session with the other UWB-enabled devices is done. Then, the live direction to the closest UWB-enabled device is displayed in *Quick Share*. We point out, while the UWB usage in the *Nearby Share* process is documented by Samsung [45], this is not the case for the usage of UWB in *Quick Share*.

5.5 SMARTTAG+

Samsung's SmartTag+ is a low-cost Internet of Things (IoT) tracking device, which can be used to find objects that are equipped with it. To manage and locate the SmartTag+, one needs to use Samsung's SmartThings app. Furthermore, its functionality is similar to the normal SmartTag, but it additionally integrates UWB, which can be used to find the SmartTag+'s location more precisely. Therefore, the SmartTag+ is part of Samsung's UWB ecosystem. Additionally, the UWB functionality is provided by the integrated NXP *SR040*. This makes the SmartTag+ the cheapest commerical device currently available that comes with an NXP UWB chip.

While there were some efforts made to analyze Samsung's normal SmartTag [5] and Apple's UWB-enabled AirTag [10, 44], we do not know of research for the SmartTag+. Hence, we analyze the SmartTag+ and its management entities on our Samsung phone.

For the following steps, we use our analysis results from Section 5.2. Additionally, we examine all files in the SmartThings app's data folder, which is located in */data/data/-com.samsung.android.oneconnect/*. We find multiple interesting files in this folder, including plugins used to control and locate the SmartTag+. We also find multiple unencrypted firmware versions of the SmartTag+ located in one plugin's folder.

Next, we explain the SmartTag+ management entities and how a UWB ranging session is set up between the SmartTag+ and a UWB-enabled Samsung smartphone. Afterwards, we examine the SmartTag+'s firmware and its Printed Circuit Board (PCB).

5.5.1 *Entities Used in the Interaction With the Smarttag+*

Different additional entities are used when a user manages or locates the SmartTag+. We show an overview in Figure 11. Moreover, the foundation is the SmartThings app. When a user sets up a SmartTag+ for the first time, the SmartThings app downloads an additional web plugin, which is called *WebTRKPlugin*. We also call it the SmartTag+'s *Management Plugin* or simply the SmartTag+'s *Plugin*, and show a screenshot of it in Figure 12. This web plugin is loaded into the SmartThings app using Android's WebView[5] and provides a controller interface for the user to control the SmartTag+. We find the files that the SmartThings app uses to display the web plugin in the corresponding data folder on our phone. The plugin consists of one HTML file and multiple JavaScript and CSS files. Additionally, we assume that the plugin is similar or equal to the normal SmartTag's plugin based on screenshots we find in a GitHub repository [5].

Furthermore, the JavaScript files contain all SmartTag+ management logic. For example, they handle the ringtone update of the SmartTag+. However, for most actions, the JavaScript execution environment is insufficient. Therefore, selected methods of the SmartThings app are called, which are accessible through the *@JavascriptInterface* annotation.

5 WebView enables apps to display web pages using remote or local resources.

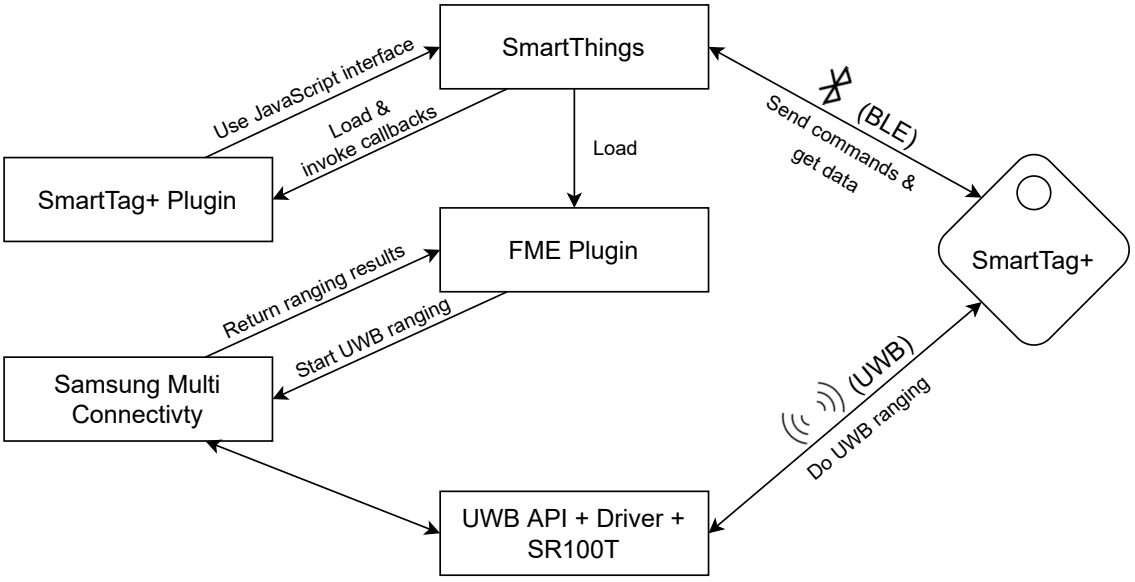

Figure 11: SmartTag+'s interaction entities.

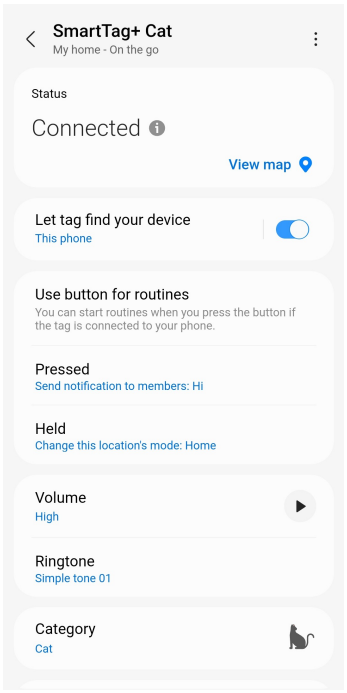

Figure 12: SmartTag+'s management plugin.

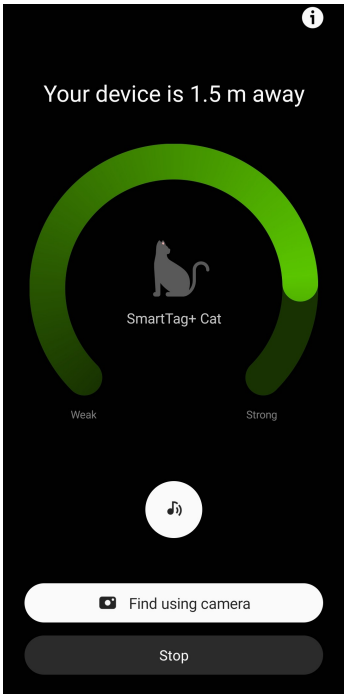

Figure 13: Finding the SmartTag+.

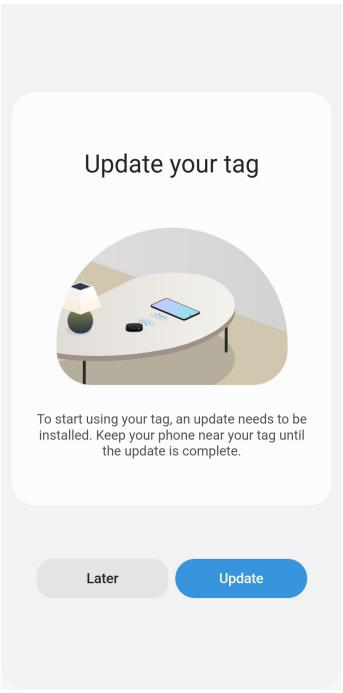

Figure 14: SmartTag+ update request.

For example, the SmartTag+'s plugin can communicate over Bluetooth Low Energy (BLE) with the SmartTag+ or can use location services using these methods. Moreover, since the JavaScript files are in the SmartThings app's data folder, we can easily modify them. After modifying one of these files, we can see our changes successfully after reopening the plugin in the SmartThings app. It is helpful for us that we can modify these JavaScript

files and that they are not obfuscated. The positive consequence is that we can more easily manipulate the communication with the SmartTag+ and more, without digging into the SmartThings app itself and writing Frida scripts, which would cost us more time. For example, we now can trigger the Over-The-Air (OTA) firmware update process, which normally only happens if a new version is available. Additionally, we can extract or manipulate the firmware sent to the SmartTag+ by modifying only a few lines of JavaScript code.

We further detect the possibility to enable a test mode in the plugin by modifying one line of code of a JavaScript file. However, the only difference between normal and test mode is the possibility of manually triggering two types of firmware updates. The first type updates the regular firmware with the latest version, independently if it is the same version. The other type's intention is downloading a test version and then updating the SmartTag+'s firmware with this test version. However, even when a different link is used to download the test firmware, only the regular firmware is downloaded here. Maybe it is necessary to be in the intranet of Samsung to get the test firmware returned, or the server simply does not host a test version because no test version is currently available.

No UWB functionality can be triggered with the SmartTag+'s management plugin. Instead, another plugin needs to be used. Once the SmartTag+'s plugin is installed, the user can install an additional plugin, which also can be installed in the SmartThings app itself. This plugin is called *SmartThings Find*, and it can be started through the SmartTag+'s plugin or directly in the SmartThings app's main menu. Unlike the SmartTag+'s plugin, this plugin is a .apk file, and it opens its own web plugin to display information. Moreover, when *SmartThings Find* is downloaded and installed by the SmartThings app, it is stored in the app's data folder. This means it is not installed on the phone as a regular app, but the SmartThings app loads it from its data folder when the plugin is started.

The *SmartThings Find* plugin is the entity that can start the establishment of UWB ranging sessions with the SmartTag+. When the user opens the plugin, the user can press "Search nearby" in the plugin. Then, the *SmartThings Find* uses the SmartThings app to prepare a UWB ranging session with the SmartTag+ over BLE. The plugin further instructs the *Samsung Multi Connectivity* service to establish a UWB ranging session with the SmartTag+. Subsequently, the *Samsung Multi Connectivity* service uses the *UWB API service* to establish a UWB ranging session with the SmartTag+, and it periodically returns the post processed ranging data to the *SmartThings Find* plugin, which includes the distance and Angle of Arrival (AoA) measurement results. Then, the user gets the distance and direction displayed live. In Figure 13, we show a screenshot of how the results are displayed to the user.

The *SmartThings Find* plugin is further used for Samsung's offline finding network and searching for unknown SmartTags nearby. The network and finding of unknown SmartTags are independent of Samsung's UWB ecosystem and therefore out of scope.

5.5.2 *SmartTag+'s Firmware*

Two versions of the SmartTag+'s firmware come with the SmartTag+'s management plugin. These can be found in the plugin subfolder of the SmartThings app's data folder. Each name starts with "UWB_TAG" followed by the version. Once a new firmware version is available, the user is asked in the plugin to start an update as we show in Figure 14.

Then, the latest version will be downloaded from Samsung's server, and the plugin sends it to the SmartTag+ using the SmartThings app. However, even if the latest version is downloaded in the update process, it is not stored in the data folder. Moreover, when we first overviewed the firmware contained in the data folder, the firmware versions were older than the current ones we find. We assume the latest firmware are included with each update of the plugin. Furthermore, the firmware is unencrypted and intended to be used on an NXP *QN9090*, which is the the SmartTag+'s main chip.

In the duration of our thesis, we extracted every version of the SmartTag+'s firmware we encountered using two methods. First, we looked in the SmartThings app data folder for every new version of the SmartTag+'s plugin. Second, whenever the plugin notified us of a new firmware version and to do an update, we extracted the newly downloaded firmware version by modifying one of the plugin's JavaScript files. In the mid of December 2021, we have six different firmware versions, whereby the oldest version we have is 0.50.30, and the latest version is 1.01.04.

Furthermore, we find two other binaries in the same folder as the SmartTag+'s firmware. These are named "ble_finder...". We always encountered only the two identical versions in the duration of our thesis, which are 1.01.23 and 1.01.26. We also do not find any usage of these files in the SmartTag+'s plugin or the SmartThings app. However, we find many variables named "ble_finder" in the *SmartThings Find* plugin and conclude based on the name a connection to Samsung's offline finding network, which the SmartTag+ is also part of. Furthermore, we find that strings of the "UWB_TAG" firmware and these files are different but often have the same meaning. Hence, we think these files might be the normal SmartTag's firmware. To check this hypothesis, we compare strings of these files and the normal SmartTag's firmware, which we find in a GitHub repository [5]. Indeed, both files have very high similarities. Thus, we conclude that these files are the normal SmartTag's firmware. Since the normal SmartTag's firmware is included in the plugin's folder, it might indicate that the plugin is simultaneously used for the SmartTag+ and the normal SmartTag. Moreover, since the normal SmartTag has no UWB integrated, itself and its firmware are out of scope.

5.5.2.1 *Firmware Analysis*

We do a brief static analysis of the SmartTag+'s firmware. Thereby, we evaluate strings and overview selected functions with Ghidra to learn about the SmartTag+'s UWB usage.

The firmware also contains the NXP *SR040*'s firmware. However, the *SR040*'s firmware is encrypted and signed like the *SR100T*'s firmware. The *QN9090* sends this firmware at a certain step to the *SR040*. Furthermore, this is the only encryption in the SmartTag+'s firmware. So the whole code that runs on the *QN9090* is not encrypted, which includes the communication with the *SR040*.

Both MK UWB kits come with a UWB tag, which is similar to the SmartTag+. This tag also contains, like the SmartTag+, the NXP *QN9090* and *SR040* on its PCB. A demo firmware can be uploaded to the tag, which source code can be found in the UWB kits. The source code consists of demo files, which make use of the UWB API that we already examined in our MK UWB kits' source code analysis. Moreover, it is possible to change the demo files, compile them, and upload the compiled firmware to the tag. When compiled, these demo files and the UWB API are included in the tag's firmware.

We compare the UWB tag's firmware from the standard UWB kit with the SmartTag+'s firmware and find a very high similarity. We do no detailed analysis of the similarity, but we can tell that Samsung uses the complete demo files and therefore, the UWB API. Since the compiled demo makes up most of the firmware, we have most of the source code for the SmartTag+'s firmware.

We further compare the size of the standard UWB kit's tag firmware with the SmartTag+'s firmware as well as look for strings that are only contained in one of both firmware. Thereby, we determine that Samsung has extended the UWB kit's demo with around 20% more functionality for the SmartTag+. We can assign some of these extensions to the functionality of sending specific BLE beacons, which are part of Samsung's offline finding network. The offline finding network is not in the scope of our thesis. Additionally, we find strings that are related to the usage of signatures and Cyclic Redundancy Check (CRC) values, which might indicate integrity checks of the firmware that is uploaded to the SmartTag+.

5.5.2.2 *Firmware Update Process*

We show the SmartTag+'s firmware update process in Figure 15. The SmartTag+'s plugin always uses the SmartThings app's exported methods, which are marked with the *@JavascriptInterface* annotation. The plugin can communicate with Samsung's servers and the SmartTag+ using these methods. Moreover, the communication between the SmartThings app's methods and the SmartTag+ is done over BLE.

When the SmartTag+'s plugin is loaded, it requests information from Samsung's servers about the latest SmartTag+ firmware, which includes the firmware version. Afterwards, the plugin requests the device data from the SmartTag+, which includes the SmartTag+'s current firmware version. Next, the plugin compares if the SmartTag+'s firmware version is equal to the latest firmware version. If both versions are equal, the plugin terminates the update process. Otherwise, it continues and prepares the sending of the latest firmware. Hereby, the plugin first checks if the latest firmware is cached, i.e. if it was already downloaded earlier. The latest firmware is only cached when the user previously declined a firmware update or terminated it, and the firmware update runs again when the plugin is reloaded. If the firmware is not cached, the plugin downloads the firmware data from Samsung's servers.

Now the firmware transfer begins. Thereby, the plugin first transmits the firmware information, which includes firmware's version and size. If the SmartTag+ responds with a positive status, then the plugin sends the firmware data in chunks, whereby the SmartTag+ responds for each chunk with a status indicator. This step is repeated until the plugin has sent the complete firmware or the returned status indicator signalizes an error. Furthermore, the plugin sends as a checksum a CRC value once for the whole firmware and together with each firmware chunk. The plugin uses CRC-16 Kermit as the CRC algorithm.

After the firmware is completely transferred, the SmartTag+ returns one of two possible status codes, which either signalizes a transfer successs or a transfer failure. We assume that the status code depends on integrity checks, which the SmartTag+ does by verifying the firmware's digital signature and CRC value.

5.5.3 *SmartTag+'s PCB*

Since the SmartTag+ is a low-cost IoT device that hosts an NXP UWB chip, and since it is easier to open than our Samsung phone, it can be valuable for us to analyze its PCB.

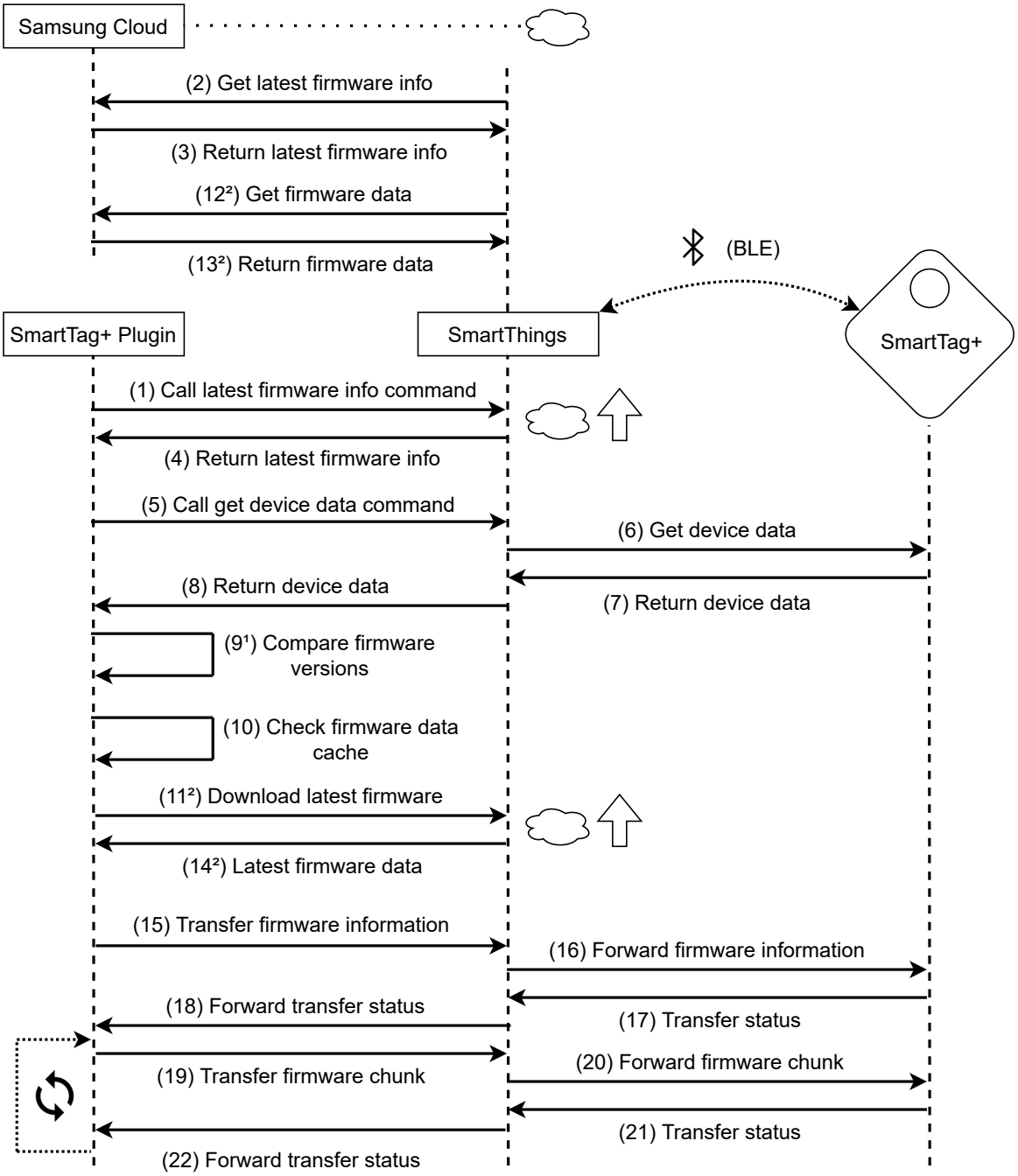

Figure 15: SmartTag+'s firmware update process. The SmartThings app gets the firmware's information and data from the cloud in separate steps.
[1] Can lead to a termination.
[2] Optional.

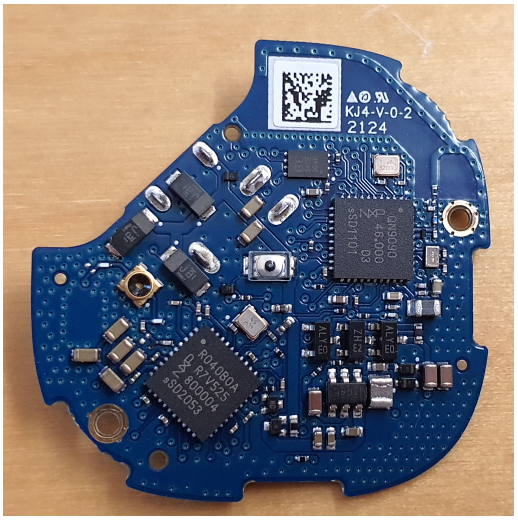

Figure 16: SmartTag+'s PCB - Upper side.

Figure 17: SmartTag+'s PCB - Lower side.

The SmartTag+ has several test pads on its PCB. These test pads have a connection to important pins of different PCB components and facilitate general PCB tests, which we use for our advantage when evaluating the SmartTag+'s hardware security. Thus, we analyze the SmartTag+'s PCB to evaluate the function of test pads. In our evaluation, we make use of our test pad analysis results, for example, when trying to get debug access to the *QN9090* or *SR040*.

In Figure 16 and Figure 17, we show the upper and lower side of the SmartTag+'s PCB. It contains the NXP *QN9090* as the host chip and the *SR040* as the UWB chip. In addition, the PCB contains a flash memory component from GigaDevice, which is labeled as *GD 34CE 2048* and is connected with the *QN9090* but not the *SR040*. The *SR040* stores the firmware on-chip [33]. Furthermore, the PCB is completely different than the normal SmartTag's PCB, and another main chip is used, which we can compare with the help of a GitHub repository [5].

To quickly learn each test pad's connected pin, we visually follow the wire from the test pad to its endpoint using high-quality photographs. Subsequently, we verify the connection using a multimeter. If we are not successful in finding a test pad's connection with this method, we probe for the test pad each possible endpoint with a multimeter. Afterwards, to learn the connected endpoint's function, we study the *QN9090*'s and *SR040*'s datasheets, which we retrieve from NXP's website [29, 33]. We do not find datasheets for other components on the PCB.

The PCB contains 36 numbered test pads. However, we could not find the test pads number 22 and 34, which might be hidden under components of the PCB. In Table 4, we show the most important test pads for us. Additionally, to facilitate future research, we include an extended evaluation of each test pad in Appendix A.3.

The most important test pads are the connections to the *QN9090*'s and *SR040*'s Serial Wire Debug (SWD) pins, which might get us debug access to the chips. Further important are the test pads that are connected to the *QN9090*'s Universal Asynchronous Receiver-Transmitter (UART) pins, which the *SR040* does not have [33]. The UART pins might get us logs or a shell. Moreover, for sniffing the Serial Peripheral Interface (SPI)

COMPONENTS	CONNECTION	TEST PAD
SR040	SWDIO pin	2
	SWCLK pin	3
	RST_N pin	5
QN9090	SWDIO pin	14
	SWCLK pin	15
	RST_N pin	10
	UART TXD pin	11
	UART RXD pin	13
QN9090, SR040	SCK - SCK - SPI line 0	27
	SS - CS - SPI line 0	30
	MISO - MOSI - SPI line 0	28
	MOSI - MISO - SPI line 0	29
QN9090, FLASH	SCK - SCK - SPI line 1	33
	SS - CS - SPI line 1	24
	MISO - MOSI - SPI line 1	31
	MOSI - MISO - SPI line 1	32

Table 4: A selection of evaluated test pads for the SmartTag+'s PCB. The first connection matches the first component in case of multiple connections.

communication, test pads are important that connect to SPI pins of the *QN9090*, *SR040*, or the flash memory component.

6 IDENTIFICATION OF ATTACK VECTORS

At this point, we understand Samsung's Ultra-Wideband (UWB) ecosystem. We know each entity and the communication workflow between those entities. Now, we want to identify attack vectors in the ecosystem, of which we also later attack selected ones to evaluate the ecosystem's security. We evaluate our previous findings from a security perspective to identify attack vectors. Furthermore, we do not want to identify them only for ourselves, but we also intend to give future research a starting point for analysis. For easier understanding by the reader, we try to keep the attack vectors we identify abstract. Additionally, while it is possible to identify attack vectors in most parts of each entity, we try to focus on parts that are directly related to the newly integrated UWB functionality. For example, we are not interested in vulnerabilities that affect the SmartThings app's network security, even if the SmartThings app is part of Samsung's UWB ecosystem.

Next, we elaborate on the abstract vectors, starting with the lowest level entity. We further present which of these attack vectors lay in our focus for the rest of our thesis. In Figure 18, we show the selection of attack vectors we target for our evaluation.

6.1 NXP UWB CHIPS

We learned about the communication protocols with NXP's UWB chips in Section 4.2. The protocol Ultra-Wideband Command Interface (UCI) is used for all UWB-related messages by all chips. On the other side, Host-Based Command/Control Interface (HBCI) is used to manage the *SR100T* and *SR150*, and it is mainly used to transfer the firmware to these chips. We further learned the workflow of transferring the firmware to the *SR100T* in Section 4.3. In addition, we analyzed the *SR100T*'s driver in Section 4.4. With all learned information, we built the *SR100T*'s state machine in Section 4.5. The two essential main states in the state machine are *UCI mode* and *HBCI mode*.

The *SR040* and *SR100T* are integrated into Samsung's UWB ecosystem, and the firmware of these chips is the first attack vector. However, we cannot analyze the firmware of these chips directly since it is encrypted and signed for each chip. Additionally, it is not publicly available which encryption and signature algorithms are used. Therefore, we need to retrieve the decrypted firmware first. We can try two methods to get the decrypted firmware. First, we can test if we can get debug access to the UWB chips on the Printed Circuit Board (PCB) of our phone or SmartTag+. If we are fortunate, we can retrieve the unencrypted firmware. Second, we can analyze the local firmware download process in depth to target information that may help us learning about the firmware encryption, signature, and more. With this information, we can try attacks on the encryption and similar. Furthermore, even when we do not have the decrypted firmware, we can try firmware downgrade attacks.

There may be another way to analyze the encrypted firmware. If we assume that NXP reuses code they also have provided for the Mobile Knowledge (MK) UWB kits, then we know parts of the UWB chips' encrypted firmware. At least a part is likely reused in the

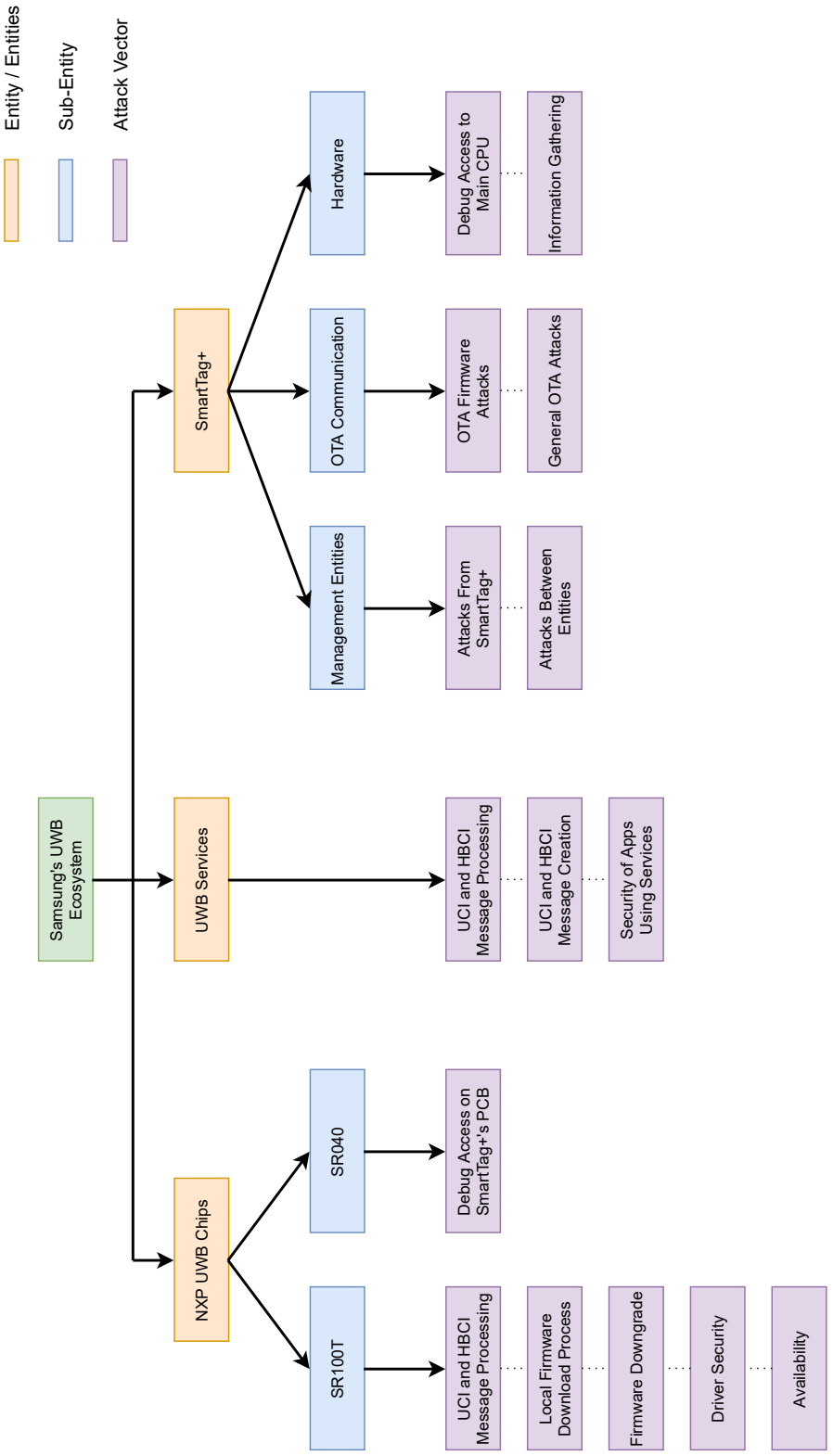

Figure 18: Selected test surface for our evaluation.

firmware. Therefore, we can look for vulnerabilities in the UWB kits' source code and then establish attacks against the UWB chips for these vulnerabilities.

Furthermore, we can test the UWB chips' availibility without having the encrypted firmware. Thereby, we can craft special messages that we send to the chip. For example, we can use a fuzzer for crafting these messages.

In our evaluation, we assess the *SR100T*'s local firmware download process. We choose the *SR100T* because we can already communicate with it. Furthermore, we try getting debug access to the *SR040* because the low-cost SmartTag+ is easier to open than our Samsung phone, it costs much less to replace the SmartTag+ in the case of bricking it, and we already have evaluated all of the SmartTag+'s test pads. Additionally, we test for the possibility of firmware downgrades and test how we can disturb the *SR100T*'s availability. We further look for vulnerabilities in the UWB kits' source code and try to trigger these through specially crafted messages.

6.2 SR100T'S DRIVER

In Section 4.4, we learned about the driver's role. Only the Linux users *uwb* and *root* can interact with it, and its main purpose it to forward UCI and HBCI messages between the user space and the *SR100T*.

The whole driver is the next attack vector. We can evaluate the driver's security in depth since the source code is publicly available. Furthermore, the source code has less than 1600 Lines Of Code (LoC), which is significant less than NXP's UWB Application Programming Interface (API) provided in the UWB kits that has more than 44000 LoC. In addition, the code is properly written and easy to understand.

6.3 UCI AND HBCI

In Section 4.2, we learned the workings of the communication protocols UCI and HBCI. We choose both protocols as our next attack vector because they are far-reaching in Samsung's UWB ecosystem, and messages pass multiple entities. Furthermore, the same types of vulnerabilities can exist in multiple entities when handling the procotols' messages. Four different entities are directly involved on a Samsung phone for sending and receiving messages of both protocols. The entities are the *SR100T*, its driver, the *UWB Hardware Abstraction Layer (HAL) service*, and the *UWB API service*. For these entities, we are interested in vulnerabilities that are related to UCI and HBCI message creation or processing. Thereby, we can look for vulnerabilities that can be attacked by another entity, which can be, for example, any app on the phone or a compromised *SR100T*.

We can test for two types of attacks, whereby not each type can be applied to each entity. The first type is attacking the processing of UCI or HBCI messages. We can craft attack messages that are both protocol conform and non-conform to trigger vulnerabilities. Furthermore, we can use the knowledge of the protocol fields. For example, we can test for buffer overflows by sending a message that declares a payload size of 100 in the header but has a larger payload.

The second type is attacking the creation of UCI and HBCI messages. For this attack type, we are limited to test the *UWB HAL service* and *UWB API service*, since only here message are created that we can control. For example, we can attack the message creation

from an external app by passing specially crafted configuration values to a vulnerable method creating UCI messages from these values.

In our evaluation, we assess the UCI and HBCI message creation and processing of each entity. Since both the *UWB API service* and *UWB HAL service* heavily make use of the MK UWB kits' source code, we primarily look for vulnerabilities in the source code, which simplifies our analysis. Then, we practically attack selected found vulnerabilities. Furthermore, since we do not have the *SR100T*'s unencrypted firmware, we practically derive attacks for every vulnerability we find in the source code because we assume that code is reused in the firmware.

6.4 UWB SERVICES

The next attack vector are both the *UWB API service* and the *UWB HAL service*, which we analyzed in Section 5.3. These play a major role in Samsung's user space. In addition, besides processes that run as the Linux *root* user, both services are the only entities on the phone permitted to communicate with the *SR100T*'s driver, because they run as the user *uwb* like the driver. In the services, we can look for vulnerabilities excluding the parts that handle UCI and HBCI messages. For example, we can look for vulnerabilities that are related to the Inter-Process Communication (IPC) with the services. We can also test if we can circumvent the protection of privileged actions in the *UWB API service*, such that any app can execute a privileged UWB-related action on the phone.

Moreover, we mostly cannot use the MK UWB kits' source code as help for finding vulnerabilities because the source code mostly contains UCI and HBCI message creation and processing, which are not part of this attack vector. Instead, we need to reverse engineer Samsung's services and libraries.

We have only a limited time frame for the analysis of the services. Therefore, we take a focus on the UCI and HBCI message handling and do not look consciously for other vulnerabilities in the services, except if these are directly connected to the communication with the *SR100T*.

