Markus Feldbacher, BSc

Design and Implementation of Sensor-based Covert Channels

Master's Thesis
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor
Ass.Prof. Dipl.-Ing. Dr.techn. Steger Christian

Institute of Technical Informatics

Graz, May 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

Every year, the number of devices that are connected to the Internet rises significantly, especially in the so-called Internet of Things- and Smart Home sector. Most of those devices have some kind of sensor embedded and applications range from environmental monitoring, over healthcare, to industrial applications. This thesis presents multiple concepts that exploit unsecured embedded sensors to build covert channels. These concepts range from very simple ideas, such as abusing unused sensor register to transfer data, to more complex covert channels that are often hard to distinguish from normal user behavior and therefore much harder to detect. To emphasize the significance of the issue, furthermore, two additional approaches are introduced that exploit the sensor stack of Android, a very common operating system for modern smartphones. For each covert channel concept various countermeasures are introduced and discussed. For the evaluation, the proposed concepts are implemented using a modular testbed application, which is extendable to different sensor- and interface hardware. As these channels are very prone to user-generated noise, a request-response packet system is used to combat the errors resulting from the noise. The evaluation shows that all of the proposed approaches are able to successfully establish a covert channel between two isolated processes. While simple channels reach a throughput of 4,8kbit/s, the more complex designs are only able to transmit data at a rate of up to 20bit/s. Because no assumptions about the target platforms are made for the design concepts, these covert channels can pose a security risk for any sensor-equipped device.

Kurzfassung

Jedes Jahr steigt die Zahl der Geräte, die an das Internet angeschlossen sind deutlich an. Insbesondere im so genannten Internet der Dinge und im Bereich Smart Home ist ein solcher Anstieg zu beobachten. Die meisten dieser Geräte haben eine Art eingebetteten Sensor und die Anwendungen reichen von der Umweltüberwachung über das Gesundheitswesen bis hin zu industriellen Anwendungen. In dieser Arbeit werden mehrere Konzepte vorgestellt, die ungesicherte eingebettete Sensoren zum Aufbau verdeckter Kanäle nutzen. Diese Konzepte reichen von sehr einfachen Ideen, wie dem Missbrauch ungenutzter Sensorregister zur Datenübertragung, bis hin zu komplexeren verdeckten Kanälen, die oft schwer vom normalen Nutzerverhalten zu unterscheiden und daher viel schwieriger zu erkennen sind. Um die Bedeutung des Themas zu unterstreichen, werden darüber hinaus zwei weitere Ansätze vorgestellt, die den Sensor-Stack von Android, einem weit verbreiteten Betriebssystem für moderne Smartphones, ausnutzen. Für jedes Konzept werden verschiedene Gegenmaßnahmen vorgestellt und diskutiert. Für die Auswertung werden die vorgeschlagenen Konzepte mit Hilfe einer modularen Testbed-Applikation umgesetzt, die auf unterschiedliche Sensor- und Schnittstellen-Hardware erweiterbar ist. Da diese Kanäle sehr anfällig für benutzergeneriertes "Rauschen" sind, wird ein Request-Response-Paket-System zur Bekämpfung der aus dem "Rauschen" resultierenden Fehler eingesetzt. Die Evaluation zeigt, dass alle vorgeschlagenen Ansätze erfolgreich in der Lage sind, einen verdeckten Kanal zwischen zwei isolierten Prozessen aufzubauen. Während einfache Kanäle einen Durchsatz von 4,8kbit/s erreichen, können die komplexeren Designs nur Daten mit einer Rate von bis zu 20bit/s übertragen. Da bei den Designkonzepten keine Annahmen über die Zielplattformen getroffen werden, können diese verdeckten Kanäle ein Sicherheitsrisiko für jedes mit Sensoren ausgestattete Gerät darstellen.

Acknowledgments

This master thesis was written during the year 2019/2020 at the Institute of Technical Informatics at Graz University of Technology.

First and foremost, I want to thank my supervisor Christian Steger for his support throughout the process of researching and writing my thesis. He gave me all the freedom I needed and pointed me in the right direction whenever I felt lost. I would also like to thank Thomas Ulz for his excellent guidance during the most important parts of this thesis. His door was always open whenever I had questions or needed feedback and a lot of great discussions and new ideas were the result. Further, I want to thank Rainer Hofmann for proofreading my master thesis.

Finally, I want to thank my parents for their unwavering support and encouragement throughout my studies. They were always understanding and without their help, I would not have been able to accomplish this goal.

Graz, May 2020 Markus Feldbacher

Danksagung

Diese Diplomarbeit wurde im (Studien)Jahr 2019/2020 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Zuallererst möchte ich meinem Betreuer Christian Steger für seine hilfreiche Unterstützung beim Verfassen dieser Masterarbeit danken. Von Beginn an hatte ich alle Freiheiten die benötigte und wenn ich unentschlossen war oder Probleme hatte half er mir stets die richtige Richtung zu finden und meine Arbeit fortzusetzen. Einen besonderen Dank möchte ich auch an Thomas Ulz richten, der mich gerade in den wichtigsten Abschnitten der Arbeit hervorragend unterstützt hat. Seine Tür war immer offen wenn ich Fragen hatte oder Feedback benötigte und durch die konstruktiven Diskussionen entstanden viele neue Ideen. Außerdem möchte ich Rainer Hofmann für das Korrekturlesen meiner Masterarbeit danken.

Abschließend möchte mich bei meinen Eltern bedanken, die mich während des gesamten Studiums unterstützten und ermutigten. Sie zeigten stets Verständnis und ohne sie wäre es mir nicht möglich gewesen dieses Ziel zu erreichen.

Graz, Mai 2020 Markus Feldbacher

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every year, the number of devices that are connected to the Internet rises significantly. Especially the number of Internet of Things (IoT)- and Smart Home devices started to grow recently. Most of those devices have some kind of sensor embedded and applications for enterprise IoT range from environmental monitoring [53], over healthcare [42] to industrial applications [7]. On the consumer side, smartphones and wearables are very common items and include a wide range of different sensors, such as accelerometers and ambient light sensors. The later is often used to automatically adjust the screen brightness, based on the ambient light level, while accelerometers are needed to recognize motion and wake up the device. In Figure 1.1, a forecast is given with multiple categories.

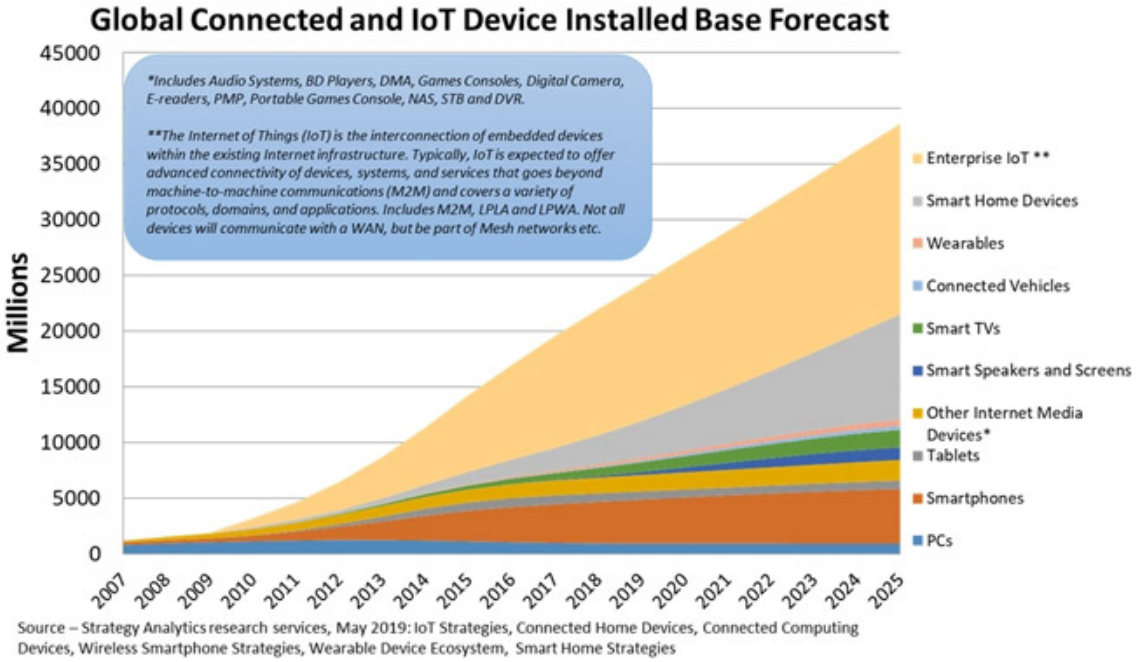


Figure 1.1: Number of devices connected to the Internet [49]

Sensors are observing the physical properties of their environment. As such properties are often private, i.e., health care data [62] or industrial espionage [48], there are some major privacy concerns. If an attacker is able to access such private data, the victim can be harmed in a personal or financial manner. For companies, it can result in significant damage to their reputation if any customer data is leaked. Therefore, it is considered a high priority to keep the private data safe and secure. Another important aspect is the trustworthiness of data [55]. Using false data injection attacks, attackers are able to manipulate cyber-physical systems by tampering with the sensor-data [34]. If the targeted systems are medical devices, such attacks could possibly threaten human lives [11]. To counteract such attacks, modern mobile operating systems are using a permission system to secure sensors interfaces. Unfortunately, these systems are often too coarse or do not apply to all sensors and therefore also present security issues [53]. These problems with the permission system are mostly associated with privacy concerns, but can also be used to build so-called covert channels, which are the main topic of this thesis.

1.1 Problem Statement

The confinement problem is a well-known topic. It describes the problem of preventing an entity (physical or software) from leaking any kind of confidential information. To achieve this goal, the entities are put into isolation. For software isolation, the most common ways are either Virtual Machines (VMs) or so-called Sandboxes. VMs simulate hardware such that the isolated program can perform like on a physical system, but without being able to break out of the VM. Furthermore, a program is not able to distinguish between a physical and a simulated system. Nevertheless, there still exists the problem of shared resources like CPU, memory, sensors, etc. Some of the related work covered in Chapter 2, shows that it is still possible to leak data across Virtual Machines. The other common technique to confine a program is sandboxing. A sandbox creates a controlled environment in which the program is executed. One of the most prominent examples is Google Android, which implements an "Application Sandbox" [22] into the kernel and uses a permission system to control all of the interactions with elements outside of the sandbox.

Side Channels. A side channel is considered information, which is leaked intentional or unintentional by an entity while performing operations and can be observed by others. For example, if a program is executing multiple calculations with different processing complexities, based on the CPU usage, other programs may be able to distinguish between the different calculations. Some effects are even observable from outside, such as timing, power consumption, or electromagnetic emanation [32, 37, 39]. For intentional side channels, also called active side channels, the attacker requires physical access to the device [17]. These channels can be used to encode data which can be observed by another process, effectively building a communication channel or so-called covert channel.

Covert Channels. In 1973, Butler W. Lampson defined covert channels as channels, which are not intended for information transfer at all [35]. These channels are built using side-channel information that is observable by at least one of the applications and can be manipulated by the other application. Covert channels can be used by two isolated

processes, which are normally prohibited to interact with each other, to communicate. A basic graphical representation of a covert channel is given in Figure 1.2.

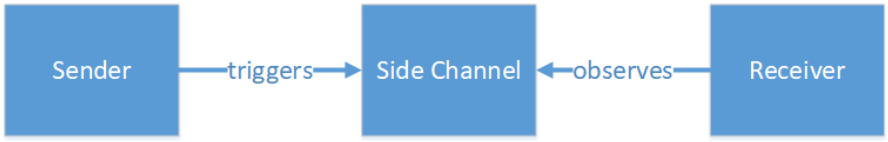


Figure 1.2: Basic concept of a covert channel

For all covert channel concepts presented in this thesis, the system model shown in Figure 1.3 applies. It is comprised of two processes which are both isolated. Direct communication between isolated processes is prohibited by a strict permission model. Both processes require access to the same sensor as part of their normal tasks. This sensor is therefore considered a shared resource that can be used to build a covert channel if not secured correctly. The model assumes that one of the processes (sender) is in possession of confidential information and the attacker wants to transmit this information to another process (receiver). Using its ability to control the behavior of the sensor, the sender encodes this information into a side channel. The receiver is able to observe the encoded information by monitoring the sensor.

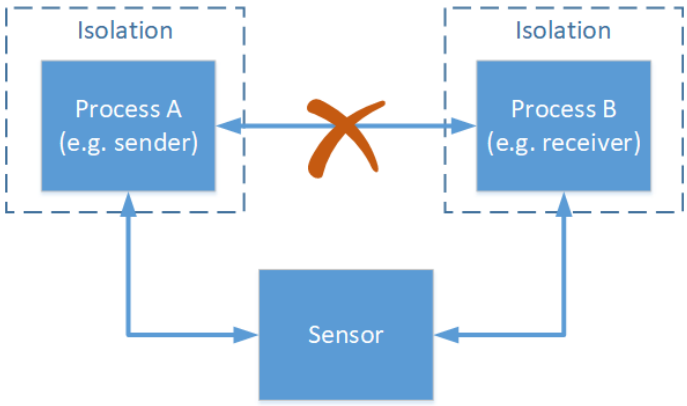


Figure 1.3: Simple system model with two processes in isolation using a sensor as to build a covert channel

1.2 Thesis Contributions

In this thesis, five covert channel concepts are presented. The first three exploit the unsecured registers of an embedded sensor. The channels allow different data rates with a trade-off in the likelihood of being detected. For each of the channels, countermeasures are proposed which have to be either implemented in hardware or software. To allow the evaluation of the covert channels on different sensors, a modular application was developed. This test application can be extended to support other sensors, hardware interfaces, and attacks. The application also integrates a packet system, which is used to control the data flow and enables error correction. To show the viability of covert channels on modern operating systems, two additional concepts are introduced that exploit the android sensor stack. These concepts are evaluated using both, the Android simulator and a real-world device. The results are compared with the other covert channel designs.
This master thesis was part of the IoSense project. [1]

1.3 Thesis Structure

This thesis is structured into five chapters. In Chapter 2, related work is presented, which is split into four sections, each covering a different side-channel category: cache-, memory-, network-, and sensor-based. For each section, a few selected publications are explained to give some insight into other popular covert channel ideas and designs. Following up, in Chapter 3, the side-channel concepts developed during this thesis are introduced and explained in more detail. The designs are split into two major categories: direct-access and managed-access. For each category, some relevant platforms and operating systems are discussed first, followed by the covert channel concepts in increasing complexity, including initialization and synchronization. For each design, ideas for countermeasures are presented as well. These designs are implemented using a layered abstraction approach that is discussed in more detail in Chapter 4. Using this implementation, the designs are evaluated in terms of throughput. Finally, in Chapter 5, a conclusion is given followed by an outlook and ideas for future work.

[1] The IoSense project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692480. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Germany, Netherlands, Spain, Austria, Belgium, Slovakia.

Chapter 2

Related Work

This chapter introduces some important publications of multiple closely related fields. These range from very fast side-channel attacks that target the cache or memory of a device, covered in Section 2.1 and Section 2.2, to considerably slower attacks on sensor-systems, covered in Section 2.4. In Section 2.3 network-based side-channels are discussed, which can be a big threat in typical IoT applications.

2.1 Cache Side-Channel Attacks

In modern processor architectures different levels of caches are used to achieve a good balance between speed and cost. The cost rises exponentially in regards to throughput, therefore the usage of the highest-speed cache is kept to a minimum. The cache hierarchy of a modern Intel CPU is shown in Figure 2.1. The CPU consists of four cores, of which each has a separate L1 and L2 cache. The L3 cache is shared between all cores and also know as Last-Level-Cache (LLC).

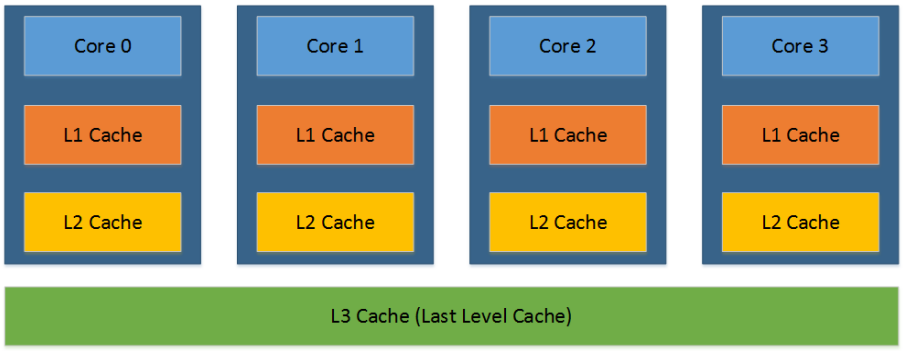


Figure 2.1: Simplified view of a modern CPU architecture.

As most systems cannot be influenced in a way that programs run on a specific core, the attacker tries to focus its exploits on the LLC which can be accessed from all cores. In modern Intel systems, the LLC is an inclusive cache, which means it contains copies of the

data stored in all L1 and L2 caches. If some of the data is removed from the LLC cache it gets also removed from the corresponding L1 and L2 cache.

Another important concept that enables the cache attacks is page sharing. This technique is implemented in modern operating systems to reduce the memory footprint of running processes. For example, if multiple processes use the same shared library, the operating system keeps only one instance of this library in memory and shares the corresponding pages with all processes. There is also a more aggressive approach of page sharing called memory de-duplication which is implemented in widely used hypervisors like VMware ESX [57]. If this form of page sharing is enabled, the system tries to find pages in memory that contain the exact same content. All but one of those pages get evicted and all references are updated to use the single remaining instance. This comes with security issues as a malicious process can alter the shared pages. To circumvent this, shared pages are mapped as copy-on-write. If a process tries to modify the contents of the page, a copy of the page is created and mapped to this process instead of the shared page. That process introduces a delay which can be exploited by the following attacks.

Prime+Probe: The Prime+Probe attack [63] was proposed in 2005 and is one of the first sophisticated cache side-channel attacks. The goal was to learn about the memory access patterns of another process and therefore to be able to perform a full key extraction during or after an encryption.

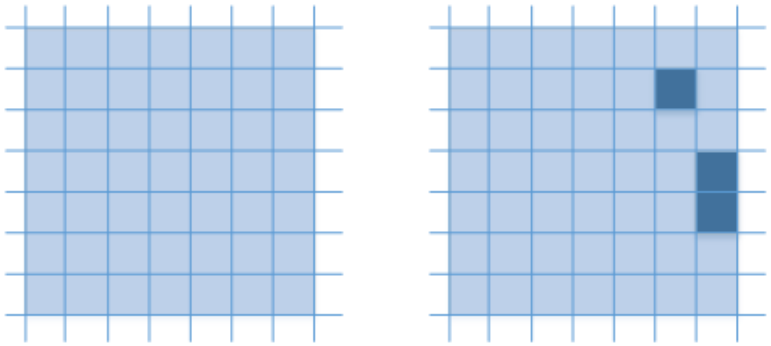


Figure 2.2: Schematics of cache states for the Prime+Probe attack

To perform the attack, first a contiguous byte array A is allocated by the attacker. Then, for each encryption of a plaintext p the attacker performs the following steps:

- (prime) Read at least one value from each memory block in A. This fills the cache with the attacker's data. See Figure 2.2a
- Trigger an encryption of p, which will cause some eviction of the attacker's memory blocks. See Figure 2.2b

- (probe) Read data from array A to check which cache sets were evicted by the victim. If a cache set was accessed during the encryption, it will contain the victim's data. Therefore the read action of the attacker will take more time as the attacker's data has to be loaded into the cache again. This timing difference is monitored and evaluated.

Flush+Reload: The Flush+Reload attack [61] is an extension of the Gullasch et al. attack [24], that allows cross-core and even cross-VM attacks by focusing the LLC. The second advantage in contrast to prior methods is the high fidelity and the resistance to false positives. This is very important, as a low-resolution attack is not able to provide the fine granularity that is required for tasks like cryptanalysis. This attack is a variant of the Prime+Probe attack described in the previous paragraph. Like Prime+Probe it consists of three phases per attack cycle:

- Flush the monitored cache lines to prepare the attack.
- Trigger an encryption of p, which will cause the victim to reload some of the previously evicted data back into the cache.
- Read the cache lines to check which cache lines were reloaded by the victim. If a cache set was accessed during the encryption, it will contain the victim's data. Therefore the read action of the attacker will take less time as the requested data is already in the cache. See Figure 2.3b Otherwise, the data has to be loaded into the cache which takes more time. This timing difference is monitored and evaluated. See Figure 2.3b

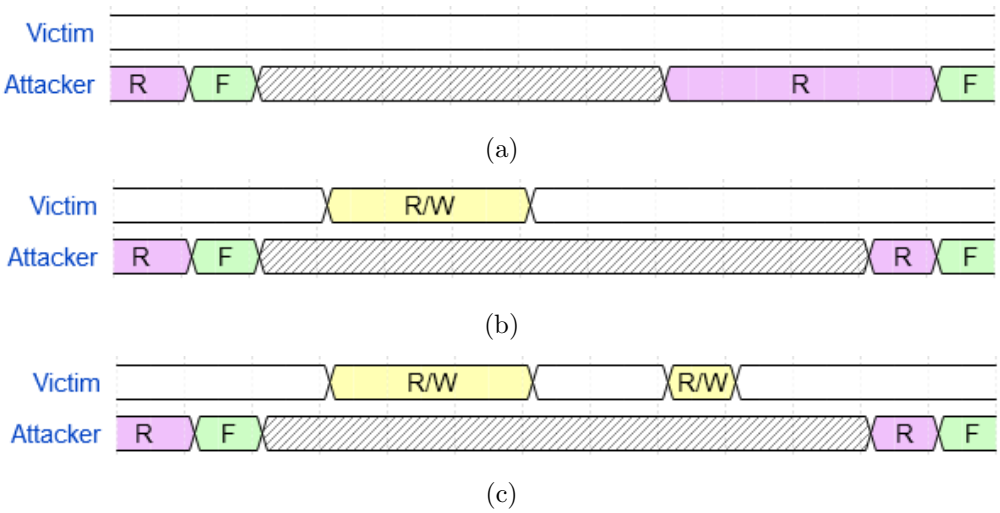


Figure 2.3: Timing diagram for the Flush+Reload attack

As pictured in Figure 2.3c, the reload time does not depend on the number of memory accesses. This means, an attacker is only able to tell if the cache line was read by one of

the victims, but not how often it was read. With this approach, the authors were able to successfully reveal 90% of the victim's key bits during a cross-VM attack. If both processes run on the same core it is even possible to recover the whole key by observing two or more signatures.

Flush+Flush: The Flush+Flush attack [23] exploits the same properties as the previously introduced Flush+Reload attack. In contrast to Flush+Reload, Flush+Flush only uses the *clflush* command and exploits the difference in execution times between data that is present in the cache and data that is not. Further, the authors implemented a monitoring tool using performance counters, that monitors cache references and cache misses of the LLC. This tool was able to detect other existing attacks as shown in Table 2.1. When an attack is mounted using Flush+Reload or Prime+Probe, the number of cache references and most important the number of cache misses skyrockets.

Technique	Cache references	Cache misses	Execution (s)	Stealthy
Flush+Reload	$1000 * 10^6$	16284602	215	No
Prime+Probe	$4222 * 10^6$	294897508	235	No
Flush+Flush	$768 * 10^6$	1741	163	Yes

Table 2.1: Comparison of the performance counters when performing 256 million encryptions with different cache attacks

Jidong Xiao et al [60] investigated the implications of memory deduplication which was already explained in the beginning of this Section. Using memory deduplication, hypervisors try to merge identical pages into one single page to increase memory efficiency. This comes with drawbacks in security as it creates a "shared medium" between multiple different processes or even different virtual machines. The goal of the authors was to design, implement and evaluate a cross-VM covert channel that is resilient to noise and ensures a high bit rate. As mentioned in Section , when a process performs a write operation on a shared page, the hypervisor first creates a copy of the original page and redirects the write operation to this new page. While this ensures that other processes that currently use the shared page are not affected, it introduces a short delay. This delay is used to build the covert channel. In the first step, both the sender and the receiver open the same file. This file is loaded into memory and the sender changes some parts based on a "protocol". Some parts of the file are still the same for both processes, so the hypervisor uses memory deduplication. Now, the receiver overwrites the whole file and therefore forces the hypervisor to perform copy-on-write operations on shared pages. The whole process is monitored by the receiver, which is now able to identify based on timing which pages the sender had accessed before. The whole procedure is visualized in Figure 2.4.

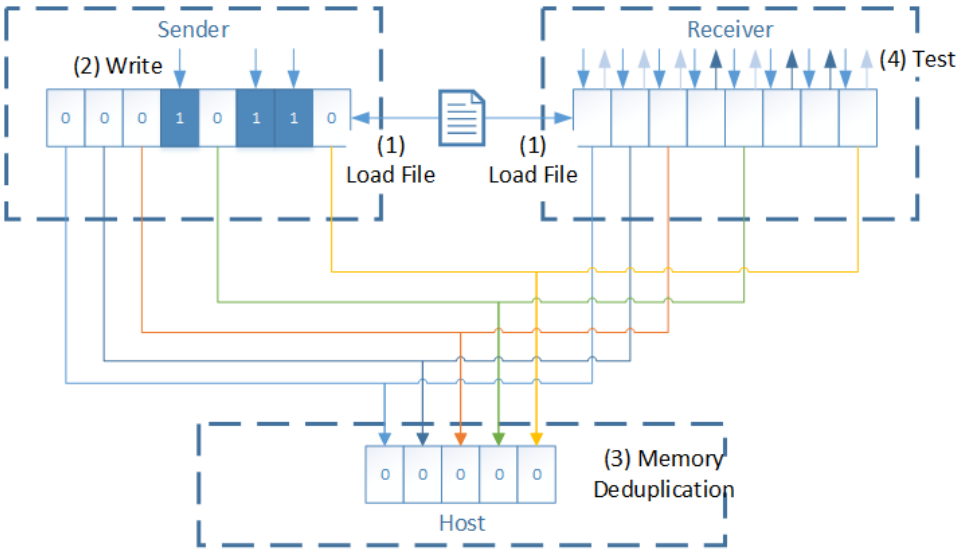


Figure 2.4: Memory covert channel overview

One of the biggest problems cache side-channels are facing is the aggressive noise introduced by other processes, the operating systems and hypervisors. Therefore, it was assumed that the noise effectively prevents attackers from using cache side-channels as functional covert channels. The research work "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud" [41] tackles that assumption and introduces the concept of error-correcting codes to cache based side-channels. They implement a packet system using a request-response architecture. Request packets only contain the id of the requested data packet. This means, the packet is really small and an algorithm with error-correction can be used. As these error-correcting codes (ECC) need to encode the input, which results in an exponential message length of $m = 2^k$ for the Hadamard code for example, they would introduce too much overhead for the data packets. Error-detecting codes (EDC), like Berger codes, calculate a checksum that is appended on the end of a packet. This adds a much lower overhead resulting in a logarithmic message length of $m = k + log(k+1)$. Table 2.2 shows the size difference of Hadamard and Berger codes for different input lengths.

Input length	Hadamard (ECC)	Berger (EDC)
1	2	2
2	4	3
5	32	6
10	1024	12
20	1048576	22

Table 2.2: Comparison of the size differences between Hadamard and Berger codes

The packet system, ECC and EDC codes are described in more detail in Section 3.3. Because they still experienced some errors, in addition to the EDC and ECC in their physical layer, the authors also introduced a data-link layer. This layer contains another error-correcting code. To achieve fast encoding and decoding speeds, Reed-Solomon codes (RS codes) with a 10% error-correction were chosen. This resulted in a channel with an achievable 0.00% error rate. For their experiments Maurice et.al implemented an SSH channel over the proposed covert channel 2.5. Other state-of-the-art channels are not able to provide the reliability required for TCP, as it expects the physical layer to transmit the data without errors. They tested the stability of the channel during different server loads and found that the connection was unstable only in high load scenarios. This shows that, given the correct implementation, covert-channels can be a real threat in cloud environments.

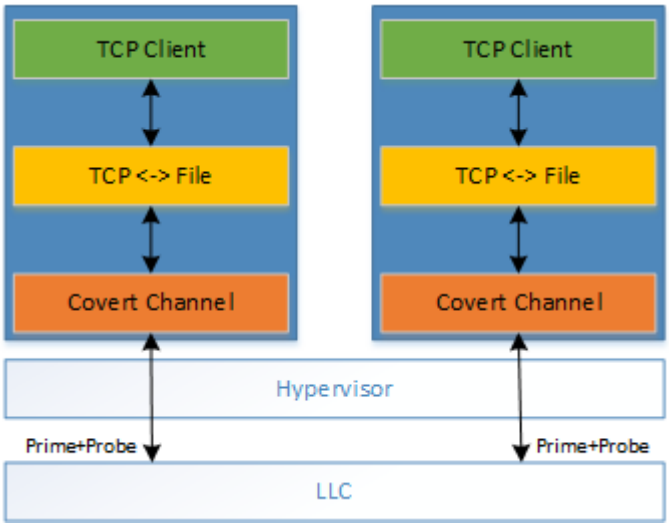


Figure 2.5: TCP connection over a cache based covert channel

2.2 Memory Side-Channel Attacks

Memory side-channel attacks follow a very similar pattern like the cache side-channel attacks discussed in Section 2.1. Instead of L1, L2 or the last-level cache, memory side-channel attacks target either the DRAM directly or files and variables present in the DRAM or saved to the hard drive disks (HDD) of a system. In general, these channels are much slower because the physical hardware has a much lower throughput. Current generation DRAM delivers raw speeds of up to 25GB/s whereas last-level-caches are able do achieve 150+GB/s. Conventional HDDs are even slower, reaching 100MB/s. Looking at the system model shown in Figure 2.6, it can be seen that memory is the next layer on top of the CPU. The memory is shared between multiple CPUs in dual-socket systems, which enables cross-CPU attacks. Modern Servers often use such a dual CPU design because of the space restrictions in datacenters. An attack that is able to establish a side-channel across different CPUs is therefore a realistic threat for modern cloud applications.

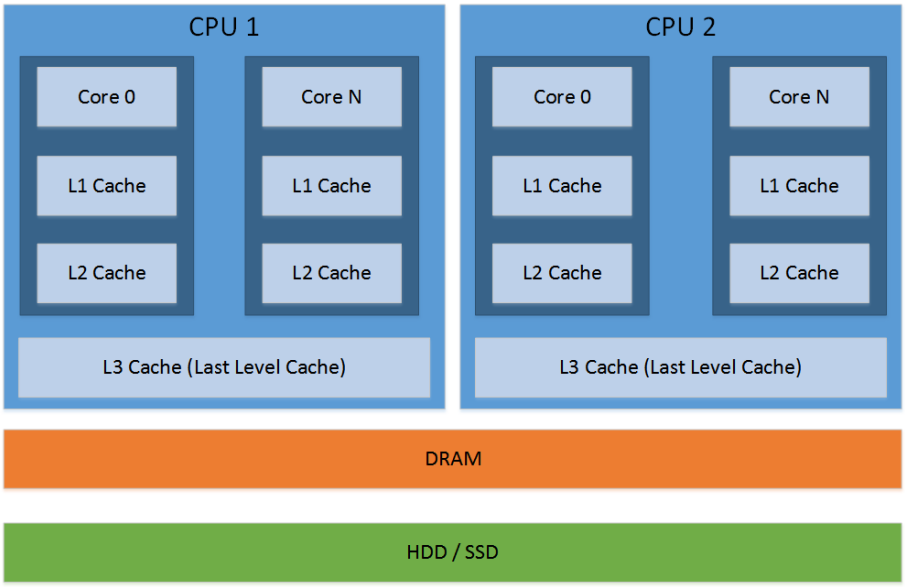


Figure 2.6: System model including main memory

Gruss et al. [43] presented methods to reverse engineer the memory address mapping of DRAM in regards to DRAM channels, ranks and banks. Using the gained knowledge, they introduce an attack that exploits the shared DRAM row buffer. The DRAM row buffer is one of the core elements in the storage organization. Other elements from high- level to low-level are: channel, DIMM, rank, chip, bank, row, column, and memory cell. The channels are the physical links between the memory controller and the DIMM modules. Multiple channels can be accessed in parallel, which speeds up the data transmission. Dual Inline Memory Modules (DIMMs) are the actual physical PCB modules that contain the DRAM chips on both sides. These chips are grouped into ranks, where each side of the DIMM represents a single rank. Each of the ranks consists of 8 banks (16 for DDR4)

which are also used in parallel for a significant speed increase on a memory level. Finally the banks consist of memory cells which are grouped into rows and columns. To further speed up multiple accesses to the same memory location, the memory modules contain row buffers that act as a cache within the DRAM. Figure 2.7 shows the location of the row buffer in the organization hierarchy. It sits directly between the individual banks and the memory bus. The access times vary between ∼20ns for cached rows, ∼40ns for empty row buffer access and ∼60ns for conflicts. A conflict happens when a row other than the cached row is accessed. The row buffer has to close the active row and activate the requested row. This procedure introduces the ∼40ns delay.

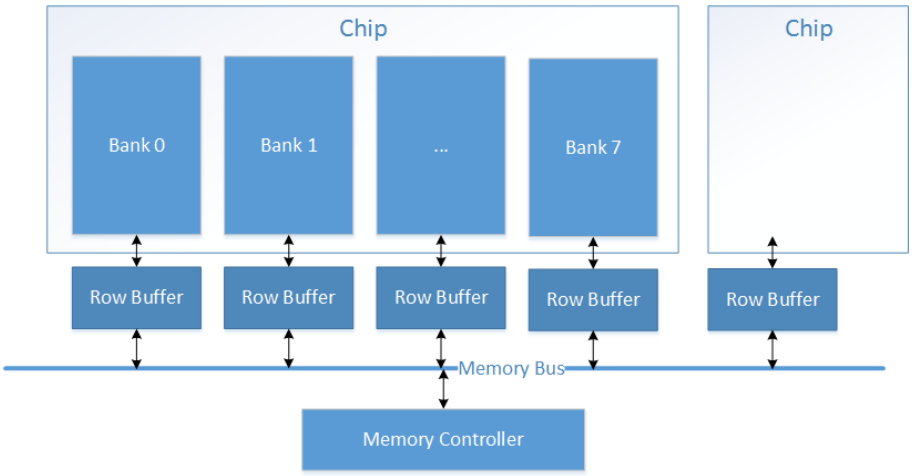


Figure 2.7: Simple view of the organization of DRAM

To be able to exploit the DRAM row buffer and build a covert channel, the attacker has to know the memory address mapping function. This function varies between different memory types and systems, so the authors introduced a fully automatic reverse engineering method. First a list of address pairs that use the same bank must be determined. This is done by accessing two addresses in an alternating fashion and measuring the access time. If the time is significantly higher, the addresses belong to the same bank but not the same row, as the delay comes from the row conflicts. With the known address pairs, the addressing functions can be reconstructed. Depending on the page size, either all partial functions up to a_{20} for 2MB pages or up to a_{30} for 1GB pages can be recovered. Bits above a_{30} or below a_5 can be ignored as they are not used for bank addressing. With a brute-force approach, the authors generated all possible linear functions for the remaining bits. These functions are then tested and filtered using the list of address pairs. Only if both addresses of a pair result in the same bank, the function is added to a list of possible addressing functions. The resulting list can then be used in the next step to build a cross-CPU covert channel.