6.5 APPS USING THE UWB API SERVICE

In Section 5.4, we describe multiple apps we detected that use the *UWB API service*. Some of these build a middleware that can be used by other apps. For example, the SmartThings app's plugin for UWB ranging establishment with the SmartTag+ uses such a middleware service, which is the *Samsung Multi Connectivity* app. In addition, we found the *UwbTest* app, which is an helper app for testing UWB functionality and that can be started with a key combination in the regular *Phone* app.

The apps that use the *UWB API service* are also part of Samsung's UWB ecosystem and also are attack vectors. Therefore, a security analysis of them is also important. Additionally, every app we detected is at the same time a system app, and some of them provide an API for other apps to use exported functions. Suppose these system apps can be attacked or misused successfully. In that case, it can be possible for a third-party app to use a privileged method of the *UWB API service* through the vulnerable system app, which normally is only accessible for system apps. Furthermore, it is also important to look for vulnerabilities in the system apps that can be attacked from the *UWB API service*.

Since the service runs as the Linux user *uwb* on the phone, it is limited for certain actions. However, a system app that runs as the Linux user *system* has much more privileges, which is an attractive target for attackers to escalate privileges.

Because of our limited time frame, we only analyze two apps that use the *UWB API service* on our phone. The first app is *UwbTest*, which also runs as the Linux user *system*. The app is directly related to the UWB ecosystem and contains predominantly code that is related to using the service. We also analyze UWB-related parts of the *Samsung Multi Connectivity* app because the workflow from the SmartTag+ plugin to the *UWB API service* passes through this app. This app also runs as the Linux user *system*. Furthermore, we skip the analysis of UWB-related parts of *Mdx Kit Service* and *Google Play services*, because we found only a limited UWB usage in these apps.

6.6 SMARTTAG+

The SmartTag+ is a low-cost Internet of Things (IoT) tracking device. In Section 5.5, we analyzed the SmartTag+ integration into Samsung's UWB ecosystem. We further analyzed its PCB.

A compromised SmartTag+ can affect the privacy and more of its users. Moreover, a normal user cannot learn that the SmartTag+ is compromised since no utility like anti-virus software can be used. Also, entities on the phone that handle the SmartTag+ need to be secured against attacks. Therefore, our last attack vectors are the SmartTag+ itself and its management interfaces on Samsung phones. These interfaces are the SmartTag+'s plugin, the SmartThings app's SmartTag+ related parts, and the *SmartThings Find* plugin.

Even if the SmartTag+ is only an IoT device with limited usage, its attack surface is relatively large. We can test for different vulnerabilities on the SmartTag+'s PCB. For example, we can test if it is possible to dump or even manipulate the firmware over Serial Wire Debug (SWD) access to the *QN9090*. We also can test if it is possible to sniff the Serial Peripheral Interface (SPI) communication between components on the PCB in order to learn secrets.

Since the SmartTag+ uses much of the MK UWB kits' source code, we also can derive attacks for vulnerabilities that we find in the source code and then attack Over-The-Air (OTA) from our phone. However, such attacks are limited since we only can control the Bluetooth Low Energy (BLE) communication with the SmartTag+, and the kits' source code is not directly related to BLE usage on the SmartTag+.

We also can look for vulnerabilities of the SmartTag+'s management interfaces, which all run on a Samsung phone. For these interfaces, we can test if a compromised SmartTag+ can attack them. We also can look for vulnerabilities that are exploitable between the interfaces or can be exploited from a lower layer, which for example, can be initiated by the *UWB API service*. Furthermore, we can try to attack the OTA firmware update process in order to downgrade or manipulate the firmware without PCB access. We also can evaluate the security of sharing a SmartTag+ with other SmartThings app users. Additionally, normally the SmartTag+'s location is shared with Samsung's servers, even if the SmartTag+ is not connected with the owner's phone. Therefore, a secure infrastructure is important, and the infrastructure's security can be tested as well.

In our evaluation, we focus on PCB attacks. We test if we can sniff SPI communication between components and if we can get SWD access to the *QN9090* and *SR040*. If this is

successful, we try to extract the firmware over SWD and afterwards, we try to manipulate it. Furthermore, we test for vulnerabilities in the management entities and focus on those that a compromised SmartTag+ can attack. In addition, we check for OTA attacks against the SmartTag+, which include to test if we can downgrade or manipulate the firmware OTA. We further test if there exist vulnerabilities in the management entities that can be attacked from the UWB services or the *Samsung Multi Connectivity* app.

7 IMPLEMENTATION OF UTILITIES

In Chapter 6, we made a selection of attack vectors that we want to assess in our evaluation. Now, we need to develop tools that aid us in our evaluation. Our goal is to implement tools that can be used for different testing scenarios at once. For example, the foundation of a Frida script that attacks the *SR100T* should also be reusable to simulate attacks coming from the *SR100T*. Therewith, we can save time that we can invest in attacks. Another goal of the tools' implementations is the reusability for future research. We intend that our tools can be used to continue our work or for other research.

Next, we describe our Ultra-Wideband Command Interface (UCI) and Host-Based Command/Control Interface (HBCI) Wireshark dissector, which can decode every UCI message we encounter. Afterwards, we present the foundations of our Frida scripts that we use to test for vulnerabilities and to attack entities. We elaborate on the implementations of selected Frida scripts that use the foundations. Subsequently, we explain modifications that we take on the *ucitool*, and we explain how we can use the *ucitool* for our analysis by describing self-developed *ucitool* scripts. Then, we depict the modifications we take on the SmartTag+ plugin's JavaScript files to test for different vulnerabilities. Last, we delineate how we set up our environment for testing Printed Circuit Board (PCB) attacks on the SmartTag+.

7.1 UCI AND HBCI WIRESHARK DISSECTOR

We do not find the specifications of UCI and HBCI publicly. Therefore, we reverse engineered both specifications in Chapter 4. However, even by knowing the specifications, we cannot quickly understand the communication contents between host and NXP's Ultra-Wideband (UWB) chips without a detailed analysis of the communication every time. We further have no simple way to provide our knowledge to other researchers in a way that does not require understanding the protocols. Thus, to make use of our received knowledge about both specifications, we decide to integrate our knowledge into a graphical utility that can display the communication comprehensibly. With this utility, other researchers and we can quickly understand the communication contents. Other researchers do not even need to understand the specifications to learn about the communication contents.

We choose to integrate our knowledge into a Wireshark dissector of which we show an extract in Figure 19. We decide to use Wireshark for multiple reasons. First, it provides a simple interface to integrate decoders of any custom protocol, which are called *Dissectors*. So we do not need to develop additional software that provides a Graphical User Interface (GUI). Second, it provides an easy-to-understand GUI that is tailored to display messages of any protocol. Third, Wireshark is a common tool, and it is part of every security researcher's toolkit. Thus, our dissector can be reused without installing additional tools.

The *ucitool* also can be used to decode UCI messages and output the contents. However, it outputs these only in a terminal, which is not easily comprehensible. In addition, it

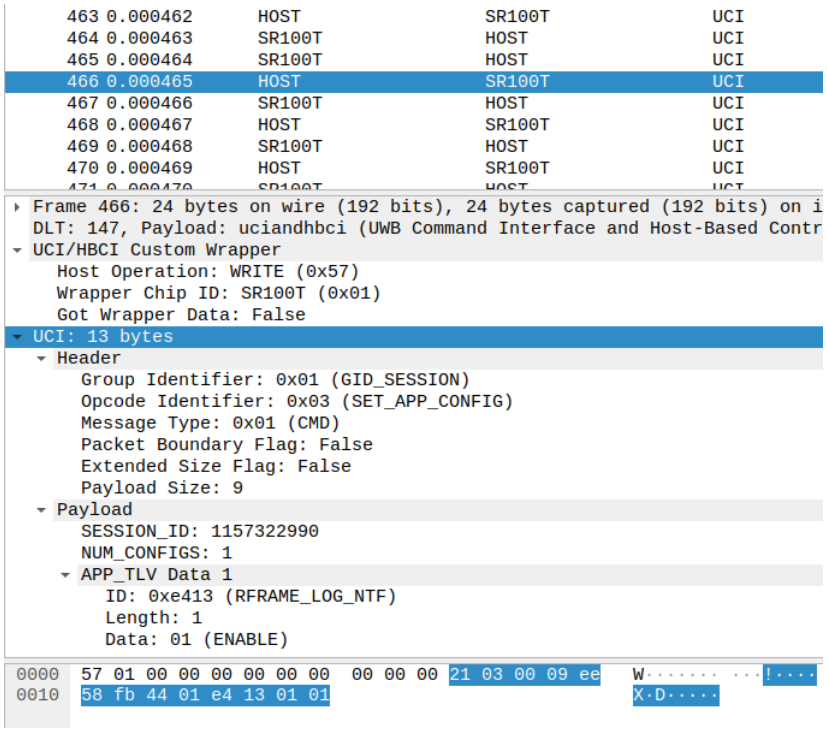

Figure 19: Extract of our Wireshark dissector.

cannot decode HBCI messages, and it is only tailored to the communication with the *SR100T*. Moreover, the *ucitool* cannot be shared with other researchers that do not have access to the Mobile Knowledge (MK) UWB kits. Therefore, we do not use the *ucitool* for decoding UCI and HBCI messages.

7.1.1 *Implementation Overview*

We implement a combined dissector that can decode UCI and HBCI messages simultaneously. In Appendix A.4 we provide a short user guide of how the dissector can be imported into Wireshark and how a hexdump of the communication can be analyzed with it. The hexdumps can be generated using one of two tools, which we describe later, and we provide a short user guide for these in Appendix A.6 and Appendix A.7.

Wireshark dissectors can be written in Lua instead of C, and we choose Lua as the programming language for our dissector since for us, it is easier and faster to use than C. Moreover, our goal is that the dissector can decode the communication with all of NXP's UWB chips. Unfortunately, we have only access to Samsung devices, and therefore, we only can test our dissector for the communication with the *SR100T* and *SR040*. However, we assume it is likely that our dissector is fully able to decode the communication with the *SR150*.

Next, we describe the implementation of the dissector's foundation. Afterwards, we detail the implementation of our UCI and HBCI decoders that are part of the dissector. Subsequently, we delineate two tools that can be used to extract the communication with the *SR100T*. One of the tools can be used for any chip. We further include the user guides in Appendix A.4, Appendix A.5, Appendix A.6, and Appendix A.7.

7.1.1.1 *Foundation*

The dissector's foundation sets up the dissector and decodes the *UCI*/*HBCI Custom Wrapper* for every message, which is a required wrapper defined by us and is independent of the UCI or HBCI message to decode. This custom wrapper is prepended to any UCI or HBCI message and is eleven bytes long. One can view it as an additional header that declares helper values. The custom wrapper's first byte declares if the message is sent or received by the host, the second byte declares the UWB chip ID, and the third byte is a flag that declares if wrapper data follows. All of these three bytes are required to be set with the correct values. Moreover, the following eight bytes are the wrapper data, and the dissector only interprets these if the wrapper data flag is set. The wrapper data's role is explained in Section 7.1.2.1.

After interpreting the *UCI*/*HBCI Custom Wrapper*, the dissector continues to decode the actual message by passing it to the decoders.

7.1.1.2 *Decoders*

The decoders make up most of the dissector's implementation. When decoding a message, the dissector first identifies if the message is a UCI or HBCI message based on the header. Then, the dissector forwards the complete message to the corresponding message decoder. Next, we detail how we decode UCI and HBCI messages.

UCI DECODER Except for the ranging data, we detected no difference of the UCI payload interpretations for different chips in the MK UWB kits' source code. Furthermore, we find that the UCI message set for the communication with the *SR040* is a subset of the other chips' message set with one exception. The message set of the communication with the *SR040* contains 15 additional messages that are specific for the *SR040*. Moreover, the *ucitool*'s YAML file contains the full UCI specification, which we can use for our decoders. We also extend the YAML file with the additional *SR040*-related messages. Now, we depict how we implement our UCI decoders based on these circumstances.

We use automatically generated header and payload decoders for all UCI messages except for the returned ranging data's payload. For generating these, we implement a tool that parses the *ucitool*'s YAML file and automatically generates decoders as Lua code from the parsed data. The tool further generates the necessary tables and fields that are used by the decoders to give the bytes a meaning. In addition, it adds references to selected tables that are used by the dissector to decode the meaning of header values and to find the correct payload decoder for given header opcodes.

We name the tool *UCI Decoder Generator* because of the described tool's functionality, and we include a user guide in Appendix A.5. The advantage of this tool is that it can parse different and modified versions of the YAML file. When an updated YAML file exists, or when one extends the YAML file by hand, the tool can automatically generate the new decoders. The *UCI Decoder Generator*'s old generated files used by the dissector can be simply replaced with the new generated files.

In summary, our dissector's foundation extracts the header values for a UCI message. These values are decoded with our generated header decoders. In addition, these values are used to find the corresponding generated payload decoder, which then decodes the payload if exisiting.

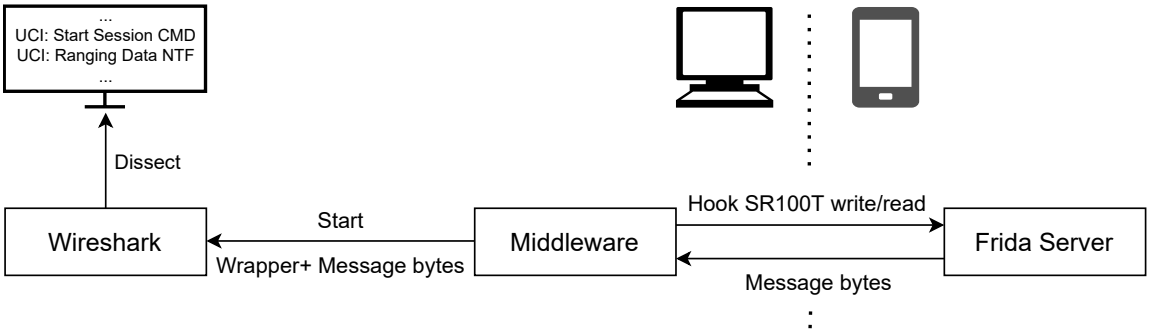

Figure 20: *Live Decoder* workflow overview.

We use a handwritten decoder for the ranging data, which is returned by NXP's UWB chips and is part of an *Notification (NTF)* UCI message. We manually write the decoder because we detect a different interpretation for the ranging data that depends on the used chip and its firmware version. Otherwise, when would use an automatically generated ranging data decoder that is tailored to the *SR100T* with an old firmware version, we could likely not decode the ranging data returned by other chips or the *SR100T* with a later firmware version.

HBCI DECODER For HBCI messages, the dissector first resolves and displays the HBCI message's header values. The dissector looks up the meaning in a handwritten table to decode the header values. Afterwards, the dissector forwards the payload to a generic decoder. Since HBCI payloads mainly only consist of data, and we do not find interpretations except for single bytes of two payloads, we do not integrate HBCI payload decoders. We only mark the payload data and Longitudinal Redundancy Check (LRC) value. Nevertheless, since we can decode all header values of HBCI messages, the user of our dissector still knows the data's meaning.

7.1.2 *Tools*

We further develop two tools that facilitate the extraction of the communication with NXP's UWB chips, which can then be reviewed in our dissector.

7.1.2.1 *Live Decoder*

We implement a tool that can live decode the communication with the *SR100T* on a Samsung phone, and it shows the real-time communication in Wireshark by using our dissector. The tool facilitates the communication extraction, and by being able to live decode the communication, we have full knowledge about what happens at the moment between the host and the *SR100T*. We name the tool *Live Decoder*, and we provide a user guide in Appendix A.6. During live decoding, the tool also creates a hexdump file, which can be reviewed in Wireshark at any time.

The *Live Decoder* consists of two parts, and in Figure 20 we show an overview of the workflow. The first part is implemented in Python and acts as a middleware. Therefore, we name it *Middleware*. Furthermore, large parts of the first part's implementation are oriented on or copied from a GitHub repository in [22].

The first part starts Wireshark and feeds it with data live, which our dissector decodes and displays. Moreover, the data is retrieved by connecting to the Frida server running on the phone and executing two Frida scripts, which build the second part. The Frida scripts hook selected methods of the *UWB Hardware Abstraction Layer (HAL) service* that write to and read from the *SR100T*. These hooked methods directly communicate with the driver, meaning the Frida scripts hook the methods at the user space's lowest level.

Additionally, the Frida scripts hook two methods of the *UWB Application Programming Interface (API) service*. The *SR100T* returns for some firmware versions standard values for two Angle of Arrival (AoA) values, and in the service are these values post-calculated based on several factors that are partly independent from the communication data. We can retrieve these post-calculated values by hooking the two services' methods. Then, the script adds the retrieved values to the *UCI*/*HBCI Custom Wrapper*, and the dissector can display the values afterwards.

The *Live Decoder* also supports live decoding when the *ucitool* is used to communicate with the *SR100T* on a Samsung phone. An additional command-line argument needs to be passed to the *Live Decoder*, which we depict in the user guide in Appendix A.6. Thereby, equivalent methods are hooked in *akash* instead of the *UWB HAL service*, which we describe more precisely in Section 7.2.

Another *Live Decoder*'s feature is the manipulation of transferred messages between host and chip. Thereby, the user can pass the exact manipulations as command-line arguments. The messages to manipulate are identified by the header. Also, the manipulation of message parts is possible by defining the index at which the message should be manipulated. With this feature, quick attacks can be generated that are directed towards the *SR100T* or the UWB services. In Appendix A.6, the user guide explains how to use this feature.

7.1.2.2 *Log Parser*

We implement an additional helper tool that can parse log files and extract the logged communication with the corresponding UWB chip. The tool generates a hexdump file from the parsed data, which can be reviewed at any time with our Wireshark dissector. We name this tool *Log Parser*, and we provide a user guide in Appendix A.7. Furthermore, the tool parses logs that we can retrieve with Logcat on our Samsung phone, and it further parses log files that are written by certain methods in the UWB API of the MK UWB kits. For example, as we later show in Section 8.8.1, it is possible to retrieve logs over Universal Asynchronous Receiver-Transmitter (UART) access on the SmartTag+'s *QN9090*. The methods of the UWB kits write to these logs, and our tool can extract the communication from them.

7.1.3 *Limitations and Workarounds*

We encountered some problems during our implementation. Additionally, our dissector and the tools have a few shortcomings that only exist when the dissector is used to decode the communication without using the *Live Decoder* or a generated hexdump from it. Next, we depict the three most relevant ones.

We first wrote many selected UCI decoders of our dissector by hand because we detected the *ucitool*'s YAML file in a later instance. As part of this work, we also analyzed

payload structures for many UCI messages. However, when we found the *ucitool*'s YAML file, we used this file for generating our UCI decoders and discarded our previous written UCI decoders because we assume the YAML file contains the complete UCI specification. This means some of our previous work got obsolete. Nevertheless, our work in reversing the payloads for implementing the decoders was still valuable. We learned important information, such as the differences in interpreting the returned ranging data. In addition, we still use our implemented ranging data decoder since a manual implementation is required to decode the ranging data for all chips and firmware versions. Also, the ranging data decoder's implementation was the most complex one.

A shortcoming of our dissector is the UCI and HBCI message detection. A message is identified by the header's first two bytes. Currently, the message detector identifies for certain header bytes a HBCI message, which in theory can also be a UCI message. However, we never detected a UCI message with these certain header bytes in the communication, and we also detected no method in the UWB kits that creates a UCI message with these bytes. A possible solution would be a detection of *UCI mode* and *HBCI mode*, which we described in Section 4.5, such that the message type is clear.

A further shortcoming exists when relying on the *Log Parser* and not using our *Live Decoder* or a hexdump generated by the *Live Decoder*. Methods writing to logs do not log the entire message when the message is large. Instead, only the first 505 bytes are logged. Furthermore, messages related to the local firmware download are not logged when we do not manipulate the read of a specific configuration value as we described in Section 4.2. Even when our *Log Parser* works, it parses only the bytes of messages that are logged. However, we only encounter such large messages for the firmware data, which are also part of the not logged local firmware download messages. Moreover, it is not critical that these messages are not logged since they are the least relevant messages. Using our *Live Decoder* or manually hooking selected methods solves this problem.

7.2 FRIDA SCRIPTS

We decide to use Frida because we can use it to intercept and manipulate methods by hooking any method we want in Samsung's user space. Moreover, with Frida, we do not need to care about low-level process manipulations or app modifications. Instead, we can hook methods by only providing simple information like the method's name or address. Furthermore, we are not limited to apps in .apk format but also can hook service executables or imported libraries. We also can hook methods of different entities in parallel. Additionally, the easy-to-write Frida scripts provide a fast way to create our hooks, and we often can reuse or extend parts of the scripts for a different testing scenario.

We develop many Frida scripts, which are used for different purposes. Most often, they are used to gather information at different instances of Samsung's UWB ecosystem or to test for various vulnerabilities. They also heavily were used when we analyzed Samsung's UWB ecosystem. Furthermore, while we develop more than 60 different Frida scripts in total, the foundation of most scripts are the same, and there exist fewer different foundations than scripts. The scripts often build on these foundations. In most cases, they only introduce minor additions for specific tests. If a script does not build on a foundation, then it often only consists of a simple hook definition of a method.

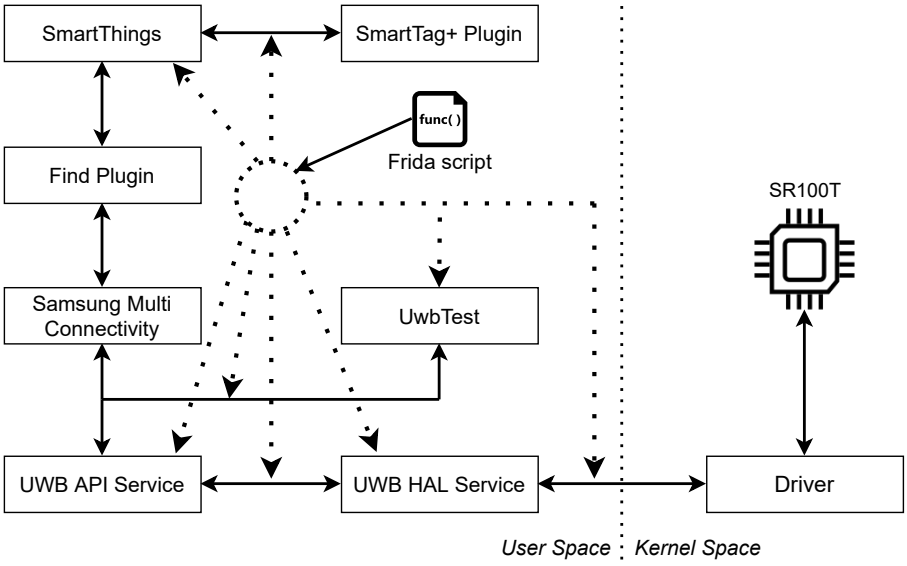

Figure 21: Locations at which we hook with Frida.

In Figure 21, we show an illustration of the user space locations in which we hook methods with Frida to test entities in Samsung's UWB ecosystem. Next, we explain the most important foundation and provide examples of how scripts use this foundation. This foundation hooks methods at the driver's interface, and it also is the most relevant for future research. Afterwards, we briefly explain an additional foundation that can be used to simulate an app using the UWB API of Samsung's UWB ecosystem.

7.2.1 *Hooking of Host-to-SR100T Communication*

Our most important foundation hooks four methods that handle the whole UCI and HBCI communication with the *SR100T*'s driver, which on the other hand just forwards messages between host and chip. Two of the methods write to the driver, and the other two read from the driver. The methods are named *phHbci_PutApdu*, *phHbci_GetApdu*, *phTmlUwb_spi_write*, and *phTmlUwb_spi_read*. Furthermore, the methods are part of the *UWB HAL service*'s library *uwb_uci.helios.so*.

The base foundation hooks the methods to extract the communication. However, an extension of the foundation also enables manipulation in both directions. Thus, messages that are sent to or received from the *SR100T* can be manipulated. Thereby, all types of manipulations are possible. For example, extending, shrinking, or replacing a message is possible. It also is possible to modify only selected bytes of a message or to do only modifications if certain conditions are true.

Additionally, the foundation can be used to hook the same methods in *akash* of the *ucitool*, which is similar to the *UWB HAL service*. Because function names of *akash* are stripped, we only can hook these methods by using the functions' addresses. We easily can find these addresses by searching in *akash* for unique strings that are also used by the previously named methods of *uwb_uci.helios.so*. After inserting the addresses at the corresponding locations in the foundation, we have support for the methods of *akash*.

7.2.1.1 *Example Usage*

Now, we delineate selected examples of how we use and extend the foundation to test for different vulnerabilities.

We use the base foundation in one Frida script of the *Live Decoder*, which we previously explained. The script extends the base foundation to enable the communication with the Python main script and to manipulate selected messages that we can define over the *Live Decoder*'s command-line arguments.

Multiple other Frida scripts use the base foundation for vulnerability testing of the *SR100T*. In these scripts, each of the four methods has an extension to enable manipulations of messages under defined conditions. Besides the *write* methods, the *read* methods also can manipulate messages because in some cases, we need to mock a specific response after manipulating a message that is sent to the *SR100T*. Moreover, the manipulation extensions are different from the ones of the *Live Decoder*. Additionally, we point out that an alternative for testing the *SR100T* with Frida scripts is to use the *ucitool*, and we also use it for the latest tests.

We further use the base foundation with the manipulation extensions when evaluating the local firmware download process. For example, we manipulate single bytes of the firmware transferred to the *SR100T* for evaluating the responses that have informative status codes, which helps us identifying selected bytes as we describe in Section 8.3.1.

We also use the foundation with manipulation extensions to test for vulnerabilities of Samsung's UWB services. Thereby, we simulate attacks that come from a compromised *SR100T*. This works by manipulating reads from the driver.

7.2.2 *App Simulator*

We can simulate an app that uses the *UWB API service*. For this, we write a Frida script foundation that creates an instance for a *UwbTest* app's certain class. After creating the instance, we call the instance's setup method, which creates a connection to the *UWB API service* using the framework jar libraries. Then, we can use the instance's global variables to call all of the *UWB API service*'s exported methods. Furthermore, we point out that this app needs to be manually opened first using the key sequence **#8928378#* in the call app.

With this foundation, we can test for vulnerabilities in Samsung's UWB services, the driver, and the *SR100T*. Depending on the API's called method, we have full control over messages that are sent to the chip through the other entities. For example, we can call the API's method *sendRawUci*, which forwards our chosen byte array up to the chip without validity checks of the content.

7.3 UCITOOL MODIFICATIONS AND SCRIPTS

In Section 4.1.2.2 we described the *ucitool*. Apart from the useful UCI specification, we also can use the *ucitool* to quickly build UWB applications and more importantly, to attack the *SR100T*. It provides an API with selected methods that can be used to send predefined UCI messages to the *SR100T*. The *ucitool* and the helper binary *akash* handle the connection with the *SR100T* and all background tasks like sending the firmware to the *SR100T*. Hereby, no entity of Samsung's user space is used.

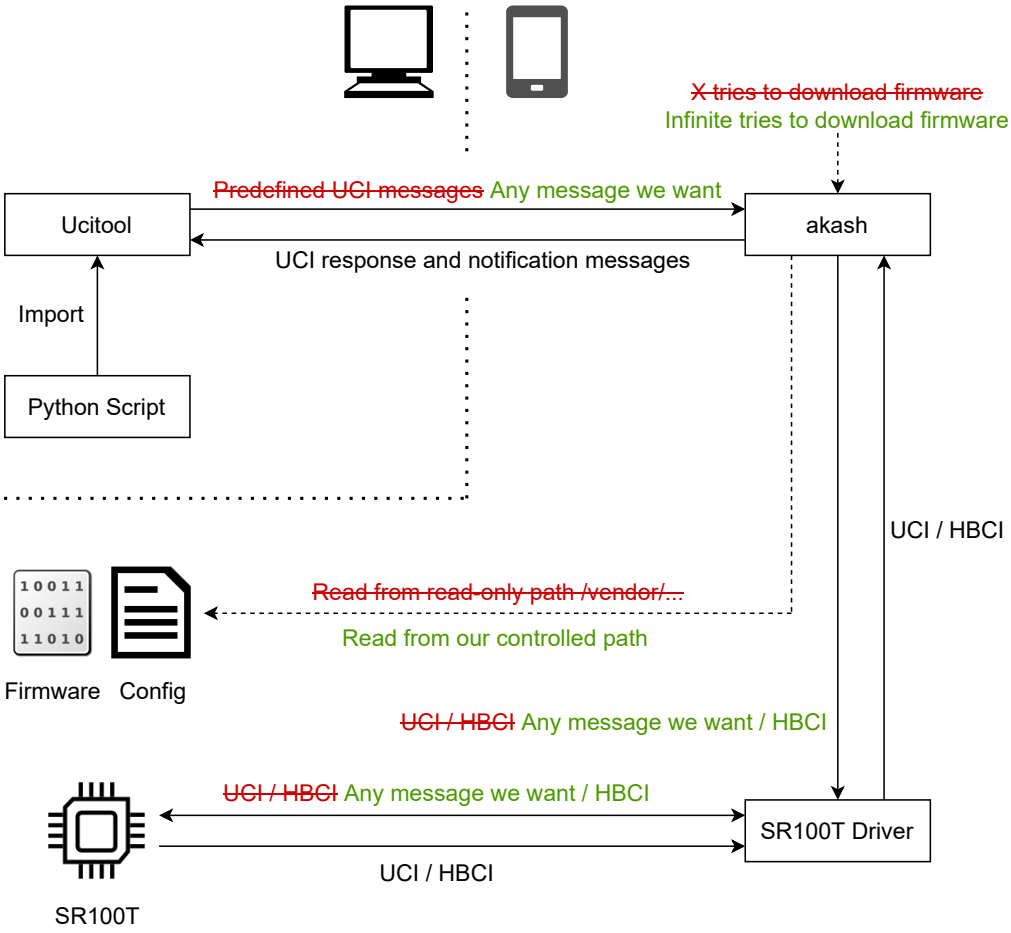

Figure 22: Workflow and modifications of *ucitool* components.

To use the API, we need to write a Python script that imports and uses the *ucitool*'s classes and methods. Furthermore, it is essential to repeatedly kill both of Samsung's UWB services. Otherwise, these interfere with *akash* that runs on the phone. The services are automatically restarted after termination. Therefore, we recommend using a script that kills them every half second or less. We provide a shell script named *killall.sh*, which does the repetitive termination of the services in a short time frame.

Currently, the *ucitool* is limited to sending only predefined UCI messages. Therefore, we make some simple modifications to the *ucitool* and *akash*. In Figure 22, we show an overview of the workflow when using the *ucitool*, and we also indicate the modifications we do. With the *ucitool*'s modification, we can fully define any message sent to the *SR100T*, and we can quickly write and send these using only one line of code per message. Furthermore, the *akash* modifications help us in a later step when we tamper with the local firmware download process or do tests for different firmware versions.

Even with our modifications, we only can send any messages we want after the chip goes into *UCI mode*, which we described in Section 4.5. The reason is that *akash* completely handles all HBCI messages including the local firmware download, and before any message from the *ucitool* is forwarded to the chip, *akash* transfers the chip into *UCI mode*. However, we already have Frida scripts that can write arbitrary messages to the chip in *HBCI mode* by using Samsung's *UWB HAL service*, which is very similar to

akash. Therefore, we adapt these scripts for *akash* instead of making modifications in *akash*, which would cost more time because the script adaptations consist only of exchanging a few lines with addresses.

Next, we depict our modifications. Afterwards, we present examples of how we make use of the modified *ucitool*.

7.3.1 *Modifications*

In the *ucitool*'s Python code we only add two small methods at the API interface, which accept raw bytes. We add these methods in *UciHost* located in *uci.py*, and using these methods does not change the regular API workflow. Moreover, we can call these methods in a Python script that imports the *ucitool*. As a result, we can send any message we want to the *SR100T*. The messages are not checked for validity in between.

Furthermore, the existing API interface methods always print the sent UCI message in a decoded form to the terminal. Our added methods do not print them by default, and to print the messages, *doprint=True* needs to be passed as the optional second argument to the method's call. We decided us for this behavior because our added methods are intended for testing, which includes sending long and malformed UCI messages. These often are not decodable and only result in unclear and superfluous terminal outputs. Thus, a user can choose which messages should be displayed in a decoded form in the terminal output.

We use Ghidra for disassembling *akash* and for instruction patching. Furthermore, since we already can send arbitrary UCI and HBCI messages with the *ucitool* modifications and Frida scripts, we only target modifications that help us for new tests.

We take three modifications. The modifications we take can be equally done in the *UWB HAL service*'s corresponding library because of the high similarity to *akash*. However, since all of Samsung's UWB services and libraries are in read-only partitions on the phone and protected by the operating system, kernel modifications are required. Furthermore, the first two modification's results also can be achieved with Frida, however, with much more work and less flexibility.

Our first modification changes two paths from which *akash* reads the firmware and configuration files. Normally, the firmware and configuration files are located in a read-only partition on the phone. Therefore, we change both paths to point to the same path we control and in which we can write and modify files. We chose the path */data/uwbb*. In this path, we store the firmware and the configuration files.

We can change any file in this path as we want, and *akash* reads it from this path. Only the filename needs to stay the same. By being able to change the file contents, different testing possibilities emerge. For example, we can test the impact of specific configuration values by changing these. Some of these configuration values are directly used to configure the *SR100T*. We also can modify the read firmware, or we can try to replace the firmware that *akash* reads with an older firmware to test if firmware downgrade attacks are possible.

Our second modification is a change of the firmware's filename to *libsr100t_chosen_fw.bin*. As a result, *akash* reads the firmware with this name from our controlled path. This modification is not essential, and we only do the modification to avoid confusion when handling different firmware files.

INFORMATION ID	VALUE
Firmware version byte sequence	0x211000
Firmware UCI version byte sequence	0x0110
Firmware MD5-hash	FFB9B3459D7FB3A4B2E6D7723ED2B869
Phone Android version	11
Phone security patch level	2021-08-01
Phone firmware version	G998BXXU3AUGM

Table 5: Information about the firmware that we retrieved from our Samsung phone and is targeted by our *ucitool* scripts.

The third modification patches only a single byte of an instruction. When *akash* downloads locally the firmware to the *SR100T*, it increments a counter after each failed try and terminates if the counter is equal to a specific number of allowed tries. However, we need to circumvent this check for our firmware download brute-force attacks, which we will describe in Section 8.3.1. Therefore, we patch the instruction that increments the counter such that only zero is added to the counter, which effectively circumvents the check for failed tries. This is the fastest and most effective way to achieve unlimited local firmware download tries.

7.3.2 *Example Scripts*

We develop several Python scripts that use the *ucitool* to communicate with the *SR100T*. Most of them are based on existing test scripts, and some are a good foundation for future work.

Moreover, we write our scripts for a certain *SR100T* firmware that we find on our device in */vendor/firmware/uwb*. In Table 5, we point out the essential information about the firmware and the phone's image from which we retrieved it. The firmware version has the byte sequence of *0x211000*, which is interpreted as *21.10.0* by Samsung but may also be *33.16.0*. Additionally, the firmware's UCI version is *0x0110*, which is interpreted as *1.10* by Samsung but may also be *1.16*. The firmware and UCI version can be retrieved from the chip using the UCI command *GET_DEV_INFO*.