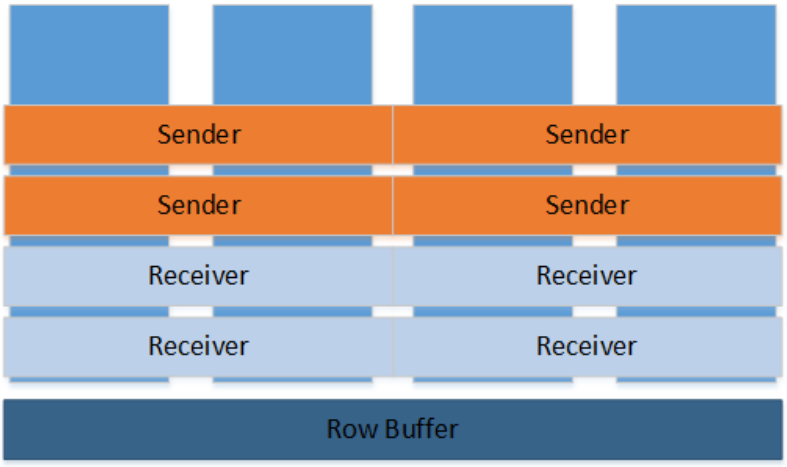


Figure 2.8: Covert channel using DRAM row buffer

Using the method described in the previous paragraph, the attacker determines possible addressing functions. Then, starting with the shortest function, he searches for address pairs that share the same bank but not the row. Receiver and sender choose rows based on these addresses, which is illustrated in Figure 2.8. The receiver accesses its memory locations periodically and measures the access time. If the sender also reads or writes to its memory, the next time the receiver tries to access it, the access time will be significantly greater because of the row conflict. The authors determined that multiple (CPU, channel, DIMM, rank, bank) tuples can be used in parallel but this will introduce noise and is only applicable to a certain point.

2.3 Network Side-Channel Attacks

This section will cover multiple different approaches to network side-channel attacks. In contrast to the previous side-channels, with network-based attacks it is possible to build covert channels between multiple independent entities. This is especially critical in modern industrial Internet of things applications that rely heavily on network structures (mesh networks, etc.). Because the network medium is significantly slower than DRAM or cache memory, the resulting side- and covert channels are also much slower. The extended system model is visualized in Figure 2.9. To keep it simple, some details shown in Figure 2.1 and 2.6 were omitted.

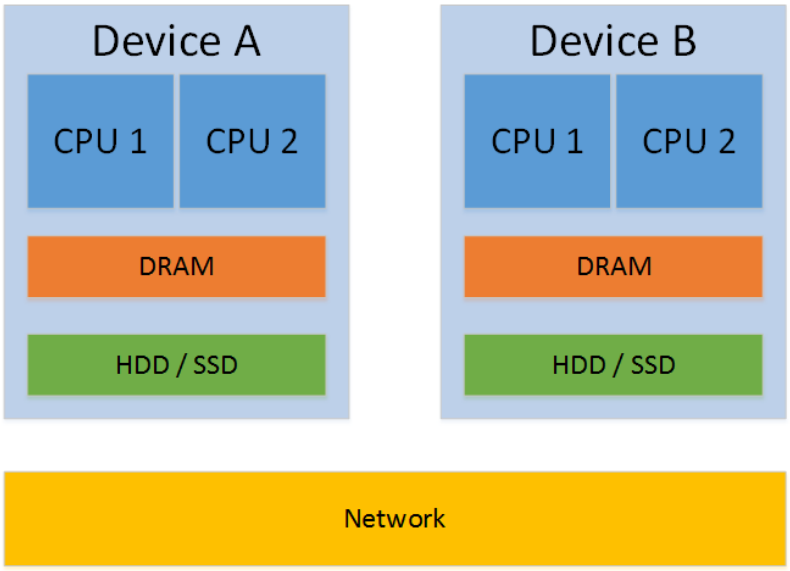


Figure 2.9: Covert channel using DRAM row buffer

In 1987 C. Gray Girling [18] investigated local area networks (LANs) and their vulnerability to covert channels. The author describes three different approaches: using the address field, the length of a data block and the time between successive transmissions. These are concepts defined on an abstract level that are independent from the network protocol. For example, if the attacker is able to send messages to 16 different addresses, he could encode 4 bits of information by sending a message to a specific address. The receiver monitors the network and keeps track of the addresses, the sender used. Another method proposed by Girling exploits the dynamic length of a data block. In most network protocols, the data block length has to be specified by the sender to tell the receiver how many bytes of data should be parsed. This can be used to encode data directly into the block length. For example, the commonly used IP header, see Figure 2.10, defines a 16 bit "Total Length" field, which according to the RCF791 specification is "the length of the datagram, measured in octets, including internet header and data." [25].

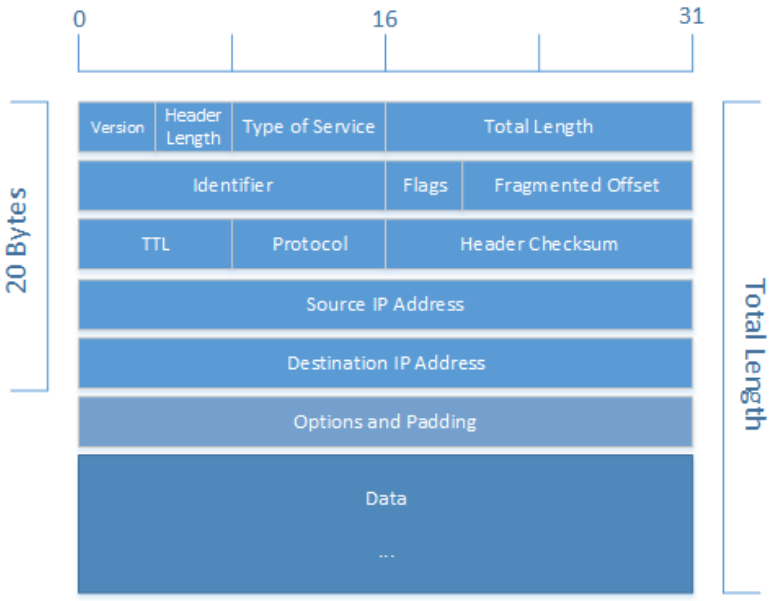


Figure 2.10: IP header format with additional options, padding and data sections

The third attack targets the time between successive transmissions. If a process is able to monitor the timestamps of packets, it can also calculate the difference between transmissions. The sender encodes data by using different delays for successive send operations. These delays are known by the receiving process. Because networks can introduce significant delays, it is important to define an appropriate distance between timing thresholds. All of these approaches can be assigned into one of two major groups, the covert storage channels and the covert timing channels. Covert storage channels use some form of shared memory to transfer data between the sender and receiver, whereas covert timing channels try to hide time modulated data in existing procedures. Since the original paper was released, many other researchers introduced different covert channel ideas related to concepts of Girling. Ahsan et al. [1] use a stego algorithm framework to inject data into TCP/IP packets which are transferred between two communicating parties. To improve the security of the channel in some versions, they also proposed the usage of a shared symmetric secret key during the encoding and decoding, such that no other entities are able to detect the plaintext information. The first convert channel design introduced by Ahsan et al. is based on the manipulation of the IP header fields. As shown in Figure 2.10, the IP header consists of many different fields and some of them can be redundant based on the application. For example, the flags field consists of three bits, representing "Reserved", "Don't Fragment" (DF), and "More Fragments" (MF), which are all related to packet fragmentation. According to the specification [25], the DF flag is used to signal that the packet should not be fragmented, so it just gets dropped if it is too large. Under normal circumstances (e.g. the packet size is smaller than the MTU) this flag will always be ignored, so it is a perfect candidate to embed one bit of information. The second approach introduced by Ahsan et al. uses part of the 16-bit identification field, also shown in Figure 2.10. This field is used by the receiver to identify related packages when

assembling fragments. It is important that no two messages share the same identification value, therefore so-called "toral automorphism systems" are employed. These systems generate statistically unrelated families of sorted sequences from a key k and dimension N. For each symbol different unique sequences can be selected from the sorted sequences to scramble the encoded input. This ensures a high resistance to statistical cryptanalysis. A simple block diagram of this procedure is visualized in Figure 2.11. In addition to these two methods, the authors show that it is also possible to apply the toral automorphism systems to the packet sorting in the IPSec protocol.

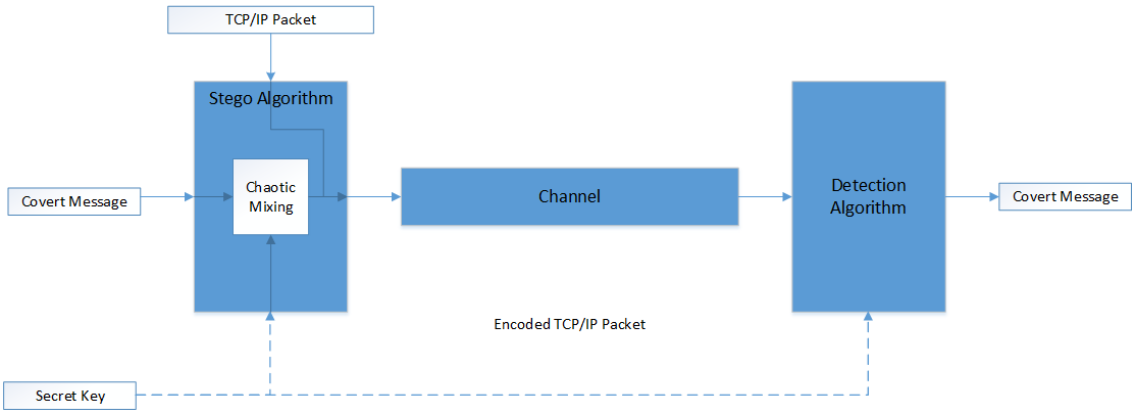


Figure 2.11: Block diagram for the proposed covert channel

The ideas, discussed in the previous paragraph can also be applied to different types of protocols and networks. For example, Lilia Frikha et al. [16] introduce two covert channel designs that use fields in the 802.11 Header to secretly transmit data between different parties in wireless local area networks. The first design is based on the sequence control field, which is very similar to the identifier field in the IP header. The second design is based on the IV field, which consists of a 3byte Initial Vector, a 6bit Pad, and a 2bit Key ID. Because the Initial Vector is chosen randomly for normal transmissions, network sniffers will mostly ignore changes to this field.

2.4 Sensor Side-Channel Attacks

In this section, sensor side-channel attacks that target wireless sensor networks and mobile devices are discussed in more detail. The first part covers attacks on various layers of a WSN as shown by Jaydip Sen [51]. After that, multiple publications introducing side-channels for smartphones are presented.

2.4.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a network of a large number of nodes, that are used to measure physical properties. This data gets sent to a central entity that stores it for further processing. In most cases, the sensor nodes have no access to a main power connection and therefore have to rely on a dedicated battery and energy harvesting techniques. Further, they are restricted in terms of memory and processing power, which limits the available security algorithms. Other problems arise because of the unreliable communication induced through connectionless protocols and the unattended operation of the sensor nodes. This opens the nodes up to denial of service attacks and physical tampering. While low-level attacks only try to prevent communication between nodes by Jamming, Collision, Exhaustion or Unfairness [51], on higher levels (network and routing) attackers are also able to influence routing.

Network	Attacks	Defence
Physical	Jamming	Spread-spectrum, priority messages, lower duty cycle, region mapping
	Tampering	Tamper-proofing, hiding
Link	Collision	Error-correcting code
	Exhaustion	Rate limitation
	Unfairness	Small frames
Network and routing	Spoofed,	Egress filtering, authentication, monitoring
	Selective forwarding	Redundancy, probing
	Sinkhole	Authentication, monitoring, redundancy
	Sybil	Authentication, probing
	Wormholes	Authentication
	Hello flood	Authentication
	Acknowledgement spoofing	Authentication
Transport	Flooding	Client puzzles
	Desynchronization	Authentication

Table 2.3: Sensor network layers, attacks and defences [58]

With mechanisms like Sinkholes and Wormholes, the attacker reroutes all traffic through a compromised node. This allows him to either save and analyze the data or advanced techniques like Spoofing, Replay attacks or Selective Forwarding [31]. Table 2.3 shows an overview of attacks and defenses for different layers in a WSN. Most of the higher level attacks can be prevented by adding some form of authentication. This is not trivial,

because WSNs lack memory and processing power and are restricted in energy. Manik Lal Das [10] presented a two-factor user authentication protocol, that is efficient for WSNs and provides strong authentication and a method for establishing the session key. This protocol is divided into two phases, a registration phase and an authentication phase. During the registration phase, new nodes are registered at the gateway using an id and password combination. The gateway creates a personalized smart card which is given to the user. Upon further communication the user has to provide his smart card and the id and password combination. Each request is verified by the gateway and sent to the sensor node, which then responds with the data. This approach will prevent spoofing and replay attacks, but cannot prevent other attacks like sinkhole, wormhole, etc. Because most attacks target different parts of the system and protocols, it is hard to design an all-purpose countermeasure.

2.4.2 Smartphones

The important role modern smartphones play in everyday life, led to an increase in research effort to find and mitigate side- and covert-channels. There are two main competitors when it comes to smartphone operating systems: Android with a market share of 73.92% and iOS with 25.19% (Figure 2.12). Because of this, most research work is related to these two operating systems. As shown by many different research groups, there exist a large number of vulnerabilities, which even led to attacks that are able to guess PINs with an accuracy of 73% [52]. Existing publications can be split into two major groups, one targeting side-channels which enable communication between processes on a single device, and the other focusing on inter-device covert channels. There is also a special version of Android for Internet of Things applications, *Android Things* [21].

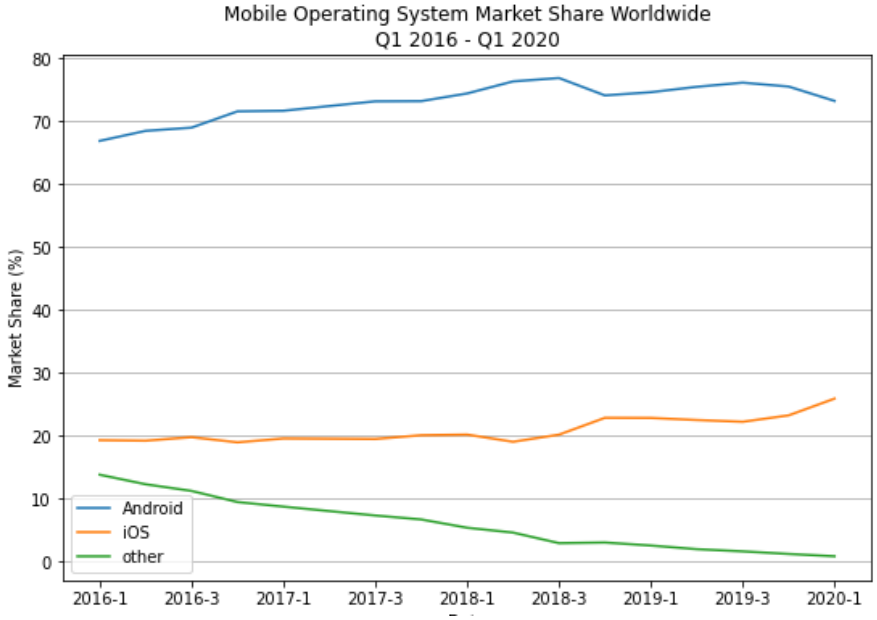


Figure 2.12: Market Share of mobile Operating Systems [54]

Because most of this operating system is equal to a standard version of Android, devices running Android Things are also vulnerable to the attacks discussed in this section. Further, the IoT version integrates a *Peripheral I/O API* which gives users direct access to the serial communication buses. This opens up possibilities for attacks directly on the sensor level. To prevent abuse and leakage of user data, Android implements a permission system. For example, if an application wants to access global storage (photos, videos, etc.) or the camera, it has to request a permission. In older Android versions, permissions had to be defined when installing a new application, even if they are never used. Newer versions (6.0+) are able to request runtime permissions on accessing the protected resource.

Adrienne Porter Felt et. al [13] performed two user studies questioning participants if they pay attention to the permissions of applications during installation and if users are able to correctly define some permissions by name. They confirmed that only 17% of the participants read the permissions and only 3% could correctly identify the scope of permissions like *READ_CONTACTS* and *READ_PHONE_STATE* which was already predicted by other researchers. The second big problem is, that some sensors can be accessed without any permissions, as they do not provide any confidential data under normal circumstances. Researchers have abused this in multiple instances. Raphael Spreitzer [52] has shown that it is possible to classify a PIN from only the light level measured during input. For each button click, the light level changes slightly based on the digit entered. If a PIN is entered multiple times, a recurring pattern can be observed as shown in Figure 2.13.

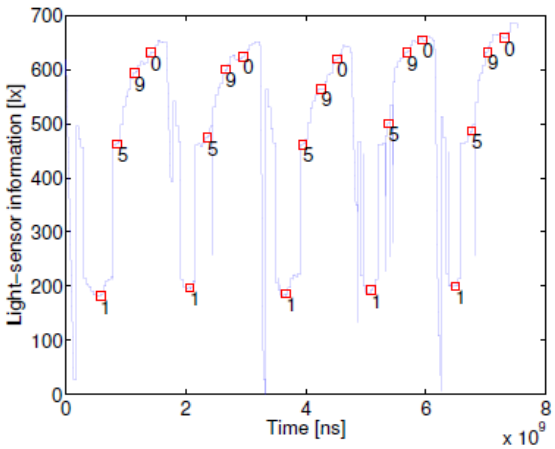


Figure 2.13: Ambient light level during PIN entry

To be able to classify patterns, first a training phase is needed. For this, the author suggests a simple application, where users have to solve mathematical puzzles and enter the result in the same way they would enter a pin. After enough samples are gathered, a second phase, the exploitation phase starts. In this phase, the user is referred to the target application, where he enters his secret pin. The measurements of the ambient light

sensor are then used to deduce the entered pin by using machine learning algorithms. This method is able to recover 65% within 5 guesses. There are also other approaches that use different sensors, like accelerometer, gyroscope, camera, and microphone, which are able to achieve 43% - 50% recovery rate [2, 12].

Chapter 3

Covert Channels Design

This chapter introduces the different attacks developed throughout this thesis. The first part will cover the very basic attacks, performed with direct hardware access. After that, a more sophisticated attack, designed to perform on a widely used mobile platform, is described in more detail and potential countermeasures are discussed for each attack. To decrease the influence of noise and therefore also decrease the error probability, a packet system is introduced in Section 3.3. This includes features, such as a request-response protocol, error correction, error detection, and dynamic packet length.