For later firmware versions, some of the scripts may not work as expected when significant changes are introduced to the UCI specification. If a script does not work because of this reason, we recommend using our modified *akash* binary that reads and downloads locally the firmware version *21.10.0*/*33.16.0* from our controlled path. We further recommend using the same firmware version when doing tests with the *ucitool* between two or more phones. Next, we present our five most important scripts, and in Table 6, we give an overview of the test scenarios targeted by our scripts.

OPCODE_DETECTOR.PY The first script sends UCI command messages by iterating through all possible *Group Identifier (GID)* and *Opcode Identifier (OID)* values, even when these are not defined. The goal is to find new messages that are not defined in the YAML file or in the UWB kits. Since the *SR100T* returns an error value when a *GID* does not

SCRIPT NAME	TEST SCENARIO
opcode_detector.py	UCI opcode detection
simple_fuzz.py	Simple fuzzing
ranging.py	Ranging establishment between two phones using *ucitool*
ranging_one_phone.py	Ranging establishment on one phone using *ucitool*
per_phone_data.py	UWB data transfer between two phones using *ucitool*

Table 6: Test scenarios of our most essential *ucitool* scripts.

exist, or when for the corresponding *GID* the *OID* does not exist, we can learn new GIDs or OIDs when no error code is returned.

SIMPLE_FUZZ.PY Our second script tests the *SR100T* for stability. It sends large amounts of differently sized messages independently of the chip's state. The messages are both UCI conform and unconform. Our script only implements a simple fuzzer, and we delegate testing with a sophisticated fuzzer to future work.

RANGING.PY The third script opens a ranging session between two phones that are connected to our PC and run *akash*. Since ranging is currently the primary application for UWB usage in phones, it is interesting for attacks. For example, we use the script to test for attacks while a ranging session is done between phones. After the ranging session establishment, the script begins to send the attack data to the chip, which is derived from the second script.

RANGING_ONE_PHONE.PY Our fourth script is derived from the third script and opens a ranging session on one phone. The ranging partner can be any chosen device and no *ucitool* needs to be used. For example, one phone runs our ucitool ranging script, and the other phone runs the *UwbTest* app.

PER_PHONE_DATA.PY NXP's UWB chips support data transfer over UWB. While we also find the corresponding API methods for UWB data transfer in the *UWB API service*, we detect no app using it. This is also the case for the file sharing apps *Nearby Share* and *Quick Share*. We assume this will be used in the future.

Therefore, our fifth script creates a UWB data exchange connection between two phones by using the *ucitool*. On both phones runs *akash*, and the data transfer is entirely done over UWB. Furthermore, one phone builds the transmitter and the other the receiver, and the script can be slightly modified to enable data transfer in both directions.

The data transfer we establish is one of five ways a phone can control data sent to and processed by another phone. Three of the ways can be achieved by establishing a test session between the phones and by using specific test UCI commands. The last possibility to transfer data over UWB presumably can be done during a ranging session using the *BLINK_DATA_TX* UCI command. We do not test the data transfer commands using UCI test commands and delegate it to future work. Moreover, we could not successfully

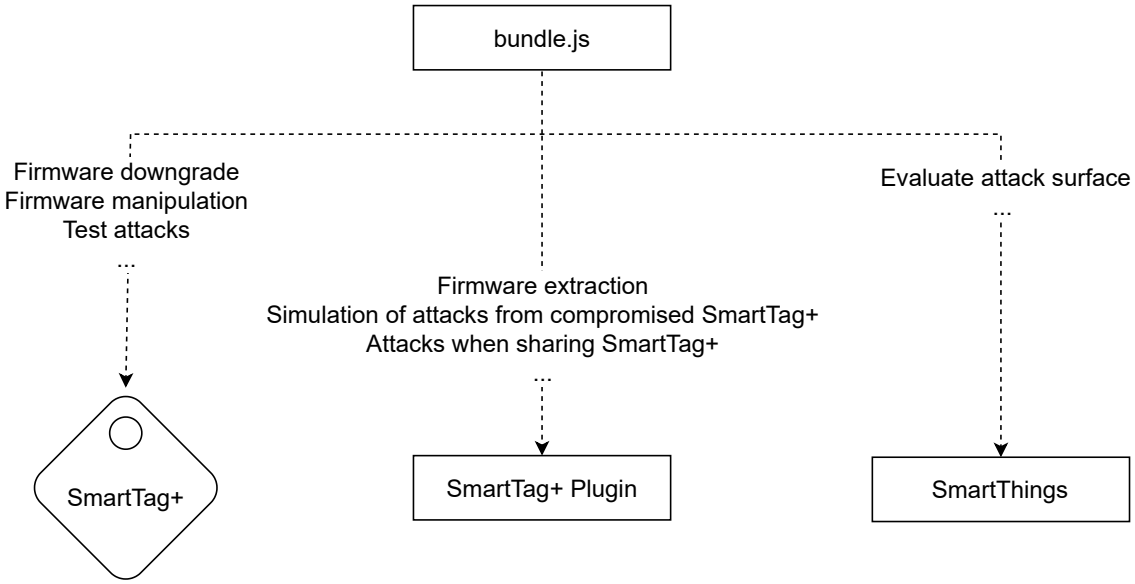

Figure 23: Testing capabilities with modified *bundle.js*.

exchange data using the *BLINK_DATA_TX* command and also delegate a working version to future work.

By having control of the data that is exchanged over UWB between phones, an attacker can attack the receiver's UWB chip as well as all other entities of the UWB ecosystem. This fact can be used to try attacks for found vulnerabilities in entities.

7.4 SMARTTAG+ PLUGIN MODIFICATIONS

The SmartTag+'s plugin easily can be modified since it is a web plugin which files are writable by the *root* user on our phone. The plugin is interesting for us because it contains all SmartTag+ management logic, and it handles the majority of the data sent to the SmartTag+. For example, the main JavaScript file stores the ringtone data sent to the SmartTag+ when the user wants to update the ringtone. We simply can exchange the ringtone data in the JavaScript file with our chosen data and save the file. Then, the plugin transfers our chosen ringtone data, which lets us completely control the SmartTag+'s ringtone.

We modify the SmartTag+ plugin's files to do several security tests efficiently. Furthermore, we take the majority of modifications in the plugin's main JavaScript file, which contains the most interesting operations. The file is called *bundle.js*, and we illustrate the test surface we can achieve with our modifications in Figure 23.

We have different goals when modifying the plugin, including learning if a firmware manipulation or downgrade is possible. Thus, we create a modified version of *bundle.js* first, which serves as a foundation for the firmware transfer attacks. This foundation manually triggers the Over-The-Air (OTA) firmware update process every time the plugin is started. On that foundation, we create our firmware transfer attacks, which modify the firmware that is about to be transferred. In the attacks, we modify selected bytes of the firmware or replace the complete firmware with an old version. Additionally, we add

code that extracts the new firmware versions sent to the SmartTag+ in the regular OTA firmware update process.

We also want to test if we can attack the SmartTag+ with other data that we can control. For this, we take multiple minor modifications at different locations in *bundle.js*. For example, we modify a length field of meta-data information sent to the SmartTag+ before the firmware data is sent. The modified length field indicates the firmware version string's size that follows the field. Here, we decrease the length indication but keep following data's size.

Furthermore, we want to test the plugin's general security and if a compromised SmartTag+ can attack the plugin. Therefore, we again take minor modifications in *bundle.js* to test for different vulnerabilities and partly to simulate attacks that come from a compromised SmartTag+. For example, we modify the firmware version that the SmartTag+ sends our phone, which is subsequently displayed in the plugin. Here, we inject JavaScript code to test for Cross-Site Scripting (XSS) vulnerabilities. Additionally, it is possible to share access to the SmartTag+ over the SmartThings app. A member has limited access rights for managing the SmartTag+ in the SmartTag+ plugin, and the owner has full access rights. Here, we also briefly check for vulnerabilities that can be attacked between members.

We further take minor modifications to test for vulnerabilities in the interface between the plugin and the SmartThings app. Moreover, because of two reasons, we also add code to *bundle.js* that uses the SmartThings app's JavaScript interface. First, we want to test for vulnerabilities. Second, we want to learn how much essential code of the SmartThings app the SmartTag+ plugin can access. This helps us learning the impact that a security vulnerability in the plugin can have. For example, we add code to *bundle.js* that tries retrieving the phone's location or that tries turning the phone's Bluetooth on and off.

7.5 SMARTTAG+ PCB ACCESS

The SmartTag+'s PCB is interesting for us. Over the UART and Serial Wire Debug (SWD) interfaces, we can try to get logs or even debug access. When we can retrieve logs from the PCB, we could evaluate our attacks against the SmartTag+. Moreover, when we can get SWD access, we could manipulate the firmware. Furthermore, we can sniff the Serial Peripheral Interface (SPI) communication between components in order to learn secrets. For all of these tests, the test pads on the PCB facilitate our efforts.

Besides screwdrivers to open and extract the SmartTag+'s PCB, we need to implement a hardware setup that consists of proper tools for our tests. Furthermore, the SmartTag+ PCB's longest diagonal is less than four centimetres small, and the diagonal of a test pad is around one millimetre in size. In addition, the test pads do not have protrusions such as header pins. Also, component pins that are not connected to a test pad can only be accessed directly over the pin. These are also very small. In conclusion, it is hard to solder cables on this small PCB. Thus, we choose tools that also help us to avoid soldering. Next, we present our hardware implementation for our tests, and we show our setup in Figure 24.

We use a Raspberry Pi 3B+ instead of the battery to power the SmartTag+ because it is easier to connect on the open PCB. Thereby, we connect the 3.3 volts pin and the ground pin of the Raspberry Pi to the corresponding battery protrusions of the SmartTag+'s

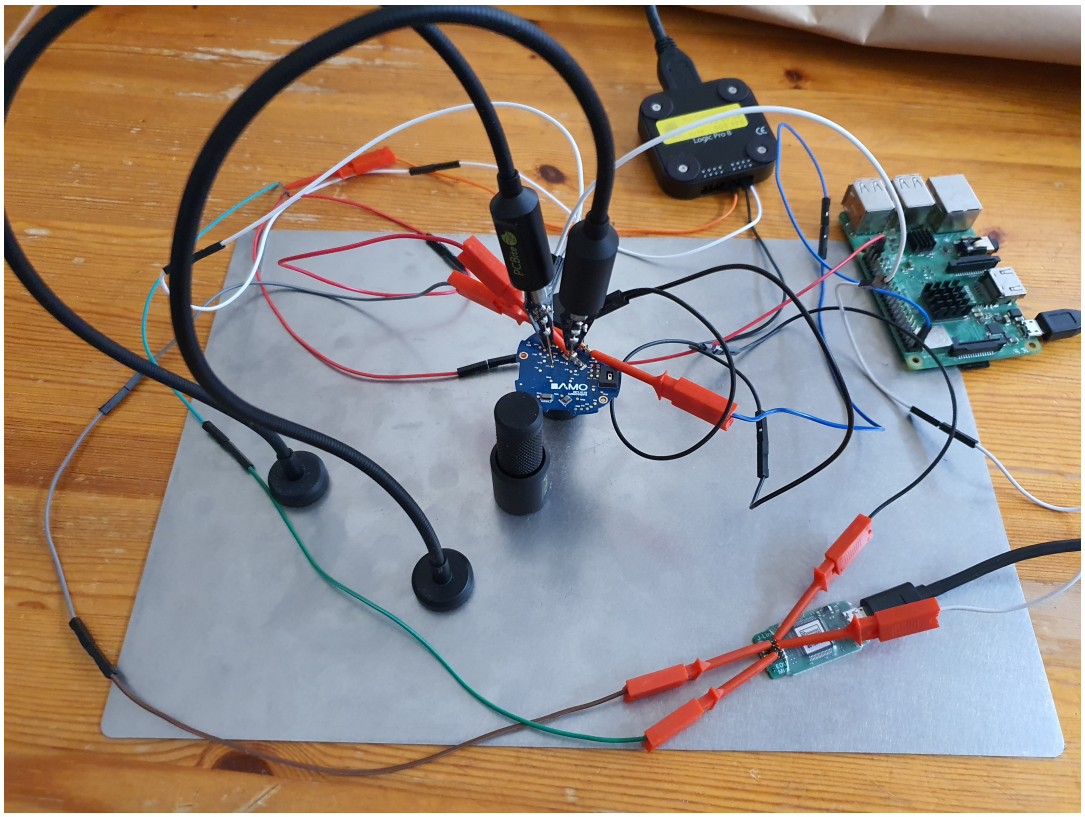

Figure 24: Hardware test setup.

PCB. Moreover, for the battery protrusions connection, we use hook grabbers, which are simple PCB pin grabbers connected to a jumper cable. As an alternative to hook grabbers, we recommend soldering a cable to the protrusions since these are relatively large in comparison to the whole PCB's size.

To solve the problem of accessing test pads without soldering, we use the PCBite set from Sensepeek[1] and two self-made PCBite replicas. The set contains PCB holders and flexible arms with very thin tips that are ideal for connecting to tiny component pins or test pads. We also can connect our jumper cables to the arms.

When testing UART and SPI access, we need a tool that can interpret the signals we retrieve from the connected test pads. We use the Logic Pro 8 as an interpreter, which is a logic analyzer from Saleae[2]. We also use the logic analyzer to record SWD access tries. However, we cannot use it to actively connect to a component over SWD.

For trying to get an SWD connection to a component, we use the Segger J-Link EDU Mini[3].

1 https://sensepeek.com/
2 https://www.saleae.com/
3 https://www.segger.com/products/debug-probes/j-link/models/j-link-edu-mini/

8 EVALUATION

In Chapter 4, we learned about the communication protocols with NXP's Ultra-Wideband (UWB) chips and the *SR100T*'s state machine. We further learned about Samsung's UWB ecosystem entities and the SmartTag+ in Chapter 5. Based on our findings, we identified attack vectors in Chapter 6, of which we made a selection for our following security tests. Afterwards, we implemented several tools in Chapter 7, which help us with our security tests and also can be used for future work.

Now, we evaluate our selected attack vectors of Chapter 6. Our goal is to assess if attacks are possible against selected entities and to make statements about the entities' general security. We do not intend to make an in-depth evaluation of selected entities, but we target results that give us a broad overview of the entities' security. We further aim to provide a starting point for future work with our tests.

ENTITY	RESULT	STATUS
SR100T	Decrypted firmware	- [1]
	Firmware manipulation	- [1]
	Firmware downgrade	+ [2]
	Crash triggering	+ [2]
	Other vulnerabilities	+ [2]
SR100T driver	Vulnerabilities	- [1]
UWB services	SR100T can attack services	+ [2]
	App can attack services	+ [2]
Apps using UWB services	Vulnerabilities	- [1]
SmartTag+ management (entities)	OTA Firmware manipulation	- [1]
	OTA Firmware downgrade	+ [2]
	SmartTag+ can attack plugin	+ [2]
	Attacks between members	+ [2]
SmartTag+ PCB	QN9090 UART read logs	+ [2]
	Full QN9090 SWD access	+ [2]
	SmartTag+ firmware extraction	+ [2]
	SmartTag+ firmware manipulation	+ [2]
	Full SR040 SWD access	- [1]

Table 7: Summary of important evaluation results per entity.
[1] = Unsuccessful attack or no vulnerability found.
[2] = Successful attack or test.

In Table 7, we show a summary of the most important evaluation results. We cannot get the *SR100T*'s decrypted firmware, but we still can identify the header and selected values. We further find a concluded vulnerability in the firmware, which we conclude based on our test results. Furthermore, we find vulnerabilities in Samsung's UWB services that can be attacked from both sides. We also can compromise the SmartTag+ over unblocked Serial Wire Debug (SWD) access and can attack its management plugin over an HTML tag injection vulnerability that leads to Cross-Site Scripting (XSS).

Next, we describe the results of gathering information from the *SR100T*. Subsequently, we present vulnerabilities that we found in the source code provided by NXP of the Mobile Knowledge (MK) UWB kits. We derive attacks against entities of Samsung's UWB ecosystem that use the code provided by NXP, and we provide the results when presenting the corresponding entity. After presenting the source code vulnerabilities, we detail our *SR100T* analysis results. Then, we delineate our results regarding the ecosystem's services and selected apps that use the services. Last, we provide our security analysis results of entities that handle the SmartTag+ on a Samsung phone, and we present our SmartTag+ hardware security analysis results.

In Chapter 9, we discuss our results, and we also briefly provide information about the vulnerability disclosure process of found vulnerabilities.

8.1 UCI AND HBCI INFORMATION GATHERING

First, we test if and how we can get additional information from the *SR100T*. Thereby, we use our knowledge of the Ultra-Wideband Command Interface (UCI) and Host-Based Command/Control Interface (HBCI) specifications and create messages that potentially get us more information from the *SR100T*. We send messages and appraise responses using our implemented *ucitool* and Frida scripts. For example, we brute-force every possible Group Identifier (GID) and Opcode Identifier (OID) with a *ucitool* script to find undeclared opcodes. Moreover, while we apply our tests to the *SR100T*, the test results likely are similar for NXP's other UWB chips.

In Appendix A.8, we give a complete overview of the results since it may be helpful for future work. Next, we briefly summarize the results.

We find seven undeclared OID and GID combinations. By setting the UCI configuration value *DUMP_SE_COMM_DATA* and sending a UCI command with an undeclared OID and GID combination, we get logs that are presumably the exchanged messages between the *SR100T* and the phone's secure element. Additionally, it is possible to enable additional logs from the *SR100T* when doing a ranging session. For example, by enabling one log type, the chip returns data about received UWB frames. Furthermore, we can request some additional information using HBCI queries, including certificate identifiers that are stored on the *SR100T* independently of the firmware.

8.2 UWB KIT VULNERABILITIES

NXP's code in the UWB kits provides an Application Programming Interface (API) that enables the communication with a UWB chip. It creates and processes UCI and HBCI messages, and an app using the API has access to UWB functionality without handling the specifics. Furthermore, NXP's code also is used by Samsung's UWB services,

VULNERABILITY GROUP	GROUP ID	FILE NAME
Reading received messages	1	phTmlUwb.cc
		phNxpUciHal_fwd.cc
Fragment chaining	2	uwb_ucif.cc
UCI payload processing	3	uwb_ucif.cc
		UwbApi_Proprietary_Internal.ccp
		UwbApi_RfTest.ccp
App controlled data processing	4	uci_hmsgs.cc

Table 8: Vulnerability groups and files that contain vulnerabilities of that group.

the SmartTag+'s firmware, and presumably partly by NXP's UWB chips. Therefore, vulnerabilities in the code can affect multiple entities at once.

In this section, we do a careful source code security analysis and focus on vulnerabilities that relate to the UCI and HBCI message processing and creation, which another entity can attack. Thereby, we also use our knowledge of the previous chapters. For the analysis, we use NXP's files that are shaped for the communication with the *SR100T*, but only minor differences exist to the files for the other chips, except for the *SR040*'s chip management protocol Software Update (SWUP), which is different and not in scope of our thesis. Our targeted vulnerabilities are particularly interesting for us because they enable attacks between different entities in Samsung's UWB ecosystem and are potentially far-reaching. Moreover, when doing our analysis, we orient on the *SR100T*'s driver for message size limits. Therefore, we consider a 4200-byte limit for messages to or from a UWB chip. We note found vulnerabilities down and assess these when analyzing entities that use the source code.

In our analysis, we find buffer overflow, integer overflow, and integer underflow vulnerabilities in different source code files. In addition, sometimes vulnerabilities can be combined to increase the amount of attacker-controlled bytes that overflow a buffer. Furthermore, the majority can be attacked from a compromised NXP UWB chip, but we also find vulnerabilities that can be attacked from an external app using NXP's API. Many vulnerabilities are similar, and we can group them into four groups.

Next, we present the vulnerabilities of each group and summarize them. In subsequent sections, we elaborate on these vulnerabilities if we can practically trigger them in entities that use NXP's code. In Table 8, we further show a summary in which source code files we find vulnerabilities of which group. In Appendix A.9, we provide a complete list including method names.

8.2.1 *Reading Received Messages*

The first group of vulnerabilities relate to reading data from the driver's interface, which receives the data from the UWB chip. A compromised chip can attack these vulnerabilities. We find six vulnerabilities, and five are the same but exist in different locations. These five vulnerabilities occur when calling a method that reads HBCI messages. A buffer is passed to this method, and a HBCI response from the chip is stored to the buffer without

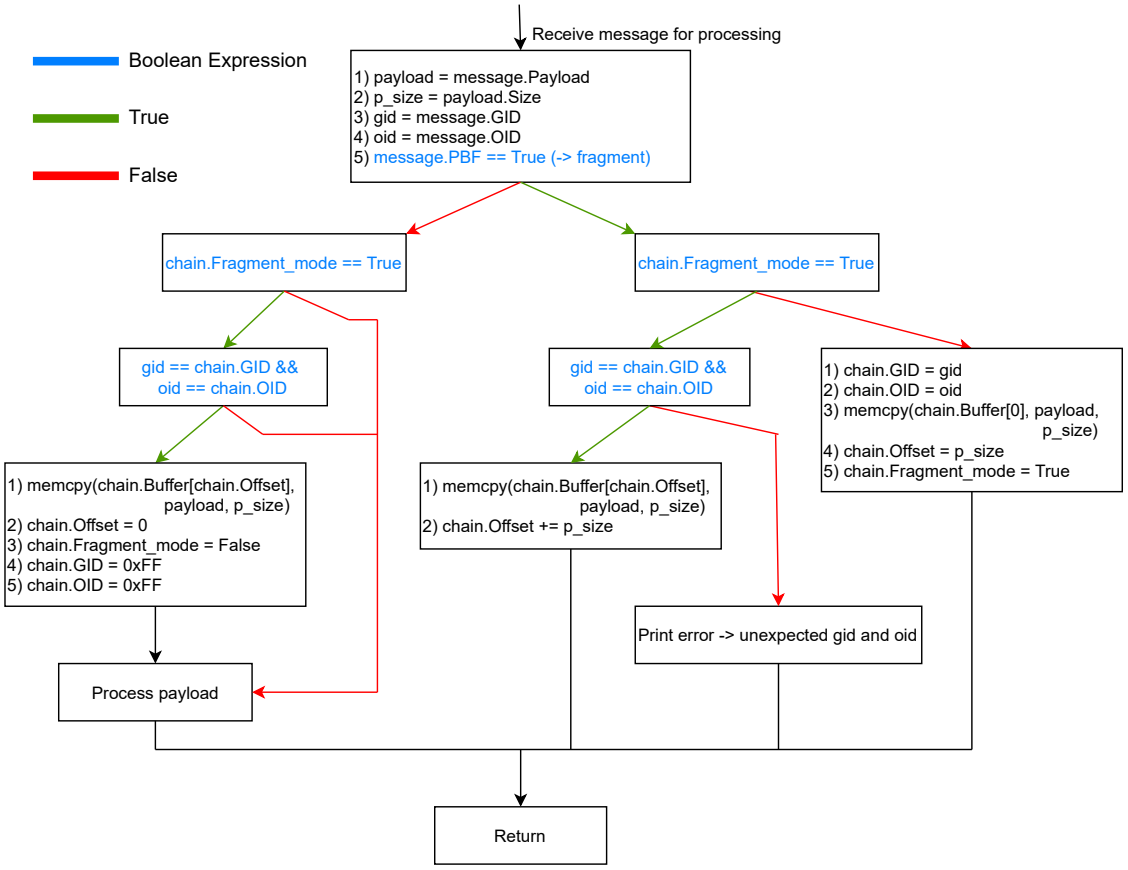

Figure 25: Workflow of processing UCI fragements.

any checks. When considering the maximum allowed message size of 4200 bytes, the buffer can overflow with over 3900 bytes.

The remaining vulnerability works similarly, but in this case, it exists when parsing a UCI message.

8.2.2 *Fragment Chaining*

As we describe in Section 4.2.1.1, a UCI packet can be divided into multiple fragments. Each fragment contains a header, and a message is considered a fragment if the *Packet Boundary Flag (PBF)* is set in the header. We only can order one vulnerability to the second group, and it exists when batching the payloads of fragments in the method *uwb_ucif_process_event*.

We show the workflow of fragment processing in Figure 25. For batching fragments' payloads, there is a global struct used named *chain*. This struct holds a 1024-byte sized buffer for payloads, and the *OID* and *GID* of the fragments follow. It further contains a 16-bit offset indicating the current position in the buffer such that a new fragment's payload can be appended at the right position. This offset value is reset to zero when a non-fragment message is received. The struct's last value is a flag indicating a *fragment*

mode. It is set when the first fragment is received. Additionally, the flag is used to determine if a fragment is the first or a following fragment of a chain of fragments.

When processing an incoming UCI message that is a fragment, the vulnerable source code appends the fragment's payload to the struct's buffer. It further increases the struct's offset value by the payload size until a message is received that is no fragment, meaning it has no *PBF* set. A check if the struct's buffer is full and gets overflowed is missing. Therefore, a compromised UWB chip can send multiple fragments and overflow this buffer with over 64000 controlled bytes. Additionally, if needed and no crash occurs before, the attacker can rewrite the buffer once the offset variable overflows.

When the first fragment of a chain of fragments is received, the struct's *OID* and *GID* are set using the fragment's header values. Following fragments do not modify the values. Moreover, if a following fragment has a different *OID* or *GID* than the struct's corresponding values, an error message is logged. As we describe next, we can use this fact when verifying if this vulnerability exists in entities.

Once the buffer overflows, the struct's *OID* is the first value in memory that an attacker can overwrite. Therefore, if we only overwrite the struct's *OID* with a non-existing *OID* value through the overflow, the processing of the following fragment would lead to printing the error message. Then, we can see the printed error message in our logs and learn that the vulnerability exists in an entity.

The vulnerability can be attacked from a compromised UWB chip. Furthermore, if an attacker remotely can influence fragmentation at the receiver's side, then remote attacks are possible. For example, an attacker controls the data when doing data transfer over UWB. If the attacker can influence fragmentation, the vulnerability can be exploited with complete control over the data.

8.2.3 *UCI Payload Processing*

All vulnerabilities of the third group relate to the payload processing of UCI messages received from a UWB chip. The result of the vulnerability is always a buffer overflow, and we can divide the vulnerabilities into two types.

The first type is a simple buffer overflow when UCI payload contents are copied into a smaller-sized buffer. Except for one occurence, the buffer always overflows with less than 128 bytes. Additionally, an integer underflow vulnerability precedes most buffer overflows of the first type. The combination of the integer underflow with the buffer overflow is the second type.

The integer underflow happens in the copy counter calculation. The copy counter is a 16-bit unsigned integer, and it is used when copying payload bytes to a buffer. It results by subtracting a constant from the payload's size passed as a parameter. The attacker needs to send a small payload of one or two bytes to underflow the integer. Then, the subtraction results in a copy counter of up to 65535 bytes through the integer underflow.

Despite sending a UCI message with a small payload, the attacker still controls bytes after the payload's buffer. The bytes that overflow the buffer are the bytes of previous messages that still exist in memory. Therefore, by carefully choosing the contents of previous messages, a buffer overflow with controlled bytes is possible. We verified this statement by reviewing the memory after the payload's buffer in Samsung's *UWB API service*, which inherits NXP's code.

Again, the vulnerability can be attacked from a compromised UWB chip, and if a remote attacker controls the UCI payload at the vulnerable method, then remote attacks are possible.

8.2.4 *App Controlled Data Processing*

We also find four vulnerabilities that can be attacked from an app interacting with NXP's API. We order them to the last group. Three of these vulnerabilities exist when different UCI messages of the group *TEST* are created. The other one exists when a message for UWB data transfer is created. All four vulnerabilities are caused by the same error, which we will describe next.

Each vulnerable method creates a UCI message containing data from a buffer *B1* passed as a parameter. Additional to *B1*, the size *S* of *B1* is passed as a 16-bit unsigned integer. In the beginning, each method allocates a second buffer *B2*. The size of *B2* is calculated with an addition of three constants and *S*. However, the addition's result is casted to a 16-bit unsigned integer. If an attacker carefully chooses *S*, then the sum exceeds the possible representable size of a 16-bit unsigned integer, and a small number of bytes are allocated for *B2* through the integer overflow. After allocating *B2*, *S* bytes of the attacker-controlled buffer *B1* are appended to *B2*. As a result, an attacker can overflow the buffer with over 65300 bytes of chosen data.

8.3 CHIP ANALYSIS

We do not have an unencrypted firmware version for any of NXP's UWB chips. Therefore, only limited possibilities remain for assessing the chips, and we take specific steps to learn about the firmware and its security. Our goal is to evaluate security against black-box attacks. In addition, we evaluate if the assumption holds that NXP's code of the UWB kits is used in the UWB chip's firmware. Thereby, we attack selected UWB kit source code vulnerabilities.

We use the *SR100T* as a representative for all of NXP's UWB chips since it is part of our Samsung phone, and we already have tools to completely control the communication.

First, we assess the *SR100T*'s local firmware download process. Afterwards, we derive attacks against the chip based on the UWB kits' vulnerabilities and depict our results. Subsequently, we present how we can crash the *SR100T* and the implications of crashes.

8.3.1 *Findings of Firmware Download Analysis*

The local firmware download transfers the encrypted firmware to the *SR100T*. Then, the chip decrypts and executes the firmware. We tamper with this process for evaluating different details and present our results next.

8.3.1.1 *Firmware Header Identification*

After the encrypted firmware is completely transferred to the *SR100T* in the local firmware download process, the *UWB Hardware Abstraction Layer (HAL) service* or respectively *akash* request the transfer status. The response's status byte is expressive and signals one of

ERROR CODE	POSITION IN FIRMWARE FILE
Header parse error	0 - 3 & 72 - 79 & 288 - 291 & 356 - 359
Invalid cipher type crypto	4
Invalid cipher type mode	4
Invalid cipher type hash	4
Invalid cipher type curve	5
Invalid ECC key length	6 - 7
Invalid header signature	8 - 71 & 80 - 287 & 292 - 355 & 360 - 511
Invalid encrypted payload hash	512+

Table 9: Error codes returned by the *SR100T* after receiving a firmware with one modified byte at position X.

25 status codes. In case of a transfer error, the status byte indicates the cause, and most error codes refer to the firmware's unencrypted header.

We use this fact to order firmware bytes to the header and firmware itself. By using a Frida script in combination with our modified *akash* version that allows unlimited firmware transfer tries, we automatically send the regular firmware to the chip and each time decrement a byte at a different position. Thus, we send the X byte sized firmware X times. For the first 512 bytes, we test every possible byte value. Afterwards, we evaluate the responses for each firmware transfer.

In Table 9, we order the firmware's bytes based on the response status codes. Because of the different header-related response codes, we conclude that the firmware's first 512 bytes are the header, which is not encrypted. Moreover, we can decode the meaning of selected header bytes with some error codes. We conclude that Elliptic-Curve Cryptography (ECC) is used for encrypting the firmware. However, we cannot identify the curve. Furthermore, we get different error messages for the fifth byte at position four.

When modifying a byte that follows the header, we always get the same error message: *Invalid_Encrypted_Payload_Hash*. Therefore, we conclude that the following bytes build the actual encrypted firmware that gets decrypted by the chip by evaluating the header.

8.3.1.2 *Production and Development Firmware*

We find on our phone two firmware versions of the *SR100T* named *libsr100t_prod_fw.bin* and *libsr100t_dev_fw.bin*. Both are encrypted, and the first version — the production version — is always used by the *UWB HAL service* and *akash* in the local firmware download process.

We do not find unexpected differences between both versions except for bytes at positions eight and nine, which are zero in the development version. This is also the case for other development firmware we find in the UWB kits and different images of our test phone.

Using our modified *akash* version that allows us to choose the transferred firmware file, we try to transfer the development version to the *SR100T*. We are unsuccessful, and the *SR100T* returns an unknown error code: 0x96. We look in NXP's source code to verify

that we do not need to use different HBCI commands for the development version, which is not the case. Furthermore, we set the bytes at position eight and nine to zero in a valid production firmware and transfer it to the chip. As expected, the chip returns an *Invalid header signature* error.

We conclude that the *SR100T* chip on our phone is a productive build and only allows production firmware executed, which might be realized using a fuse. Yet, we question why the development firmware version is included on the phone.

8.3.1.3 *Accepted Firmware Versions*

As we already described, we can choose the firmware file that our modified *akash* version sends to the *SR100T* in the local firmware download process. We use it to test which firmware is accepted by the chip. Thereby, we transfer all *SR100T* firmware versions we encounter during our thesis. Some of them are included in the UWB kits. We further try to transfer the *SR040*'s and *SR150*'s firmware, which we also retrieve from the kits.

While modified versions or a development firmware are discarded by the *SR100T* on our phone, we successfully can transfer any valid production firmware version of the *SR100T*. Additionally, production firmware versions for the *SR150* are accepted as well but not firmware versions for the *SR040*.

8.3.2 *Black-Box Attacks*

We do not have the decrypted firmware. The only way to verify if the *SR100T* inherits the vulnerable NXP source code is to derive attacks. When we achieve unexpected behavior or crashes of the chips, we can conclude that the vulnerability exists. Yet, we cannot be sure if we always trigger the vulnerability we target. It might be possible that the unexpected behavior or crash have a different source, which may even be another vulnerability.

Apart from derived attacks, we also test for potential vulnerabilities that are independent of NXP's source code. We divide the tests into two attack types. The first type concentrates on the features of UCI and HBCI messages. We mainly focus on specific bytes in the header or payload that can lead to vulnerabilities when parsed or used wrongly. For example, we send valid UCI messages in which we decrease the value of specific length fields in the payload while keeping the data's length same. The second type is fuzzing the *SR100T*, for example, by sending differently sized random data.

Because of our limited time frame, we only test selected messages for the first attack type and only do simple fuzzing for the second type. We delegate sophisticated tests to future work. For sending attack messages, we primarily use our *ucitool* scripts.

We do our tests for three different production firmware versions that we extracted from different builds of our Samsung phones. The firmware versions are represented by the byte sequences *0x211000*, *0x273000*, and *0x300000*, which are interpreted differently by different entities. The last firmware is the latest version, and it is part of the Samsung S21 Ultra image release of January 2022. Moreover, we retrieve the firmware versions using the *GET_DEV_INFO* UCI command.

Furthermore, to detect that we trigger a vulnerability successfully, we rely on two different crash logs that the chip returns. In the case of a crash and the *SR100T* is still able to respond, without considering the header, the *SR100T* sends a 48-byte debug UCI

message. Additionally, we can request from the chip a 232-byte UCI error message, which is done automatically by the *UWB HAL service* and *akash*.

Next, we present the results of our tests regarding inherited NXP code vulnerabilities and both other attack types. Afterwards, we assess crash logs that we get during our testing.

8.3.2.1 *Inherited NXP Code Vulnerabilities*

When appraising if the *SR100T*'s firmware inherits vulnerabilities of NXP's code, we filter for possible vulnerabilities. Then, we test them by sending carefully crafted messages using *ucitool* scripts.

FRAGMENT CHAINING ATTACK We successfully get crashes for all tested firmware versions when consecutively sending two UCI fragments to the *SR100T*. After sending the second fragment, the chip sends a crash notification and a debug log message.

We only get crashes if the fragments use an extended payload length, which is declared by the *Extended (EXT)* flag in the UCI header. For example, we use payload lengths of 256 or 512 bytes. Moreover, we send the same two UCI fragment messages without declaring the *PBF*, thus, making them no fragments but normal UCI messages. The only difference is the unset *PBF* flag, which is signalized by only one bit in the UCI header. When sending these two messages, we get no crash, and the chip continues working as expected. We also do not get a crash when sending more than two of these non-fragment messages. Therefore, we conclude that the fragment chaining vulnerability exists in the *SR100T*'s firmware.

When triggering a crash through the concluded fragment chaining vulnerability, we only get an unexpressive debug log message from the chip for each firmware version. Additionally, the crash log we request afterwards is also unexpressive. Both messages mainly consist of zeros. Nevertheless, at a specific position in each debug log message, a few bytes are not zeros and look like an address. For two of our three firmware versions, this address is the same. The other address is very similar. We assume this is the address at which the chip crashes.

OTHER NXP CODE VULNERABILITIES We cannot trigger any other vulnerability we found in NXP's source code. The chip continues working and returns expected responses despite carefully deriving attacks.

8.3.2.2 *Further Attacks*

Attacks by exploiting UCI or HBCI characteristics are not successful besides attacking the fragment chaining vulnerability. We did not thoroughly test other protocol features because of our limited time frame, and future work can expand on these attacks. Nevertheless, we choose the most likely payload fields that can lead to potential crashes because of a vulnerability. We create multiple different attacks and analyze the responses. Each time, the chip responds as expected, and no crash occurs.

Except for the fragment chaining vulnerability, we are also unsuccessful for attacks exploiting header characteristics. For example, we send UCI messages declaring a 1000-byte sized payload in the header, but the payload consists of 4000 bytes.

Yet, we can trigger unexpected behavior and crashes of the *SR100T* by sending differently sized random data. We achieve this using a simple self-implemented fuzzer in a *ucitool* script. Moreover, we also create valid UCI messages as fuzzing data but only get unexpected behavior or crashes when sending many large messages in a short time frame. Thereby, the fuzzing data's contents do not matter.

UNEXPECTED BEHAVIOUR Unexpected behavior of the *SR100T* comes in the form of responses that are not UCI conform, responses that contain parts of our attack data, or both. For example, when sending consecutively large messages that only consist of the byte 0x41, the chip eventually returns the UCI message *0x690000064141414101 01*, which is a valid UCI message but unrelated to the communication.

CRASHES Except when triggering the fragment chaining vulnerability, we can only crash the chip when sending many large messages in a short time frame, independent of the attack data. The direct result is often undefined behavior, and eventually, the chip crashes. For the three different firmware versions, we observe slightly different undefined behavior due to our attacks. Moreover, we also observe a slightly different tolerance. Yet, the eventual result is the same, namely a crash.

We make a further observation. When crashing the *SR100T* while doing UWB ranging with the SmartTag+, often after the chip crashes and the firmware is transferred to the chip again, the chip returns too large measured distances when doing ranging again. The distance is then always enlarged by around 40 meters, and in our tests we never saw a different deviation.

Furthermore, when crashing the latest firmware through our simple fuzzing attacks, we sometimes see four bytes of our attack data in the requested error log between other byte sequences. An additional byte of our attack data is placed eight bytes before the other four bytes in the error log. We assume the four bytes of our attack data display the value of a register. In the next section, we assess the debug and error logs more closely.

In older firmware versions, we also sometimes see our attack data in the returned debug log sent after the chip's crash notification. However, this happens seldom, unlike the observation in the latest firmware. Moreover, we achieve this only while the chip does UWB ranging and when sending the attack data to the driver's device handle in */dev/sr100* using the Linux *echo* command. We never observed a debug log containing our attack data when using a *ucitool* script to establish a ranging session and attacking the chip during the ongoing ranging session. Besides the ongoing ranging session condition for triggering a crash, we assume sending data directly over the driver speeds the sending rate, which is needed to trigger a crash while ranging.

8.3.3 *Crash Analysis*

Two types of crash-related logs exist. We do a close inspection of the UWB kits and all related entities in Samsung's UWB ecosystem, but we do not find the interpretation for any of both crash logs. Therefore, we only can make assumptions about the contents, which we present next.

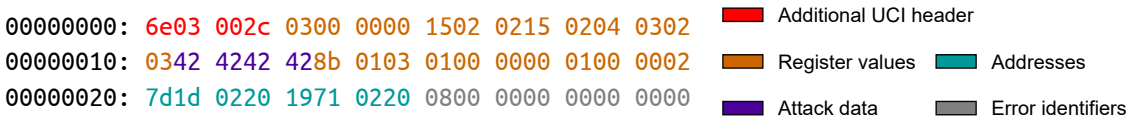

Figure 26: Dump of an encountered debug message and our assumed meaning for each byte.

DEBUG MESSAGE Two types of crashes exist for the *SR100T* in *UCI mode*, as we described in Section 4.5. The *SR100T* is still responsive when one of these crash types occurs. Then, the *SR100T* first sends a crash notification, and subsequently an additional debug message. This debug message cannot be requested. In both older firmware versions, the message is 52 bytes in size with a payload of 48 bytes. The latest firmware returns a 12 byte larger message. Next, we only consider the 52-byte message because we only have debug message samples mainly consisting of zeros for the latest firmware.

In Figure 26, we show a dump of the payload from a debug message we received, which contains parts of our attack data. The first four-byte sequence is a UCI header, and it is always part of the debug messages' payload independently of the real header. We mark it in red. Furthermore, the purple marked attack data in the debug log is at a position that does not hold an address in logs that do not contain our attack data. Moreover, in all debug logs, we do not find any address from position 0x0 to 0x1F. We assume these values are register values, which we mark orange. The attack data is also part of the register values.

Marked in blue, the four-byte sequences at positions 0x20 and 0x24 are likely addresses. At 0x24, we always have an address in all debug log messages from older firmware versions. At position 0x2C in a debug log from the latest firmware, the address is very similar to the addresses in the older firmware version's debug messages. Depending on the firmware version, we assume the crash happens at the address displayed in 0x24 or 0x2C.

The last eight bytes always are the same in each debug message. We assume that these are error identifiers.

ERROR MESSAGE After receiving the crash notification, it is possible to request an error message from the chip. When no error occurred before, the chip returns only zeros when requesting it.

The error message consists of 232 bytes, excluding the header. We further differentiate between two types of error messages. The first type consists of many different bytes, and the second type consists mainly of zeros. Next, we only consider the first type.

In Figure 27, we show an example error message we got from the chip after crashing the latest firmware with our simple fuzzing script. It contains four bytes of our attack data at position 0x5C, which we mark in purple. We compare this error message to other error messages from the chip we gathered during our thesis to make the correct conclusions.

All error messages have 4-byte sequences that are likely addresses, share the same address range, and are similar to the debug message's dump in Figure 26. The addresses often are located at different positions in each message. Only the last six 4-byte sequences always store an address, which we mark in blue. We conclude this might be a call stack,

```
00000000: 0200 0000 c880 0320 3002 0000 6c8c 0320
00000010: 0100 0000 92ca 0120 e3c3 0120 f2c3 0120
00000020: 0000 0021 0100 0000 0000 0000 0000 0000
00000030: 0000 0000 0000 0000 0000 0000 0410 0040
00000040: 00e1 00e0 1766 0320 da58 0120 0434 0040
00000050: 1c00 0040 4100 0000 001d 0120 4141 4141
00000060: 640c 0120 ec1c 0120 0000 0000 0000 0000
00000070: 0000 0000 f81c 0120 d01b 0120 0000 0000
00000080: 0000 0000 0000 0000 0000 0000 0000 0000
00000090: 0000 0000 0000 0000 0000 0000 0000 0000
000000a0: b55b 0320 0082 0000 0000 0000 0000 0000
000000b0: 0000 0000 0000 0000 38af 0520 38af 0520
000000c0: 0400 0000 bc01 0010 792d 01d8 0000 0000
000000d0: ea08 014c 2011 0248 8014 020c df01 0108
000000e0: 6008 0340 0005 0120
```

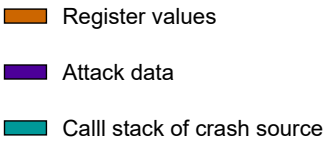

Figure 27: Dump of an encountered error message and our assumed meaning for each byte.

whereby the last address is the error's source. We further assume that the previous bytes display register values since the *SR100T* likely is an ARM Cortex-M33 chip as its two silbings [33, 34], and all of its core register values fit into the error message [2]. We mark them orange, and the attack data is part of the register values.

8.4 SR100T'S DRIVER

We do a careful security analysis of the driver's source code and find no vulnerability. Instead, as we describe in the next section, the driver even prevents many vulnerabilities in Samsung's UWB services because it discards messages that are larger than 4200 bytes in size.

8.5 UWB SERVICES

We consider both the *UWB API service* and *UWB HAL service* as Samsung's UWB services. They build the middleware between external apps and the *SR100T*'s driver, which forwards messages from and to the chip. Moreover, both services make heavy use of NXP's code provided in the UWB kits.

When assessing the security of both services, we target vulnerabilities that can be attacked from both services' sides. These attacks can come from an external app or from the *SR100T*, which can be a compromised *SR100T* or a remote attack that uses the *SR100T* to forward the attack payload. For example, a remote attack might be possible when doing data transfer over UWB.

We focus on the vulnerabilities we found in NXP' source code. Our goal is to find and practically verify at least one vulnerability for each side.

Next, we first describe entities' five behaviors and characteristics in Samsung's UWB ecosystem that prevent many attacks against potential vulnerabilities, including some of

the inherited NXP code vulnerabilities. Except of the driver's message size limitation, we conclude that Samsung implements all vulnerability preventions we describe next. After describing the vulnerability preventions, we present found vulnerabilities.

8.5.1 *Vulnerability Preventions*

The *SR100T*'s driver only forwards messages that do not exceed the 4200-byte limit. Otherwise, the message is discarded. When simulating attacks from a compromised *SR100T*, we are bound to this limit and cannot simulate attacks with larger messages.

Additionally, in Samsung's UWB services, bytes directly read from the driver are always copied to are large enough buffer that meets the driver's size limit. Moreover, UCI payloads of messages received from the *SR100T* are always copied to buffers that also are allocated large enough and meet the driver's size limit. Thus, all vulnerabilities of our defined first group and some of the third group are prevented, which both are based on smaller allocated buffers.

Many vulnerabilities also are prevented on the other side, meaning these that can be attacked from an app using UWB functionality over Samsung's UWB services. Vulnerabilities of the fourth group can be attacked through the integer overflow if an external app can control a Java array that is almost 65336 bytes in size. This array needs to be forwarded to the vulnerable method as well as the size that is calculated by a helper method in between. In addition, some other potential vulnerabilities we find in native code of Samsung's UWB services could be attacked when we can send arrays that are sized larger than 256 bytes in size.

However, in the call chain between the called *UWB API service*'s exported method and the targeted vulnerable method, most times, one of the first methods in the call chain only accepts an 8-byte unsigned integer for the array's size. When this method is called, it results in casting the array's size value to an 8-bit unsigned integer that is smaller than 256. Consequently, in the following methods, the array's 8-bit size value is considered and used for any copy operation, and the vulnerable methods cannot be attacked. Sometimes an array size check is also done, which prevents forwarding arrays larger than 255 bytes.

In conclusion, for attacks, we need to find call chains to vulnerable methods that do not cast the array's size parameter from a 16-bit value to an 8-bit value. We also find one, which we describe after the next section.

8.5.2 *Fragment Chaining Attack*

In the method *uwb_ucif_process_event* of *UWB API service*'s library *libuwb-uci.so*, fragments are chained and UCI headers are processed. To appraise if our previously found fragment chaining vulnerability of NXP's code is inherited, we first do a brief static analysis of *libuwb-uci.so*, and we conclude that the vulnerability might be inherited. Therefore, we test our finding practically and simulate an attack by the *SR100T*. We want to prove that the vulnerability exists and can be attacked. Because of our limited time frame, we aim to verify the vulnerability and not to write an exploit.

To practically trigger the vulnerability, we use one of our Frida scripts that hooks the driver's interface methods of the *UWB HAL service*, and exchanges read UCI messages from the *SR100T* with valid UCI fragment messages. By hooking at this location, we can

ensure we do not skip any validity checks, and the manipulated message passes through all instances until it arrives at the vulnerable method.

Our goal is to trigger the error printing, which normally only happens when a following fragment of a chain of fragements is received with a different *OID* or *GID* than the first fragment. When the first fragment of a chain of fragments is received, these both one-byte values are stored in the same global struct as the buffer that stores all fragment's payloads. In memory, the values follow the buffer directly, and the *OID* follows first.

In our attack, we create a chain of fragments, which all have the same valid *OID* and *GID* declared in the header. The payload is a sequence of the byte 0x41. When the vulnerability exists, we would eventually overflow the buffer and overwrite the buffer's following values in the struct with the byte 0x41. After we overflow the buffer and overwrite at least the *OID*, the vulnerable code would print the error message when processing the following received fragment since the *OID* and *GID* are both different than the byte 0x41.

We successfully trigger the overflow and see in the logs the printed error. We learn that the overflowed buffer's size is 4192 bytes instead of 1024. The rest turns out as expected. In conclusion, a compromised *SR100T* can attack Samsung's UWB services. Moreover, remote attacks might be possible if an attacker can control the creation of fragments and at least parts of the fragments' payloads.

8.5.3 *App Controlled Data Processing Attacks*

The evaluation of vulnerabilities attackable by external apps is a three-step process. First, we need to find methods doing operations that potentially can be attacked by an app using exposed *UWB API service*'s methods. For example, we filter out methods in native code that copy *X* bytes of a buffer *A* to another buffer *B*, whereby the contents of buffer *A* are controlled by us. Sometimes C's *memcpy* function is used to copy bytes, but the copy operations are often directly done without using a helper method. Second, we need to identify all exposed *UWB API service*'s methods that forward our data to the corresponding potential vulnerable methods. Last, we check if our attack data gets forwarded as expected to the vulnerable method, and if positive, we test the attack by trying to trigger a crash.

We further target vulnerabilities that can be attacked with one method call exclusive calling setup methods. Moreover, we give the highest priority to the methods in Samsung's code equal to the vulnerable methods in NXP's source code, which can be attacked by an app using NXP's code.

Because of the limitations regarding casting the buffer's size parameter to an 8-bit value in specific methods, we fail most often in the third step for methods identified in the first step. Nevertheless, we find one call chain that allows using an exposed API method to send a large buffer for attacking a vulnerable method successfully, which is one of the identified ones in NXP's source code. The vulnerable method is *UWA_PerRxTest* of the *UWB API service*'s library *libuwb-uci.so*. In NXP's source code the same method is named *uci_snd_test_per_rx_cmd*.

For our attack, we use our Frida script that uses the *UwbTest* app's methods to establish a connection to the *UWB API service*. The *UwbTest* app is independent of Samsung's UWB services and is used as our external app. In Figure 28, we show an overview of the call

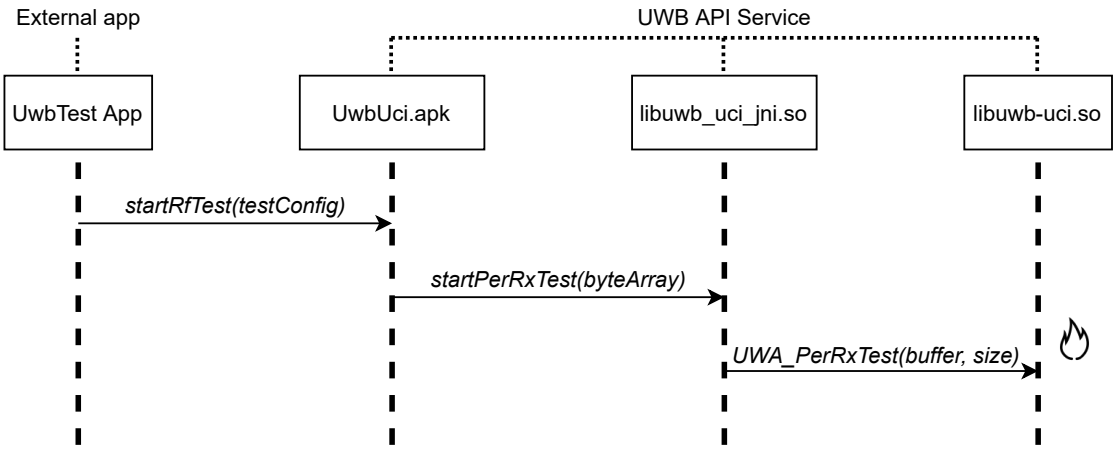

Figure 28: Overview of the call chain's important methods and arguments used to attack a vulnerable method in Samsung's *UWB API service*.

chain to attack *UWA_PerRxTest*. We only mark essential arguments. Next, we describe the workflow and only highlight important methods and their behavior. We also only name essential arguments.

After creating the connection to the *UWB API service* in the Frida script, we establish the attack by calling the exposed *startRfTest(testConfig)* method. We pass an instance of a configuration class as a parameter. The created instance holds a byte array defined by us, which gets extracted from the instance and forwarded to *startPerRxTest(byteArray)*.

The method *startPerRxTest(byteArray)* is in the *UWB API service*'s library *libuwb_uci_jni.so* in native code. It creates a buffer from the array and calculates the buffer's size using the array. Both values are passed to *UWA_PerRxTest(buffer, size)* of the library *libuwb-uci.so*.

In *UWA_PerRxTest(buffer, size)* the size of our array gets added to a constant that holds the value 0x18. The result is stored in a 16-bit unsigned integer variable. Thus, for example, when sending an array sized 65514 bytes, the addition's result is 65538, which a 16-bit unsigned integer cannot represent. Instead, the result value overflows and holds the value two, meaning we have an integer overflow.

The problem is that the result value is used to allocate space for a second buffer to which the passed buffer is copied afterwards. The passed size argument determines the number of bytes to copy from the passed buffer to the second buffer. Therefore, when the integer overflow occurs, a buffer overflow follows.

We can trigger the buffer overflow successfully using our Frida script, which results in a crash of the *UWB API service*. In the logs, we also get a crash dump.

8.5.4 *Other Vulnerabilities*

We did not test practical attacks regarding the vulnerabilities of group three. Yet, using Ghidra, we discover that the vulnerabilities exist and likely can be attacked. Future work can practically demonstrate attacks.

Because of our limited time frame, we do no thorough tests for other vulnerabilities, and we have no findings for tests we take. We delegate further testing to future work, and the next step for future work can be to evaluate remote attacks against services.

8.6 API APPS

We do a security analysis of the *UwbTest* app's and the Samsung Multi Connectivity app's UWB-related parts. Both parts only consist of a few classes that use the *UWB API service* over the framework libraries, and we do not have any noteworthy findings.

8.7 SMARTTAG+'S MANAGEMENT ENTITIES

We assess the security of different entities that are responsible for managing the SmartTag+. Thereby, we focus on attacks that a compromised SmartTag+ can exploit and remote attacks against the SmartTag+ that these entities can establish. We also check if attacks are possible from lower entities of Samsung's UWB ecosystem.

Furthermore, all of our tests for SmartTag+ vulnerabilities are established through the SmartTag+ plugin since here the most data sent to the SmartTag+ is handled, and it is easy to manipulate. We use our manipulated versions of the plugin's main JavaScript file named *bundle.js*, which manipulations we presented in Section 7.4.

First, we depict our results for attacks against the SmartTag+. Afterwards, we come to the results regarding the entities' security.

8.7.1 *Remote Attacks Against the SmartTag+*

First, we test firmware attacks. Our goal is to learn if we can downgrade or manipulate the firmware Over-The-Air (OTA). Therefore, using one of our modified *bundle.js* files, we trigger the OTA firmware update process and inject our firmware. In Figure 29, we illustrate the manipulated workflow of *bundle.js*. Our modifications trigger the firmware update process after the plugin is started and inject our chosen firmware at the right location before it is sent to the SmartTag+. The update process is triggered by a manipulated line in *bundle.js*, which always returns a non-existing old firmware version to the method that checks if the SmartTag+ runs the current firmware.

8.7.1.1 *Firmware Downgrade*

We gather multiple firmware versions of the SmartTag+ during our work, and we test firmware downgrades with each older version. Each firmware downgrade is successful, and we can downgrade the SmartTag+'s firmware to the oldest version we have, which has the version 0.50.30. The current version is 1.01.04 in January 2022.

Additionally, we point out that after each successful firmware downgrade, we first do a regular OTA firmware update to the latest version before we test the next older version. Furthermore, we can quickly check our success in the SmartTag+ plugin's *Information* menu. Here, the plugin displays the SmartTag+'s firmware version, which the SmartTag+ sends to our plugin.

8.7.1.2 *Firmware Manipulation*

To test if firmware manipulations with the OTA firmware update process are possible, we manipulate single bytes of different firmware versions and test if the SmartTag+ accepts these. We further take our tests while the SmartTag+ runs different firmware versions,

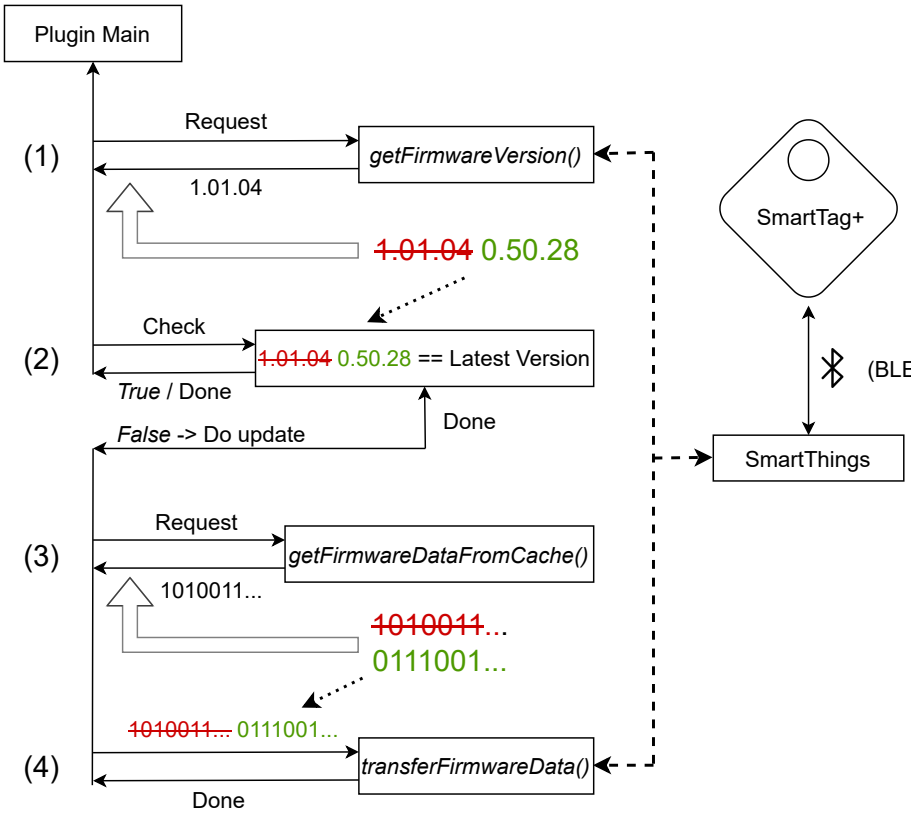

Figure 29: Modifications of the SmartTag+'s plugin to trigger the firmware update mechanism and inject a custom firmware.

whereby we use the firmware downgrade attack to choose the SmartTag+'s current firmware. Unfortunately, the SmartTag+ does not accept any manipulated firmware, and after the firmware is transferred, the SmartTag+ returns a single error code, which does not explain the failure reason. We further validate that we make no error and closely analyze the code of *bundle.js* that is responsible for the firmware update. For example, the plugin sends Cyclic Redundancy Check (CRC) values together with each firmware chunk and one time for the whole firmware, and we check that these values are correct for our manipulated firmware.

In the SmartTag+'s Universal Asynchronous Receiver-Transmitter (UART) logs retrieved over Printed Circuit Board (PCB) access, several messages indicate that a signature check fails for the manipulated firmware we transfer, and a certificate stored on the SmartTag+ is used to verify the signature. Therefore, we conclude that signature checks are correctly implemented and prevent OTA firmware manipulations.

8.7.1.3 *Other attacks*

We also use our modified *bundle.js* files to exchange the contents of selected messages that are sent to the SmartTag+. For example, we considerably increase the size of these messages to test for crashes and similar. Thereby, we simultaneously observe the SmartTag+'s UART logs. While doing this, we achieve crashes and undefined behavior. For example, after sending a large string instead of the firmware meta-data information that

is sent in the firmware update process, we have to reset the SmartTag+. Unfortunately, the UART logs do not explain the crashes.

We conclude that there are a lot of undiscovered vulnerabilities. Moreover, also NXP's source code vulnerabilities might be inherited since some parts of the SmartTag+'s firmware use this code. However, a closer analysis is out of scope because of our limited time frame. Therefore, we continue with our analysis at this point, and delegate the firmware analysis to future work.

8.7.2 *Security of SmartTag+'s Management Entities*

When evaluating the management entities, we focus on vulnerabilities that directly correlate to handling data received from the SmartTag+. We also briefly appraise the attack surface between entities and look for vulnerabilities that can be attacked from another entity. Moreover, access to the SmartTag+ can be shared, and a member only can find the SmartTag+ and change the ringtone. We also briefly assess if attacks between members are possible.

We find two vulnerabilities in the SmartTag+ plugin and no vulnerability in the SmartThings app that is related to handling the SmartTag+.

8.7.2.1 *Cross-Site Scripting*

The first finding is an HTML injection vulnerability, which leads to XSS. For specific displayed values, we can inject these HTML tags. Furthermore, script tags are sanitized by the plugin, but JavaScript in HTML tags not. Therefore, for example, we can inject an HTML image tag and JavaScript into the image tag's *onerror* attribute.

Only locations at which JavaScript can be injected are important for us. To identify vulnerable locations in the plugin, we use this payload: *<img src="a" onerror="alert(1)">*, which results in a pop-up as in Figure 30. Thereby, we identify three locations, and the foundation of each vulnerability is that attacker-controlled data is dynamically appended to the HTML Document Object Model (DOM). Next, we depict these locations.

The first identified location is the SmartTag+'s name field, which can be changed by the SmartTag+'s owner or by any member when using the vulnerability described in Section 8.7.2.2. As a result, attacker-controlled JavaScript is executed when a member opens the plugin.

The second identified location is the selected ringtone's name. An attacker with a superuser-enabled phone can manipulate the ringtone data sent to the SmartTag+. Additionally, the attacker can also modify the ringtone's name and inject the HTML tags into the name. This name is displayed to other members, and once a member opens the plugin, the member is attacked. In addition, while the size of the SmartTag+'s name field is limited, we do not find limits for the ringtone's name field. For larger payloads, an attacker needs to remotely load code when attacking over the SmartTag+'s name field. Over the ringtone's name field, the attacker can inject the complete payload.

While the first two locations can only be abused between members, the third location enables attacks from a compromised SmartTag+. The SmartTag+'s firmware contains the firmware version as a string, and it sends the bytes of this string to our phone over Bluetooth Low Energy (BLE). Subsequently, on our phone, these bytes are converted back to a string and forwarded to the SmartTag+'s plugin, which displays the firmware

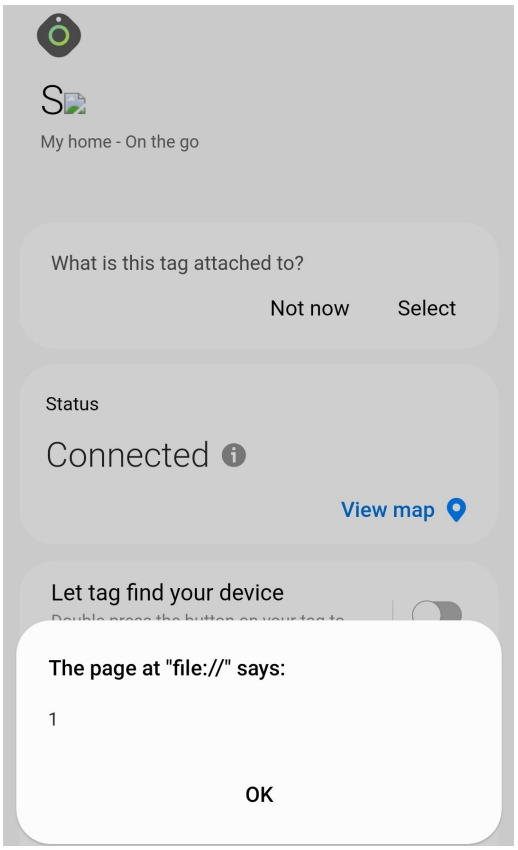

Figure 30: Verification of cross-site scripting.

Figure 31: Location retrieval over cross-site scripting.

version in the *Information* menu. However, HTML tags can be injected into this firmware version string.

In Section 8.8.2, we practically demonstrate that HTML tags can be sent from a compromised SmartTag+.

CALLING SMARTTHINGS' METHODS With XSS it is also possible to call methods from the SmartThings app that are exported with the *@JavascriptInterface* annotation. These methods allow access to certain phone information and management functions. For example, it is possible to retrieve the phone's location or enable/disable Bluetooth. We demonstrate this possibility in Figure 31, where we show a screenshot of a successful location retrieval over JavaScript.

8.7.2.2 *Member Privilege Escalation*

We further find a vulnerability that allows any member of a shared SmartTag+ to get full privileges for SmartTag+ management. All logic of checking if a user is the owner of a shared SmartTag+ is implemented in *bundle.js*. An attacker with a superuser-enabled phone can manipulate the checks, and it is sufficient to exchange one line in *bundle.js* for getting full privileges. Then, for example, an attacker gets firmware update privileges or can change the SmartTag+'s name.

We reported this vulnerability to Samsung, and they consider it working as intended. However, we do not agree with this assessment.

8.7.2.3 *No Attacks From Lower Layers*

We do not encounter any data from lower layers of Samsung's UWB ecosystem that reaches the management entities on our phone, and we do not find any vulnerability that can be attacked from lower layers. Therefore, we consider the attack surface of attacks that come from lower entities as very small or even not existent.

8.8 SMARTTAG+ HARDWARE ATTACKS

Now, we evaluate hardware-level attacks against the SmartTag+. Thereby, we have three goals. First, we want to learn if and how much we can gather information from the SmartTag+'s PCB. For example, we test if we can get logs or learn secrets by sniffing signals from the test pads or selected component pins. Second, we want to assess if we can extract the SmartTag+'s firmware over SWD access to the *QN9090*, or if we even can manipulate the firmware. Third, we intend to learn if we can extract the *SR040*'s decrypted firmware and if we further can manipulate it.

For our tests, we use the setup described in Section 7.5. Next, we present the results of our targeted goals.

8.8.1 *Information Gathering*

We can successfully live extract logs from the *QN9090*'s UART transmitter pin. The logs include the UCI messages exchanged between *QN9090* and *SR040*. Furthermore, it is comprehensible through the logs which operations the SmartTag+ does at the moment. We also learn that the signature of a transferred firmware is checked before it is replaced with the old one.

Moreover, it is possible to sniff the Serial Peripheral Interface (SPI) communication between the *QN9090* and *SR040* as well as *QN9090* and the flash memory component using the test pads. By doing this, we do not learn any additional information about the *SR040* or secrets written to the flash memory component that are independent of the firmware.

8.8.2 *SmartTag+ Firmware Extraction and Manipulation*

We assess the attack surface over a potential debug access to the *QN9090*. First, we test if we can establish an SWD connection to the *QN9090*, which is the SmartTag+'s main chip and executes the firmware. Thereby, we are successful. We can halt and resume the firmware's execution. Additionally, we can dump the firmware.

To test if we can manipulate the firmware, we modify selected bytes inclusive executed instructions of the firmware we dumped from the *QN9090*. For example, we modify the firmware string in the dumped firmware. Afterwards, we upload the modified firmware to the *QN9090* over SWD, and we are successful. We see our success in the UART output and the SmartTag+ plugin to which the SmartTag+ sends its firmware version. Both

Figure 32: Non-existing firmware version sent by manipulated firmware.

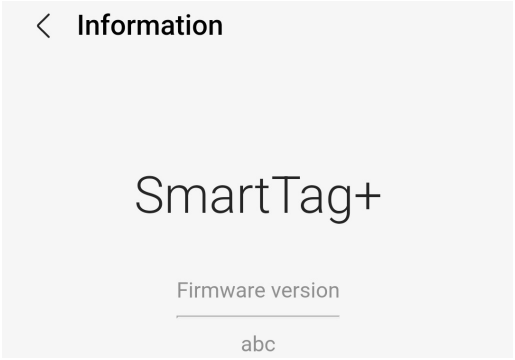

Figure 33: Injected horizontal rule HTML tag.

times our non-existing firmware version is displayed. This means we can compromise the SmartTag+ completely. In Figure 32, we show a screenshot of a displayed non-existing firmware version.

We further test if we can get SWD access and manipulate the firmware for three firmware versions: *1.00.08*, *1.00.10*, and *1.01.04*, which is the latest firmware version in January 2022. We are successful for each version.

SENDING HTML TAGS In Section 8.7.2.1, we already presented a vulnerability that a compromised SmartTag+ can attack. The compromised SmartTag+ can execute JavaScript in the plugin by injecting HTML tags into the sent firmware version.

Currently, the SmartTag+'s firmware only sends the first seven bytes of the firmware version string. For practically testing the XSS attack, we first need to modify the firmware's code related to reading the seven bytes, and upload the modified firmware to the SmartTag+ over SWD afterwards. However, this involves patching several code fragments in Ghidra, and it is out of scope with our limited time frame. Nevertheless, for a simple proof-of-concept, we still verify that a compromised SmartTag+ can send HTML tags, which are displayed without checks in the plugin afterwards. For this test, we modify the firmware's hard-coded firmware version string and inject an HTML horizontal rule tag (*<hr>*), which fits in our byte limit. Afterwards, we upload the firmware to the SmartTag+ over SWD. Then, we successfully can see the result in the plugin. In Figure 33, we show a screenshot of the result. Future work can modify the firmware's code fragments to send longer strings with an injected HTML tag containing a JavaScript payload.

8.8.3 *Failure of SR040 Firmware Extraction*

Despite existing test pads that are connected to the *SR040*'s SWD pins, we cannot establish a fully working SWD connection with the *SR040*.

The Segger J-Link EDU Mini, which is our SWD connection tool, can successfully do the SWD connection setup's first steps with the *SR040* and fails afterwards. Thereby, it finds the addresses of the Serial Wire Debug Port (SW-DP) registers and can read the Debug Port Identification Register (DPIDR). Using our logic analyzer, we further detect that some reads of the *RDBUFF* register fail, which is one of the SW-DP registers.