3.1 Direct-Access Sensors

For this thesis, the sensor access type is an important aspect as it defines the available interactions. In this section, different platforms and attack designs are covered that rely on direct access to the sensor interfaces. This could be either direct access to the hardware interface, e.g. Inter-Integrated Circuit (I2C), or via a device driver that allows users to fully control all aspects of the sensor. The more sophisticated attacks need less permissions than the very basic ones.

3.1.1 Platforms

Two different platforms were chosen for the design, implementation, and evaluation of the covert channels. The first one is a very simple sensor node manufactured by Texas Instruments. Such nodes are commonly used in wireless sensor networks because of their connectivity and low power requirements. A variety of operating systems is available, which are mostly open-source projects. Section 3.1.1.1 covers some of the more popular operating systems for sensor nodes in more detail. For the second platform, a basic Linux environment was chosen, as this is the most common system used for edge nodes or sensor nodes that require more processing power than typical wireless sensor nodes.

3.1.1.1 Texas Instruments

Texas Instruments [26] is a large manufacturer of embedded processors and integrated circuit products for power management and signal chain processing (example: sensors for pressure, temperature, humidity, etc.). In terms of processing power and operating system

features, this platform is the most basic one of the three platforms discussed throughout this thesis.

The operating system is a critical point in this thesis as it handles the access to hardware buses and therefore sensors, process scheduling and multithreading. The latter is specially important for smaller microcontroller systems because of their limited processing power. These systems do not always support multithreading because of the overhead it entails and the relevant platforms mostly consist of only one processor core. Systems that only support one process are not in the scope of this thesis. Some of the more known systems, which support multiple processes, include:

- Contiki
- FreeRTOS
- Zephyr
- TI-RTOS

Contiki: Contiki [8] is an open-source operating system designed for Internet of Things applications. It requires very little memory, making it a good choice for many low-cost devices. As the CC2650 SensorTag is one of the officially supported hardware platforms, drivers for the sensors listed in Table 4.2 already exist. These drivers provide basic functionality to enable and configure a sensor, to wait for a result ready flag and to read the result and the status registers. This limits the design of the covert channel as a process is not able to read and write the same register. Furthermore, it is also not able to distinguish between individual bits of the result ready information for hybrid sensors[1]. To get full access to the sensors, either a custom driver or a direct implementation of the I2C protocol inside the process is required.

FreeRTOS: FreeRTOS [38] is developed by Real Time Engineers Ltd. in a partnership with multiple large companies like ARM, Infineon, Microchip, NXP, Texas Instruments and others. It is distributed under an MIT open-source license and the stewardship of Amazon. Amazon Web Services provides multiple Software as a Service (SaaS) that integrate FreeRTOS into the Amazon cloud. The FreeRTOS kernel is built to support a wide variety of different platforms and is able to fit on very small systems because of a very small compiled binary image (6Kb). There exists also a commercial option, OpenRTOS, which is compatible with the base version but adds warranty and legal protection. For applications with higher security requirements, like industrial, medical, and automotive, there exists a special version called SafeRTOS. The CC2650 SensorTag is not supported directly, but its microprocessor, the MSP430, is supported. There exists some information about porting the FreeRTOS kernel to the CC2650 SensorTag, but this is out of scope for this thesis.

[1] sensors with more than one internal sensors

Zephyr: Like the other RTOS systems, the Zephyr OS [64] is based on a small kernel that is designed to be able to run on memory-constrained platforms. It is open-source and supported by companies such as Intel, Nordic Semiconductor, and NXP. The official repository supports more than 200 different hardware platforms, including the CC2650 SensorTag.

TI-RTOS: TI-RTOS [29] is a real time operating system developed by Texas Instruments specifically for their own microcontroller product line. It bundles multiple components into a downloadable package, including:

- **SYS/BIOS:** A scalable real-time kernel
- **Drivers:** Generall drivers, supporting all available platforms
- **NDK:** Network Developer's Kit
- **UIA:** Unified Instrumentation Architecture
- **XDCtools:** Configuration tools
- **Device specific libraries**

TI-RTOS also includes some examples, which were used in the evaluation part of this thesis. A graphical representation of the TI-RTOS core structure can be seen in Figure 3.1. Important parts of this structure are the real-time kernel, because it handles the multi-threading, and the drivers as they handle the sensor communication. The modules only include basic functionality, such as an I2C bus implementation, but no specific drivers for external components, like sensors. Thus, if a user wants to interact with sensors connected to the I2C bus, either a custom driver library has to be implemented or the I2C bus needs to be directly accessed in the program code.

Scheduling: The scheduling is handled by Texas Instruments' real-time kernel. In a typical application there are three types of threads supported by the scheduler:

- **Interrupt Service Routines (ISRs):** High priority hardware- and software-interrupts
- **Tasks:** User applications are implemented as tasks.
- **Idle:** The idle thread is executed when no other tasks or ISRs are ready.

These thread types are listed in descending priority. Additionally, tasks have their own priority, which is used to determine the order, tasks should be executed in. There are two types of scheduling:

- **Preemptive Scheduling:** Tasks can be preempted by calling *sleep()* or when a higher priority task gets ready to execute.
- **Time-sliced Scheduling:** Every thread gets the same amount of time to run.

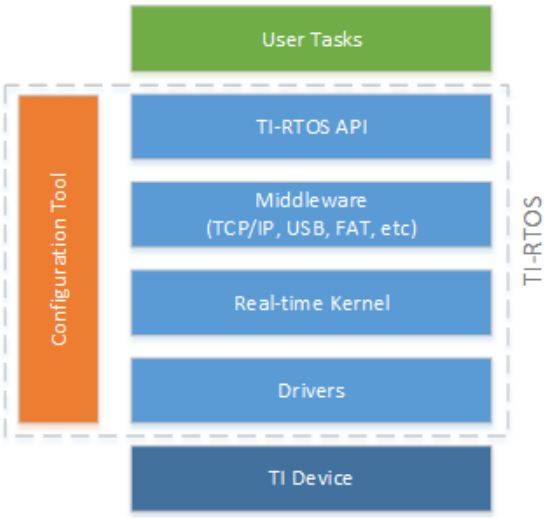


Figure 3.1: TI-RTOS: graphical overview

If all tasks in a system have the same priority, the scheduler will rotate through them in the same order. An example of preemptive scheduling is visualized in Figure 3.2. The ISR preempts all other tasks, but has a very short execution time. Task A is a high priority task, that runs before all other tasks. Task B and task C have the same priority, so the scheduler alternates between them.

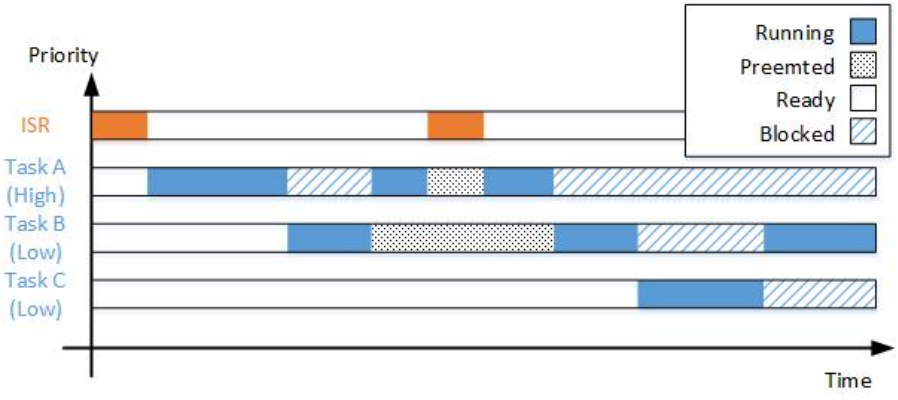


Figure 3.2: TI-RTOS: scheduling example

This alternation is very helpful for building channels between two processes as it can be used as a synchronization mechanism.

3.1.1.2 Linux

Besides the already covered operating systems especially designed for sensor nodes, Linux is the most used platform for wireless sensor networks. It is frequently used for developing lower-level software, which often requires some form of sensor access. In the mobile sector, Android is the most used operating system and is based on the Linux kernel. Therefore, Linux is a very good pick for implementing and testing covert channels.

In Linux, there are two major ways an application can access hardware interfaces: device drivers and dev-mappings. Device drivers are either bundled with the installed distribution or have to be provided by the manufacturer. For example, the Raspberry Pi uses Raspbian, a Linux distribution which is a custom version of Debian. To access a sensor over the integrated I2C bus, first the corresponding driver has to be enabled. The bundled driver maps the I2C interface to a file descriptor in the */dev* directory. This file descriptor can then be accessed using the file open function in C++. For more restricted distributions, a custom device driver may exist that abstracts the hardware interfaces and only allows access to certain parts. Another important aspect is multi-threading. In contrast to previously covered operating systems, Linux implements real multithreading and multiple processes can and will be executed in parallel. Therefore the sending and receiving process need to implement a synchronization mechanism.

3.1.2 Attack Design

3.1.2.1 Unused Register Attack

In this section an attack is introduced, which exploits unused registers of a sensor to transmit data between two processes. The general idea is shown in Figure 3.3. Most of the current sensors contain internal registers that serve different purposes. Some registers are used to save result readings, others are used to configure the behavior of the sensor. While the result and status registers are most certainly read-only registers, the configuration registers are read- and write-able. To be able to make slight adjustments, the configuration register needs to be read first. Then, the desired bits are changed and the whole byte is sent back to the sensor. For example, when a user wants to change the measurement speed, he can read the configuration register and change the relevant bits to fit his needs. This is done via bit masking using and (&) and or (|) operations. If done correctly, other bits are unaffected by these actions. After the modification, the user sends the new configuration back to the sensor. There is also a third type of sensor register, which is defined as "*reserved*". These registers can also be used to transmit data, but depending on implementation of the sensors, this could affect its functionality. In general, datasheets advise against changing of such reserved registers.

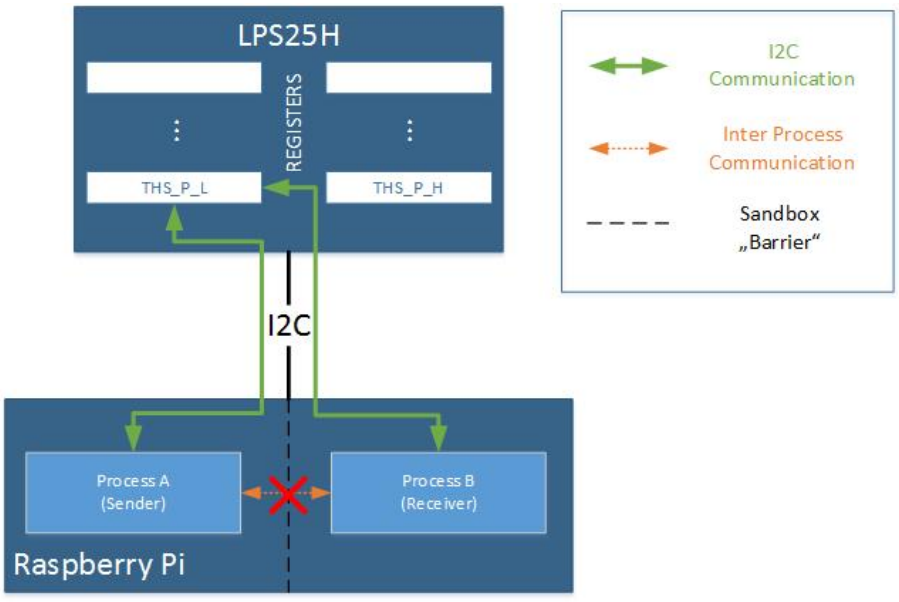


Figure 3.3: Schematic overview: sender and receiver communicating over unused register

A good example is the LPS25H[46] sensor manufactured by STMicroelectronics. This sensor is an absolute piezoresistive pressure sensor that has an included threshold interrupt feature. The user is able to configure the sensor to output a signal on an interrupt pin, if the measured pressure exceeds a defined threshold. Therefore, the respective flag in the configuration register must be set to enable threshold interrupts. Afterwards, the 8-bit registers *"THS_P_H"* and *"THS_P_L"* are used to define the upper and lower part of the

7	6	5	4	3	2	1	0
THS15	THS14	THS13	THS12	THS11	THS10	THS9	THS8
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
7	6	5	4	3	2	1	0
THS7	THS6	THS5	THS4	THS3	THS2	THS1	THS0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Table 3.1: THS_P_H and THS_P_L: threshold pressure registers

7	6	5	4	3	2	1	0
PD	ODR2	ODR1	ODR0	DIFF_EN	BDU	RESET_AZ	SIM
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Table 3.2: CTRL_REG1: Control register 1

threshold. The threshold registers are shown in Table 3.1 and the relevant configuration registers are shown in Table 3.2. If the *DIFF_EN* flag in the *CTRL_REG1* register is false, all interrupts are disabled. Therefore, the *THS_P_H* and *THS_P_L* registers are not used by the current application, but are still read- and write-able. An adversary is able to exploit that, by using these registers as a covert channel between two (isolated) processes. To establish a communication, one process acts as the sender and the other process acts as the receiver. The threat model for this example was introduced in Section 1.1.

Initialization: For the initialization of the covert channel, the sender first has to write a specified codeword to the register, e.g. *01010101*. Afterwards, the sender has to wait until the receiver "acknowledges" the initialization by sending an inverted synchronization bit, e.g. the Most Significant Bit (MSB), to the register (***1****1010101*). This completes the initialization step and the sender can now start the data transmission. If, at any time during the process, an unexpected value is read from the register, it has to be checked if the interrupt mechanism was activated. In that case, no more write actions should occur until the interrupt mechanism is deactivated again. This decreases the detectability of the covert channel. After the mechanism is deactivated by the user that has previously activated it, the transmission can either be resumed or restarted, depending on the preferred strategy.

Synchronization: Depending on the type of scheduling, the sender and receiver processes might have to implement a synchronization mechanism. Without this mechanism, two types of errors can occur: deletion- and insertion-errors. The first type, deletion-errors, can appear, if the sender is scheduled more frequently than the receiver. In this case, the sender will overwrite the transmitted data in the register before the receiver was able to read it. Because the receiver has no way to detect this error, it will read the new data and append it to the already received data, causing corruption. The other type of errors, insertion-errors, occur when the receiver is scheduled more frequently than the sender. The receiver will keep reading data from the sensors register, interpreting it as new data, independently of two consecutive data fragments containing the different data or not. This leads to duplicate data fragments, which also cause data corruption. In very

simple operating systems with pseudo-multi-threading, like Texas Instruments' TI-RTOS, threads get scheduled in the same order every time, so no additional synchronization is necessary (see Section 3.1.1.1 for more information about the TI-RTOS scheduling). For more complex operating systems, like Raspbian, that actually support real multithreading, synchronization is very important to prevent said errors.

For this attack, a simple synchronization algorithm, that is easy to implement, is chosen: The first bit of the exploited register(s) is used as a synchronization bit. This comes with a trade-off. Because there are only $n-1$ bits left for data, the data rate will be a bit lower, compared to a transmission without synchronization. An example structure, for an 8bit register, is illustrated in Figure 3.4. This example is also used for the evaluation in Section 4.3.

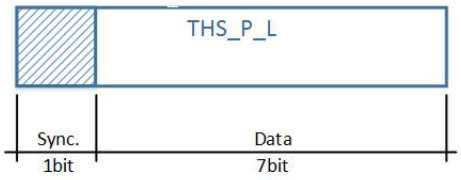


Figure 3.4: Register structure, for the 8bit THS_P_L register with synchronization bit

Before writing new data to the register(s), the synchronization bit is read. To guarantee changes in the register, detectable by the opposite process, the synchronization bit is inverted. Then $n-1$ bits of data get appended to the inverted synchronization bit. After that, the sending process is ready to write the new data block to the register(s). If the receiver wants to send an acknowledgment (ack), it also inverts the synchronization bit, but the appended data is irrelevant and left unchanged.
This procedure is illustrated in Figure 3.5 and explained step by step in the following part:

- The sender writes *0110010* to the register with the synchronization bit set to *0*: ***0****0110010*
- The receiver reads the data from the register, inverts the synchronization bit ($0 \rightarrow 1$) and sends it back to the sensor as an acknowledgment: ***1****0110010*
- Until the changed synchronization bit is detected, the sender keeps reading the target register.
- The sender writes the next data fragment *1110001* to the register, again with an inverted synchronization bit ($1 \rightarrow 0$): ***0****1110001*
- The receiver waits for the change and reads the new data fragment.
- After the acknowledgment, the next cycle starts and is repeated until the transmission is complete

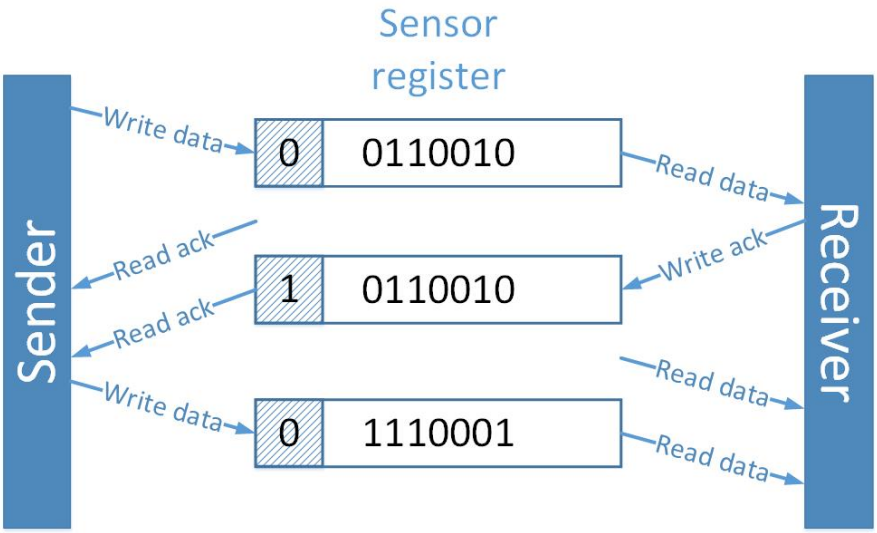


Figure 3.5: Synchronization procedure between sender and receiver to initialize a new channel

Countermeasures: There exist many feasible countermeasures against this attack, that are easy to implement. One possible approach would be, to change the threshold registers to write-only. The only drawback is, that the user has to keep track of the threshold values in the application instead of reading it from the sensor before any modification. Another approach, which has to be implemented on the sensor-side, is to set all unused registers to read-only mode by default. Only if such a register is enabled, it is set to read/write mode. For example on the LPS25H, the threshold registers should only be interactive if the corresponding *DIFF_EN* flag in the *CTRL_REG1* register is set. Figure 3.6 shows a diagram of the threshold logic of the LPS25H sensor.