We carefully checked the web for explanations to help us understand the failure, but we did not find anything. If the SWD protocol is not customized on the *SR040*, a possibility would be to manipulate the SmartTag+'s firmware to include a development version of the *SR040*, which the *QN9090* sends to the *SR040*. The development version might be accepted since the *SR040*'s firmware transfer process is different from the *SR100T*'s local firmware download process. Furthermore, when the development version is accepted by the *SR040*, then the SWD connection establishment might be successful. We delegate this test to future work.

9 DISCUSSION

In Chapter 8, we evaluated the security of entities from Samsung's Ultra-Wideband (UWB) ecosystem, including the *SR100T* as a representative for all NXP UWB chips. Some of our attacks are successful, and we find several vulnerabilities in different entities. Next, we discuss our results and the impact of found vulnerabilities.

9.1 NXP'S UWB CHIPS

We do not have an unencrypted firmware version for any of NXP's UWB chips and find no ways to extract the firmware from the chips. Also, trying to access the firmware over the *SR040*'s Serial Wire Debug (SWD) interface on the SmartTag+ was not successful. Nevertheless, we understand the communication with the chips and the *SR100T*'s state machine. With this knowledge and by using our *ucitool* and Frida scripts for practical tests, we identify security issues for the *SR100T*, which also are helpful for future work. Our Wireshark dissector additionally aids us when doing our tests, for example, when interpreting responses from the chip.

In our evaluation, we show that any valid production firmware is accepted by the *SR100T*, which also means that a firmware downgrade is possible. This finding is helpful for future research that needs specific firmware versions. It has no security impact for an end-user since a superuser-enabled phone is needed to transfer a chosen firmware version, and only attacks against the phone itself are possible.

We further conclude in our evaluation that the source code's fragment chaining vulnerability is inherited by the *SR100T*'s firmware. While we only can crash the chip, a sophisticated attacker might be able to create a working exploit. Moreover, in the future, an unencrypted version for one of NXP's UWB chips might become public. Then, the fragment chaining vulnerability would likely be included and quickly found. Therefore, a fix is important now since it is easy to attack with much attacker-controlled bytes. Furthermore, the fragment chaining vulnerability strengthens our assumption that the encrypted firmware inherits at least parts of NXP's source code.

We also show that we can crash the *SR100T* with simple fuzzing. Sometimes we even see our fuzzing data in one of the two crash logs. It has no direct security impact besides hindering the availability. Yet, it shows that the *SR100T* has a low tolerance, and the crashes might be triggered through memory corruptions. Additionally, it shows that NXP did not test the chip's robustness with fuzzing, even though fuzzing is a standard measure for improving code robustness or discovering and preventing vulnerabilities.

9.1.1 *Impact and Security Assessment*

Currently, attacks against the fragment chaining vulnerability only are possible from a system app on our Samsung phone because the command needed for the attack is only available for privileged apps. The same applies to attacks against the *SR100T*'s availability.

When Samsung opens more of its UWB ecosystem's functionality to all apps, then a third-party app also can attack the vulnerability. Moreover, besides Samsung phones, other devices that use an NXP UWB chip likely are also affected. Here, an app might not need to be a privileged app.

In conclusion, we find the fragment chaining vulnerability, and the *SR100T* has a low tolerance against our simple fuzzing attacks. In addition, NXP's code has a general lack of security on which we elaborate in the next section. Therefore, we conclude the *SR100T*'s security and the security of NXP's other UWB chips can be defined as *security by obscurity*.

9.1.2 *Vulnerability Disclosure*

We reported the concluded fragment chaining vulnerability to Samsung in the mid of January 2022 and requested Samsung to forward the information to NXP. Currently, our report still is under analysis, and we do not expect a feedback before the end of our thesis.

9.2 SERVICES AND APPS OF SAMSUNG'S UWB ECOSYSTEM

The *UWB Application Programming Interface (API) service* and the *UWB Hardware Abstraction Layer (HAL) service* build both of Samsung's UWB services. In our evaluation, we find vulnerabilities in Samsung's UWB services that can be attacked from both sides, which means from an app using the services or from a compromised *SR100T*. We further indicate that remote attacks might be possible. Furthermore, we do not find vulnerabilities in apps using Samsung's UWB services.

Both of Samsung's UWB services inherit source code from NXP, which also is provided in the Mobile Knowledge (MK) UWB kits. The source code contains several vulnerabilities, of which most are inherited as well in the services. This shows that the blind adoption of source code also inherits its vulnerabilities.

9.2.1 *Attacks From SR100T*

We practically verified the inherited fragment chaining vulnerability for the *UWB API service*. There are several vulnerabilities left that a compromised *SR100T* can attack, which we did not test practically but discovered using Ghidra. Furthermore, remote attacks might be possible that use the *SR100T* to forward attacks.

The core problem of these vulnerabilities lies in assuming that only expected messages are received from the *SR100T*. Most often, we do not find any check for the received data's validity. Moreover, independent of an attack, locations at which the vulnerabilities exist are often also susceptible to crashes when unexpected messages are received, which only need to vary from expected messages slightly.

Thorough checks for the validity of received messages would prevent most attacks. In addition, the other vulnerabilities would be prevented when checking that the number of bytes in a copy operation does not exceed the destination buffer's size.

9.2.2 *Attacks From Apps*

Attacks from an app against Samsung's UWB services are critical. We demonstrate an attack in our evaluation. We also describe behaviors and characteristics of methods in the services that prevent attacking other vulnerabilities. However, the problem is that the core vulnerabilities still exist. Most often, no checks are implemented to prevent these. Therefore, we conclude that most of these behaviors and characteristics are not implemented for preventing vulnerabilities.

In our evaluation, we demonstrate exactly this problem with our attack against the vulnerable method *UWA_PerRxTest*, which results by finding a call chain to a vulnerable method without the non-intended attack preventions. Also, since the core vulnerabilities are not prevented, when new features are added to the services, new paths to vulnerable methods might emerge that an attacker can use for attacks.

It is important to fix the core vulnerabilities. All of our discovered ones can be prevented when implementing two checks at the corresponding vulnerable locations. First, checking is needed for a buffer overflow before copying bytes to a buffer. In addition, the second check should ensure that the result of an addition fits into the result object.

9.2.3 *Vulnerability Disclosure*

We reported both practically verified vulnerabilities to Samsung at the end of November 2021. We further marked NXP's source code vulnerabilities and reported them to Samsung in the same report, even if we did not test them or could attack them successfully. Additionally, we requested to forward the information to NXP. A fix is outstanding, but the vulnerabilities were already acknowledged, and a patch is under development, which will be released in March 2022. Moreover, the vulnerability in *UWA_PerRxTest* that we could successfully attack was previously known to Samsung and was fixed in parallel to our report.

9.3 SMARTTAG+

In Chapter 8, we find several vulnerabilities regarding the SmartTag+ and its management entities. Overall, these vulnerabilities show that the security of Internet of Things (IoT) devices from large companies also is a concern. Next, we discuss the vulnerabilities.

9.3.1 *Cross-Site Scripting in Plugin*

We find that an HTML injection vulnerability exists in the SmartTag+'s management plugin. This vulnerability can be used to run arbitrary JavaScript in the plugin. Members of a shared SmartTag+ can attack other members, and a compromised SmartTag+ can attack members as well. An attacker can execute JavaScript in the victim's plugin, for example, to forward a user to a malicious website. Furthermore, an attacker also can call the SmartThings app's exported methods, which considerably extends the impact. The exported methods allow, for example, managing other SmartTag+s or retrieving the location of the victim's phone.

The problem is that elements dynamically added to the HTML Document Object Model (DOM) are not sanitized. This includes the SmartTag+'s name and the SmartTag+'s sent firmware version. A simple fix for this vulnerability is to sanitize each element that is dynamically added to the HTML DOM.

Furthermore, the HTML tag injection vulnerability exists in the SmartTag+ plugin version *1.2.11-11* and later. We do not test earlier versions, but these likely also are affected.

9.3.1.1 *Vulnerability Disclosure*

We reported the vulnerability to Samsung in October 2021, and a fix was provided at the end of December 2021. The vulnerability is fixed in version *1.2.15-6*.

9.3.2 *SmartTag+ Firmware Downgrade*

The work in [10] demonstrates that the AirTag's firmware can be downgraded Over-The-Air (OTA). We show in our evaluation that the SmartTag+'s firmware can be downgraded as well. Furthermore, we assume that the firmware of Samsung's normal SmartTag also can be downgraded, but we do not test it.

By transferring any valid firmware to the SmartTag+, it is possible to send vulnerable firmware versions. An attacker might be able to fully compromise the SmartTag+ OTA afterwards. The attacker might have different goals. For example, when sharing a SmartTag+ with other members, the attacker first downgrades the SmartTag+ to a vulnerable version. Then, the vulnerable SmartTag+ is attacked. Afterwards, the compromised SmartTag+ is used to establish attacks against another member. Additionally, attacking a vulnerable downgraded firmware might be a way for an attacker to circumvent the protection against OTA firmware manipulations.

A fix for the vulnerability would be comparing the received firmware's version with the current running version. If the received firmware's version is not greater than the current version, an update to the received firmware should be declined.

9.3.2.1 *Vulnerability Disclosure*

We reported the vulnerability to Samsung in October 2022. At the beginning of February 2022, four months after the report submission, an acknowledgment of the vulnerability and a fix are outstanding.

9.3.3 *SmartTag+ Hardware Security*

The works in [5] and [44] show that the main chip of Apple's AirTag and Samsung's normal SmartTag can be attacked to enable SWD access. We show that the SmartTag+'s different main chip has an enabled SWD interface, and firmware extraction and manipulation of the SmartTag+ are possible without exploiting a vulnerability.

In conclusion, future research can use the SmartTag+ as a low-cost programmable hacking device with included UWB and Bluetooth Low Energy (BLE) functionality, which also is attractive for attackers. Moreover, the binding between the phone and SmartTag+ does not reset after the firmware manipulation. This indicates that the SmartTag+ does

not store user data but only its non-programmable ID. Therefore, an attacker can steal a SmartTag+, manipulate the firmware, and attack the SmartTag+'s user without following noticeable indications for the victim. With the HTML tag injection vulnerability, we also showcase that the attacker can attack members with a compromised SmartTag+.

9.3.3.1 *Vulnerability Disclosure*

We reported the vulnerability to Samsung at the beginning of January 2022, and the issue was previously known to Samsung. We further learned from Samsung that this vulnerability is unfixable because the deactivation of SWD pins cannot be done in the firmware. Moreover, all SmartTag+'s manufactured before July 2021 are affected, and later products should not be affected. However, in practice, we consider new commercially available SmartTag+'s still as affected. We bought a SmartTag+ from Samsung's official German shop in the end of November 2021, and the SmartTag+'s production date is March 2021. Therefore, we conclude that this issue still is current. Security researchers and attackers that need a SmartTag+ with an enabled SWD interface also can try buying one from different sources, and it is likely that still unsecured SmartTag+s are shipped. Security researchers and attackers that need a SmartTag+ with an enabled SWD interface also can try buying one from different sources, and still unsecured SmartTag+s are likely shipped. Depending on the time difference to our thesis, we recommend buying a SmartTag+ from unofficial sources or buying a used one.

10 CONCLUSIONS

In our thesis, we looked into the security of Samsung's Ultra-Wideband (UWB) ecosystem and the usage of NXP's UWB chips. Our goals were first to learn about the ecosystem's entities and their communication. Subsequently, we aimed to identify relevant attack vectors. With our developed understanding, we developed tools and used them for a security evaluation of selected attack vectors. Finally, in our evaluation, we found several vulnerabilities, which we discussed afterwards.

First, we analyzed the communication with NXP's UWB chips in Chapter 4. Ultra-Wideband Command Interface (UCI) is the first protocol and is used by all chips for UWB-related messages. It is a standard by the Fine Ranging (FiRa) Consortium [31] and is only available to its members. In addition, the protocol Host-Based Command/Control Interface (HBCI) is used to manage the *SR100T* and *SR150*. We conclude it is a non-publicly available proprietary protocol by NXP. Since both protocol specifications are not available to us, we reverse engineered the specifications using the source code provided by NXP in the Mobile Knowledge (MK) UWB kits. We further used the *ucitool*'s YAML file that contains UCI opcodes and payload structure identifiers.

Furthermore, in Chapter 4, we used the *SR100T* as a representative of NXP's UWB chips since it is integrated on our test phone. We analyzed its local firmware download process, which is responsible for transferring the encrypted firmware to the *SR100T*. With our understanding of the protocols and the following analysis of the *SR100T*'s driver, we further built the *SR100T*'s state machine. Two essential modes exist in the state machine: *UCI mode* and *HBCI mode*.

Subsequently, in Chapter 5, we examined Samsung's UWB ecosystem entities with a focus on each entity's role. NXP's *SR100T* chip is used in Samsung phones. Moreover, two services are responsible for providing UWB functionality on the phone and for communicating with the *SR100T*. The first service provides an Application Programming Interface (API) for external apps. It is further responsible for UCI message creation and processing. This service forwards its messages to the second service, which implements the Hardware Abstraction Layer (HAL) for UWB functionality on the phone. It is further responsible for transferring the firmware to the *SR100T*. Additionally, we detected that essential parts of both services use NXP's source code from the UWB kits.

Afterwards, also in Chapter 5, we briefly examined apps using the UWB API on the phone, which include a middleware service used by a SmartThings app's plugin to establish a UWB ranging session with the SmartTag+.

Additionally, we analyzed the SmartTag+ and its management entities in Chapter 5. The SmartTag+ is controlled by a plugin that is easy modifiable, and we used it for understanding the Over-The-Air (OTA) firmware update process. We further analyzed the SmartTag+'s Printed Circuit Board (PCB) and noted down the connections of test pads on the PCB.

After we developed an understanding of the entities and the *SR100T*'s usage, in Chapter 6, we identified relevant attack vectors and selected specific ones for our evaluation.

For our further work, in Chapter 7, we implemented several utilities and made modifications to pre-existing tools and entities. Most results of this work are further valuable for future work in different scenarios. Our first implementation was the Wireshark dissector. It can decode the UCI and HBCI messages exchanged with NXP's UWB chips. With an additional tool, we even can use it to decode the live communication with the *SR100T* on our Samsung phone. We further implemented several Frida scripts for hooking and manipulating methods at different locations in Samsung's UWB ecosystem. For example, we can use the scripts to simulate attacks against the UWB services from both sides. In addition, we took modifications on the *ucitool* and its helper binary *akash*. As a result, we can send any message we want to the *SR100T* as well as choose the transferred firmware.

Finally, in Chapter 8, we first evaluated our selected identified attack vectors and discussed our results subsequently in Chapter 9. In Chapter 8, we first showed that NXP's provided source code has several vulnerabilities, which can be divided into four groups. The core problem of most vulnerabilities is trusting incoming messages. Furthermore, the goal of this security analysis was to provide a foundation for deriving attacks against entities using the source code, which includes the *SR100T*.

The encrypted firmware of NXP's UWB chips prevented a close analysis in our thesis. Nevertheless, in Chapter 8, we could identify the unencrypted header and selected bytes of the header for the *SR100T*'s firmware. Moreover, we derived an attack against the fragment chaining vulnerability, and we practically demonstrated that the vulnerability presumably is inherited by the *SR100T*'s firmware. Additionally, we could crash the chip and sometimes see our fuzzing data in a returned crash log. In conclusion, we defined the security of NXP's UWB chips as *security by obscurity* in Chapter 9.

We further demonstrated in Chapter 8 that Samsung's UWB services could be attacked from an external app and a compromised *SR100T*. In addition, remote attacks might be possible. The vulnerabilities are inherited from NXP's code. Most of them that could be attacked from an app are prevented through indirect preventions. However, the core vulnerabilities still exist, and we showed a call chain to circumvent the preventions.

In Chapter 8, we also showed that the SmartTag+'s firmware could be downgraded OTA but not manipulated. In addition, we found an HTLM tag injection vulnerability in the SmartTag+'s management plugin. Attacks between members sharing a SmartTag+ are possible, and a compromised SmartTag+ can attack members as well.

Afterwards, we assessed the SmartTag+'s hardware security in Chapter 8. The Serial Wire Debug (SWD) interface of the SmartTag+'s main chip is enabled, and it is possible to extract and manipulate the firmware. As a result, the SmartTag+ can be fully compromised. We also showed that HTML tags could be sent over a manipulated SmartTag+ firmware to the vulnerable management plugin. We concluded in Chapter 9 that the lack of the SmartTag+'s security and the security of its entities show that the security of Internet of Things (IoT) products from large companies also is a concern.

We point out that we reported all discovered vulnerabilities to Samsung. If they were related to NXP's vulnerable source code, we requested Samsung to forward the information to NXP.

OUTLOOK FOR FUTURE WORK

There is work left for analyzing the implementation security of NXP's UWB chips. The essential step is to retrieve an unencrypted version for at least one of the chips. An unencrypted version is helpful to assess the chips' security, and it might also be helpful for future work that evaluates the physical-layer security of NXP's chips. We recommend as a next step a sophisticated hardware attack that aims the firmware extraction for an NXP UWB chip. We further recommend the low-cost SmartTag+ with the integrated *SR040* as the test device. As an alternative, successfully transferring a development firmware version might lead to enabled SWD access and the following firmware extraction.

Once an unencrypted firmware version is available, we recommend first assessing the security of UCI message processing. The core vulnerabilities in NXP's source code emerge through the trust that only expected UCI messages are received. While we detected several vulnerabilities in NXP's source code and reported them, it is probable that undetected vulnerabilities still exist since the code lacks general security. Additionally, when analyzing an unencrypted firmware version, one of the first steps should be to understand crash logs. Understanding the crash logs can be very helpful when executing attacks against an NXP UWB chip with encrypted firmware. Moreover, our modified *ucitool* and the implemented scripts can be used for attacks against the *SR100T*. It is probable that the *ucitool* and its helper binary *akash* work on any Android device featuring the *SR100T*.

Parts of both Samsung UWB services also use NXP's source code. We showed that the adoption of NXP's source code also inherited its vulnerabilities. Since the source code provides a fully working API to communicate with NXP's UWB chips, it is probable that other vendors that feature an NXP UWB chip also use this source code. Therefore, it is likely that the vulnerabilities also exist in applications that inherit the source code. For example, Xiaomi features the *SR100T* in the Mi Mix 4 phone [28]. We assume that NXP's code is inherited by services that provide UWB functionality on this phone. The vulnerabilities are probably also inherited. We further point out that attacks in other devices than mobile phones that integrate NXP's code also might be possible.

Remote attacks against the UWB services were only briefly covered by us without success. We recommend first fully evaluating the attack surface of remote attacks when doing data transfer over UWB. Here, an attacker has the most control over data that reaches the victim's services. Furthermore, since UWB ranging is currently the main application of UWB in a mobile device, remote attacks while doing ranging are also essential to evaluate. Our *ucitool* scripts provide a good foundation when testing remote attacks. For example, they implement UWB ranging or data transfer between two devices. An attack can be built on top of them.

Based on Samsung's response to our SmartTag+ SWD access report, we assume that SWD is disabled on the main chip for newly manufactured SmartTag+s. Therefore, the next step is to evaluate if one can bypass the protection as in [24]. Additionally, a SmartTag+ firmware analysis might find vulnerabilities that are remotely attackable.

UWB will likely be fully integrated into the Android Open Source Project (AOSP) for the Android 13 release [48, 52]. Currently, the HAL is missing in Android 12, and the Google Pixel 6 Pro uses custom software to implement the HAL, which is independent of the AOSP's Android 12 branch [48, 52]. Nevertheless, future work can start to analyze

the Pixel's UWB ecosystem now. The API exists already [52, 53] and it is probable that the Pixel 6 Pro uses a similar HAL implementation as it will be integrated into the AOSP for Android 13.

Furthermore, since UWB is not fully implemented in the AOSP [48, 52, 53], other vendors like Samsung started integrating UWB functionality using self-implemented proprietary entities. Once UWB is integrated into the AOSP, this will lead to problems when merging the functionality with pre-existing UWB integration like in Samsung's phones. Probably, a simple replacement of the services and co. is not possible. For example, in the latest *UwbUci.apk* version from January 2022, which implements Samsung's UWB API accessible by apps, we presumably detected an AOSP UWB API wrapper, which likely will be used when merging the functionality for Android 13. We conclude this is Samsung's way of merging the functionality. For Samsung phones, we assume that the AOSP's UWB API will be exposed to third-party apps, and internally Samsung's UWB services still will be used. We further assume that Samsung's UWB API simultaneously will be available to Samsung's apps since these currently use Samsung's API. Future work can investigate the security issues that emerge from merging the functionality.

Moreover, in general, when the chip is addressable over UCI, future work that analyzes the physical-layer security or usage of UWB chips might use our Wireshark dissector for decoding the communication with the chips. Since UCI is a non-public standard by the FiRa Consortium, and the consortium has members that include major companies, which have access to the procotol specification [16, 31], many UWB chips likely use UCI as a protocol in the future.

FINAL WORDS

UWB was integrated into recent smartphones and IoT devices, including devices from Samsung that feature NXP's *SR100T* and *SR040* UWB chip. The main use case is a precise position estimation between devices, which enables using a smartphone as a car key by doing *secure UWB ranging*. However, new attack vectors emerge with the integration of UWB. In our thesis, we analyzed the security of Samsung's UWB ecosystem, including NXP's UWB chips and Samsung's SmartTag+. Our results show that security issues exist for the integration of UWB functionality into Samsung phones, including the integrated *SR100T* UWB chip. The core of all issues is a lack of writing code with security in mind and a lack of general testing for written code. Our results further show that major security issues exist for Samsung's only UWB-enabled IoT device. In conclusion, Samsung's UWB ecosystem is not mature for deployment yet.

APPENDIX

A.1 STEPS TO ENABLE SUPERUSER ACCESS ON A SAMSUNG GALAXY S21 ULTRA

1. Download the right firmware for the phone, for example, from SamMobile[1].
2. In the developer options, turn the setting "unlock the bootloader" on.
3. Turn the phone off.
4. Reboot in download mode, by pressing the volume up and down key at once on the powered off phone and then connecting the phone cable to the PC.
5. A long press of volume up and the following press of "yes" will unlock the bootloader finally.
6. Start and set up the device.
7. Push the latest Magisk[2] app to the device and install it.
8. Push the file of the downloaded firmware that begins with "AP..." to the device.
9. Open Magisk on the phone and patch the "AP..." file.
10. Pull the patched "AP..." file from the phone to the host.
11. Download and execute Odin[3] on Windows. Alternatively, use a Windows virtual machine, copy the firmware files to the virtual machine if necessary, and then download and start Odin.
12. In Odin, put the "BL..." file to the *BL* instance, the patched "AP..." file to the *AP* instance, the "CP..." file to the *CP* instance, and the "CSC..." (not HOME CSC) file to the *CSC* instance. Then, disable auto-reboot.
13. Power off the phone and put it into download mode like in step 4. Then, press shortly the volume up key to go into the regular download mode.
14. The phone should now appear in Odin. Now, press start in Odin.
15. Restart the device after Odin was successful. After the device setup, install the Magisk app that is already there and disable auto-updates in the phone menu.

A.2 IMPORTANT PATHS IN THE MK UWB KITS

In Table 10, we show the paths to the most important files in the MK UWB kits in relation to the targeted NXP chip, after each zip file was extracted as a folder. Each extracted zip

1 https://www.sammobile.com/
2 https://github.com/topjohnwu/Magisk
3 https://forum.xda-developers.com/t/patched-odin-3-13-1.3762572/

UWB KIT EDITION	NXP CHIP	PATH
Standard Edition	SR040 & SR100T	MK-UWB%20KIT/USB%20flash%20drive%20KIT%20SR150_SR040%20%20v3.2/Software/MK%20UWB%20SDK/source/MK%20UWB%20SDK%20v1.2.0/resources/NXPSoftware%20packages/SR040%20-%20UWB%20Tracker%20SDK/UWBIOT_v02.00.00_MCUx/
Standard Edition	SR150	MK-UWB KIT/USB%20flash drive%20KIT%20SR150_SR040%20%20v3.2/Software/MK%20UWB%20SDK/source/MK%20UWB%20SDK%20v1.2.0/resources/NXP%20Software%20packages/SR150%20-%20UWB%20IoT%20SDK/UWB01_SW_FreeRTOS_RHD_A19.2/2020-12-04_UWB01_SW_FreeRTOS_RHD_A19.2/SOURCE/UwbCoreSDK/uwb_core/
Mobile Edition	SR040	SW-Documentation-MK-UWB_Kit-mobiledition/SW%20&%20Documentation%20-%20MK%20UWB%20Kit%20Mobile%20edition/Software/SR040%20Tag%20v03.07.00/UWBIOT_SR040_v03.07.00/
Mobile Edition	SR150	SW-Documentation-MK-UWB_Kit-mobiledition/SW%20&%20Documentation%20-%20MK%20UWB%20Kit%20Mobile%20edition/Software/SR150%20Anchor%20v03.04.00/UWBIOT_SR150_v03.04.00_MCUx/

Table 10: Paths to the most important files in the Mobile Knowledge (MK) Ultra-Wideband (UWB) kits. "%20" means the space character.

file has the same name and is in the same corresponding folder. Except for the *SR150* code in the mobile edition, all the UWB-related files are located in the subfolder *uwbiot-top* and the gist in its subfolder *libs*. The folder structure of each *uwbiot-top* folder is the same, and important files in these folders are mostly similar with the same contents. If they vary, then only slightly.

The most important differences between the contents of the *uwbiot-top* folder lay in the *libs* subfolder. The differences are in source code files, which extend the standard set of Ultra-Wideband Command Interface (UCI) parameters. These are used as part of proprietary UCI messages, whereby some different parameters are defined and used. Additionally, the ranging data returned from the chips is slightly different parsed.

A.3 COMPLETE SMARTTAG+ TEST PAD EVALUTION

In Table 11, we show the complete test pad evaluation. We could not find the test pads with the numbers 22 and 34, which might be hidden under components of the Printed Circuit Board (PCB). We also found two unnumbered test pads, which might be these two test pads. However, we do not find a connection for both unnumbered test pads. Moreover, test pad 22 might be test pad 27 and vice versa because we cannot clearly read the number of this test pad.

A.4 WIRESHARK DISSECTOR USER GUIDE

Now, we explain how to set up the dissector and how to import a hexdump to dissect.

A.4.1 *Setup*

First, place the Lua files of the dissector in the Wireshark plugin path. For example, on a Linux system, this can be in the path */home/username/.local/lib/wireshark/plugins/*. Second, open Wireshark and then, go to *Edit -> Preferences... -> Protocols -> DLT_USER -> Edit*. In the pop-up, add a mapping by pressing the plus sign. Ensure that *USER 0* is set under *DLT*. Afterwards, set the *Payload protocol* of the mapping to *uciandhbci* and press ok.

Values marked with * in the dissector are post-processed and interpreted as such in Samsung's services. We reverse engineered the interpretation, and in the dissector, we add how the values are interpreted.

A.4.2 *Import of a Hexdump File*

First, open Wireshark. Then, go to *File -> Import from Hex Dump...*. In the pop-up, select the hexdump file. Afterwards, set in the pop-up *USER 0* as the *Encapsulation Type*. Now, press *Import*.

A.4.3 *Layout of a Hexdump File*

Now, we describe how a hexdump file needs to be formatted. We assume that the messages of the communication are already extracted.

A custom wrapper needs to be generated for each message, which is 11 bytes long and prepended to the actual packet bytes. The first byte declares if the packet is written (0x57) or received (0x52) by the host. The second byte declares the UWB chip, whereby 0x00 = *SR040*, 0x01 = *SR100T*, 0x02 = *SR150*, and 0x03 = *SR100T* with a new firmware version. The *SR100T* formats the ranging data in new firmware versions like the *SR150*. Therefore, use for current versions 0x03 as the chip ID. The other 9 bytes can be 0x00 if no additional information is integrated into the custom wrapper, which should be the case when communication should be dissected that was not retrieved with the Frida code of the *Live Decoder*. For example, for a write to the *SR100T* with no information in the custom wrapper, the wrapper should be: *57 01 00 00 00 00 00 00 00 00 00*.

TEST PAD	CONNECTION 1	CONNECTION 2	NOTE/SYMBOL
1	QN9090 pin 19	-	RSTN
2	SR040 pin 18	-	SWDIO
3	SR040 pin 17	-	SWCLK
4	SR040 pin 8	-	PA_CAP_N
5	SR040 pin 31	-	RST_N
6	SR040 pin 14	-	Unknown function
7	SR040 pin 13	-	Unknown function
8	-	-	Unknown
9	QN9090 pin 28	-	VBAT
10	QN9090 pin 27	-	RSTN
11	QN9090 pin 11	-	UART TXD
12	Buzzer_P	-	SmartTag+'s sound system
13	QN9090 pin 12	-	UART RXD
14	QN9090 pin 16	-	SWDIO
15	QN9090 pin 15	-	SWCLK
16	QN9090 pin 9	-	Unknown function
17	Power (+)	-	
18	QN9090 pin 23	-	Unknown function
19	QN9090 pin 20	-	IO supply voltage
20	Button	-	SmartTag+'s button
21	Buzzer_N	-	SmartTag+'s sound system
22	-	-	Not found. May be TP 27
23	SR040 pin 11	-	VDD_GLOB
24	QN9090 pin 8	Flash pin 5	SS/CS of SPI line 1
25	Power (-)	-	
26	SR040 pin 19	-	Test point
27	QN9090 pin 3	SR040 pin 22	SCK of SPI line 0. May be TP 22
28	QN9090 pin 4	SR040 pin 23	MOSI/MISO of SPI line 0
29	QN9090 pin 4	SR040 pin 21	MISO/MOSI of SPI line 0
30	QN9090 pin 6	SR040 pin 20	SS/CS of SPI line 0
31	QN9090 pin 14	Flash pin 7	MISO/MOSI of SPI line 1
32	QN9090 pin 7	Flash pin 4	MOSI/MISO of SPI line 1
33	QN9090 pin 13	Flash pin 3	SCK of SPI line 1
34	-	-	Not found
35	QN9090 pin 10	-	Unknown function
36	QN9090 pin 26	-	TRST

Table 11: Complete evaluation of all SmartTag+ test pads. *SYMBOL* refers to declared symbol in the data sheet of the corresponding chip [29, 33]. We number the pin numbers of the flash component beginning from the left upper side. The left upper side is pin 1 and the right lower side is pin 8.

Furthermore, each message needs to be declared in a new line and prepended with six zeros without a space. Each following byte needs to be appended with a space in between. For example, a line for a sent packet from the host to the chip (*SR100T*), which contains only the header (always 4 bytes long) and no payload, should be declared like this: *000000 57 01 00 00 00 00 00 00 00 00 20 02 00 00*.

A.5 GENERATOR

Note that we cannot provide the YAML file with the UCI specification due to copyright reasons. Therefore, one needs to get the standard MK UWB kit, and in the kit is the YAML file.

To generate the decoders, run *python3 generator.py -f INPUTFILE*, whereby *INPUTFILE* defines the path to the YAML file. The resulting decoders are written to the folder *generated*.

A.6 LIVE DECODER USER GUIDE

First, optionally kill both of Samsung's UWB services, which run under the processes *vendor.samsung.hardware.uwb@1.0-service* and *com.samsung.android.uwb*. When killing these processes, one gets a trace from the beginning where the firmware is transferred to the *SR100T*. For example, use our provided *killuwb.sh* script.

Second, start the Frida server on the phone and then run *python3 main.py -i CHIP_ID* on your computer. As the argument for the chip ID use "1" for older *SR100T* versions, and use "3" for later versions. If the *ucitool* is used to communicate with the *SR100T*, then run *python3 main.py -i CHIP_ID -u yes*. Note that when using a later version of *akash*, an update of the Frida script with the corresponding addresses of the hooked methods in *akash* is needed. Thereby, in the Frida script *write_read.js*, four addresses need to be changed. For example, one can find the new addresses as described in Section 7.2.1. This step takes a few minutes when using Ghidra. Further note, when using the *ucitool*, both of Samsung's services need to be killed as described in Section 7.3.

It is also possible to manipulate messages and parts that are sent to or received from the *SR100T*. For this, use the *-m* option for any manipulation. Thereby, separated by a double dot, first define the header (4 bytes) of the packet that should be manipulated as a hex string. Then, define the index at which should be manipulated as an integer, and afterwards, define the hex values that should be written at the index. The argument can be used as often as wanted. It is also possible to manipulate the same packet at different indexes by using the argument multiple times and defining the same header. Example: *6200003D:33:1A01* manipulates the distance of the range measurement returned by the *SR100T* to 1A01 as a hex value, which results in 282 as int (LE). Example: *6200003D:2:AABB* manipulates the header, such that the header indicates a payload size of 0xBBAA.

A.7 LOG PARSER USER GUIDE

Run *python3 extract_uci_and_hbci.py -f INPUTFILE -o OUTPUTFILE -i CHIP_ID* to generate a hexdump of a log file. *INPUTFILE* is the path to the filename that contains logs, and

GID	GID NAME	NEW OID
1	Session	32
14	Proprietary	32, 33, 35, 36
15	Internal	1, 2

Table 12: Found undeclared UCI opcodes displayed in decimal form.

OUTPUTFILE is the path to the filename to which the hexdump should be written. Depending on the UWB chip used for the communication in the log, *CHIP_ID* should be "0" for the *SR040*, "1" for the *SR100T*, "2" for the *SR150*, and "3" for the *SR100T* with a new firmware version. The tool works for logs retrieved with Logcat on a Samsung phone and logs of a SmartTag+ retrieved over Universal Asynchronous Receiver-Transmitter (UART). Logs generated by any entity that uses the UWB Application Programming Interface (API) of the UWB kits can also be parsed, but this was not tested.

A.8 UCI AND HBCI INFORMATION GATHERING

In this section, we give an overview of which interesting information can be requested from the *SR100T* and likely from NXP's other UWB chips.

A.8.1 *Undeclared UCI opcodes*

One of our implemented *uctiool* scripts, which we presented in Section 7.3.2, iterates through all possible *Group Identifiers (GIDs)* and *Opcode Identifiers (OIDs)* opcodes. In result, we find seven *OIDs* that are not declared anywhere in the *ucitool* or MK UWB kits. However, we cannot derive the meaning for any opcode we found. In Table 12, we show the newly found *OIDs* with the corresponding *GID*.

A.8.2 *HBCI Queries*

We find that using Host-Based Command/Control Interface (HBCI) messages, we can request from the *SR100T* two interesting information, which are likely independent of the firmware that is send to the *SR100T* in a later instance. First, we can query it for a six-byte value, which presumably is a key ID of the installed root Certificate Authority (CA)'s public key. Second, we can request a two-byte value, which presumably is the ID of the installed certificate's public key from NXP. Both previous assumptions are based on the opcode's name. We further assume the chip uses the certificates with the corresponding IDs for validating the transferred firmware and other checks.

Moreover, we can request additional other data, which might be interesting for future work. In Table 13, we show the responses for all queries, and we additional include the HBCI message used to retrieve the data from the chip. Unfortunately, no HBCI payload interpretation exists except for a few selected bytes of two HBCI responses, which are not noteworthy. Therefore, we only can guess the meaning of each response's content.