There are also methods, that can be implemented on the driver-side. For example, by restricting the access to the I2C bus, the users could be forced to use a special driver that only allows certain actions such as reading a status flag or the result registers. This would prevent any read access to the configuration registers, which will therefore also prevent the proposed attack. Looking at available LPS25H drivers, some are implemented in a way, that prevents this attack entirely, while others are very basic and can be exploited. For example, the official SenseHat driver [45] is written in python and uses the RTIMULib C++ driver [44] for sensor interactions. In the RTIMU library, there are no functions available that allow a user to read or write to one of the threshold registers. Therefore, if this driver is the only way to interact with the sensor, the attack described in this section is not possible. Another example, a javascript driver [6] developed by Thomas Byrne, implements functions for reading and writing most of the available registers. This makes it possible for an adversary to perform the exploit on the unused registers of the sensor. Because these drivers are userspace drivers, they do not prevent an attack from directly accessing the I2C bus.

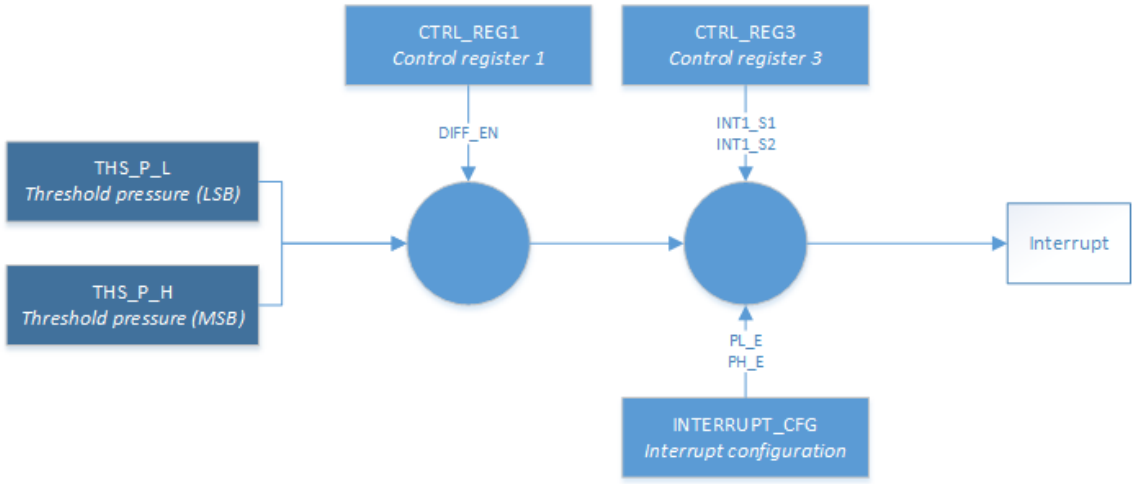


Figure 3.6: LPS25H Thresshold Logic

3.1.2.2 Bit Manipulation Attack

This section describes the second attack, which tries to find single bits in configuration registers, that will not impact the behavior of the sensor in a meaningful way. It is very similar to the first attack, but will also work in an environment that is a bit more restrictive. For example, if a sensor implements a simple countermeasure to the first attack and block read- and write-access to unused sensor registers, this attack will still work. There are different kinds of bits that are good candidates for small manipulations. A conceptual illustration is shown in Figure 3.7.

As mentioned in the last section, sensors often have some registers that are marked as "*reserved*" in the datasheet. These registers should not be changed, because it could impact the behavior of the sensors in a negative way. This also applies to reserved bits. Another candidate for manipulation are settings that have multiple states with the same sensor behavior. An example is the mode of the OPT3001 ambient light sensor [28]. Because there are three different modes, this setting uses two bits. These bits provide four states, but only three are actually used and the last one is a duplicate of state 3 (see Table 3.3).

Mode	M[0]	M[1]
Shutdown (default)	0	0
Single-shot	0	1
Continuous conversion	1	0
Continuous conversion	1	1

Table 3.3: OPT3001 operational modes

If an attacker switches between state 3 and 4, the sensor behavior does not change and

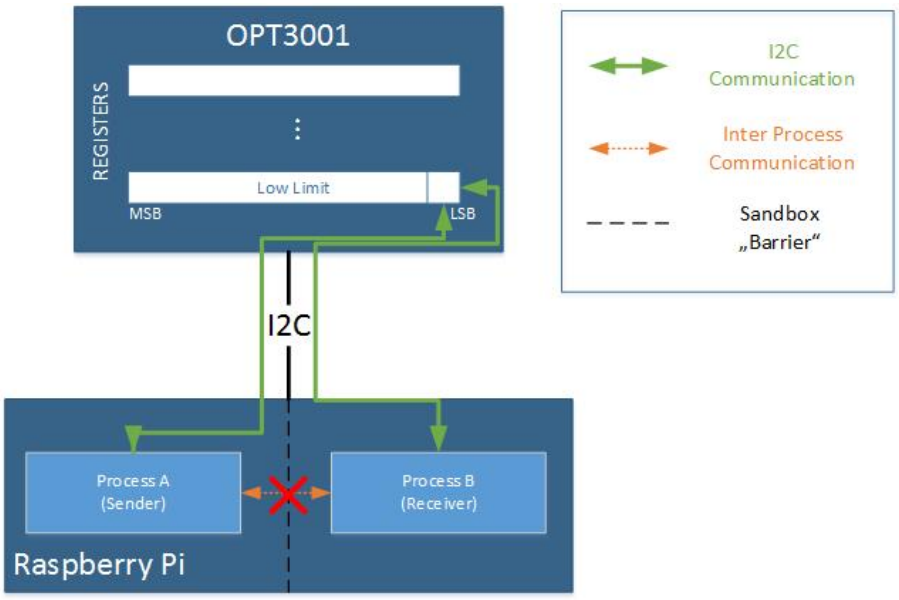


Figure 3.7: Schematic overview: sender and receiver communicating over one threshold bit

potential users are not disturbed. With only one bit, the synchronization method used in the previous attack, does not work and another method has to be used. This is described in more detail in Section 3.1.2.2. Instead of exploiting the mode bits in the configuration register to transmit information, the least significant bit in the high- or low-limit registers can be modified. This has very little impact on the reported value:

$$\begin{aligned} r_{full} &= 40.95\,\text{lux} \\ r_{lsb} &= 0.01\,\text{lux} \\ p_{lsb} &= \frac{r_{lsb}}{r_{full}} * 100 = \frac{0.01}{40.95} * 100 \\ &= 0.0244\% \end{aligned}$$

Depending on the actual use case, a threshold change of $\sim 0.02\%$ for the light level should be neglectable. To make use of the synchronization method introduced in the last chapter, at least two bits are necessary, where one bit is used for synchronization and the other for actual data. These can either be from the same register or from two different registers. If two bits of the threshold register are used, the change increases to a maximum of $\sim 0.07\%$. Alternatively both threshold registers can be used to keep the influence on the sensor behavior at a minimum.

Initialization: The covert channel is initialized by the sender. A bit is flipped each cycle to signal the availability of data. After a specified amount of bits was flipped,

the sender waits for the receiver to respond. If the receiver is not available, the sender starts the flipping procedure over again. The receiver monitors the bit and waits until the flipping of the bit has stopped. Then the receiver starts by requesting the first data packet. The sender should recognize the request, but some of the starting bits may have been missed because of various delays. From this point onwards, the standard packet procedures explained in Section 3.3 are used to determine the start of a request. The receiver keeps requesting the same packet until the sender is able to decode the request and starts the transmission of the payload.

Synchronization: If the attack is able to exploit two or more bits, at least one data bit and one synchronization bit, the synchronization method introduced in Section 3.1.2.1 can be used. In this case, instead of the register size, n equals the number of available bits. It also has to be specified which bit is used as the synchronization bit. In cases where only one bit is available, a different synchronization method has to be used. One such method is only available for operating systems where the scheduler works in a deterministic way, like TI-RTOS. As stated in Section 3.1.1, tasks with the same priority are scheduled using a round-robin approach. Therefore, each time a task executes, it can be certain that all other tasks where executed in between. The sender is now able to transmit n bits per cycle as long as it is ensured that each operation is atomic. This is important because as soon as the sender yields or enters a blocked state, the receiving process will be scheduled and will try to read the transmitted bits, which may be incomplete. If possible it is advised to use the first approach as it works independently of the used operating system.

Countermeasures: Looking at the countermeasures introduced in Section 3.1.2.1, not all of them will work on this attack. The first proposal was to change some registers to write-only. If the attacker chooses some threshold bits for example, this approach will prevent him from reading those bits and therefore prevent the covert channel, like in the last attack. But for the bit manipulation attack, the attack could also choose to use some bits in a configuration register that needs to be read- and write-able. For example, if the configuration register contains status bits, the users must be able to read those bits to determine if the sensor readings are ready. A slightly adapted version of this countermeasure could prevent both attacks by setting individual bits to be either read or writeable, but not both. This could be implemented using bitmasks on the sensor side. The second approach, proposed in Section 3.1.2.1, is to lock registers, that are currently not used. This comes with the same drawbacks as the first countermeasure: depending on the register bits chosen by the adversary, it would not have any impact. On the driver-side, restrictions could be implemented to prevent attackers from reading and writing to the same bits. For example, status bits could be only readable, while configuration bits are write-only. This would prevent both attacks.

3.1.2.3 Read Only Attack

In this section, the third attack is introduced. This attack is designed to emulate normal user behavior as close as possible, to avoid detection and to decrease the number of feasible countermeasures. The drawback for this attack is the lower data rate compared to the previous attacks, which is shown in detail in Section 4.3.

There exist many sensors that have multiple internal "sub-sensors", in order to sense different physical properties. A good example is the LPS25H pressure sensor from ST [46]. This sensor provides two different sensor readings: a pressure and a temperature reading. Each of those readings is linked to an individual status bit in the status register. For sensors with only one internal "sub-sensor", the status bit is reset after a user reads the status register. In case of multiple "sub-sensors", the status bits are only reset when a user reads the corresponding result register. The LPS25H datasheet notes this as "P_DA is set to 1 whenever a new pressure sample is available. P_DA is cleared when PRESS_OUT_H (2Ah) register is read." [46]. This behaviour allows an attacker to detect when and which result registers are read by polling the status register. The resulting information can be used to build a covert-channel. A schematic overview of this idea is given in Figure 3.8. Because the attack relies only on polling of the status register to check the status flags and reading of the result registers, it is very hard to distinguish from normal user activities. In addition, compared to the previous attacks, no write operations are required.

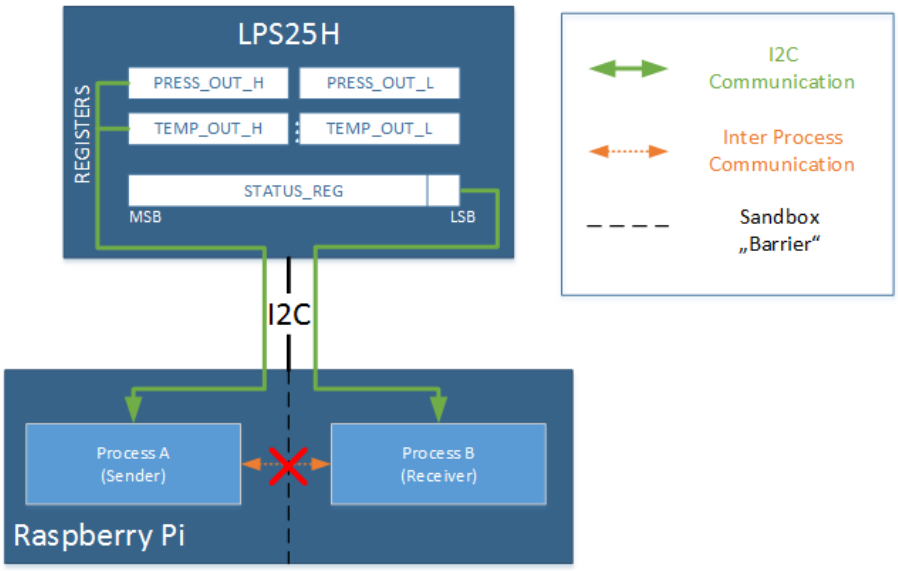


Figure 3.8: Proposed "read only" covert channel

A key element of this attack is the encoding of data using only read operations. This thesis proposes two ways to achieve this encoding. Either by converting the data to time encoded information or directly encode the information in the status flags. In both cases, the processes first need to wait for the sensor to finish one measuring cycle. After that, the information is encoded using read operations on the result register with or without delays. For example, the first version would encode a logical 0 as a 0ms delay and a logical 1 as a 5ms delay. Using this type of encoding, the throughput is limited by the bus speed and the sensor speed. The LPS25H sensor has a maximum sensing frequency of

$$f_{\text{sens}} = 25Hz$$

which results in a new sensor reading every

$$t_{\text{sens}} = \frac{1}{f_{\text{sens}}} = 40\text{ms}.$$

As shown in Section A.1, one read operation using a 400kHz I2C bus takes about

$$t_{\text{read}} = 0.1\text{ms}.$$

So overall, the duration of one cycle should be around

$$t_{\text{cycle}} = (40 + x * 0.1)\text{ms}$$

where x is the number of queued I2C read operations. With one active user process, one sender process, and one receiver process, there can be at most 3 queued operations at the same time. The theoretical maximum cycle time is

$$\begin{aligned} x_{\max} &= 3 \\ t_{\text{cycle, max}} &= (40 + x_{\max} * 0.1) \\ &= 40.3\text{ms}. \end{aligned}$$

The throughput depends on the chosen delay Δ which is added after each cylce. The theoretical maximum throughput is

$$\begin{aligned} \Delta &= x_{\max} * 0.1 = 0.3 \\ t_0 &= t_{\text{cycle, min}} = 40.1\text{ms} \\ t_1 &= t_0 + \Delta = 40.4\text{ms} \\ t_{\text{mean}} &= \frac{t_0 + t_1}{2} = 40.25\text{ms} \\ \text{throughput} &= \frac{1000}{t_{\text{mean}}} = 24.8\text{bps} \end{aligned}$$

but it would also be very susceptible to timing delays. A much more conservative Δ of 5ms would yield a throughput of 23.5bps and give enough room for the compensation of timing errors. Table 3.4 lists different Δ and the achievable throughput for the LPS25H in 25Hz mode.

Δ	Throughput	Efficiency
0.1	24.8	100
1	24.6	98.9
2	24.3	97.7
5	23.5	94.2
10	22.2	89.0
20	20.0	80.1

Table 3.4: Theoretical throughput for different delta values

The other version of this attack encodes the information directly into the status bits. This can be achieved in two ways, as shown in Table 3.5. In the case of the LPS25H, using the 1-bit variant, a bit is transmitted by either reading the pressure result register or the temperature result register. For example, reading the pressure result register could be interpreted as 0 and reading the temperature result register as 1. The 2-bit variant can encode multiple states into one operation, but using the state 11 (results ready, no result register read) to encode data, some special synchronization is necessary. This thesis will, therefore, focus on the 1-bit variant and all of the following sections will only refer to this variant.

P_DA	T_DA	1-bit encoding	2-bit encoding
0	0	error	00
0	1	0	01
1	0	1	10
1	1	ready	11 or ready

Table 3.5: Information encoded into status bits

Initialization: The initialization is based on the packet system introduced in Section 3.3. The receiver tries to requests packets until the sender starts responding, while the sender waits until it receives a valid request. This method ensures the correct communication setup. To reduce stress on the sensor, the receiver delays its requests by a significant amount of time in relation to sensor and bus speeds ($\gg$ 40ms). An example procedure is shown in Figure 3.9.

Synchronization: As mentioned in the previous subsection, this attack encodes information by reading result registers, which in turn resets their respective status flags. These changes are then observed by the receiving process. Because it is not possible to force the sensor to set these flags, it is required to wait for a full measurement cycle, before new information can be encoded. This is used as a simple synchronization mechanism. As soon as both status flags are set, the sender and receiver know that one cycle is completed and new data can be encoded. One such cycle is shown in Figure 3.10. Note that "write data" stands for "encodes one bit, by resetting one status flag".
In the first step, the sender reads the pressure result register to reset its status flag. The receiver detects the change, reads the status register and decodes the information. Afterwards, both receiver and sender poll the sensors status register, waiting for the measurement cycle to finish. As soon as both status flags are set to 1, the sender encodes new information, by reading one of the two result registers and the cycle is complete. It is important to note, as it is not known which process runs first, the sender needs to ensure, the receiver also recognized the completed measurement before new information is written. Otherwise the receiver would not detect or dismiss new information as errors. To ensure that both parties know that the sensor is ready, the sender waits half a cycle before writing new data. If the receiver still misses one bit, the packet system introduced

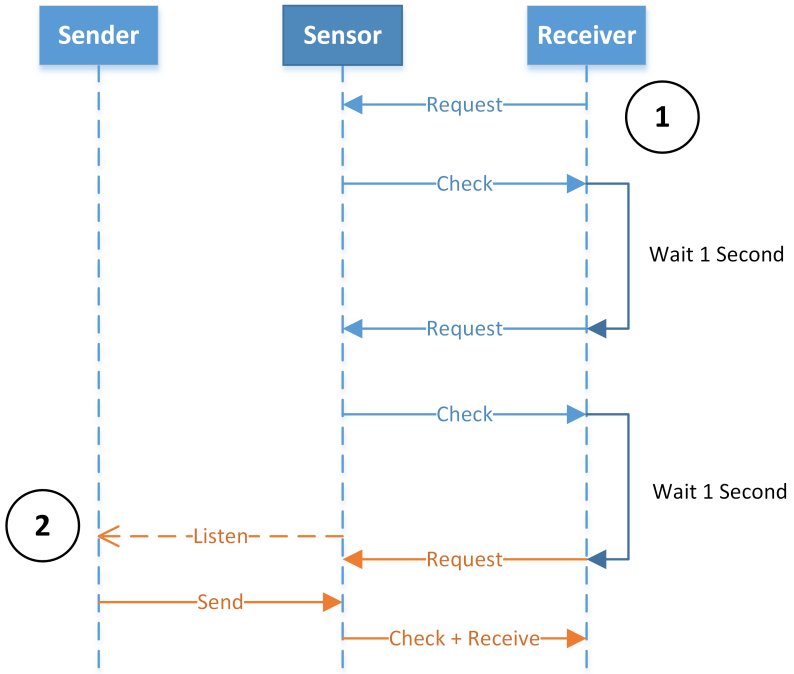


Figure 3.9: Initialization procedure for the read-only method

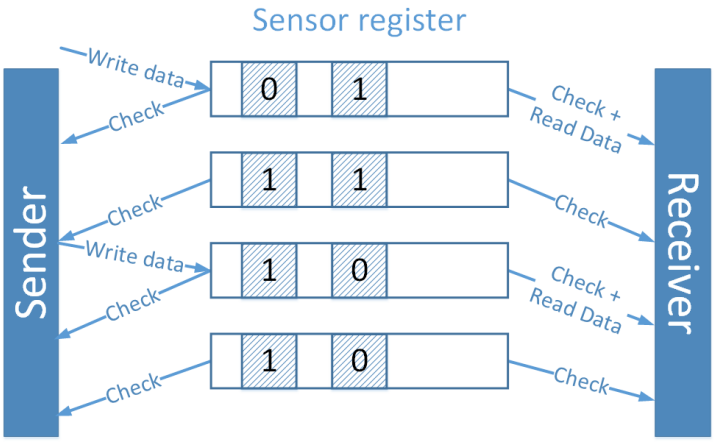


Figure 3.10: One read cycle

in Section 3.3 and the error detection codes introduced in Section 3.3.2 will take care of it.