NAME	HBCI QUERY	RESPONSE DATA
Chip_ID	0x01310000	0x30523050302D3030B13245-382200300008C879
Helios_ID	0x01320000	0xB100000016
CA_Root_Pub_Key	0x01330000	0x40888C2D4301
NXP_Pub_Key	0x01340000	0x03C5
ROM_Version	0x01350000	0x104523AFD4C8
Dev_Lifecycle	0x01360000	0xBF1F000000000000000000-0000000000BE006000F0022-0AD2D97B000000000000000-0000000003F000000002C

Table 13: Responses of HBCI queries.

```
▾ UCI: 259 bytes
  ▾ Header
      Group Identifier: 0x0e (GID_PROPRIETARY)
      Opcode Identifier: 0x0b (DBG_RFRAME_LOG_NTF)
      Message Type: 0x03 (NTF)
      Packet Boundary Flag: True
      Extended Size Flag: False
      Payload Size: 255
  ▾ Payload
      SESSION_ID: 3210489765
      NUM_RFRAME_MEASUREMENT: 4
    ▾ Data Frame
        MAPPING: 1
        DEC_STATUS: 5 (STATUS_RX_DEC_NO_DATA)
        NLOS: 0
        FIRST_PATH_INDEX: 53254
        MAIN_PATH_INDEX: 53248
        SNR_MAIN_PATH: 43
        SNR_FIRST_PATH: 43
        SNR_TOTAL: 11289
        RSSI: 46431
        CIR-MAIN-POWER: 4950423
        CIR-FIRST-PATH-POWER: 4950423
        NOISE-VARIANCE: 249
        CFO: 65521
        AoA_PHASE: 14041
        CIR_SAMPLES: fffffdfffbff0400f9fffefff8ff0d00f8ff0b001f004000…
    ▸ Data Frame
    ▸ Data Frame
```

Figure 34: Decoded RFRAME measurement.

A.8.3 *Ranging Logs*

When establishing a ranging session, or also while a ranging session is running, it is possible to enable different log messages by applying a configuration for the ranging session. There exist four of these messages and when enabled, the *SR100T* returns periodically the corresponding log message as an *Notification (NTF)* UCI message. Moreover, these logs messages may be interesting for future work that evaluates physical-layer attacks against the *SR100T*.

NUMBER	SE_COMM_DATA MESSAGES
1	0x6e1a0014078oca00fe02df230bfe07df2304003532429000
2	0x6e1a003d1603a4040010a0000003965453000000010402oo-oooooo256f218410a0000003965453000000010402000000a5-odbfoc0a9f7e0201034c030000009000
3	0x6e1a002f0e83500000008466a2675476a471d001f00000000-ooooooooooooo103600f0893148443 1e36be82aec39166799d-9000
4	0x6e1a0019158782330010084 1c0509f14798e52b191c659b3-80a2029000
5	0x6e1a002a0e87ca004708a69354b949a8a040001a56c5b74b-b5f8f2977cb9edd203198e87d78fbd5c60067ded9000

Table 14: Returned secure element logs. The used command to trigger the previously enabled logs is: *0x2E240000*.

The first log message that can be enabled by applying a configuration value are *RFRAME* measurements. *RFRAMES* — short for ranging frame — are UWB frames with a set ranging bit flag. They are exchanged between devices when doing UWB ranging [21]. An *RFRAME* measurement returned from the *SR100T* includes data like Channel Impulse Response (CIR) samples or the Received Signal Strength Indication (RSSI), which presumably comes from a received *RFRAME*. In Figure 34, we show the contents of a decoded *RFRAME* measurement that is part of a message with multiple measurements and is decoded by our dissector.

For a ranging session it is also possible to enable logs of the CIR. Furthermore, it is possible to enable PHY Service Data Unit (PSDU) logs. PSDU is a data field of a UWB frame [21], and the logs presumably come from the PSDU of a received *RFRAME*.

The fourth log message that can be enabled is *DATA_LOGGER*, and we do not the effect of enabling it since we encounter no additional messages by enabling it. We also cannot learn the meaning of it by looking in the UWB kits' source code.

A.8.4 *Other*

It is possible to apply a device configuration that enables logs presumably of the communication between *SR100T* and the secure element on the phone. The configuration value is named *DUMP_SE_COMM_DATA*, and when enabled, the *SR100T* returns additional *NTF* UCI packets. We test this successfully with a self-implemented *ucitool* script. Furthermore, we only do a brief analysis of the log contents and under which circumstances logs are returend. Thereby, we learn that we get five logs as *NTF* UCI messages, only when we send a specific UCI command to the chip: *0x2E240000*. This command also is one of the undeclared ones we find in Appendix A.8.1. In Table 14, we show the whole log messages we get. Unfortunately, we cannot decode the logs, and find no other commands that trigger these *NTF* UCI messages. We delegate a thorough analysis to future work.

FILE NAME	VULNERABLE METHOD	GROUP
uwb_ucif.cc	uwb_ucif_process_event	2
	uwb_ucif_proc_core_set_config_status	3
	uwb_ucif_proc_core_get_config_rsp	3
	uwb_ucif_proc_app_get_config_status	3
	uwb_ucif_proc_app_set_config_status	3
	uwb_ucif_proc_ranging_data	3
	uwb_ucif_proc_app_data_rcve_ntf_status	3
	uwb_ucif_proc_get_device_capability_rsp	3
	uwb_ucif_proc_test_get_config_status	3
	uwb_ucif_proc_test_set_config_status	3
	uwb_ucif_proc_rf_test_data	3
uci_hmsgs.cc	uci_snd_app_data_send_cmd	4
	uci_snd_test_per_rx_cmd	4
	uci_snd_test_uwb_loopback_cmd	4
	uci_snd_test_periodic_tx_cmd	4
UwbApi_Proprietary_Internal.ccp	uci_snd_test_uwb_loopback_cmd	3
	handle_schedstatus_ntf	3
	handle_do_calibration_ntf	3
UwbApi_RfTest.ccp	ufaTestDeviceManagementCallback	3
phNxpUciHal_fwd.cc	phHbci_GetStatus	1
	phHbci_QueryInfo	1
	phHbci_PutCommand	1
phTmlUwb.cc	phTmlUwb_TmlReaderThread	1

Table 15: Vulnerable methods of files and the groups of the vulnerabilities.

In the YAML file that contains the UCI specification we also find configuration IDs that are related to information about the stack of threads on the *SR100T*. We do not know how to request the information since we do not find a corresponding UCI opcode. Furthermore, we know from other opcodes how configuration values can be requested from the *SR100T*. Therefore, we try unsuccessfully a brute force attack, that creates any possible UCI message by iterating through all *GIDs* and *OIDs*, and defines for each message in the payload a valid configuration value request. We assume that these configuration values cannot be requested from the *SR100T* running a producation firmware.

A.9 UWB KIT VULNERABILITIES

In Table 15, we show all vulnerable methods of NXP's source code contained in the UWB kits. The first group of vulnerablitites relate to the reading of received messages

from the driver. Only the fragment chaining vulnerability corresponds to group two. Vulnerabilities of the third group relate to the processing of UCI messages. Last, the fourth group are vulnerabiliites that happen when processing app controlled data.

A.10 HBCI SPECIFICATION

In this section, we present all opcodes for the different HBCI classes. We resolve opcodes for the class *General* in Table 16, *Test* in Table 17, *Patch_ROM* in Table 18, *HIF_Image* in Table 19, and *IM4_Image* in Table 20.

CID	CN	SCID	SCN	OID	ON
0	General	1	Query	0x21	phHbci_General_Qry_Status
				0x31	phHbci_General_Qry_Chip_ID
				0x32	phHbci_General_Qry_Helios_ID
				0x33	phHbci_General_Qry_CA_Root_Pub_Key
				0x34	phHbci_General_Qry_NXP_Pub_Key
				0x35	phHbci_General_Qry_ROM_Version
				0x36	phHbci_General_Qry_Device_LC
0	General	2	Answer	0x21	phHbci_General_Ans_HBCI_Ready
				0x23	phHbci_General_Ans_Mode_Patch_ROM_Ready
				0x24	phHbci_General_Ans_Mode_HIF_Image_Ready
				0x25	phHbci_General_Ans_Mode_IM4_Image_Ready
				0x31	phHbci_General_Ans_Chip_ID
				0x32	phHbci_General_Ans_Helios_ID
				0x33	phHbci_General_Ans_CA_Root_Pub_Key
				0x34	phHbci_General_Ans_NXP_Pub_Key
				0x35	phHbci_General_Ans_ROM_Version
				0x36	phHbci_General_Ans_Device_LC
				0x41	phHbci_General_Ans_Boot_Success
				0xD1	phHbci_General_Ans_Boot_Autoload_Fail
				0xD2	phHbci_General_Ans_Boot_GPIOConf_CRC_Fail
				0xD3	phHbci_General_Ans_Boot_TRIM_CRC_Fail
				0xD4	phHbci_General_Ans_Boot_GPIOTRIM_CRC_Fail
				0xE1	phHbci_General_Ans_HBCI_Fail
				0xE3	phHbci_General_Ans_Mode_Patch_ROM_Fail
				0xE4	phHbci_General_Ans_Mode_HIF_Image_Fail
				0xE5	phHbci_General_Ans_Mode_IM4_Image_Fail
0	General	3	Command	0x23	phHbci_General_Cmd_Mode_Patch_ROM
				0x24	phHbci_General_Cmd_Mode_HIF_Image
				0x25	phHbci_General_Cmd_Mode_IM4_Image
0	General	4	Ack	0x01	phHbci_Valid_APDU
				0x81	phHbci_Invalid_LRC
				0x82	phHbci_Invalid_Class
				0x83	phHbci_Invalid_Instruction
				0x84	phHbci_Invalid_Segment_Length

Table 16: HBCI opcodes for class *General*. CID = Class ID, CN = Class Name, SCID = Subclass ID, SCN = Subclass Name, OID = Opcode ID, and ON = Opcode Name.

CID	CN	SCID	SCN	OID	ON
1	Test	1	Query	0x1	WRITE_STATUS
				0x2	AUTH_STATUS
				0x3	JTAG2AHB_STATUS
				0x4	PAYLOAD_STATUS
				0x8	DEV_STATUS
				0x9	ATTEMPT_REMAINING
1	Test	2	Answer	0x1	WRITE_SUCCESS
				0x2	AUTH_SUCCESS
				0x3	JTAG2AHB_SUCCESS
				0x4	PAYLOAD_SUCCESS
				0x8	DEV_UNLOCKED
				0x9	ATTEMPT_REMAINING
				0x81	OTP_FULL
				0x82	INVALID_PWD_LEN
				0x83	AUTH_FAIL
				0x84	DEV_LOCKED
				0x85	JTAG2AHB_FAIL
				0x86	PAYLOAD_FAIL
1	Test	3	Command	0x1	WRITE_PWD
				0x2	AUTH_PWD
				0x24	ENABLE_JTAG2AHB
				0x25	DOWNLOAD_PAYLOAD

Table 17: HBCI opcodes for class *Test*. CID = Class ID, CN = Class Name, SCID = Subclass ID, SCN = Subclass Name, OID = Opcode ID, and ON = Opcode Name.

CID	CN	SCID	SCN	OID	ON
2	Patch_ROM	1	Query	0x1	phHbci_Patch_ROM_Qry_Patch_Status
2	Patch_ROM	2	Answer	0x1	phHbci_Patch_ROM_Ans_Patch_Success
				0x81	phHbci_Patch_ROM_Ans_File_Too_Large
				0x82	phHbci_Patch_ROM_Ans_Invalid_Patch_File_Marker
				0x83	phHbci_Patch_ROM_Ans_Too_Many_Patch_Table_Entries
				0x84	phHbci_Patch_ROM_Ans_Invalid_Patch_Code_Size
				0x85	phHbci_Patch_ROM_Ans_Invalid_Global_Patch_Marker
				0x86	phHbci_Patch_ROM_Ans_Invalid_Signature_Size
				0x87	phHbci_Patch_ROM_Ans_Invalid_Signature
2	Patch_ROM	3	Command	0x1	phHbci_Patch_ROM_Cmd_Download_Patch

Table 18: HBCI opcodes for class *Patch_ROM*. CID = Class ID, CN = Class Name, SCID = Subclass ID, SCN = Subclass Name, OID = Opcode ID, and ON = Opcode Name.

CID	CN	SCID	SCN	OID	ON
5	HIF_Image	1	Query	0x1	phHbci_HIF_Image_Qry_Image_Status
5	HIF_Image	2	Answer	0x1	phHbci_HIF_Image_Ans_Image_Success
				0x4	phHbci_HIF_Image_Ans_Header_Success
				0x5	phHbci_HIF_Image_Ans_Quickboot_Settings_Success
				0x6	phHbci_HIF_Image_Ans_Execution_Settings_Success
				0x81	phHbci_HIF_Image_Ans_Header_Too_Large
				0x82	phHbci_HIF_Image_Ans_Header_Parse_Error
				0x83	phHbci_HIF_Image_Ans_Invalid_Cipher_Type_Crypto
				0x84	phHbci_HIF_Image_Ans_Invalid_Cipher_Type_Mode
				0x85	phHbci_HIF_Image_Ans_Invalid_Cipher_Type_Hash
				0x86	phHbci_HIF_Image_Ans_Invalid_Cipher_Type_Curve
				0x87	phHbci_HIF_Image_Ans_Invalid_ECC_Key_Length
				0x88	phHbci_HIF_Image_Ans_Invalid_Payload_Description
				0x89	phHbci_HIF_Image_Ans_Invalid_Firmware_Version
				0x8A	phHbci_HIF_Image_Ans_Invalid_ECID_Mask
				0x8B	phHbci_HIF_Image_Ans_Invalid_ECID_Value
				0x8C	phHbci_HIF_Image_Ans_Invalid_Encrypted_Payload_Hash
				0x8D	phHbci_HIF_Image_Ans_Invalid_Header_Signature
				0x8E	phHbci_HIF_Image_Ans_Install_Settings_Too_Large
				0x8F	phHbci_HIF_Image_Ans_Install_Settings_Parse_Error
				0x90	phHbci_HIF_Image_Ans_Payload_Too_Large
				0x91	phHbci_HIF_Image_Ans_Quickboot_Settings_Parse_Error
				0x92	phHbci_HIF_Image_Ans_Invalid_Static_Hash
				0x93	phHbci_HIF_Image_Ans_Invalid_Dynamic_Hash
				0x94	phHbci_HIF_Image_Ans_Execution_Settings_Parse_Error
				0x95	phHbci_HIF_Image_Ans_Key_Read_Error
5	HIF_Image	3	Command	0x1	phHbci_HIF_Image_Cmd_Download_Image

Table 19: HBCI opcodes for class *HIF_Image*. CID = Class ID, CN = Class Name, SCID = Subclass ID, SCN = Subclass Name, OID = Opcode ID, and ON = Opcode Name.

CID	CN	SCID	SCN	OID	ON
6	IM4_Image	1	Query	0x1	phHbci_IM4_Image_Qry_IM4_Status
				0x2	phHbci_IM4_Image_Qry_IM4M_Status
				0x3	phHbci_IM4_Image_Qry_IM4P_Status
				0x4	phHbci_IM4_Image_Qry_File_Descriptor_Status
				0x5	phHbci_IM4_Image_Qry_Payload_Status
6	IM4_Image	2	Answer	0x1	phHbci_IM4_Image_Ans_IM4_Success
				0x2	phHbci_IM4_Image_Ans_IM4M_Success
				0x3	phHbci_IM4_Image_Ans_IM4P_Success
				0x4	phHbci_IM4_Image_Ans_File_Descriptor_Success
				0x5	phHbci_IM4_Image_Ans_Payload_Success
				0x81	phHbci_IM4_Image_Ans_IM4M_Too_Large
				0x82	phHbci_IM4_Image_Ans_IM4M_Parse_Error
				0x83	phHbci_IM4_Image_Ans_Invalid_Chip_ID
				0x84	phHbci_IM4_Image_Ans_Invalid_Helios_ID
				0x85	phHbci_IM4_Image_Ans_Invalid_IM4M_Leaf_Certificate
				0x86	phHbci_IM4_Image_Ans_Invalid_IM4M_Manifest_Signature
				0x87	phHbci_IM4_Image_Ans_IM4P_Too_Large
				0x88	phHbci_IM4_Image_Ans_Invalid_IM4P_Hash
				0x89	phHbci_IM4_Image_Ans_IM4P_Parse_Error
				0x8A	phHbci_IM4_Image_Ans_Invalid_IM4P_Signature
				0x8B	phHbci_IM4_Image_Ans_File_Descriptor_Too_Large
				0x8C	phHbci_IM4_Image_Ans_Invalid_File_Descriptor
				0x8D	phHbci_IM4_Image_Ans_Payload_Too_Large
				0x8E	phHbci_IM4_Image_Ans_Invalid_Encrypted_Payload_Hash
				0x8F	phHbci_IM4_Image_Ans_Invalid_Download_Settings
6	IM4_Image	3	Command	0x1	phHbci_IM4_Image_Cmd_Download_IM4
				0x2	phHbci_IM4_Image_Cmd_Download_IM4M
				0x3	phHbci_IM4_Image_Cmd_Download_IM4P
				0x4	phHbci_IM4_Image_Cmd_Download_File_Descriptor
				0x5	phHbci_IM4_Image_Cmd_Download_Payload

Table 20: HBCI opcodes for class *IM4_Image*. CID = Class ID, CN = Class Name, SCID = Subclass ID, SCN = Subclass Name, OID = Opcode ID, and ON = Opcode Name.

A.11 UCI SPECIFICATION

Now, we declare all UCI opcodes and payload identifiers. We include opcode identifiers, resolvers, etc., in automatically generated tables. In the start Table 21, we give an overview of all UCI messages. This table can be used as the base for decoding a UCI message. It contains a reference for each OID, and the reference points to the payload identifiers for each existing message type of an OID. We point out that when no payload identifiers are declared for a message type, then the UCI message may still exist but just has no payload. For example, the UCI message *GET_DEV_INFO* can be sent as a command. Then the message has no payload, and no payload identifer exists for the command. We further point out that an index value indicates the position of a byte in the payload.

If a resolver exists for a payload identifer value in the referenced OID table, then the value references to yet another table, which resolves the meaning of the payload identifier's value. These resolver tables follow after all payload identifiers.

Furthermore, in UCI, data streams that are resolvalbe like *APP_TLV* are formatted based on the Type-Length-Value (TLV) scheme, and can be identified by having "TLV" in their name. Beginning with the first byte of the stream, the *ID* and an optional second *SUB-ID* are each one byte and resolve the type. The following byte defines the length *L* of the following data that is *L* bytes sized. Afterwards, subsequent bytes are resolved again based on the TLV scheme until no bytes are unresolved.

GID	GID NAME	OID	OID NAME	IDENTIFIERS
0	GID_CORE	0	DEVICE_RESET	Table 22
		1	DEVICE_STATUS_NTF	Table 23
		2	GET_DEV_INFO	Table 24
		3	GET_CAPS_INFO	Table 25
		4	SET_CONFIG	Table 26
		5	GET_CONFIG	Table 27
		6	DEV_SUSPEND	Table 28
		7	GENERIC_ERROR_NTF	Table 29
1	GID_SESSION	0	SESSION_INIT	Table 30
		1	SESSION_DEINIT	Table 31
		2	SESSION_STATUS_NTF	Table 32
		3	SET_APP_CONFIG	Table 33
		4	GET_APP_CONFIG	Table 34
		5	SESSION_GET_COUNT	Table 35
		6	SESSION_GET_STATE	Table 36
		7	SESSION_UPDATE_CONTROLLE...LIST	Table 37
2	GID_RANGING	0	RANGE_START	Table 38
		1	RANGE_STOP	Table 39
		2	RANGE_INTERVAL_UPDATE_REQ	Table 40
		3	RANGE_GET_RANGING_COUNT	Table 41
		4	BLINK_DATA_TX	Table 42
3	GID_DATA_CTRL	0	DATA_CREDIT_NTF_G3	Table 43
		1	DATA_TRANSMISSION_STATUS_NTF_G3	Table 44
9	GID_DATA_CTRL	0	DATA_CREDIT_NTF_G9	Table 45
		1	DATA_TRANSMISSION_STATUS_NTF_G9	Table 46
13	GID_TEST	0	TEST_CONFIG_SET	Table 47
		1	TEST_CONFIG_GET	Table 48
		2	TEST_PERIODIC_TX	Table 49
		3	TEST_PER_RX	Table 50
		4	TEST_TX	Table 51
		5	TEST_RX	Table 52
		6	TEST_LOOPBACK	Table 53
		7	TEST_STOP_SESSION	Table 54
		8	TEST_SS_TWR	Table 55
14	GID_PROPRIETARY	0	DEVICE_INIT	Table 56
		1	SE_DO_BIND	Table 57
		3	DBG_BIN_LOG	Table 58
		4	DBG_CIR0_LOG_NTF	Table 59
...	...	...	...	...

GID	GID NAME	OID	OID NAME	IDENTIFIERS
...	...	...	...	...
14	GID_PROPRIETARY	5	DBG_CIR1_LOG_NTF	Table 60
		6	DBG_GET_ERROR_LOG	Table 61
		9	DBG_PSDU_LOG_NTF	Table 62
		10	SE_GET_BINDING_COUNT	Table 63
		11	DBG_RFRAME_LOG_NTF	Table 64
		12	SE_GET_BINDING_STATUS	Table 65
		13	SE_DO_TEST_LOOP	Table 66
		14	SE_DO_TEST_CONNECTIVITY	Table 67
		15	GET_ALL_UWB_SESSIONS	Table 68
		16	SE_COMM_ERROR_NTF	Table 69
		17	SET_CALIBRATION	Table 70
		18	GET_CALIBRATION	Table 71
		19	BINDING_STATUS	Table 72
		20	SCHEDULER_STATUS_NTF	Table 73
		21	UWB_SESSION_KDF_NTF	Table 74
		22	UWB_WIFI_COEX_IND_NTF	Table 75
		23	WLAN_UWB_IND_ERR_NTF	Table 76
		24	DO_CALIBRATION	Table 77
		25	QUERY_TEMPERATURE	Table 78
		28	GENERATE_TAG	Table 79
		29	VERIFY_CALIB_DATA	Table 80
		34	UWB_WLAN_COEX_MAX_ACTIVE..._NTF	Table 81
		0	!SR040! - R4_LOG_NTF	Table 82
		17	!SR040! - R4_RADIO_CONFIG_DOWNLOAD	Table 83
		18	!SR040! - R4_ACTIVATE_SWUP	Table 84
		32	!SR040! - R4_TEST_START	Table 85
		33	!SR040! - R4_TEST_STOP	Table 86
		34	!SR040! - R4_TEST_INITIATOR_RA...DATA	Table 87
		35	!SR040! - R4_STACK_TEST	Table 88
		36	!SR040! - R4_DEVICE_SUSPEND	Table 89
		37	!SR040! - R4_TEST_LOOPBACK	Table 90
		38	!SR040! - R4_SET_TRIM_VALUES	Table 91
		39	!SR040! - R4_GET_ALL_UWB_SESSIONS	Table 92
		40	!SR040! - R4_GET_TRIM_VALUES	Table 93
		43	!SR040! - R4_SESSION_NVM_MANAGE	Table 94
		44	!SR040! - R4_GET_LUT_CRC	Table 95
		45	!SR040! - R4_GET_TRNG	Table 96

Table 21: UCI specification overview.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	0	DEVICE_RESET	
CMD payload index	0	RESET_CONFIG	
RSP payload index	0	UCI_STATUS	Table 100

Table 22: Payload identifiers - DEVICE_RESET. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	1	DEVICE_STATUS_NTF	
NTF payload index	0	DEVICE_STATUS	Table 101

Table 23: Payload identifiers - DEVICE_STATUS_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	2	GET_DEV_INFO	
RSP payload index	0	UCI_STATUS	Table 100
	1	UCI_MAJOR_VERSION	
	2	UCI_MINOR_VERSION	
	3	MANUFACTURE_LEN	
	4 - N	DEVICE_TLV	Table 126

Table 24: Payload identifiers - GET_DEV_INFO. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	3	GET_CAPS_INFO	
RSP payload index	0	UCI_STATUS	Table 100
	1	PARAMETERS	
	2 - N	DEVICE_TLV	Table 126

Table 25: Payload identifiers - GET_CAPS_INFO. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	4	SET_CONFIG	
CMD payload index	0	PARAMETERS	
	1 - N	DEVICE_TLV	Table 126
RSP payload index	0	UCI_STATUS	Table 100
	1	N_PARMETERS	
	2 - N	DEVICE_FAIL_STATUS	

Table 26: Payload identifiers - SET_CONFIG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	5	GET_CONFIG	
CMD payload index	0	PARAMETERS	
	1 - N	DEVICE_PARAMS	
RSP payload index	0	UCI_STATUS	Table 100
	1	PARAMETERS	
	2 - N	DEVICE_TLV	Table 126

Table 27: Payload identifiers - GET_CONFIG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	6	DEV_SUSPEND	
RSP payload index	0	UCI_STATUS	Table 100

Table 28: Payload identifiers - DEV_SUSPEND. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	0	GID_CORE	
OID	7	GENERIC_ERROR_NTF	
NTF payload index	0	UCI_STATUS	Table 100

Table 29: Payload identifiers - GENERIC_ERROR_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	0	SESSION_INIT	
CMD payload index	0 - 3	SESSION_ID	
	4	SESSION_TYPE	Table 104
RSP payload index	0	UCI_STATUS	Table 100

Table 30: Payload identifiers - SESSION_INIT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	1	SESSION_DEINIT	
CMD payload index	0 - 3	SESSION_ID	
RSP payload index	0	UCI_STATUS	Table 100

Table 31: Payload identifiers - SESSION_DEINIT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	2	SESSION_STATUS_NTF	
NTF payload index	0 - 3	SESSION_ID	
	4	SESSION_STATUS	Table 102
	5	SESSION_REASON_CODE	Table 103

Table 32: Payload identifiers - SESSION_STATUS_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	3	SET_APP_CONFIG	
CMD payload index	0 - 3	SESSION_ID	
	4	NUM_CONFIGS	
	5 - N	APP_TLV	Table 125
RSP payload index	0	UCI_STATUS	Table 100
	1	N_PARAMETERS	
	2 - N	APP_FAIL_STATUS	

Table 33: Payload identifiers - SET_APP_CONFIG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	4	GET_APP_CONFIG	
CMD payload index	0 - 3	SESSION_ID	
	4	NUM_CONFIGS	
	5 - N	APP_PARAMS	
RSP payload index	0	UCI_STATUS	Table 100
	1	NUM_CONFIGS	
	2 - N	APP_TLV	Table 125

Table 34: Payload identifiers - GET_APP_CONFIG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	5	SESSION_GET_COUNT	
RSP payload index	0	UCI_STATUS	Table 100
	1	ACTIVE_SESSION_COUNT	

Table 35: Payload identifiers - SESSION_GET_COUNT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	6	SESSION_GET_STATE	
CMD payload index	0 - 3	SESSION_ID	
RSP payload index	0	UCI_STATUS	Table 100
	1	SESSION_STATUS	Table 102

Table 36: Payload identifiers - SESSION_GET_STATE. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	1	GID_SESSION	
OID	7	SESSION_UPDATE_CONTROLLE...LIST	
CMD payload index	0 - 3	SESSION_ID	
	4	CONTROLLEE_UPDATE_ACTION	Table 105
	5	NUM_OF_CONTROLLES	
	6 - N	CONTROLEE_LIST	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0 - 3	SESSION_ID	
	4	REMAINING_MULTICAST_LIST_SIZE	
	5	NUM_OF_CONTROLLES	
	6 - N	STATUS_LIST_M_SUBFIELD	Table 224

Table 37: Payload identifiers - SESSION_UPDATE_CONTROLLER_MULTICAST_LIST. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	2	GID_RANGING	
OID	0	RANGE_START	
CMD payload index	0 - 3	SESSION_ID	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index SR040	0 - 3	SEQUENCE_NUM	
	4 - 7	SESSION_ID	
	8	RCR_IND	
	9 - 12	CUR_RNG_INTERVAL	
	13	MEASUREMENT_TYPE	
	14	RFU	
	15	MAC_ADDR_MODE	
	16 - 23	RFU	
	24	NUM_RANGING_MEAS	
	25 - X	MEASUREMENT_DATA	Table 97
	(X+1) - (X+2) -> Optional	VENDOR_SPEC_LENGTH	
	(X+3) - N -> Optional	VENDOR_SPEC	
NTF payload index SR100T - Old FW	0 - 3	SEQUENCE_NUM	
	4 - 7	SESSION_ID	
	8	RCR_IND	
	9 - 12	CUR_RNG_INTERVAL	
	13	MEASUREMENT_TYPE	
	14	ANTENNA_PAIR_INFO	
	15	MAC_ADDR_MODE	
	16 - 23	RFU	
	24	NUM_RANGING_MEAS	
	25 - X	MEASUREMENT_DATA	Table 98
	X + 1 -> Required	AUTH_INFO_PRESENT	
	(X + 1) - N -> Optional	AUTHENTICATION_TAG	
NTF payload index SR100T or SR150	0 - 3	SEQUENCE_NUM	
	4 - 7	SESSION_ID	
	8	RCR_IND	
	9 - 12	CUR_RNG_INTERVAL	
	13	MEASUREMENT_TYPE	
	14	RFU	
	15	MAC_ADDR_MODE	
	16 - 23	RFU	
	24	NUM_RANGING_MEAS	
	25 - X	MEASUREMENT_DATA	Table 99
	(X+1) - (X+2) -> Required	VENDOR_SPEC_LENGTH	
	(X+3) - N -> Optional	VENDOR_SPEC	

Table 38: Payload identifiers - RANGE_START. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	2	GID_RANGING	
OID	1	RANGE_STOP	
CMD payload index	0 - 3	SESSION_ID	
RSP payload index	0	UCI_STATUS	Table 100

Table 39: Payload identifiers - RANGE_STOP. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	2	GID_RANGING	
OID	2	RANGE_INTERVAL_UPDATE_REQ	
CMD payload index	0 - 3	SESSION_ID	
	4 - 5	RANGING_INTERVAL	
RSP payload index	0	UCI_STATUS	Table 100

Table 40: Payload identifiers - RANGE_INTERVAL_UPDATE_REQ. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	2	GID_RANGING	
OID	3	RANGE_GET_RANGING_COUNT	
CMD payload index	0 - 3	SESSION_ID	
RSP payload index	0	UCI_STATUS	Table 100
	1 - 4	SESSION_RANGING_COUNT	

Table 41: Payload identifiers - RANGE_GET_RANGING_COUNT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	2	GID_RANGING	
OID	4	BLINK_DATA_TX	
CMD payload index	0 - 3	SESSION_ID	
	4	REPEAT_COUNT	
	5	APP_DATA_LEN	
	6 - N	APP_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100

Table 42: Payload identifiers - BLINK_DATA_TX. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	3	GID_DATA_CTRL	
OID	0	DATA_CREDIT_NTF_G3	
NTF payload index	0 - 3	SESSION_ID	
	4	UCI_STATUS	Table 100
	5	NUM_CREDITS	

Table 43: Payload identifiers - DATA_CREDIT_NTF_G3. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	3	GID_DATA_CTRL	
OID	1	DATA_TRANSMISSION_STATUS_NTF_G3	
NTF payload index	0 - 3	SESSION_ID	
	4	UCI_STATUS	Table 100

Table 44: Payload identifiers - DATA_TRANSMISSION_STATUS_NTF_G3. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	9	GID_DATA_CTRL	
OID	0	DATA_CREDIT_NTF_G9	
NTF payload index	0 - 3	SESSION_ID	
	4	UCI_STATUS	Table 100
	5	NUM_CREDITS	

Table 45: Payload identifiers - DATA_CREDIT_NTF_G9. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	9	GID_DATA_CTRL	
OID	1	DATA_TRANSMISSION_STATUS_NTF_G9	
NTF payload index	0 - 3	SESSION_ID	
	4	UCI_STATUS	Table 100

Table 46: Payload identifiers - DATA_TRANSMISSION_STATUS_NTF_G9. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	0	TEST_CONFIG_SET	
CMD payload index	0 - 3	SESSION_ID	
	4	NUM_TEST_CONFIG	
	5 - N	TEST_TLV	Table 129
RSP payload index	0	UCI_STATUS	Table 100
	1	NUMBER_OF_PARAMS	
	2 - N	TEST_PARAMS	

Table 47: Payload identifiers - TEST_CONFIG_SET. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	1	TEST_CONFIG_GET	
CMD payload index	0 - 3	SESSION_ID	
	4	NUM_TEST_CONFIG	
	5 - N	TEST_PARAMS	
RSP payload index	0	UCI_STATUS	Table 100
	1	NUMBER_OF_PARAMS	
	2 - N	TEST_TLV	Table 129

Table 48: Payload identifiers - TEST_CONFIG_GET. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	2	TEST_PERIODIC_TX	
CMD payload index	0 - N	PSDU_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100

Table 49: Payload identifiers - TEST_PERIODIC_TX. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	3	TEST_PER_RX	
CMD payload index	0 - N	PSDU_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1 - 4	ATTEMPTS	
	5 - 8	ACQ_DETECTS	
	9 - 12	ACQ_REJECTS	
	13 - 16	RX_FAIL	
	17 - 20	CIR_SYNC_READY	
	21 - 24	SFD_FAIL	
	25 - 28	SFD_FOUND	
	29 - 32	PHR_DEC_ERR	
	33 - 36	PHR_BIT_ERROR	
	37 - 40	PSDU_DEC_ERROR	
	41 - 44	PSDU_BIT_ERROR	
	45 - 48	EOF	
	49 - 50	RSSI_RX1	
	51 - 52	RSSI_RX2	
	53 - 54	SNR_RX1	
	55 - 56	SNR_RX2	
	57 - 58	RX_CFO_EST	

Table 50: Payload identifiers - TEST_PER_RX. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	4	TEST_TX	
CMD payload index	0 - N	PSDU_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1 - 4	TX_DONE_TS_INT	
	5 - 6	TX_DONE_TS_FRAC	

Table 51: Payload identifiers - TEST_TX. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	5	TEST_RX	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1	RX_TOA_FIRST_PATH	Table 123
	2 - 5	RX_DONE_TS_INT	
	6 - 7	RX_DONE_TS_FRAC	
	8 - 9	AoA1	
	10 - 11	AoA2	
	12 - 13	PDoA1	
	14 - 15	PDoA2	
	16 - 17	PDoA1_Index	
	18 - 19	PDoA2_Index	
	20 - 21	RSSI_RX1	
	22 - 23	RSSI_RX2	
	24 - 25	FIRST_PATH_INDEX_RX1	
	26 - 27	FIRST_PATH_INDEX_RX2	
	28 - 29	MAX_PATH_INDEX_RX1	
	30 - 31	MAX_PATH_INDEX_RX2	
	32	SNR_MAIN_PATH_RX1	
	33	SNR_MAIN_PATH_RX2	
	34	SNR_FIRST_PATH_RX1	
	35	SNR_FIRST_PATH_RX2	
	36	TOA_GAP	
	37 - 38	PHR	
	39 - N	PSDU_DATA	

Table 52: Payload identifiers - TEST_RX. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	6	TEST_LOOPBACK	
CMD payload index	0 - N	PSDU_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1	RX_TOA_FIRST_PATH	Table 123
	2 - 5	TX_TS_INT	
	6 - 7	TX_TS_FRAC	
	8 - 11	RX_TS_INT	
	12 - 13	RX_TS_FRAC	
	14 - 15	AoA1	
	16 - 17	AoA2	
	18 - 19	PDoA1	
	20 - 21	PDoA2	
	22 - 23	PDoA1_Index	
	24 - 25	PDoA2_Index	
	26 - 27	RSSI_RX1	
	28 - 29	RSSI_RX2	
	30	SNR_MAIN_PATH_RX1	
	31	SNR_MAIN_PATH_RX2	
	32	SNR_FIRST_PATH_RX1	
	33	SNR_FIRST_PATH_RX2	
	34 - 35	SNR_AVERAGE_RX1	
	36 - 37	SNR_AVERAGE_RX2	
	38 - 39	FIRST_PATH_INDEX_RX1	
	40 - 41	FIRST_PATH_INDEX_RX2	
	42 - 43	MAX_PATH_INDEX_RX1	
	44 - 45	MAX_PATH_INDEX_RX2	
	46 - 47	PHR	
	48 - N	PSDU_DATA	

Table 53: Payload identifiers - TEST_LOOPBACK. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	7	TEST_STOP_SESSION	
RSP payload index	0	UCI_STATUS	Table 100

Table 54: Payload identifiers - TEST_STOP_SESSION. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	13	GID_TEST	
OID	8	TEST_SS_TWR	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1 - 4	MEASUREMENT	

Table 55: Payload identifiers - TEST_SS_TWR. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	0	DEVICE_INIT	
CMD payload index	0	PLATFORM_ID	Table 118
	1	VARIANT_ID	Table 119
RSP payload index	0	UCI_STATUS	Table 100

Table 56: Payload identifiers - DEVICE_INIT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	1	SE_DO_BIND	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1	HELIOS_BINDING_COUNT	
	2	BIND_STATUS	Table 107

Table 57: Payload identifiers - SE_DO_BIND. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	3	DBG_BIN_LOG	
RSP payload index	0 - N	UCI_STATUS	Table 100
NTF payload index	0 - N	DEBUG_DATA	

Table 58: Payload identifiers - DBG_BIN_LOG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	4	DBG_CIR0_LOG_NTF	
NTF payload index	0 - 3	SESSION_ID	
	4 - N	CIR0_DATA	

Table 59: Payload identifiers - DBG_CIR0_LOG_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	5	DBG_CIR1_LOG_NTF	
NTF payload index	0 - 3	SESSION_ID	
	4 - N	CIR1_DATA	

Table 60: Payload identifiers - DBG_CIR1_LOG_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	6	DBG_GET_ERROR_LOG	
RSP payload index	0 - N	DEBUG_DATA	

Table 61: Payload identifiers - DBG_GET_ERROR_LOG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	9	DBG_PSDU_LOG_NTF	
NTF payload index	0 - 3	SESSION_ID	
	4 - N	PSDU_LOG_DATA_M_SUBFIELD	Table 225

Table 62: Payload identifiers - DBG_PSDU_LOG_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	10	SE_GET_BINDING_COUNT	
RSP payload index	0	UCI_STATUS	Table 100
	1	BINDING_STATUS	
	2	HELIOS_BINDING_COUNT	
	3	SE_BINDING_COUNT	

Table 63: Payload identifiers - SE_GET_BINDING_COUNT. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	11	DBG_RFRAME_LOG_NTF	
NTF payload index	0 - 3	SESSION_ID	
	4	NUM_RFRAME_MEASUREMENT	
	5 - N	RFRAME_MEASUREMENT_M_SUBFIELD	Table 223

Table 64: Payload identifiers - DBG_RFRAME_LOG_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	12	SE_GET_BINDING_STATUS	
RSP payload index	0	SE_STATUS	Table 108
NTF payload index	0	BIND_STATUS	Table 107
	1	SE_BINDING_COUNT	
	2	HELIOS_BINDING_COUNT	
	3	SE_BINDING_COUNT	

Table 65: Payload identifiers - SE_GET_BINDING_STATUS. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	13	SE_DO_TEST_LOOP	
CMD payload index	0 - 3	SE_CMD	
RSP payload index	0	SE_TEST_LOOP_STATUS	Table 109
NTF payload index	0	TEST_STATUS	
	1 - 2	LOOP_COUNT	
	3 - 4	LOOP_PASS_COUNT	

Table 66: Payload identifiers - SE_DO_TEST_LOOP. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	14	SE_DO_TEST_CONNECTIVITY	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	SE_AID_STATUS	Table 110
	1	WTX_COUNT	

Table 67: Payload identifiers - SE_DO_TEST_CONNECTIVITY. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	15	GET_ALL_UWB_SESSIONS	
RSP payload index	0	UCI_STATUS	Table 100
	1	SESSION_COUNT	
	2 - N	SESSION_INFO_M_SUBFIELD	Table 226

Table 68: Payload identifiers - GET_ALL_UWB_SESSIONS. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	16	SE_COMM_ERROR_NTF	
NTF payload index	0	UCI_STATUS	Table 100
	1 - 2	CLS_AND_INS	
	3 - 4	STATUS_CODES	

Table 69: Payload identifiers - SE_COMM_ERROR_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	17	SET_CALIBRATION	
CMD payload index	0	CHANNEL_ID	
	1	CALIB_PARAM	Table 120
	2 - N	CALIBRATION_VALUE	
RSP payload index	0	UCI_STATUS	Table 100

Table 70: Payload identifiers - SET_CALIBRATION. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	18	GET_CALIBRATION	
CMD payload index	0	CHANNEL_ID	
	1	CALIB_PARAM	Table 120
RSP payload index	0	UCI_STATUS	Table 100
	1	CALIB_STATE	Table 122
	2 - N	CALIB_VALUE	

Table 71: Payload identifiers - GET_CALIBRATION. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	19	BINDING_STATUS	
NTF payload index	0	BIND_STATUS	Table 107

Table 72: Payload identifiers - BINDING_STATUS. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	20	SCHEDULER_STATUS_NTF	
NTF payload index	0	NUM_OF_SESSIONS	
	1 - N	SESSION_DATA_M_SUBFIELD	Table 227

Table 73: Payload identifiers - SCHEDULER_STATUS_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	21	UWB_SESSION_KDF_NTF	
NTF payload index	0	NUM_OF_PARAMS	
	1 - N	KDF_NTF_TLV	Table 128

Table 74: Payload identifiers - UWB_SESSION_KDF_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	22	UWB_WIFI_COEX_IND_NTF	
NTF payload index	0	UWB_WIFI_COEX_IND_STATUS	Table 114
	1 - 4	SLOT_INDEX	

Table 75: Payload identifiers - UWB_WIFI_COEX_IND_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	23	WLAN_UWB_IND_ERR_NTF	
NTF payload index	0	WLAN_UWB_IND_ERR_STATUS	Table 115
	1 - 4	SLOT_INDEX	

Table 76: Payload identifiers - WLAN_UWB_IND_ERR_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	24	DO_CALIBRATION	
CMD payload index	0	CHANNEL_ID	
	1	CALIB_PARAM	Table 120
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1 - N	CALIB_VALUE	

Table 77: Payload identifiers - DO_CALIBRATION. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	25	QUERY_TEMPERATURE	
RSP payload index	0	UCI_STATUS	Table 100
	1	TEMPERATURE	
NTF payload index	0	UCI_STATUS	Table 100
	1 - N	CALIB_VALUE	

Table 78: Payload identifiers - QUERY_TEMPERATURE. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	28	GENERATE_TAG	
CMD payload index	0 - N	KEY	
	0	TAG_OPTION	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1 - 16	CMAC_TAG_IFVAL_UCI_STATUS_VAL_0	

Table 79: Payload identifiers - GENERATE_TAG. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	29	VERIFY_CALIB_DATA	
CMD payload index	0 - N	KEY	
	0 - N	CMAC_TAG	
	0	TAG_OPTION	
	1 - 2	TAG_VERSION	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100

Table 80: Payload identifiers - VERIFY_CALIB_DATA. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	34	UWB_WLAN_COEX_MAX_ACTIVE..._NTF	
NTF payload index	0	STATUS	

Table 81: Payload identifiers - UWB_WLAN_COEX_MAX_ACTIVE_GRANT_DUARTION_EXCEEDED_WAR_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	0	!SR040! - R4_LOG_NTF	
NTF payload index	0 - N	LOG_DATA	

Table 82: Payload identifiers - !SR040! - R4_LOG_NTF. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	17	!SR040! - R4_RADIO_CONFIG_DOWN...LOAD	
CMD payload index	0 - N	RADIO_DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 83: Payload identifiers - !SR040! - R4_RADIO_CONFIG_DOWNLOAD. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	18	!SR040! - R4_ACTIVATE_SWUP	
CMD payload index	0 - N	COMMAND_DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 84: Payload identifiers - !SR040! - R4_ACTIVATE_SWUP. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	32	!SR040! - R4_TEST_START	
CMD payload index	0 - N	TEST_DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 85: Payload identifiers - !SR040! - R4_TEST_START. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	33	!SR040! - R4_TEST_STOP	
CMD payload index	0 - N	TEST_DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 86: Payload identifiers - !SR040! - R4_TEST_STOP. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	34	!SR040! - R4_TEST_INITIATOR_RA...DATA	
CMD payload index	0 - N	RANGE_DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100
NTF payload index	0 - N	RANGE_DATA_NTF	

Table 87: Payload identifiers - !SR040! - R4_TEST_INITIATOR_RANGE_DATA. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	35	!SR040! - R4_STACK_TEST	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 88: Payload identifiers - !SR040! - R4_STACK_TEST. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	36	!SR040! - R4_DEVICE_SUSPEND	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 89: Payload identifiers - !SR040! - R4_DEVICE_SUSPEND. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	37	!SR040! - R4_TEST_LOOPBACK	
CMD payload index	0 - N	PSDU_DATA	
RSP payload index	0	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100
	1	RX_TOA_FIRST_PATH	Table 123
	2 - 5	TX_TS_INT	
	6 - 7	TX_TS_FRAC	
	8 - 11	RX_TS_INT	
	12 - 13	RX_TS_FRAC	
	14 - 15	AoA1	
	16 - 17	AoA2	
	18 - 19	PDoA1	
	20 - 21	PDoA2	
	22 - 23	PDoA1_Index	
	24 - 25	PDoA2_Index	
	26 - 27	RSSI_RX1	
	28 - 29	RSSI_RX2	
	30	SNR_MAIN_PATH_RX1	
	31	SNR_MAIN_PATH_RX2	
	32	SNR_FIRST_PATH_RX1	
	33	SNR_FIRST_PATH_RX2	
	34 - 35	SNR_AVERAGE_RX1	
	36 - 37	SNR_AVERAGE_RX2	
	38 - 39	FIRST_PATH_INDEX_RX1	
	40 - 41	FIRST_PATH_INDEX_RX2	
	42 - 43	MAX_PATH_INDEX_RX1	
	44 - 45	MAX_PATH_INDEX_RX2	
	46 - 47	PHR	
	48 - N	PSDU_DATA	

Table 90: Payload identifiers - !SR040! - R4_TEST_LOOPBACK. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	38	!SR040! - R4_SET_TRIM_VALUES	
CMD payload index	0	NUM_DATA	
	1	CALIBRATION_ID	
	2	CALIBRATON_LEN	
	3	CHANNEL_NO	
	4 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100
NTF payload index	0	UCI_STATUS	Table 100

Table 91: Payload identifiers - !SR040! - R4_SET_TRIM_VALUES. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	39	!SR040! - R4_GET_ALL_UWB_SESSI...IONS	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 92: Payload identifiers - !SR040! - R4_GET_ALL_UWB_SESSIONS. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	40	!SR040! - R4_GET_TRIM_VALUES	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 93: Payload identifiers - !SR040! - R4_GET_TRIM_VALUES. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	43	!SR040! - R4_SESSION_NVM_MANAGE	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 94: Payload identifiers - !SR040! - R4_SESSION_NVM_MANAGE. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	44	!SR040! - R4_GET_LUT_CRC	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 95: Payload identifiers - !SR040! - R4_GET_LUT_CRC. Go to start at Table 21.

INFORMATION	IDENTIFIER / INDEX	VALUE	RESOLVER
GID	14	GID_PROPRIETARY	
OID	45	!SR040! - R4_GET_TRNG	
CMD payload index	0 - N	DATA	
RSP payload index	0 - N	UCI_STATUS	Table 100

Table 96: Payload identifiers - !SR040! - R4_GET_TRNG. Go to start at Table 21.

TYPE	INDEX	VALUE	RESOLVER
ONE WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	FRAME_TYPE	
	3	NLOS	
	4 - 5	AoA_AZIMUTH	
	6	AoA_AZIMUTH_FOM	
	7 - 8	AoA_ELEVATION	
	9	AoA_ELEVATION_FOM	
	10 - 17	TIMESTAMP	
	18 - 21	BLINK_FRAME_NUM	
		!! Do not add 6 to following indexes any more (-> Cleared) !!	
	22 - 33 or 28 - 33	RFU (Depends on MAC_ADDR_MODE = 0 or 1)	
	34	DEV_INFO_SIZE	
	35 - X	DEV_INFO (Size depends on DEV_INFO_SIZE)	
		-> Add #DEV_INFO_SIZE to following indexes	
	36	BLINK_PAYLOAD_SIZE	
	37 - Y	BLINK_PAYLOAD (Size depends on BLINK_PAYLOAD_SIZE)	
TWO WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	UCI_STATUS	Table 100
	3	NLOS	
	4 - 5	DISTANCE	
	6 - 7	AOA_AZIMUTH	
	8	AOA_AZIMUTH_FOM	
	9 - 10	AOA_ELEVATION	
	11	AOA_ELEVATION_FOM	
	12 - 13	AOA_DEST_AZIMUTH	
	14	AOA_DEST_AZIMUTH_FOM	
	15 - 16	AOA_DEST_ELEVATION	
	17	AOA_DEST_ELEVATION_FOM	
	18	SLOT_INDEX	
		!! Do not add 6 to following indexes any more (-> Cleared) !!	
	19 - 30 or 25 - 30	RFU (Depends on MAC_ADDR_MODE = 0 or 1)	

Table 97: Resolver - RANGING_DATA_SR040. TYPE = MEASUREMENT_TYPE. Go to start at Table 21.

TYPE	INDEX	VALUE	RESOLVER
ONE WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	FRAME_TYPE	
	3	NLOS	
	4 - 5	AoA1	
	6 - 7	AoA2	
	8 - 9	PDoA1	
	10 - 11	PDoA2	
	12 - 13	PDoA1_INDEX	
	14 - 15	PDoA2_INDEX	
	16 - 23	TIMESTAMP	
	24 - 27	BLINK_FRAME_NUM	
	28 - 29	RSSI_RX1	
	30 - 31	RSSI_RX2	
	32 - 41	RFU	
	42	DEV_INFO_SIZE	
	43 - X	DEV_INFO (Size depends on DEV_INFO_SIZE)	
		-> Add #DEV_INFO_SIZE to following indexes	
	44	BLINK_PAYLOAD_SIZE	
	45 - Y	BLINK_PAYLOAD (Size depends on BLINK_PAYLOAD_SIZE)	
TWO WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	UCI_STATUS	Table 100
	3	NLOS	
	4 - 5	DISTANCE	
	6 - 7	AoA1	
	8 - 9	AoA2	
	10 - 11	PDoA1	
	12 - 13	PDoA2	
	14 - 15	PDoA1_Index	
	16 - 17	PDoA2_Index	
	18 - 19	AoA_DEST1	
	20 - 21	AoA_DEST2	
	22	SLOT_INDEX	
	23 - 24	RSSI_RX1	
	25 - 26	RSSI_RX2	
	27 - 28	DISTANCE2	
		!! Continue only if MAC_ADDR_MODE = 0 !!	
	29 - 34	RFU	

Table 98: Resolver - RANGING_DATA_SR100T_OLD_FW. TYPE = MEASUREMENT_TYPE. Go to start at Table 21.

TYPE	INDEX	VALUE	RESOLVER
ONE WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	FRAME_TYPE	
	3	NLOS	
	4 - 5	AoA_AZIMUTH	
	6	AoA_AZIMUTH_FOM	
	7 - 8	AoA_ELEVATION	
	9	AoA_ELEVATION_FOM	
	10 - 17	TIMESTAMP	
	18 - 21	BLINK_FRAME_NUM	
		!! Do not add 6 to following indexes any more (-> Cleared) !!	
	22 - 33 or 28 - 33	RFU (Depends on MAC_ADDR_MODE = 0 or 1)	
	34	DEV_INFO_SIZE	
	35 - X	DEV_INFO (Size depends on DEV_INFO_SIZE)	
		-> Add #DEV_INFO_SIZE to following indexes	
	36	BLINK_PAYLOAD_SIZE	
	37 - Y	BLINK_PAYLOAD (Size depends on BLINK_PAYLOAD_SIZE)	
TWO WAY	0 - 1 or 0 - 7	MAC_ADDR (Depends on MAC_ADDR_MODE = 0 or 1)	
		-> Add 6 to following indexes if MAC_ADDR_MODE = 1	
	2	UCI_STATUS	Table 100
	3	NLOS	
	4 - 5	DISTANCE	
	6 - 7	AOA_AZIMUTH	
	8	AOA_AZIMUTH_FOM	
	9 - 10	AOA_ELEVATION	
	11	AOA_ELEVATION_FOM	
	12 - 13	AOA_DEST_AZIMUTH	
	14	AOA_DEST_AZIMUTH_FOM	
	15 - 16	AOA_DEST_ELEVATION	
	17	AOA_DEST_ELEVATION_FOM	
	18	SLOT_INDEX	
		!! Do not add 6 to following indexes any more (-> Cleared) !!	
	19 - 30 or 25 - 30	RFU (Depends on MAC_ADDR_MODE = 0 or 1)	

Table 99: Resolver - RANGING_DATA_SR150_SR100T. TYPE = MEASUREMENT_TYPE. Go to start at Table 21.

ID	VALUE
0	STATUS_OK
1	STATUS_REJECTED
2	STATUS_FAILED
3	STATUS_SYNTAX_ERROR
4	STATUS_INVALID_PARAM
5	STATUS_INVALID_RANGE
6	STATUS_INVALID_MESSAGE_SIZE
7	STATUS_UNKNOWN_GID
8	STATUS_UNKNOWN_OID
9	STATUS_READ_ONLY
10	STATUS_CMD_RETRY
17	STATUS_SESSION_NOT_EXIST
18	STATUS_SESSION_DUPLICATE
19	STATUS_SESSION_ACTIVE
20	STATUS_MAX_SESSIONS_EXCEEDED
21	STATUS_SESSION_NOT_CONFIGURED
22	STATUS_ACTIVE_SESSIONS_ONGOING
23	STATUS_ERROR_MULTICAST_LIST_FULL
24	STATUS_ERROR_ADDRESS_NOT_FOUND
25	STATUS_ERROR_ADDRESS_ALREADY_PRESENT
32	STATUS_RANGING_TX_FAILED
33	STATUS_RANGING_RX_TIMEOUT
34	STATUS_RANGING_RX_PHY_DEC_FAILED
35	STATUS_RANGING_RX_PHY_TOA_FAILED
36	STATUS_RANGING_RX_PHY_STS_FAILED
37	STATUS_RANGING_RX_MAC_DEC_FAILED
38	STATUS_RANGING_RX_MAC_IE_DEC_FAILED
39	STATUS_RANGING_RX_MAC_IE_MISSING
48	STATUS_DATA_TRANSFER_ERROR
49	STATUS_DATA_NO_CREDIT_AVAILABLE
80	STATUS_BINDING_SUCCESS
81	STATUS_BINDING_FAILURE
82	STATUS_BINDING_LIMIT_REACHED
83	STATUS_CALIBRATION_IN_PROGRESS
84	STATUS_DEVICE_TEMP_REACHED_THERMAL_RUNAWAY
112	STATUS_NO_SE
113	STATUS_SE_RSP_TIMEOUT
114	STATUS_SE_RECOVERY_FAILURE
...	...

ID	VALUE
...	...
115	STATUS_SE_RECOVERY_SUCCESS
116	STATUS_SE_APDU_CMD_FAIL
117	STATUS_SE_AUTH_FAIL
129	STATUS_RANGING_PHY_RX_SECDEC_FAILED
130	STATUS_RANGING_PHY_RX_RSDEC_FAILED
131	STATUS_RANGING_PHY_RX_DEC_FAILED
132	STATUS_RANGING_PHY_RX_ERR_FAILED
133	STATUS_RANGING_PHY_RX_PHR_DECODE_FAILED
134	STATUS_RANGING_PHY_RX_SYNC_SFD_TIMEOUT
135	STATUS_RANGING_PHY_RX_PHR_DATA_RATE_ERROR
136	STATUS_RANGING_PHY_RX_PHR_RANGING_ERROR
137	STATUS_RANGING_PHY_RX_PHR_PREAMBLE_DUR_ERROR
138	STATUS_MAX_ACTIVE_GRANT_DUR_EXD_WARN_NTF
144	STATUS_DATA_TRANSFER_ERROR
145	STATUS_NO_CREDIT_AVAILABLE

Table 100: Resolver - UCI_STATUS. Go to start at Table 21.

ID	VALUE
0	STATUS_INIT
1	STATUS_READY
2	STATUS_ACTIVE
3	STATUS_SE_BINDING_UNKNOWN
4	STATUS_SE_UNBOUND
5	STATUS_SE_BOUND_UNLOCKED
6	STATUS_SE_BOUND_LOCKED
255	STATUS_ERROR

Table 101: Resolver - DEVICE_STATUS. Go to start at Table 21.

ID	VALUE
0	SESSION_STATE_INIT
1	SESSION_STATE_DEINIT
2	SESSION_STATE_ACTIVE
3	SESSION_STATE_IDLE
255	SESSION_ERROR

Table 102: Resolver - SESSION_STATUS. Go to start at Table 21.

ID	VALUE
0	STATE_CHANGE_WITH_SESSION_MANAGEMENT_COMMANDS
1	MAX_RANGING_ROUND_RETRY_COUNT_REACHED
2	MAX_RANGING_BLOCKS_REACHED
32	ERROR_SLOT_LENGTH_NOT_SUPPORTED
33	ERROR_INSUFFICIENT_SLOTS_PER_RR
34	ERROR_MAC_ADDRESS_MODE_NOT_SUPPORTED
35	ERROR_INVALID_RANGING_INTERVAL
36	ERROR_INVALID_STS_CONFIG
37	ERROR_INVALID_RFRAME_CONFIG
128	NO RANGING DATA IN SE
129	KEY FETCH FAILURE
130	DYNAMIC STS NOT SUPPORTED
131	SESSION TERMINATED BY IN BAND STOP SIGNAL
132	FEATURE_NOT_SUPPORTED_FOR_MODEL_ID

Table 103: Resolver - SESSION_REASON_CODE. Go to start at Table 21.

ID	VALUE
0	SESSION_RANGING
1	SESSION_DATA_TRANSFER
208	SESSION_DEVICE_TEST_MODE

Table 104: Resolver - SESSION_TYPE. Go to start at Table 21.

ID	VALUE
0	UPDATE
1	DELETE

Table 105: Resolver - CONTROLLEE_UPDATE_ACTION. Go to start at Table 21.

ID	VALUE
0	READY FOR RANGE
1	IDLE
2	BUSY
3	RFU
255	ERROR

Table 106: Resolver - RNG_NTF_STATUS. Go to start at Table 21.

ID	VALUE
0	NOT_BOUND
1	BOUND_UNLOCK
2	BOUND_LOCK
3	UNKNOWN
4	NO_SE

Table 107: Resolver - BIND_STATUS. Go to start at Table 21.

ID	VALUE
0	SUCCESS
1	FAIL

Table 108: Resolver - SE_STATUS. Go to start at Table 21.

ID	VALUE
0	SUCCESS
1	NO_TEST
255	ERROR

Table 109: Resolver - SE_TEST_LOOP_STATUS. Go to start at Table 21.

ID	VALUE
0	SELEC_SUCCESS
1	SE_ERROR
2	INFINITE_WTX
3	I2C_FAIL_BETWEEN_UWB_AND_ESE
4	I2C _FAILE WITH_IRQ_LOW
5	I2C_FAIL_WITH_IRQ_HIGH
6	I2C_TIMEDOUT
7	I2C_WRITE_TIMEOUT_WITH_IRQ_HIGH

Table 110: Resolver - SE_AID_STATUS. Go to start at Table 21.

ID	VALUE
0	TEST_CMPLT
1	TEST_ABORTD

Table 111: Resolver - SE_TEST_STATUS. Go to start at Table 21.

ID	VALUE
1	Hard Fault
2	Bus Fault
4	Secure Fault
8	Usage Fault
16	Watchdog
32	CoolFlux Fault
64	Assert Fault log

Table 112: Resolver - EXCEPTION_STATUS. Go to start at Table 21.

ID	VALUE
0	SCHD_STATUS_SESSION_SUCCESS
1	SCHD_STATUS_SESSION_CANNOT_SCHEDULE
2	SCHD_STATUS_SESSION_SYNC_FAILURE
3	SCHD_STATUS_SESSION_WIFICOEX_PROTO_VIOLATION

Table 113: Resolver - SCHEDULER_STATUS. Go to start at Table 21.

ID	VALUE
0	HIGH_TO_LOW
1	LOW_TO_HIGH

Table 114: Resolver - UWB_WIFI_COEX_IND_STATUS. Go to start at Table 21.

ID	VALUE
1	WLAN_UWB_IND_HIGH_AT_RR_START
2	WLAN_UWB_IND_HIGH_ DURING_RR

Table 115: Resolver - WLAN_UWB_IND_ERR_STATUS. Go to start at Table 21.

ID	VALUE
0	STATUS_RX_ACQ_FAILURE
1	STATUS_RX_SECDEC_FAILURE
2	STATUS_RX_RSDEC_FAILURE
3	STATUS_RX_DEC_FAILURE
4	STATUS_RX_DEC_SUCCESS
5	STATUS_RX_DEC_NO_DATA
6	STATUS_PHY_RX_ERR
7	STATUS_RX_STS_FAILURE
8	STATUS_RX_TOA_DETECT_FAILURE
9	STATUS_RX_PHR_DEC_FAILURE
10	STATUS_RX_SYNC_SFD_FAILURE
11	STATUS_PHR_DATA_RATE_ERROR
12	STATUS_RX_PHR_RANGING_ERROR
13	STATUS_RX_PHR_PREAMBLE_DUR_ERROR

Table 116: Resolver - DEC_STATUS. Go to start at Table 21.

ID	VALUE
0	STATUS_OK_MULTICAST_LIST_UPDATE
1	STATUS_ERROR_MULTICAST_LIST_FULL
2	STATUS_ERROR_KEY_FETCH_FAIL
3	STATUS_ERROR_SUB_SESSION_ID_NOT_FOUND

Table 117: Resolver - SESSION_UPDATE_CONTROLLER_MULTICAST_LIST_STATUS. Go to start at Table 21.

ID	VALUE
0	uninitialized
1	NXP_REF
42	SSG
115	RHODES
11	AMOTECH

Table 118: Resolver - PLATFORM_ID. Go to start at Table 21.

ID	VALUE
1	V1
2	V2
3	GN20_V1

Table 119: Resolver - VARIANT_ID. Go to start at Table 21.

ID	VALUE
0	VCO_PLL
1	TX_POWER
2	XTAL_CAP_GM_CTRL
3	RSSI_CALIB_CONSTANT1
4	RSSI_CALIB_CONSTANT2
5	SNR_CALIB_CONSTANT
6	MANUAL_TX_POW_CTRL
7	PDOA_OFFSET
8	PA_PPA_CALIB_CTRL
9	TX_TEMPERATURE_COMP

Table 120: Resolver - CALIB_PARAM. Go to start at Table 21.

ID	VALUE
0	Forbid_rsp_for_next_command
1	Forbid_rsp_infinitely
2	Force_FW_Assert_or_crash
3	Forbid_session_status_NTF_for_next_session_state_cha...ands

Table 121: Resolver - ERR_OPTION. Go to start at Table 21.

ID	VALUE
0	DEFAULT
1	CUSTOM_NOT_INTEGRITY_PROTECTED
2	CUSTOM_AUTH_PENDING
3	CUSTOM_DEVICE_SPECIFIC_TAG_AUTHENTICATED
4	CUSTOM_MODEL_SPECIFIC_TAG_AUTHENTICATED

Table 122: Resolver - CALIB_STATE. Go to start at Table 21.

ID	VALUE
0	RX1
1	RX2

Table 123: Resolver - RX_TOA_FIRST_PATH. Go to start at Table 21.

ID	VALUE
0	I2C_INTERFACE_IDLE
1	I2C_INTERFACE_BUSY

Table 124: Resolver - INTERFACE_STATUS. Go to start at Table 21.

ID	SUB-ID	TAG	LEN	RESOLVER
0	-	DEVICE_TYPE	1	Table 130
1	-	RANGING_CONFIG	1	Table 131
2	-	STS_CONFIG	1	Table 132
3	-	MULTI_NODE_MODE	1	Table 133
4	-	CHANNEL_ID	1	Table 134
5	-	NUMBER_OF_CONTROLEES	1	Table 135
6	-	SRC_MAC_ADDRESS	[2, 8]	Table 136
7	-	DST_MAC_ADDRESS_LIST	2	Table 137
8	-	SLOT_DURATION	2	Table 138
9	-	RANGING_INTERVAL	4	Table 139
10	-	STS_INDEX	4	Table 140
11	-	MAC_TYPE	1	Table 141
12	-	RANGING_ROUND_CONTROL	1	Table 142
13	-	AOA_RESULT_REQ	1	Table 143
14	-	RNG_DATA_NTF	1	Table 144
15	-	RNG_DATA_NTF_PROXIMITY_NEAR	2	Table 145
16	-	RNG_DATA_NTF_PROXIMITY_FAR	2	Table 146
17	-	DEVICE_ROLE	1	Table 147
18	-	RFRAME_CONFIG	1	Table 148
19	-	RX_MODE	1	Table 149
20	-	PREAMBLE_CODE_INDEX	1	Table 150
21	-	SFD_ID	1	Table 151
22	-	PSDU_DATA_RATE	1	Table 152
23	-	PREMABLE_DUR	1	Table 153
24	-	RX_ANTENNA_PAIR_SEL	1	Table 154
25	-	MAC_CFG	1	Table 155
26	-	RANGING_TIME_STRUCT	1	Table 156
27	-	SLOTS_PER_RR	1	Table 157
28	-	TX_ADAPTIVE_PAYLOAD_POWER	1	Table 158
29	-	TX_ANTENNA_SELECTION	1	Table 159
30	-	RESPONDER_SLOT_INDEX	1	Table 160
31	-	PRF_MODE	1	Table 161
32	-	MAX_CONTENTION_PHASE_LENGTH	1	Table 162
33	-	MAX_CONTENTION_PHASE_UPDATE_LENGTH	1	Table 163
34	-	SCHEDULED_MODE	1	Table 164
35	-	KEY_ROTATION	1	Table 165
36	-	KEY_ROTATION_RATE	1	Table 166
37	-	SESSION_PRIORITY	1	-
...	...	...	...	...

ID	SUB-ID	TAG	LEN	RESOLVER
...	...	...	...	...
38	-	MAC_ADDRESS_MODE	1	Table 167
39	-	VENDOR_ID	2	-
40	-	STATIC_STS_IV	6	-
41	-	NUMBER_OF_STS_SEGMENTS	1	Table 168
42	-	MAX_RR_RETRY	2	Table 169
43	-	UWB_INITIATION_TIME	4	Table 170
44	-	RANGING_ROUND_HOPPING	1	Table 171
45	-	BLOCK_STRIDING	1	Table 172
46	-	RESULT_REPORT_CONFIG	1	Table 173
47	-	IN_BAND_TERMINATION_ATTEMPT_COUNT	1	-
48	-	SUB_SESSION_ID	4	-
49	-	TDOA_REPORT_FREQUENCY	2	-
50	-	BLINK_RANDOM_INTERVAL	2	-
51	-	AUTHENTICITY_TAG	1	-
52	-	MAX_NUMBER_OF_BLOCKS	2	-
227	0	TOA_MODE	1	Table 174
227	1	CIR_CAPTURE_MODE	1	Table 175
227	2	MAC_PAYLOAD_ENCRYPTION	1	Table 176
227	3	RX_ANTENNA_POLARIZATION_OPTION	1	Table 177
227	4	RX_ANTENNA_SELECTION_RFM	undefined	-
227	5	SESSION_SYNC_ATTEMPTS	1	Table 178
227	6	SESSION_SHED_ATTEMPTS	1	Table 179
227	7	SCHED_STATUS_NTF	1	Table 180
227	8	TX_POWER_DELTA_FCC	1	Table 181
227	9	TEST_KDF_FEATURE	1	Table 182
227	10	DUAL_AOA_PREAMBLE_STS	1	Table 183
227	11	TX_POWER_TEMP_COMP	1	Table 184
227	12	WIFI_COEX_MAX_TOL_COUNT	1	Table 185
227	13	ADAPTIVE_HOPPING_THRESHOLD	undefined	-
227	14	RX_MODE_2	undefined	-
227	15	RX_ANTENNA_SELECTION	undefined	-
227	16	TX_ANTENNA_SELECTION_2	undefined	-
227	17	MAX_CONTENTION_PHASE_LENGTH_2	undefined	-
227	18	CONTENTION_PHASE_UPDATE_LENGTH_2	undefined	-
227	22	INBAND_DATA_TX_BLOCKS	undefined	-
227	23	INBAND_DATA_RX_BLOCKS	undefined	-
227	24	RANGING_SUSPEND_MODE	undefined	-
227	25	RX_ANTENNA_SELECTION_RFM_2	undefined	-
227	160	WRAPPED_RDS	undefined	-
...	...	...	...	...

ID	SUB-ID	TAG	LEN	RESOLVER
...	...	...	...	...
228	0	THREAD_SECURE	undefined	-
228	1	THREAD_S_ISR	undefined	-
228	2	THREAD_NS_ISR	undefined	-
228	3	THREAD_SHELL	undefined	-
228	4	THREAD_PHY	undefined	Table 186
228	5	THREAD_RANGING	undefined	Table 187
228	6	THREAD_SE	undefined	-
228	7	THREAD_UWB_WLAN_COEX	undefined	-
228	16	DATA_LOGGER	undefined	Table 188
228	17	CIR_LOG_NTF	undefined	Table 189
228	18	PSDU_LOG_NTF	undefined	Table 190
228	19	RFRAME_LOG_NTF	undefined	Table 191
228	20	TEST_CONTENTION_RANGING_FEATURE	undefined	Table 192

Table 125: Resolver - APP_TLV. Go to start at Table 21.