Packet Synchronization: A second mechanism is needed to detect data packet borders to avoid shifting. This is achieved by inserting an additional delay of the same size as the sensor measurement time. When the receiver detects this longer delay, it knows that this marks the end of a packet, and therefore also the start of a new data packet, and adjusts accordingly. In case of an error induced timeout or multiple timeouts, the connection is reset and the receiver tries to request the first data packet again.

Countermeasures: To prevent the proposed covert channel, the direct access to the hardware interface has to be abstracted by providing a custom device driver. This driver has to limit the ability of users to read status flags. One way would be to provide users with getters for both sensor values and return a value on each call independent of if a new value is available or not. If the measured value is fluctuating a lot, the users may still be able to determine if a new value or a stale value was returned by comparing sequential return values.

3.2 Managed-Access Sensors

This section covers attacks on platforms that implement a managed access to sensors. For these platforms, the accessible information of a specific sensor highly depends on the implementation. Some platforms may allow users to read and write registers using provided methods, while others implement a strict abstraction that only allows users to subscribe to sensor events. In these cases, users have no means to access registers directly. In Section 3.2.1, the Google Android platform is introduced and important aspects like the permission system and the sensor stack are covered in some detail. Afterwards, in Sections 3.2.2 and 3.2.3, two state-of-the-art attacks are proposed that bypass security features and enable covert channels between applications.

3.2.1 Platform - Google Android

In this section, all relevant parts of the Google Android ecosystem are introduced. First, the sensor stack is discussed in more detail as it is responsible for handling the user-sensor interaction. Then, a short overview of Androids' permission system is given, which is intended to protect users from malicious applications.

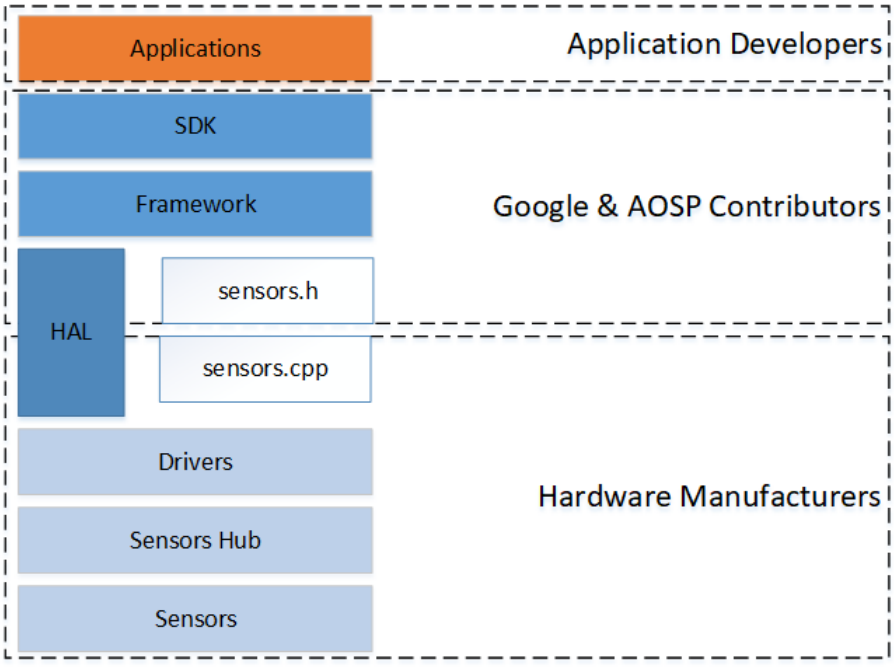


Figure 3.11: Layers of the Android sensor stack and their respective owners [20]

The biggest difference to the previously discussed attacks, is the introduction of the *Sensor Stack*, shown in Figure 3.11. This stack prevents users from accessing the sensor interfaces directly. It consists of six layers underneath the application where each layer has its own purpose and owner. The lower layers have to be supplied by the hardware manufacturer

and contain the sensors itself, the *Sensor Hub*, the *Drivers*, and part of the *Hardware Abstraction Layer (HAL)*. Because the main System on Chip (SoC) requires a considerable amount of power, manufacturers can include an optional sensors hub, which is either integrated or a separate hardware that is able to perform some low-level calculations while the main SoC is suspended. The sensor hub should use as little power as possible to enable low power applications. Another usage scenario for the sensor hub is *Sensor Fusion*. Often, sensor readings can be enhanced by adding information from other types of sensors. As an example of an integrated sensing hub: Qualcomm recently announced a new Version of its *Sensing Hub* for the Snapdragon 865 5G mobile platform which features low power AI functionality for sensors [47]. The next layer contains the drivers which directly communicate with the underlying hardware. These drivers are developed by the sensor chip manufacturers. This layer can also be integrated into the HAL, which is maintained by the system integrator. Both of these layers build the connection between the Android framework and the sensor hardware. The interface for the HAL is defined by Android and implemented by the hardware manufacturers, as shown in Figure 3.11. In the framework layer, the implementation switches from low-level C/C++ to Java, to provide the functions used by the SDK. Up until this point, all layers are only capable of serving a single application. The framework handles the multiplexing to allow multiple applications access to sensors at the same time. It is also responsible for enabling/disabling sensors and sensor configurations such as the sampling frequency. However, the multiplexing introduces some side-effects. When multiple applications request readings from the same sensor, the framework will set the sensor speed to the higher one of the two sampling rates. This means, that both applications will receive the events at the faster speed. As discussed in Section 3.2, this can be exploited to build a covert channel. The last layer defines the Sensors SDK API, which is the programming interface that application developers use. The SDK implements functions to register and unregister callbacks for available sensors. The user supplies a minimum update frequency and the SDK and underlying layers handle the rest.

Permission System: To protect the user from malicious activities, Android implements a strict permission system [19]. The goal of this system is that "no app, by default, has permission to perform any operations that would adversely impact other apps, the operating system, or the user". If an app wants to perform any secured action, it has to state the required permission in its manifest file. Android defines two types of permissions:

- **normal:** low-risk permissions in regards to privacy or device operation
- **dangerous:** higher risk permissions, that could affect privacy or device operation

Normal permissions are automatically granted by the system and only used to inform the user about certain interactions with elements outside of the app's sandbox. In contrast, *dangerous* permissions require an explicit agreement from the user. This agreement is either accepted on install (install-time requests), where a list of required permissions is shown to the user and has to be accepted to continue the installation. Another method, which is available since Android 6.0, are runtime requests. Instead of requiring the user to accept all permissions on install, the individual permissions are requested during runtime when the application tries to perform an action that requires a permission. In this case,

a popup is displayed which states the permission name and has to be either accepted or denied by the user. If the user denies, the action cannot be performed, which may restrict certain features of the application. Otherwise the permission setting is saved and no further popups are displayed.

3.2.2 Interval Attack

The first attack, introduced in this section is based on the intervals between sensor events. As explained in the previous section, Android implements a sensor stack with many different layers to abstract access to sensors. There are some sensors, that require additional permissions, for example the GPS sensor, while others don't require any permission at all. If an application subscribes to a sensor, the framework layer of the sensor hub configures the underlying hardware to report values at the desired rate. Besides a callback function and a reference to a sensor object, the sensor API only requires the sampling period in μs as an additional parameter:

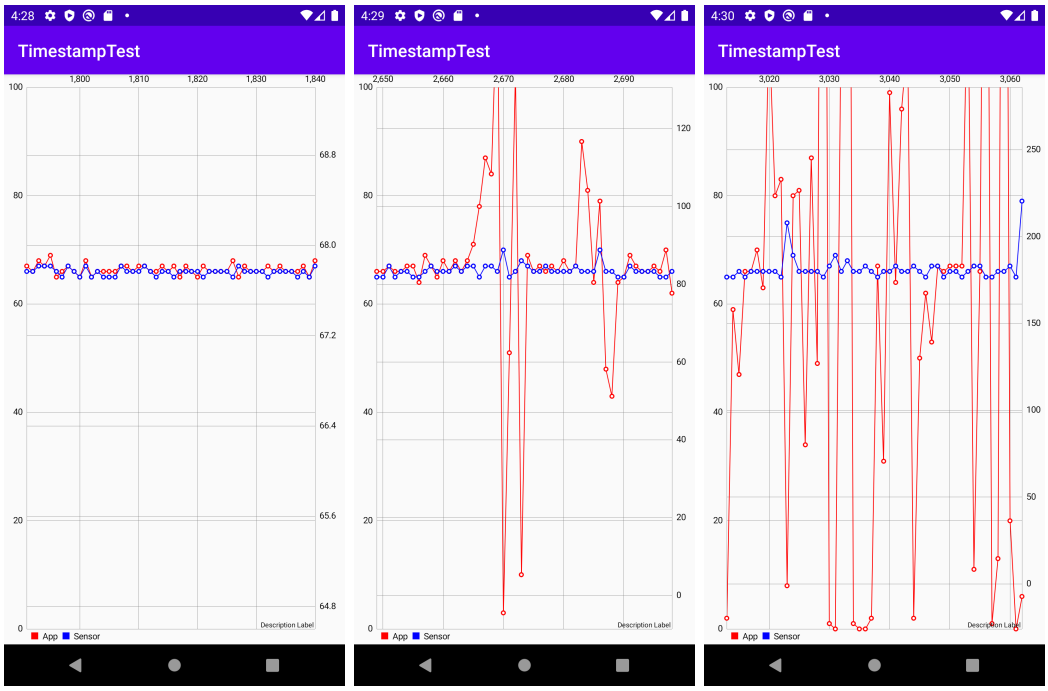
```
public boolean registerListener (SensorEventListener listener,
                                 Sensor sensor,
                                 int samplingPeriodUs)
```

After subscribing, the framework calls the listeners' *onSensorChanged* function each time a new value is available, using the configured sampling period as a maximum. The only parameter is a *SensorEvent* which includes the requested value(s), a timestamp and other information as shown in Table 3.6. Because it is the timestamp at which the event happened and not when it is received by the application, it directly corresponds to the sensing interval. To visualize the difference between the measurements timestamp, and the applications' timestamp at which the event was received, a simple testing application was developed. The results are shown in Figure 3.12. While the CPU is idle, both timings seem very similar for a sampling interval of 65ms. But as soon as the CPU usage gets higher, the variance between application timestamps spikes, which can be seen in Figure 3.12b and 3.12c, while the sensor timestamps stay consistent.

Data type	Field name	Description
int	accuracy	The accuracy of this event.
Sensor	sensor	The sensor that generated this event.
long	timestamp	The time in nanosecond at which the event happened.
final float[]	values	The length and contents of the values array depends on which sensor type is being monitored.

Table 3.6: SensorEvent fields [20]

This renders the application timestamp unusable for the side-channel attack described in this section. Another problem occurs when a higher sampling frequency is used. As shown in Figure 3.13, for a sampling frequency of 10ms, the application timestamp is very inaccurate and varies between 10ms and 40ms, while sensor timestamps stay consistent at 10ms and are therefore the preferred choice.



(a) Idle (b) Medium (c) High

Figure 3.12: Timing differences between application and sensor for different CPU usages

As previously explained, sensors are a shared medium that is abstracted by the *Sensor Stack*. On modern mobile operating systems multiple processes can be subscribed to a single sensor at the same time by using the provided subscription functions and callbacks. Each subscription is active until it is either canceled by calling an unsubscribe function or when the process is terminated. For normal user applications, the Android developer guide advises to cancel all subscriptions on pause and subscribe again if the application is continued [20]. This ensures that the sensors do not keep on measuring while the application is not used (e.g., in the background), which would cause a significant battery drain. For some services it is necessary to keep monitoring while in the background. Therefore, Android does not automatically unregister subscriptions. In regards to the *samplingPeriodUs* parameter, the developer guide states: "The delay that you specify is only a suggested delay. The Android system and other applications can alter this delay. As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify (that is, you should choose the slowest sampling rate that still meets the needs of your application)." This does already hint at the core of the problem: other Android applications are able to alter the delay at which the sensor events are sent to all subscribed applications. An adversary can use this to his advantage by integrating a covert-channel in two applications and transmitting data between them without the user's permission. For example, one application could be a simple offline password safe which does not need any internet connectivity. If the application wanted to secretly send passwords to a remote location, it would require two permissions: "an-

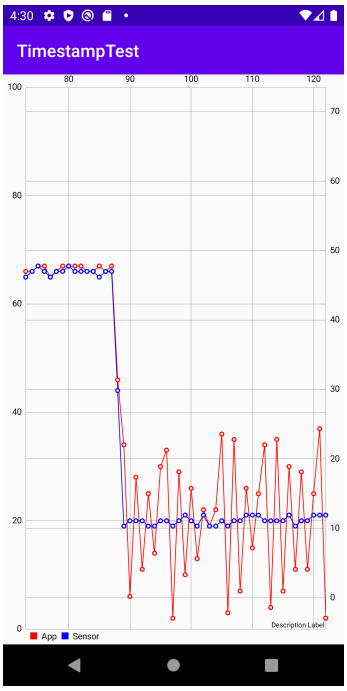


Figure 3.13: Timestamp variation for small intervals

droid.permission.INTERNET" and "android.permission.ACCESS_NETWORK_STATE". As mentioned in Section 3.2.1, Android shows a list of required permissions before installing an application from the app store. If the internet permission is on the list of required permissions from an offline password manager it would be obvious that something is not quite right. On the other hand, if the application would be a weather app instead of an offline password manager, the internet permission would justified because the weather data will be downloaded from remote locations. Under normal circumstances applications run in a sandbox environment where it is not possible to access information about other applications. Therefore, applications cannot exchange information without requesting specific permission to do so. If the adversary is able to control both apps and include a covert channel, he will be able to use that channel to build up a communication between the applications without the need for extra permissions. This undermines the security- and sandbox-system provided by the operating system.

To maximize the throughput of the channel, the smallest possible intervals are chosen. These depend on multiple factors. First there are physical limits, like reporting intervals and maximum reporting speed, which are specified by the hardware sensor. Further, the specific driver implementation supplied by the manufacturer matters, as it is responsible to provide the upper layers with information about the sensor hardware capabilities. Besides using the smallest intervals, it is also important to test the stability of these intervals and adjust the difference and thresholds accordingly. Otherwise the entities could have problems distinguishing between real data and noise. In Section 4.3, this technique is evaluated using both the Android Emulator on Windows and a physical Android device.

Because the thresholds of the sensors are global and can only be pulled below the current threshold, it is important that the current threshold is not equal to the minimum threshold of the particular sensor. During testing, some applications were found (mostly browsers) that use the maximum reporting frequency regardless. When such an app is running, the sensor events are always reported at the fastest possible speed and the frequency can not be altered until the application is closed and the reporting speed is lowered. This effectively disables the covert channel and no more transmissions are possible using the proposed method. Therefore, a new method is introduced in the next section.

3.2.3 Subscription Attack

The subscription attack exploits abnormal events that occur when an application subscribes to a sensor. Like the first attack, this attack targets sensors that can be used without special permissions. As mentioned in the last section, all applications subscribed to a specific sensor receive the update events at the same interval. This interval is the minimum of all subscription intervals. If one of the applications already uses the lowest possible interval, no further changes are possible until this application removes its listener. A typical application that shows this behavior, are browsers. Chrome, Firefox, Opera, and the Android browser all subscribe at the maximum possible frequency. This was discovered because one idea was to build a covert channel between an Android application and a web application. This would enable new powerful ways for the attacker to gain access to secret data. For example, a password safe could implement a link to the FAQ page which opens a browser and redirects to the attacker's server. While the user is reading, the Android application starts transmitting all stored passwords over the covert channel to a JavaScript application running in the background of the FAQ page. Through experimentation, it was discovered that the sensor framework generates event outliers each time a new subscription occurs. In a low noise environment, with low system load and little to no sensor usage, these outliers can be detected by another application. Using a time-based encoding scheme, the sender can encode data. This data can be decoded by a receiver application. The design of the covert channel is visualized in Figure 3.14. Because a low noise environment is hard to achieve in a real-world scenario, this covert channel design was not developed beyond the initial design. A small proof of concept application using the initial design was implemented, which is covered in Section 4.3.