ID	SUB-ID	TAG	LEN	RESOLVER
0	-	DEVICE_STATUS	1	Table 101
1	-	LOW_POWER_MODE	1	Table 193
144	-	TEST_UCI_VERSION	2	-
227	0	DEVICE_NAME	1	-
227	1	FW_VERSION	3	Table 194
227	2	NXP_UCI_VERSION	3	Table 195
227	3	NXP_CHIP_ID	16	-
227	4	FW_BOOT_MODE	1	-
228	0	DELAY_CALIBRATION	8	Table 196
228	1	AOA_CALIBRATION_CTRL	128	-
228	2	DPD_WAKEUP_SRC	1	Table 197
228	3	WTX_COUNT_CONFIG	1	Table 198
228	4	DPD_ENTRY_TIMEOUT	2	-
228	5	WIFI_COEX_FEATURE	4	Table 199
228	6	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_1_CH5	72	-
228	7	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_1_CH6	72	-
228	8	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_1_CH8	72	-
228	9	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_1_CH9	72	-
228	10	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_2_CH5	72	-
228	11	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_2_CH6	72	-
228	12	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_2_CH8	72	-
228	13	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_2_CH9	72	-
228	14	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_3_CH5	72	-
228	15	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_3_CH6	72	-
228	16	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_3_CH8	72	-
228	17	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_3_CH9	72	-
228	18	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_4_CH5	72	-
228	19	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_4_CH6	72	-
228	20	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_4_CH8	72	-
228	21	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_4_CH9	72	-
228	22	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_5_CH5	72	-
228	23	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_5_CH6	72	-
228	24	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_5_CH8	72	-
228	25	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_5_CH9	72	-
228	26	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_6_CH5	72	-
228	27	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_6_CH6	72	-
228	28	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_6_CH8	72	-
228	29	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_6_CH9	72	-
...	...	...	...	...

ID	SUB-ID	TAG	LEN	RESOLVER
...	...	...	...	...
228	30	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_7_CH5	72	-
228	31	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_7_CH6	72	-
228	32	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_7_CH8	72	-
228	33	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_7_CH9	72	-
228	34	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_8_CH5	72	-
228	35	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_8_CH6	72	-
228	36	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_8_CH8	72	-
228	37	AOA_FINE_CALIB_CTRL_RX_ANT_PAIR_8_CH9	72	-
228	38	DDFS_TONE_CONFIG_ENABLE	1	-
228	39	DDFS_TONE_CONFIG	72	-
228	40	TX_TELEC_CONFIG	4	-
229	0	DUMP_SE_COMM_DATA	1	Table 200

Table 126: Resolver - DEVICE_TLV. Go to start at Table 21.

ID	SUB-ID	TAG	LEN	RESOLVER
0	-	MEMORY_ALLOCATIONS	16	Table 201
1	-	DYNAMIC_MEMORY_OBJECT_COUNTS	28	Table 202
2	-	SECURE_REGION_MAIN_STACK_DETAILS	8	Table 203
3	-	NON_SECURE_REGION_MAIN_STACK_DETAILS	8	Table 204
4	-	SECURE_REGION_PROCESS_STACK_DETAILS	52	Table 205
5	-	DYNAMIC_MEMORY_DETAILS	8	Table 206
6	-	NON_SECURE_REGION_EACH_APPLICATION_...AILS	64	Table 207
7	-	NON_SECURE_REGION_OS_IDLE_THREAD_ST...AILS	8	Table 208
8	-	NON_SECURE_REGION_OS_TIMER_THREAD_S...AILS	8	Table 209

Table 127: Resolver - MEM_TLV. Go to start at Table 21.

ID	SUB-ID	TAG	LEN	RESOLVER
0	-	KDF_BLOCK_INDEX	2	-
1	-	KDF_STS_INDEX	4	-
2	-	KDF_CONFIG_DIGEST	4	-
3	-	KDF_DERIVED_AUTH_IV	4	-
4	-	KDF_DERIVED_AUTH_KEY	4	-
5	-	KDF_DERIVED_PAYLOAD_KEY	4	-
6	-	KDF_SALTED_HASH	4	-

Table 128: Resolver - KDF_NTF_TLV. Go to start at Table 21.

ID	SUB-ID	TAG	LEN	RESOLVER
0	-	NUM_PACKETS	4	Table 210
1	-	T_GAP	4	Table 211
2	-	T_START	4	Table 212
3	-	T_WIN	4	Table 213
4	-	RANDOMIZE_PSDU	undefined	Table 214
5	-	RAW_PHR	2	Table 215
6	-	RMARKER_TX_START	4	Table 216
7	-	RMARKER_RX_START	4	Table 217
8	-	STS_INDEX_AUTO_INCR	undefined	Table 218
229	0	RSSI_AVG_FILT_CNT	undefined	Table 219
229	1	RSSI_CALIBRATION_OPTION	undefined	Table 220
229	2	AGC_GAIN_VAL_RX	2	Table 221
229	3	TEST_SESSION_STS_KEY_OPTION	1	Table 222

Table 129: Resolver - TEST_TLV. Go to start at Table 21.

ID	VALUE
0	CONTROLEE
1	CONTROLLER

Table 130: Resolver - DEVICE_TYPE. Go to start at Table 21.

ID	VALUE
0	ONE_WAY
1	SS_TWR
2	DS_TWR

Table 131: Resolver - RANGING_CONFIG. Go to start at Table 21.

ID	VALUE
0	NO_SE_STATIC_STS
1	SE_DYNAMIC_STS
2	SE_DYNAMIC_STS_FOR_CONTROLEE_INDIVIDUAL_KEY
3	STATIC_STS_TDOA

Table 132: Resolver - STS_CONFIG. Go to start at Table 21.

ID	VALUE
0	SINGLE_DEVICE_TO_SINGLE_DEVICE
1	ONE_TO_MANY
2	MANY_TO_MANY
3	RESERVED

Table 133: Resolver - MULTI_NODE_MODE. Go to start at Table 21.

ID	VALUE
5	CH_5
6	CH_6
8	CH_8
9	CH_9
N	CH

Table 134: Resolver - CHANNEL_ID. Go to start at Table 21.

ID	VALUE
1	SINGLE_ANCHOR

Table 135: Resolver - NUMBER_OF_CONTROLEES. Go to start at Table 21.

ID	VALUE
35840	MASTER_ADDR

Table 136: Resolver - SRC_MAC_ADDRESS. Go to start at Table 21.

ID	VALUE
36096	ANCHOR_ADDR

Table 137: Resolver - DST_MAC_ADDRESS_LIST. Go to start at Table 21.

ID	VALUE
2000	DEFAULT_RANGING

Table 138: Resolver - SLOT_DURATION. Go to start at Table 21.

ID	VALUE
192	DEFAULT_RANGING

Table 139: Resolver - RANGING_INTERVAL. Go to start at Table 21.

ID	VALUE
0	STS_ZERO

Table 140: Resolver - STS_INDEX. Go to start at Table 21.

ID	VALUE
0	CRC_16
1	CRC_32

Table 141: Resolver - MAC_TYPE. Go to start at Table 21.

ID	VALUE
0	MEASUREMENT_REPORT_PHASE
2	RANGING_CONTROL_PHASE
3	DEFAULT

Table 142: Resolver - RANGING_ROUND_CONTROL. Go to start at Table 21.

ID	VALUE
0	NO_AOA_REPORT
1	NEG_90_TO_90

Table 143: Resolver - AOA_RESULT_REQ. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE
2	ENABLE_IN_PROXIMITY_RANGE

Table 144: Resolver - RNG_DATA_NTF. Go to start at Table 21.

ID	VALUE
0	DEFAULT_RANGING

Table 145: Resolver - RNG_DATA_NTF_PROXIMITY_NEAR. Go to start at Table 21.

ID	VALUE
20000	DEFAULT_RANGING

Table 146: Resolver - RNG_DATA_NTF_PROXIMITY_FAR. Go to start at Table 21.

ID	VALUE
0	RESPONDER
1	INITIATOR
2	MASTER_ANCHOR

Table 147: Resolver - DEVICE_ROLE. Go to start at Table 21.

ID	VALUE
0	NO_STS
1	STS_FOLLOWS_SFD
2	STS_FOLLOWS_PSDU
3	STS_FOLLOWS_SFD_NO_PPDU

Table 148: Resolver - RFRAME_CONFIG. Go to start at Table 21.

ID	VALUE
0	DUAL_RX
1	RX1
2	RX2

Table 149: Resolver - RX_MODE. Go to start at Table 21.

ID	VALUE
10	DEFAULT_BPRF

Table 150: Resolver - PREAMBLE_CODE_INDEX. Go to start at Table 21.

ID	VALUE
0	DEFAULT_BPRF
1	HPRF_1
2	BPRF
3	HPRF_3

Table 151: Resolver - SFD_ID. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 152: Resolver - PSDU_DATA_RATE. Go to start at Table 21.

ID	VALUE
0	ZERO
1	DEFAULT

Table 153: Resolver - PREMABLE_DUR. Go to start at Table 21.

ID	VALUE
0	SINGLE_RX
1	DEFAULT
2	ANTENNA_2
4	ANTENNA_4
8	ANTENNA_8

Table 154: Resolver - RX_ANTENNA_PAIR_SEL. Go to start at Table 21.

ID	VALUE
0	DEFAULT_PER
3	DEFAULT_RANGING_DATA

Table 155: Resolver - MAC_CFG. Go to start at Table 21.

ID	VALUE
0	INTERVAL_BASED_SCHEDULING
1	BLOCK_BASED_SCHEDULING

Table 156: Resolver - RANGING_TIME_STRUCT. Go to start at Table 21.

ID	VALUE
24	DEFAULT_SLOTS

Table 157: Resolver - SLOTS_PER_RR. Go to start at Table 21.

ID	VALUE
0	DISABLED
1	ENABLED

Table 158: Resolver - TX_ADAPTIVE_PAYLOAD_POWER. Go to start at Table 21.

ID	VALUE
1	ANTENNA_1
2	ANTENNA_2

Table 159: Resolver - TX_ANTENNA_SELECTION. Go to start at Table 21.

ID	VALUE
0	RESERVED
1	RESPONDER_1

Table 160: Resolver - RESPONDER_SLOT_INDEX. Go to start at Table 21.

ID	VALUE
0	BPRF
1	HPRF

Table 161: Resolver - PRF_MODE. Go to start at Table 21.

ID	VALUE
50	DEFAULT

Table 162: Resolver - MAX_CONTENTION_PHASE_LENGTH. Go to start at Table 21.

ID	VALUE
5	DEFAULT

Table 163: Resolver - MAX_CONTENTION_PHASE_UPDATE_LENGTH. Go to start at Table 21.

ID	VALUE
0	CONTENTION_BASED_RANGING
1	TIME_SCHEDULED_RANGING

Table 164: Resolver - SCHEDULED_MODE. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 165: Resolver - KEY_ROTATION. Go to start at Table 21.

ID	VALUE
5	DEFAULT

Table 166: Resolver - KEY_ROTATION_RATE. Go to start at Table 21.

ID	VALUE
0	MAC_ADDR_TWO_BYTES
1	MAC_ADDR_EIGHT_AND_TWO_IN_HDR
2	MAC_ADDR_EIGHT_AND_EIGHT_IN_HDR

Table 167: Resolver - MAC_ADDRESS_MODE. Go to start at Table 21.

ID	VALUE
1	ONE_STS_SEGMENT
2	TWO_STS_SEGMENT

Table 168: Resolver - NUMBER_OF_STS_SEGMENTS. Go to start at Table 21.

ID	VALUE
0	DEFAULT_RETRY

Table 169: Resolver - MAX_RR_RETRY. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 170: Resolver - UWB_INITIATION_TIME. Go to start at Table 21.

ID	VALUE
0	DISABLED
1	ENABLED

Table 171: Resolver - RANGING_ROUND_HOPPING. Go to start at Table 21.

ID	VALUE
0	DEFAULT_BLOCK_STRIDING_DISABLED
1	BLOCK_STRIDING_ENABLED

Table 172: Resolver - BLOCK_STRIDING. Go to start at Table 21.

ID	VALUE
0	ALL_REPORTS_DISABLED
1	TOF_REPORT
15	ALL_REPORTS_ENABLED

Table 173: Resolver - RESULT_REPORT_CONFIG. Go to start at Table 21.

ID	VALUE
0	FIRSTPATH_ON_RX1
1	FIRSTPATH_ON_RX2
2	FIRSTPATH_ON_RX1RX2

Table 174: Resolver - TOA_MODE. Go to start at Table 21.

ID	VALUE
0	PRE_SYNC_RX1
1	PRE_SYNC_RX2
2	PRE_STS_RX1
3	PRE_STS_RX2
4	POST_SYNC_RX1
5	POST_SYNC_RX2
6	POST_STS_RX1
7	POST_STS_RX2
0x8-0xF	RFU
84	DEFAULT

Table 175: Resolver - CIR_CAPTURE_MODE. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 176: Resolver - MAC_PAYLOAD_ENCRYPTION. Go to start at Table 21.

ID	VALUE
0	STRAIGHT
1	REVERSE

Table 177: Resolver - RX_ANTENNA_POLARIZATION_OPTION. Go to start at Table 21.

ID	VALUE
3	DEFAULT

Table 178: Resolver - SESSION_SYNC_ATTEMPTS. Go to start at Table 21.

ID	VALUE
3	DEFAULT

Table 179: Resolver - SESSION_SHED_ATTEMPTS. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE_ALL_SESSION
2	ENABLE_FAILURE_SESSION

Table 180: Resolver - SCHED_STATUS_NTF. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 181: Resolver - TX_POWER_DELTA_FCC. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 182: Resolver - TEST_KDF_FEATURE. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 183: Resolver - DUAL_AOA_PREAMBLE_STS. Go to start at Table 21.

ID	VALUE
0	DEFAULT_DISABLE
1	ENABLE

Table 184: Resolver - TX_POWER_TEMP_COMP. Go to start at Table 21.

ID	VALUE
3	DEFAULT

Table 185: Resolver - WIFI_COEX_MAX_TOL_COUNT. Go to start at Table 21.

ID	VALUE
7	DEFAULT_RANGING

Table 186: Resolver - THREAD_PHY. Go to start at Table 21.

ID	VALUE
23	DEFAULT_RANGING

Table 187: Resolver - THREAD_RANGING. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 188: Resolver - DATA_LOGGER. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 189: Resolver - CIR_LOG_NTF. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 190: Resolver - PSDU_LOG_NTF. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 191: Resolver - RFRAME_LOG_NTF. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 192: Resolver - TEST_CONTENTION_RANGING_FEATURE. Go to start at Table 21.

ID	VALUE
0	DISABLED
1	ENABLED

Table 193: Resolver - LOW_POWER_MODE. Go to start at Table 21.

INDEX	VALUE
0	FW_MAJ
1	FW_MIN
2	FW_RC

Table 194: Resolver - FW_VERSION. Go to start at Table 21.

INDEX	VALUE
0	UCI_MAJ
1	UCI_MIN
2	UCI_PATCH

Table 195: Resolver - NXP_UCI_VERSION. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 196: Resolver - DELAY_CALIBRATION. Go to start at Table 21.

ID	VALUE
0	SE_INTERFACE
1	GPIO1
2	GPIO3

Table 197: Resolver - DPD_WAKEUP_SRC. Go to start at Table 21.

ID	VALUE
20	DEFAULT_MIN
180	MAX

Table 198: Resolver - WTX_COUNT_CONFIG. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE
2	DEBUG_VERBOSE

Table 199: Resolver - WIFI_COEX_FEATURE. Go to start at Table 21.

ID	VALUE
0	DISABLE
1	ENABLE

Table 200: Resolver - DUMP_SE_COMM_DATA. Go to start at Table 21.

INDEX	VALUE
0-3	SE_PROC_STACK
4-7	SE_MAIN_STACK
8-11	NS_MAIN_STACK
12-15	DYNAMIC_MEM

Table 201: Resolver - MEMORY_ALLOCATIONS. Go to start at Table 21.

INDEX	VALUE
0-3	THREAD_OBJS
4-7	TIMER_OBJS
8-11	EVENT_FLG_OBJS
12-15	MUTEX_OBJS
16-19	SEMAPHORE_OBJS
20-23	MEM_POOL_OBJS
24-27	NUM_MSG_Q_OBJS

Table 202: Resolver - DYNAMIC_MEMORY_OBJECT_COUNTS. Go to start at Table 21.

INDEX	VALUE
0-3	TOTAL_SE_STACK
4-7	USED_SE_STACK

Table 203: Resolver - SECURE_REGION_MAIN_STACK_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-3	TOTAL_NS_STACK
4-7	USED_NS_STACK

Table 204: Resolver - NON_SECURE_REGION_MAIN_STACK_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-1	SE_NUM_SLOTS
2-3	SE_SLOT_SIZE
4-7	SE_THREAD1
8-9	SE_STACK_SIZE1
10-13	SE_THREAD2
14-15	SE_STACK_SIZE2
16-19	SE_THREAD3
20-21	SE_STACK_SIZE3
22-25	SE_THREAD4
26-27	SE_STACK_SIZE4
28-31	SE_THREAD5
32-33	SE_STACK_SIZE5
34-37	SE_THREAD6
38-39	SE_STACK_SIZE6
40-43	SE_THREAD7
44-45	SE_STACK_SIZE7
46-49	SE_THREAD8
50-51	SE_STACK_SIZE8

Table 205: Resolver - SECURE_REGION_PROCESS_STACK_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-3	TOTAL
4-7	USED

Table 206: Resolver - DYNAMIC_MEMORY_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-3	NS_THREAD1
4-5	TOTAL_NS_STACK1
6-7	USED_NS_STACK1
8-11	NS_THREAD2
12-13	TOTAL_NS_STACK2
14-15	USED_NS_STACK2
16-19	NS_THREAD3
20-21	TOTAL_NS_STACK3
22-23	USED_NS_STACK3
24-27	NS_THREAD4
28-29	TOTAL_NS_STACK4
30-31	USED_NS_STACK4
32-35	NS_THREAD5
36-37	TOTAL_NS_STACK5
38-39	USED_NS_STACK5
40-43	NS_THREAD6
44-45	TOTAL_NS_STACK6
46-47	USED_NS_STACK6
48-51	NS_THREAD7
52-53	TOTAL_NS_STACK7
54-55	USED_NS_STACK7
56-59	NS_THREAD8
60-61	TOTAL_NS_STACK8
62-63	USED_NS_STACK8

Table 207: Resolver - NON_SECURE_REGION_EACH_APPLICATION_THREAD_STACK_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-3	TOTAL_IDLE_THREAD_STACK
4-7	USED_IDLE_THREAD_STACK

Table 208: Resolver - NON_SECURE_REGION_OS_IDLE_THREAD_STACK_DETAILS. Go to start at Table 21.

INDEX	VALUE
0-3	TOTAL_TIMER_THREAD_STACK
4-7	USED_TIMER_THREAD_STACK

Table 209: Resolver - NON_SECURE_REGION_OS_TIMER_THREAD_STACK_DETAILS. Go to start at Table 21.

ID	VALUE
1000	DEFAULT

Table 210: Resolver - NUM_PACKETS. Go to start at Table 21.

ID	VALUE
2000	DEFAULT

Table 211: Resolver - T_GAP. Go to start at Table 21.

ID	VALUE
450	DEFAULT

Table 212: Resolver - T_START. Go to start at Table 21.

ID	VALUE
750	DEFAULT

Table 213: Resolver - T_WIN. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 214: Resolver - RANDOMIZE_PSDU. Go to start at Table 21.

ID	VALUE
0	DEFAULT
16384	RANGE_ENABLED

Table 215: Resolver - RAW_PHR. Go to start at Table 21.

ID	VALUE
0	DEFAULT
1000	LOOPBACK

Table 216: Resolver - RMARKER_TX_START. Go to start at Table 21.

ID	VALUE
0	DEFAULT
1000	LOOPBACK

Table 217: Resolver - RMARKER_RX_START. Go to start at Table 21.

ID	VALUE
0	DEFAULT
1	STS_IDX_INCR

Table 218: Resolver - STS_INDEX_AUTO_INCR. Go to start at Table 21.

ID	VALUE
0	DEFAULT

Table 219: Resolver - RSSI_AVG_FILT_CNT. Go to start at Table 21.

ID	VALUE
1	DEFAULT

Table 220: Resolver - RSSI_CALIBRATION_OPTION. Go to start at Table 21.

ID	VALUE
1	DEFAULT

Table 221: Resolver - AGC_GAIN_VAL_RX. Go to start at Table 21.

ID	VALUE
0	PROPRIETARY
1	IEEE

Table 222: Resolver - TEST_SESSION_STS_KEY_OPTION. Go to start at Table 21.

INDEX	VALUE
0	MAPPING
1	DEC_STATUS
2	NLOS
3 - 4	FIRST_PATH_INDEX
5 - 6	MAIN_PATH_INDEX
7	SNR_MAIN_PATH
8	SNR_FIRST_PATH
9 - 10	SNR_TOTAL
11 - 12	RSSI
13 - 16	CIR_MAIN_POWER
17 - 20	CIR_FIRST_PATH_POWER
21 - 22	NOISE_VARIANCE
23	CFO
25 - 26	AoA_PHASE
27 - 90	CIR_SAMPLES

Table 223: Resolver - RFRAME_MEASUREMENT_DATA. Go to start at Table 21.

INDEX	VALUE	RESOLVER
0 - 3	SUB_SESSION_ID	
4	SESSION_UPDATE_CONTROLLER_MULTICAST_LIST_STATUS	Table 117

Table 224: Resolver - STATUS_LIST. Go to start at Table 21.

INDEX	VALUE
0	SLOT_INDEX
1	PSDU_SIZE
2 - N	PSDU_DATA (Depends on PSDU_SIZE)

Table 225: Resolver - PSDU_LOG_DATA. Go to start at Table 21.

INDEX	VALUE	RESOLVER
0 - 3	SESSION_ID	
4	SESSION_TYPE	Table 104
5	SESSION_STATE	

Table 226: Resolver - SESSION_INFO. Go to start at Table 21.

INDEX	VALUE	RESOLVER
0 - 3	SESSION_ID	
4	SCHEDULER_STATUS	Table 113
5 - 8	NO_OF_SUCCESSFUL_SCHEDULEING	
9 - 12	NO_OF_UN_SUCCESSFUL_SCHEDULEING	
13	PRIORITY	

Table 227: Resolver - SESSION_DATA. Go to start at Table 21.

BIBLIOGRAPHY

[1] Admin of rtl-sdr.com. *Roundup of Software Defined Radios*. https://www.rtl-sdr.com/roundup-software-defined-radios/. Accessed: 2021-11-17.

[2] Arm Limited. *Core registers*. https://developer.arm.com/documentation/100235/0004/the-cortex-m33-processor/programmer-s-model/core-registers. Accessed: 2022-01-18.

[3] D. Barnwell. *What's the deal with Ultra Wideband*. Mar. 11, 2021. https://www.bmw.com/en/innovation/bmw-digital-key-plus-ultra-wideband.html. Accessed: 2021-10-14.

[4] B. Barrett. *The Biggest iPhone News Is a Tiny New Chip Inside It*. Sep. 12, 2019. https://www.wired.com/story/apple-u1-chip/. Accessed: 2022-01-02.

[5] L. Bongiorni. *Samsung-SmartTag-Hack*. May 25, 2021. https://github.com/whid-injector/Samsung-SmartTag-Hack. Accessed: 2021-12-11.

[6] Bosch. *Perfectly Keyless*. https://www.bosch-mobility-solutions.com/de/loesungen/software-und-services/perfectly-keyless/. Accessed: 2021-10-14.

[7] Bundesnetzagentur. *Service WLAN*. https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Allgemeines/Bundesnetzagentur/Publikationen/service/WLAN.pdf?__blob=publicationFile&v=9. Accessed: 2021-11-17.

[8] Car Connectivity Consortium. *About Us*. https://global-carconnectivity.org/about/. Accessed: 2021-10-15.

[9] Car Connectivity Consortium. *Digital Key*. https://global-carconnectivity.org/. Accessed: 2022-01-31.

[10] J. Classen. *AirTag: AirTechno and Firmware Downgrade*. YouTube. Oct. 28, 2021. https://www.youtube.com/watch?v=C4JyI_WUNJ8. Accessed: 2021-12-10.

[11] J. Classen and A. Heinrich. *Wibbly Wobbly, Timey Wimey – What's Really Inside Apple's U1 Chip*. Presentation at Black Hat USA 2021, August 2021.

[12] I. Dotlic, A. Connell, H. Ma, J. Clancy, and M. McLaughlin. "Angle of Arrival Estimation Using Decawave DW1000 Integrated Circuits." In: *14th Workshop on Positioning, Navigation and Communications (WPNC)* (2017), pp. 1–6.

[13] Federal Communications Comission. *Revision of Part 15 of the Commission's Rules Regarding Ultra WideBand Transmission Systems*. Apr. 22, 2002. https://docs.fcc.gov/public/attachments/FCC-02-48A1.pdf. Accessed: 2021-10-08.

[14] FiRa Consortium. *Membership Information*. https://www.firaconsortium.org/membership/information. Accessed: 2021-10-13.

[15] FiRa Consortium. *Mission & Goals*. https://www.firaconsortium.org/about/mission. Accessed: 2021-10-13.

[16] FiRa Consortium. *Our Members*. https://www.firaconsortium.org/about/members. Accessed: 2021-11-19.

[17] FiRa Consortium. *UWB Use Cases*. https://www.firaconsortium.org/discover/use-cases. Accessed: 2022-01-31.

[18] L. Flueratoru, S. Wehrli, M. Magno, and D. Niculescu. "On the Energy Consumption and Ranging Accuracy of Ultra-Wideband Physical Interfaces." In: *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. 2020, pp. 1–7.

[19] Google. *Factory Images for Nexus and Pixel Devices*. https://developers.google.com/android/images. Accessed: 2021-12-15.

[20] iFixit. *Samsung Galaxy S21 Ultra Teardown*. Mar. 11, 2021. https://www.ifixit.com/Teardown/Samsung+Galaxy+S21+Ultra+Teardown/141188. Accessed: 2021-12-07.

[21] E. Karapistoli, F. Pavlidou, I. Gragopoulos, and I. Tsetsinas. "An overview of the IEEE 802.15.4a Standard." In: *IEEE Communications Magazine* 48.1 (2010), pp. 47–53. DOI: 10.1109/MCOM.2010.5394030.

[22] T. Kröll and S. Sterz. *ARIstoteles*. Aug. 17, 2021. https://github.com/seemoo-lab/aristoteles. Accessed: 2021-09-16.

[23] P. Leu, G. Camurati, A. Heinrich, M. Roeschlin, C. Anliker, M. Hollick, S. Čapkun, and J. Classen. *Ghost Peak: Practical Distance Reduction Attacks Against HRP UWB Ranging*. Preprint. Nov. 9, 2021. https://arxiv.org/abs/2111.05313. Accessed: 2021-12-22.

[24] LimitedResults. *nRF52 Debug Resurrection (APPROTECT Bypass) Part 1*. June 10, 2020. https://limitedresults.com/2020/06/nrf52-debug-resurrection-approtect-bypass/. Accessed: 2022-01-04.

[25] Mobile Knowledge. *Products*. https://www.themobileknowledge.com/products/. Accessed: 2022-01-10.

[26] NXP Semiconductors. *NXP Introduces Secure UWB Fine Ranging Chipset to Allow Broad Deployment in Mobile Devices*. Sep. 17, 2019. https://www.nxp.com/company/about-nxp/nxp-introduces-secure-uwb-fine-ranging-chipset-to-allow-broad-deployment-in-mobile-devices:NW-UWB-FINE-RANGING. Accessed: 2021-10-20.

[27] NXP Semiconductors. *NXP JCOP4.2 on P73N2MoBo.2C2, Rev. 1.2*. Feb. 21, 2019. https://commoncriteriaportal.org/files/epfiles/[ST-LITE]%20JCOP4_2_P73_SecurityTarget-Lite_Public_v1.2_20190221.pdf. Accessed: 2021-12-21.

[28] NXP Semiconductors. *NXP Trimension™ Ultra-Wideband Technology Powers Xiaomi MIX4 Smartphone to Deliver New "Point to Connect" Smart Home Solution*. Sep. 26, 2021. https://www.nxp.com/company/about-nxp/nxp-trimension-ultra-wideband-technology-powers-xiaomi-mix4-smartphone-to-deliver-new-point-to-connect-smart-home-solution:NW-NXP-TRIMENSION-ULTRA-WIDEBAND-TECHNOLOGY-POWERS. Accessed: 2021-12-16.

[29] NXP Semiconductors. *QN9090(T)/QN9030(T) Bluetooth Low Energy 5.0 wireless MCU, Rev. 1.2*. June 23, 2021. https://www.nxp.com/products/wireless/bluetooth-low-energy/qn9090-30-bluetooth-low-energy-mcu-with-armcortex-m4-cpu-energy-efficiency-analog-and-digital-peripherals-and-nfc-tag-option:QN9090-30?tab=Documentation_Tab. Accessed: 2021-12-14.

[30] NXP Semiconductors. *Secure Ultra-Wideband (UWB)*. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb:UWB-TRIMENSION. Accessed: 2021-10-24.

[31] NXP Semiconductors. *Secure Ultra-Wideband (UWB) Positioning and ranging optimized for IoT use cases*. https://www.nxp.com/docs/en/fact-sheet/UWB-IOT-FS.pdf. Accessed: 2021-10-15.

[32] NXP Semiconductors. *Secure UWB Development Kits That Interoperate with Apple U1*. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb/secure-uwb-development-kits-that-interoperate-with-apple-u1:UWB_DEV_KITS. Accessed: 2022-01-05.

[33] NXP Semiconductors. *SR040 Ultra-Wideband Transceiver, Rev. 1.2*. Oct. 5, 2021. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb/trimension-sr040-reliable-uwb-solution-for-iot:SR040?tab=Documentation_Tab. Accessed: 2021-11-24.

[34] NXP Semiconductors. *SR150 Ultra-Wideband Transceiver Rev. 1.0*. Nov. 22, 2021. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb/trimension-sr150-secure-uwb-solution-for-iot-devices:SR150?tab=Documentation_Tab. Accessed: 2021-12-02.

[35] NXP Semiconductors. *Trimension™ SR040: Reliable UWB Solution for IoT*. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb/trimension-sr040-reliable-uwb-solution-for-iot:SR040. Accessed: 2022-02-03.

[36] M. Owen. *Everything you need to know about Ultra Wideband in the iPhone 12 and HomePod mini*. Oct. 18, 2020. https://appleinsider.com/articles/20/10/18/everything-you-need-to-know-about-ultra-wideband-in-the-iphone-12-and-homepod-mini. Accessed: 2021-10-14.

[37] H. Pirch and F. Leong. *Introduction to Impulse Radio UWB Seamless Access Systems*. https://www.firaconsortium.org/sites/default/files/2020-04/fira-introduction-impulse-radio-uwb-wp-en.pdf. Accessed: 2021-10-10.

[38] Qorvo Inc. *DW3000 FAMILY USER MANUAL*. https://www.qorvo.com/products/d/da008154. Accessed: 2022-01-31.

[39] Qorvo Inc. *DW3110*. https://www.qorvo.com/products/p/DW3110. Accessed: 2022-01-31.

[40] Qorvo Inc. *DW3120*. https://www.qorvo.com/products/p/DW3120. Accessed: 2022-01-31.

[41] Qorvo Inc. *DW3210*. https://www.qorvo.com/products/p/DW3210. Accessed: 2022-01-31.

[42] Qorvo Inc. *DW3220*. https://www.qorvo.com/products/p/DW3220. Accessed: 2022-01-31.

[43] RF Cafe. *Ultra Wideband (UWB) Manufacturers& Services*. https://www.rfcafe.com/vendors/components/ultra-wideband-uwb.htm. Accessed: 2022-02-05.

[44] T. Roth. *Hacking the Apple AirTags*. Presentation at DEF CON 2021, August 2021.

[45] Samsung. *How to use Nearby Share on your Galaxy smartphone*. Jan. 26, 2021. https://insights.samsung.com/2021/01/26/how-to-use-nearby-share-on-your-galaxy-smartphone-2/. Accessed: 2021-12-19.

[46] Samsung. *Introducing the New Galaxy SmartTag+: The Smart Way to Find Lost Items*. Apr. 8, 2021. https://news.samsung.com/us/introducing-the-new-galaxy-smarttag-plus/. Accessed: 2021-12-16.

[47] Samsung. *Unlock a New Experience: Galaxy Users Can Now Use Secure Digital Key With the Genesis GV60*. Sep. 30, 2021. https://news.samsung.com/global/unlock-a-new-experience-galaxy-users-can-now-use-secure-digital-key-with-the-genesis-gv60. Accessed: 2021-10-15.

[48] B. Schoon. *Google reiterates that Pixel 6 will have UWB as it works to expand support in Android 13*. Aug. 25, 2021. https://9to5google.com/2021/08/25/google-pixel-6-uwb-mention/. Accessed: 2021-12-19.

[49] T. Schreiber. *Android Binder*. https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2012/03/binder.pdf. Accessed: 2021-11-12.

[50] D. M. Schwaycer. *Instantly share files with people around you with Nearby Share*. Aug. 4, 2020. https://blog.google/products/android/nearby-share/. Accessed: 2021-11-19.

[51] M. Singh, M. Roeschlin, E. Zalzala, P. Leu, and S. Čapkun. "Security Analysis of IEEE 802.15.4z/HRP UWB Time-of-Flight Distance Measurement." In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '21. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 227–237.

[52] K. Vyas. *Further evidence shows Google's Pixel 6 will have ultra-wideband (UWB) support*. Aug. 26, 2021. https://www.xda-developers.com/pixel-6-uwb-support-evidence/. Accessed: 2021-10-14.

[53] K. Vyas. *Google has added an Ultra-wideband (UWB) API in Android*. Jan. 25, 2021. https://www.xda-developers.com/google-adding-ultra-wideband-uwb-api-android/. Accessed: 2021-10-14.

[54] S. Wegner, D. Yang, R. Trandafir, and A. Jani. *What's inside the Google Pixel 6 Pro Teardown?* https://www.techinsights.com/blog/teardown/google-pixel-6-pro-teardown. Accessed: 2021-12-15.

[55] C. Werwitzke. *BMW: Erste Übergaben von i4 und iX an Kunden*. Nov. 29, 2021. https://www.electrive.net/2021/11/29/bmw-erste-uebergaben-von-i4-und-ix-an-kunden/. Accessed: 2022-02-03.

[56] M. Yavari and B. G. Nickerson. "Ultra wideband wireless positioning systems." In: *Dept. Faculty Comput. Sci., Univ. New Brunswick, Fredericton, NB, Canada, Tech. Rep. TR14-230* 40 (2014).

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Martin Reza Heyden, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

THESIS STATEMENT

pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Martin Reza Heyden, have written the submitted Master's Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 APB), the thesis would be graded with 5.0 and counted as one failed examination attempt. The thesis may only be repeated once.

Darmstadt, February 7, 2022

Martin Reza Heyden