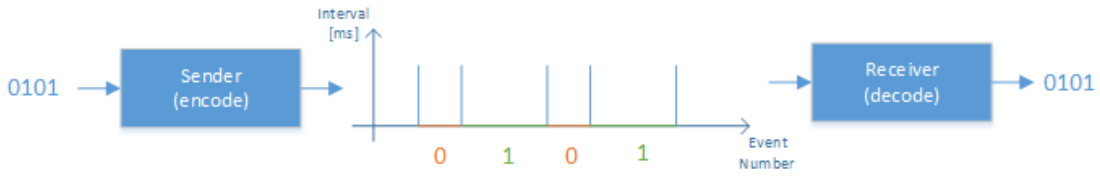


Figure 3.14: Covert channel design based on subscriptions

3.3 Packet System

This section introduces several concepts, that are applicable to all previously mentioned attacks. These concepts include error detection codes, that are used to detect single or multiple bit-flip errors in the payload. Further, error correction codes are introduced that support some amount of error recovery in addition to error detection. To be able to control and manage the flow of data, a simple packet-based system is designed. This system includes different packet types, packet ordering, and different types of error detection. For noisy channels, two mechanisms get introduced, which increase the versatility of the attacks: templating and adaptive packet size.

Based on [41], a packet-based system was designed that handles transmission errors, retransmission and other tasks in regards to data flow. This system follows a simple request-response principle. Before the transmission starts, the sender keeps listening for incoming requests. The structure of such a request is shown in Figure 3.15. To keep things compact, the request only contains the Hadamard encoded sequence number, which is explained in more detail in Section 3.3.2. In [5], it is shown that for a $k \leq 7$ the Hadamard error correction codes are optimal. With this fact in mind and the exponential encoded size of 2^k, it was decided to only use two bits for the sequence number. Because this number is only used to differentiate between the newest and the last few packets, such a short sequence number is sufficient.

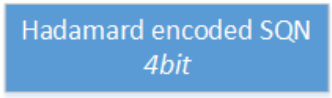


Figure 3.15: Structure of a request packet (REQ)

Figure 3.16: Structure of a response packet (RES)

Upon receiving a valid request, the sender starts the transmission of the associated data packet. These response packets have a different structure as request packets, shown in Figure 3.16. The first bit determines the type of the packet. In its current version, there are two supported types: data(0) and commands(1). Depending on the chosen type, the next section contains ether arbitrary data or one of the reserved commands. All available commands are listed in Table 3.7. The SQN field contains the current sequence number, which is incremented with each packet, regardless of data or command type. To provide a mechanism that is able to detect corrupted packets, an error detection code is appended

to the end, which is the only feasible option. Error correction codes would cause too much overhead, because of the large packet size and their 2^k relation.

Code	Command
00	Increment packet size
01	Decrement packet size
10	Stop transmission
11	Reverse data direction

Table 3.7: Available commands

3.3.1 Error Detection Codes

Error Detection Codes are used to detect errors in a payload, that can result from a noisy channel. The covert channels discussed in this thesis use a shared medium, e.g., the I2C bus and every user interaction with that medium has a high chance to cause bit errors during a transmission. Therefore, the channels are treated as noisy channels.

Parity Bit: One of the most basic error detection systems is a parity bit. The parity bit gets added at the end of the payload. If the number of payload bits set to true is even, the parity bit is set to false, otherwise it is set to true. This results in a data packet with an even number of set bits. If the receiver gets a packet with an odd number of set bits, he knows that at least one bit-flip has occurred and the packet is corrupted. This approach is very simple, but it comes with a drawback: it is only able to detect an odd number of bit-flips. This is demonstrated in the following example:

(1) Alice wants to send 1001 to Bob.

(2) Alice counts the bits set to true: $1 + 0 + 0 + 1 = 2$.

(3) She adds a parity bit which is set to false, to ensure that the resulting packet has an even parity.

(4) This packet, 10010, is then sent to Bob.

(5a) Even number of bit-flips: Bob receives 11110. He calculates the parity $1+1+1+1+0(\mod 2) = 0$". Because the parity is even, Bob thinks the packet data is correct and accepts the packet.

(5b) Odd number of bit-flips: Bob receives 11111, which equals three bit-flips, the calculated parity is $1+1+1+1+1(\mod 2) = 1$, which is correctly detected as corrupted data.

In the context of this thesis' channels, depending on the transmission time, multiple user interferences can occur which would result in multiple bit-flips. Therefore, the parity bits do not provide an adequate countermeasure and another method needs to be used. A good

example are Berger Codes, which are discussed in the next paragraph. For very short data transmissions, error correction codes are also feasible. These are discussed in in Section 3.3.2.

Berger Code: Berger Code [3] is an error detection code, introduced by J.M. Berger in 1982. Depending on the type of covert channel used, most errors are so-called unidirectional errors. Unidirectional errors are errors where bit-flips only occur in one direction, meaning only zeros flip to ones or only ones flip to zeros. Some covert channels in the thesis can only be influenced in one direction and are reset on a timer, which is a typical asymmetric channel that is susceptible to unidirectional errors. For these types of errors, Berger Codes have been proven optimal by C.V. Freiman [15].
Berger Codes are often used because of the simple principle and implementation. It gets computed by summing up all of the zeros of a payload. This means, that for a payload with size n the error detection code has a size of $k = \log(n+1)$. Another reason to choose Berger Code is its straight forward implementation.

In general, there are three error cases to be considered:

(1) Error in the data section

(2) Error in the error detection code section

(3) Error in both, the data and the check section

For (1), the number of zeros changes and the error detection code does not match this change which is detected by the receiver by calculating the Berger Code from the received data and comparing it to the received Berger Code. If an error is introduced in the error detection code section, case (2), the same statement holds true: the error detection code does not match the data which can be detected by the receiver in the same way as in the first case. For the third case, the unidirectional error property is important. If only one type of bit-flip can occur, either the number of zeros decreases or increases in both the data and the error detection code. Because the Berger Code counts zeros, it is indirectly proportional to the data. A proportional change to both parts will therefore invalidate the error detection code in every possible case.

3.3.2 Error Correction Codes

In addition to error detection, Error Correction Codes (ECC) are also able to recover mutated information, but are much more sophisticated than the simple error detection codes. The size overhead, caused by the encoding, is also much higher. Figure 3.17 shows a size comparison between an error detection and an error correction code. This is of course only an example and overhead sizes vary greatly between different codes. Because of the higher overhead, error correction codes are only used on small data request packets. More details about the packet system can be found in Section 3.3. The following paragraphs introduce two error correction codes, a very simple repetition code and the more sophisticated Hadamard encoding.

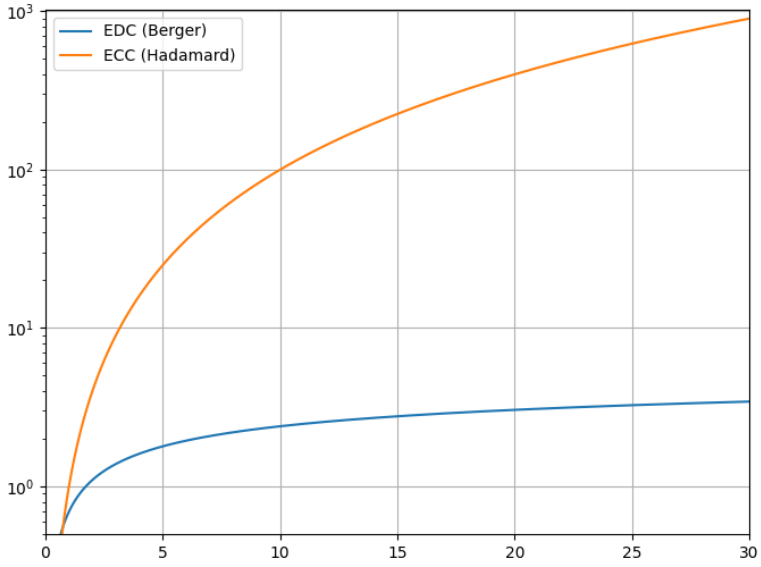


Figure 3.17: Code size comparison between Berger and Hadamard codes

Repetition Code: A repetition code is a very simple way to achieve error correction. Instead of sending the data directly over the channel, each bit is sent a specified number of times. The receiver is then able to detect the received bit using a majority vote. If most repetition bits received are false, the data bit is also assumed to be false and vice versa. A few simple encoding/decoding examples using a (3,1) repetition code are shown in Table 3.8. (3,1) means that a single data bit is represented by three repetition bits. The first row represents a transmission without any interference, so no errors are present. In the second row some bit-flips occurred, resulting in 010 and 001 instead of 000. Because only one bit-flip occurred per data bit, the receiver is able to decode the data correctly. In the last row, multiple bit-flips have occurred for the third data bit. In this case, the receiver interprets 001 as a 0 data bit. For $\mathbb{Z}_2$, odd repetition codes are proven optimal [40].

Data	Repetition Code	Received	Data
0 0 0	000 000 000	000 000 000	0 0 0
0 1 0	000 111 000	010 111 001	0 1 0
1 0 1	111 000 111	110 100 001	1 0 0

Table 3.8: Example encodings/decodings of a (3,1) repetition code.

Hadamard Code: The Hadamard code [59], also called Walsh-Hadamard or Walsh code, is a more sophisticated error correction code. It is based on the Hadamard matrices

and identifies as a $[2^k, k, 2^{k-1}]_2$-code, where k is the message length, 2^k is the block length and 2^{k-1} is the distance. There exist multiple different ways to construct a Hadamard code: using inner products, a generator matrix or general Hadamard matrices.

A Hadamard matrix is defined as a square matrix H of order n, with entries ± 1, such that

$$HH^T = nI_n$$

is satisfied, where I_n is the unit matrix of order n. An important property of these matrices is the ability to create a new Hadamard matrix by calculating the tensor product of two Hadamard matrices with order equal to the product of the order of the factors. This means, that matrix $X = ||x_{ij}||$ and $Y = ||y_{ij}||$ with orders l and k, result in a matrix $Z = ||x_{ij}Y||$ with order lk. Some examples of Hadamard matrices:

$$H_1 = \begin{bmatrix} 1 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Hadamard codes are then constructed by using H_{4m} and $H'_{4m} = -H_{4m}$ matrices and replacing all -1 entries with 0s. For example:

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$\tilde{H}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\tilde{H}'_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

This results in the code:

$$C_4 = 1111, 1010, 1100, 1001, 0000, 0101, 0011, 0110$$

The decoding is then performed the following way:

1. Change the received word back into the ± 1 form (change 0s to -1s) w.
2. Compute $s = wH_{4m}$

3a. If w is equal to a codeword, the transmission was successful and the syndrome s will be either $\pm 4me_k$, where e_k is the k^{th} row of the $n \times n$ identity matrix I_n.

3b. Otherwise, $s \neq \pm 4me_k$. If the number of errors is below $n-1$, the position of the entry in $\tilde{s} = w\tilde{H}^t_{4m}$ with the largest absolute value will equal the correct row number in H_{4m} or H'_{4m}.

3c. If the number of errors is bigger than $n-1$, decoding is not possible.

For this thesis, the actual implementation is done via lookup tables, as shown in Section 4.1. To keep the overhead at a minimum, a very small Hadamard matrix is chosen. Further, only the error detection part is used. This means, that a simple comparison of the received word to the codewords in the lookup table is sufficient. The implementation of this design is much easier to accomplish for very small data types.

3.3.3 Adaptive Packet Size

In general there are two types of interferences when communicating over a covert channel: temporary and permanent. A user accessing the sensor to get the latest temperature value on-demand, would be a simple temporary interference that could lead to the corruption of a single packet. This case is already handled by the protocol using retransmissions. In the case of a temperature logger that accesses the sensor every second for example, the system may be unable to recover by using retransmissions. Figure 3.18 illustrates this case. In the first row, the user accesses are shown as yellow segments. The length of the payload causes the user interference to hit every packet at least once, which can result in a lot of corrupted packets and therefore, retransmissions. The response for packet 1 has to be retransmitted at least two times which results in an overhead of 200% or more. This overhead would be tolerated, because scaling also causes some amount of overhead, but in the worst-case scenario, the communication would not be possible at all.

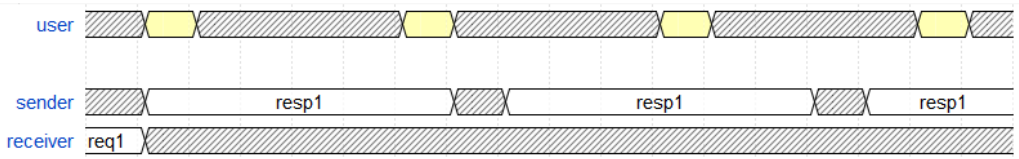


Figure 3.18: Packet size is too big, which causes many retransmissions.

To accommodate for this special case, an extension to the packet system was developed: "Adaptive Packet Size". This extension allows the sender to reduce the packet size if multiple retransmissions are encountered. The exact number of retransmissions before scaling can be configured based on the environment (see 4.1 for more details).

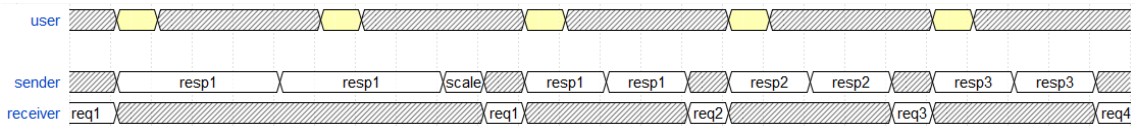


Figure 3.19: Packet size is too big, so the sender starts to scale down.

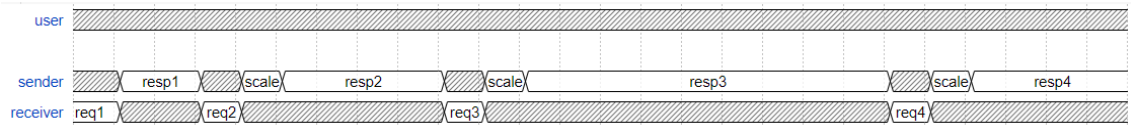


Figure 3.20: Packet size is too small, so the sender starts to scale up.

Chapter 4

Implementation

This chapter covers all the implementational details. As the attacks are highly dependent on the platform and the sensor access type, individual implementations are needed. In Section 4.2 two hardware platforms that were used for the implementation are described in more detail. After the initial tests, the focus was shifted to a Linux based platform, as there would be too many variables otherwise. Besides the platform, there are also a lot of different interfaces and sensor types which each need a separate implementation for every attack. To combat this complexity, a modular testbed application was developed which is introduced in Section 4.1. In the end, an evaluation is given in Section 4.3 including additional features such as the packet system introduced in Section 3.3.

4.1 Testbed

As mentioned, the complexity of testing multiple different attacks on multiple different sensors with multiple different interfaces increases the complexity of the testing system. To lower said complexity and introduce some flexibility, a layer-based abstraction approach was used to implement the testbed. In the upcoming Sections 4.1.2 to 4.1.5 all of the individual abstraction layers are described in more detail. The current section tries to give a general overview of the testbed implementation and also introduces all of the tools, that were used during development.

CLion: CLion is a cross-platform Integrated Development Environment (IDE) made by JetBrains [30]. It is very similar to the popular Intellij IDEA, but instead of Java it is targeted to C/C++ development. This IDE was chosen because of the powerful code completion features and its high reliability. The first idea was to develop directly on the target platform, a Raspberry Pi 3B+. This was until a big drawback of such a powerful IDE comes into play, the very high memory consumption of 1-2GB. Because the Pi 3B+ only comes with 1GB of main memory, the IDE slows down significantly at some points. One other problem is the resource consumption at compile-time, which is nearly impossible to run beside the already high resource consumption of the IDE. To prevent these problems, a different approach was chosen, namely cross-compilation. Using this method, only the compiled application is transferred to the Raspberry, which means all of the heavy lifting is done by an external system. This approach is explained in more detail

in the next paragraph. A screenshot of the CLion IDE is shown in Figure 4.1.

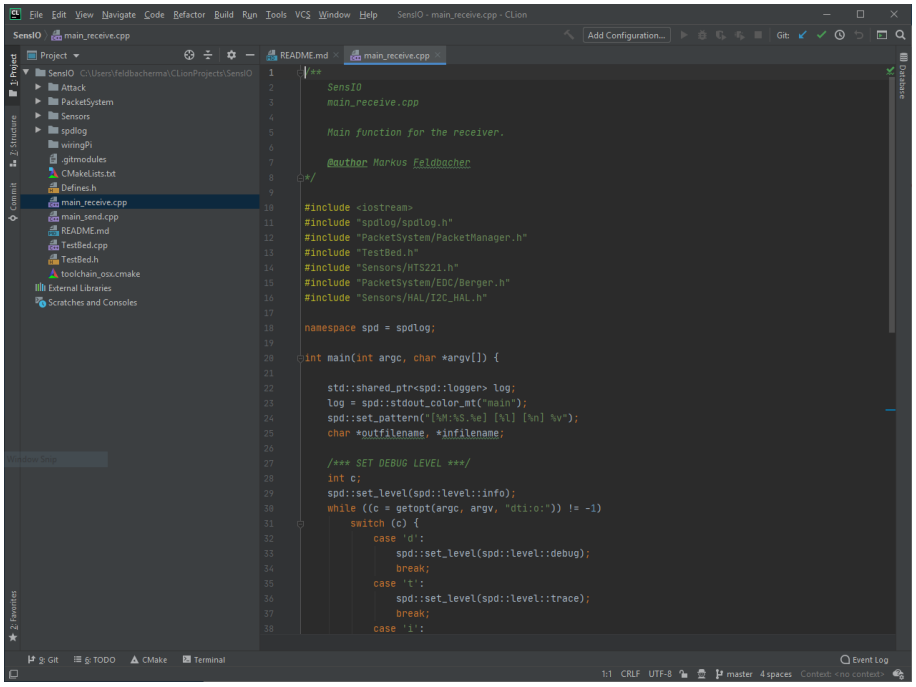


Figure 4.1: Jetbrains CLion IDE

Cross-compilation: As mentioned in the last paragraph, the Raspberry Pi 3B+ lacks resources to run the IDE and compile the code, which means an alternative solution is required. Therefore, all resource-intensive tasks are moved to an external pc, where all of the development work is done. Because a Macbook Pro running OSX was used during this thesis, the following details are in regards to this system. Figure 4.2 gives an overview of the cross-compilation procedure. There are two entities involved, the development pc and the Raspberry Pi 3B+. The development and cross-compilation is done on the pc side. For the cross-compilation step, a tool suite called *croostool-NG* [9] is used with the following GNU tools:

- GNU make
- GNU Compiler Collection (gcc)
- GNU C Library (glibc)
- GNU Binutils

The resulting Cmake file contains all of the required instructions to generate the correct make file using the *armv8-rpi3-linux-gnueabihf* environment. A simple *make* command compiles the source code into a ready to use binary file for the Raspberry Pi. This binary

is then copied to the Pi 3 via FTP. Filezilla [33] is used as the FTP client for this step. As CLion also supports FTP, it could be used instead, but this was not tested.

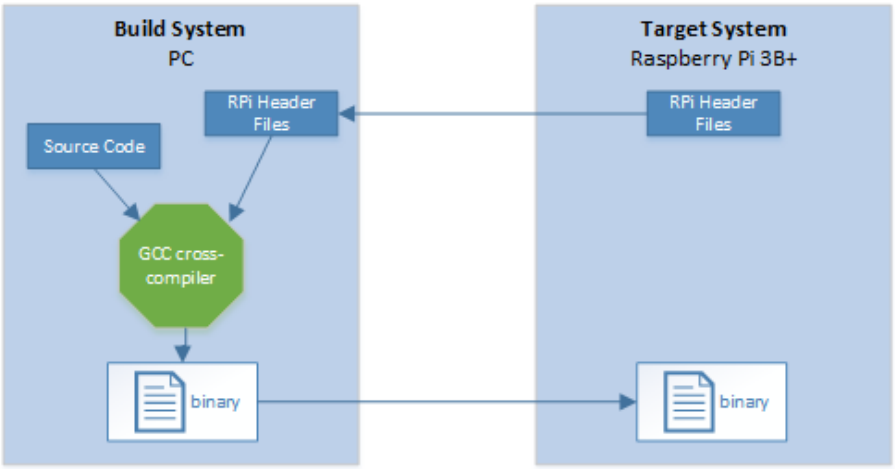


Figure 4.2: Cross-compilation procedure

4.1.1 Architecture Overview

To combat the high complexity the arises from multiple target sensors, different attacks and interfaces which should all be able to be mixed and matched as needed, a "covert channel testbed" was designed and implemented. To maximize flexibility, a modular, layer-based approach was chosen. All in all the testbed application consists of four layers which each abstract a different part of the system. Figure 4.3 shows a graphical overview of this layered architecture. Starting from the bottom, the first layer is the *Hardware Abstraction Layer (HAL)*. In this layer, all of the low-level interface actions are implemented. This can be an I2C or SPI interface for example. These two bus types are very common and therefore most sensors and host platforms support them. The goal of this layer is to provide basic read and write functionality to the upper layers independent of the actual interface. More details about the structure of the modules in this layer can be found in Section 4.1.2. Next up is the *Sensor Abstraction Layer (SAL)*. This layer consists of modules that contain all of the implementations specific to particular sensors. Because attacks are highly dependent on the used sensor, this layer has a relatively tight coupling with the Attack Abstraction Layer. The Sensor Abstraction Layer tries to encapsulate implementation specifics and provides a set of standardized functions and address maps which are used by the upper layers. Again, more details in regards to this layer and its modules can be found in Section 4.1.3.
The next layer, the *Attack Abstraction Layer (AAL)*, contains modules that implement the attacks designed in Section 3.1. For the covert channel, this is the last layer needed for its core functionality, which is the transmission of data. The Attack Abstraction Layer provides functions to send and receive single bits or bytes over the channel. By using the

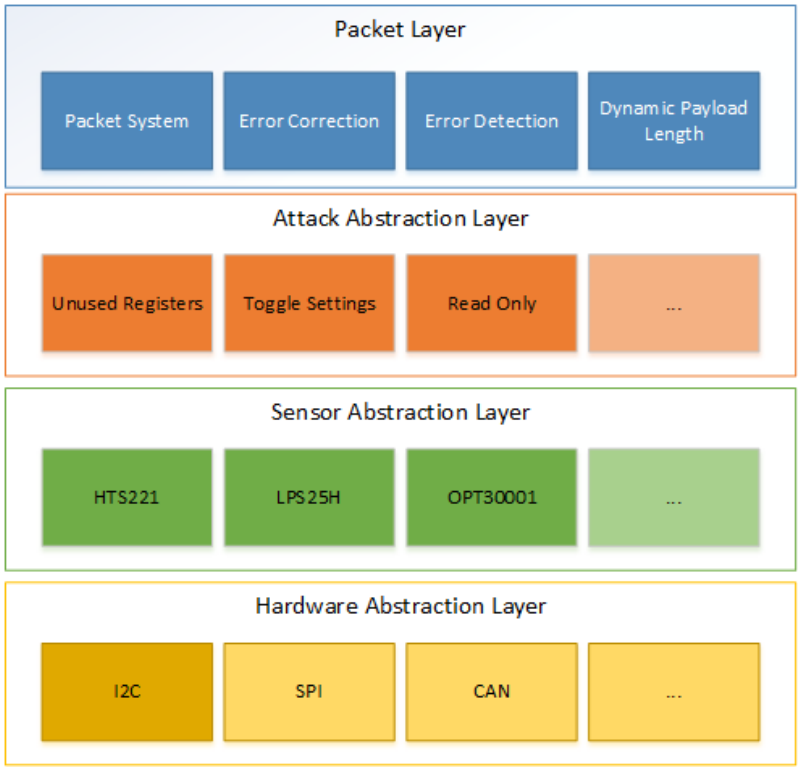


Figure 4.3: The testbed consists of four layers that abstract various parts of the system.

provided functions, the underlying attack does not matter. This core package has some drawbacks. Typically, covert channels are quite noisy, so a lot of errors occur which renders the channel unusable for data transmission in most cases. There is also no flow control, therefore the entities are required to either purely act as a sender or receiver and the flow can not be reversed. To combat these drawbacks, a fourth layer is introduced, the *Packet Layer (PaL)*. The design principles of this layer are described in Section 3.3. In short, this layer implements a packet system that is based on a request-response protocol. It also contains modules for error correction and error detection to detect errors introduced by noise. These errors are then handled by the protocol via retransmissions. Because sensors are often accessed in a fixed time interval, the Packet Layer implements a module that is able to dynamically change the size of the packets such that they are able to fit in between sensor accesses. This method was introduced in Section 3.3.3. The last element of the stack is the *Testbed Controller*. This module interacts with all four layers and is used to select a specific profile, consisting of an interface type (HAL), a sensor type (SAL), an attack-type (AAL), and additional configuration for the packet system for example. The controller is also responsible for the definition of the sender and receiver process and the initial synchronization.

4.1.2 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is the bottom layer of the stack. It is the closest to the hardware interface out of all layers. From a top-down perspective, it should provide functions that allow very basic read and write actions via a universal access using the underlying hardware interface. A header file is used to define the interface which has to be implemented by each module in the HAL. This interface defines the following two functions:

```
virtual int read(byte_t slaveAddr, byte_t regAddr,
                 unsigned int length, byte_t *data) = 0;

virtual int write(byte_t slaveAddr, byte_t regAddr,
                  unsigned int length, byte_t *data) = 0;
```

Both of these functions require the same parameters, the address of the target sensor slaveAddr, the address of the target register *regAddr*, the length of the data which should read or written and a pointer to the memory containing either the input data for a write operation or enough empty space to receive the data from a read operation. A basic overview is given in Figure 4.4

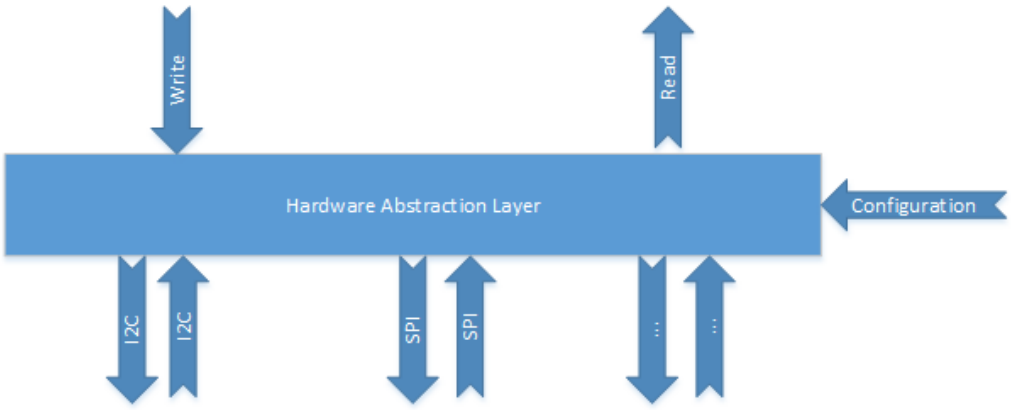


Figure 4.4: Hardware abstraction layer.

Currently, the only supported hardware interface is I2C. More information about the I2C internals and definitions are described in Appendix A.1. This section, only covers implementational details regarding the testbed. Besides the two basic functions read and write, the HAL also needs to implement all of the initialization required for the different hardware interfaces in the respective constructors. In the case of the I2C interface, this is just a simple *open()* call with the path to the desired file descriptor and a flag to define read and/or write access. This is possible because Linux implements a kernel module that maps the I2C interface to a file that is accessible in the user-space and therefore can be used by applications. In other cases, there might exist a kernel driver implementation that handles all calls to the I2C interface. This driver implementation can work in a very similar way as the dev interface by providing basic read and write functionality. As a

more universal approach the dev interface method was chosen for this thesis. The driver approach could be subject to future work. After a handle to the I2C interface is acquired, read and write operations are performed using the *ioctl* [36] import. As defined by the I2C interface, first a message is sent containing the desired operation and the target register address. Afterwards, depending on the operation, data can be received or written. For the messages the structure *i2c_msg* was used which is provided by the Linux I2C header files. Each i2c_msg contains the target address, some flags (operation type, etc.), the data length, and the data itself. To transfer the data from the input or to the output buffer a simple *memcpy* operation is used. This makes sure that only the specified amount of bytes are transmitted. All of the messages are then collected into a *i2c_rdwr_ioctl_data* array, which is passed into the *ioctl* call alongside the interface handle and the *I2C_RDRW* flag to specify a read/write operation. Additionally, to prevent larger data, the maximum thresholds *MAX_READ_LEN* and *MAX_WRITE_LEN* can be defined in the header file. In Figure 4.5 the I2C interface mapping from hardware to the user space is shown in a more visual way.

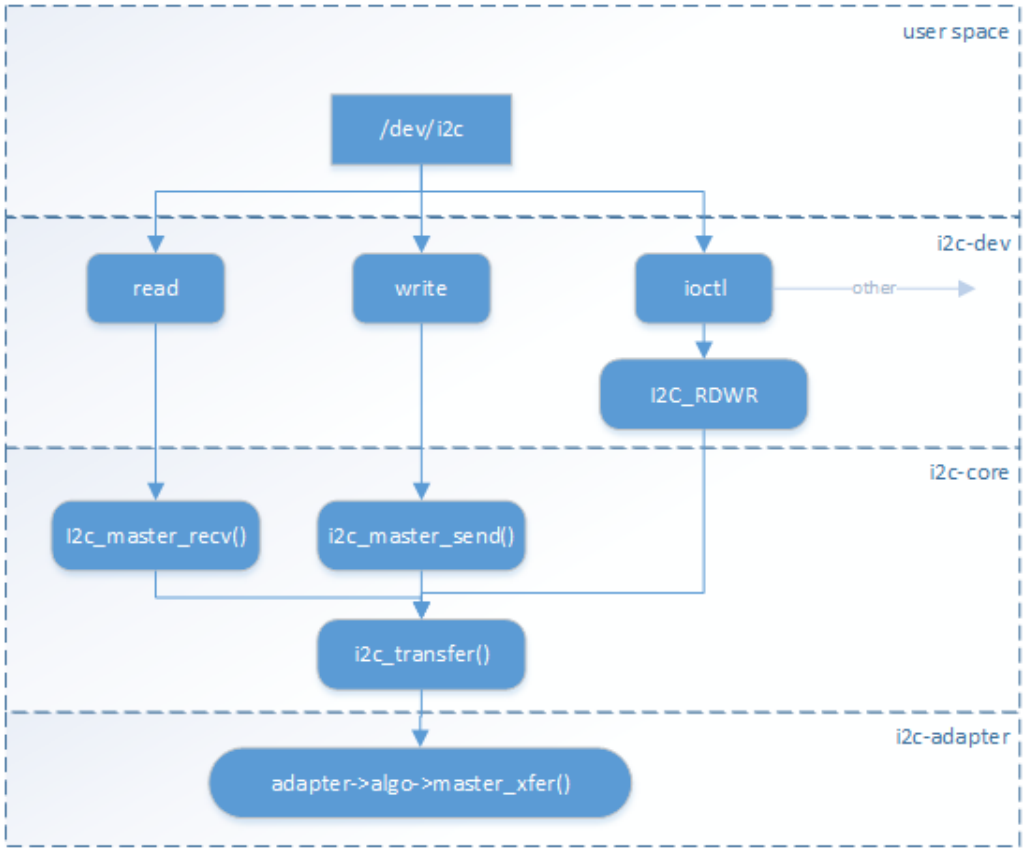


Figure 4.5: The testbed consists of four layers that abstract various parts of the system. [4]

4.1.3 Sensor Abstraction Layer

The *Sensor Abstraction Layer (SAL)* is used to abstract all of the functions specific to a sensor and has to provide standardized functions that are used by the AAL for example. Like for the other layers, a base header file is available, that defines all of the functions that need to be implemented for each sensor module. If a sensor is not able to fulfill the requirements for the implementation of one of the functions, it should return a null value instead. The following functions are available:

```
virtual int isEnabled() = 0;

virtual int enable() = 0;

virtual int disable() = 0;
```

These functions are used to enable or disable a sensor. The operation mode has to be defined globally. For all of the attacks introduced in this thesis, the same operational mode is sufficient. This mode is defined as *continuous* and maximum reporting speed. If new attack designs need special modes, an extension could be implemented that lets users define the mode when enabling a sensor for example. This is subject to future work. Further, there are some functions, that return register addresses based on a specific property:

```
virtual std::vector<int> getUnusedRegisters() = 0;

virtual std::vector<int> getSettingRegisters() = 0;

virtual std::vector<bool> getResultFlags() = 0;

virtual std::vector<int> getResultRegisters() = 0;
```

Some of these functions are very simple and just return the address of the requested register in regards to the specific sensor. Others need to implements some logic to determine the state of registers. For example, the *getUnusedRegisters* functions need to check the current settings of the sensor to determine which registers are in use. In the case of the LPS25H sensor, both the CTRL_1 register and the CTRL_3 register need to be checked if the threshold is enabled and also the reporting is enabled. If only one or none of these flags are set, the THS_P_H and THS_P_L register are "unused". Additionally, there are some functions that return more information about the sensor which is needed for some attacks:

```
virtual int getSensorCount() = 0;

virtual int getCycleTime() = 0;
```

The last two functions are used to read and write whole registers. They are very similar to the read and write functions of the HAL, but do not need any information about the hardware interface, as this is handled by the HAL:

```
virtual int readRegister(int registerAddress, int size,
                byte_t &data) = 0;

virtual int writeRegister(int registerAddress, int size,
                byte_t &data) = 0;
```

4.1.4 Attack Abstraction Layer

The *Attack Abstraction Layer* contains the main implementations for the attacks that where introduced in Section 3.1. Like with the previous layers, there is a base definition class that specifies all of the functions that need to be implemented. For each attack, a separate file is available such that implementations do not overlap. The classes need to be initialized using two parameters *EDC* and *sensor*. Sensor is an instance of the target sensors abstraction and EDC is an instance of the chosen error detection code. This is important because some size calculations are performed during data transmissions. There are two main functions related to sending and receiving data:

```
virtual int send(Packet packet) = 0;

virtual int receive(Packet &packet, int scale, bool reReceive) = 0;
```

These functions are used to send and receive packets over the covert channel. The packet parameter is a reference to the input or output data. Because the receiving entity needs to calculate the expected size of the data packet, it requires a *scale* parameter. After setting the correct payload size, the size of the error detection code is calculated and added. In hindsight, it would probably be better to handle these kinds of calculations, that are related to the packet, in the respective layer. This would loosen the bond between the packet and the attack layer which is desirable. To be able to distinguish between original messages and resent messages, a reReceive flag can be set. The next two functions are used to control the flow:

```
virtual int request(byte_t req) = 0;

virtual int waitForRequest(byte_t &sqnHad, bool reReceive,
                           bool initial) = 0;
```

Before each transmission, the sender calls the *waitForRequest* function and waits for incoming data requests. This function has a built-in timeout. If the receiver does not request a new packet for a specified amount of time, the sender starts a retransmission of the previous packet. To disable this timeout for the initial request, which could take a lot longer than normal, the function uses the *initial* flag. On the other end, the receiver calls the *request* function with a request packet number, to request a packet from the sender. The theory for this system is explained in more detail in Section 3.3. Besides these core functions, there are also some helper functions, that provide different timing methods:

```
virtual int wait(int cycles) = 0;

void waitS(int sec);

void waitMs(long ms);
```

The later two, *waitS* and *waitMs*, are static functions which are the based on real-time. They are mostly used for timeouts and small time shifts during transmissions. The *wait* function is used to initiate a wait for a specified number of cycles. The cycle time has to be defined by the sensor hardware and attack-type. For example, for the attack introduced in Section 3.1.2.3, on cycle is based on the time, the sensor needs for a full measuring cycle. This depends on the configured reporting interval and the limits of the hardware.

4.1.5 Packet Layer

In this section, the implementation of the packet layer is described in more detail. This layer consists of multiple modules which can be categorized into three parts: the error detection code (EDC) and correction (ECC), the packet structure, and the packet manager. In Figure 4.6, the data flow for the sending entity is shown. First, the packet factory is used to generate packets from the input data. These packets are then sent using the packet manager. The testbed acts as the controlling entity.

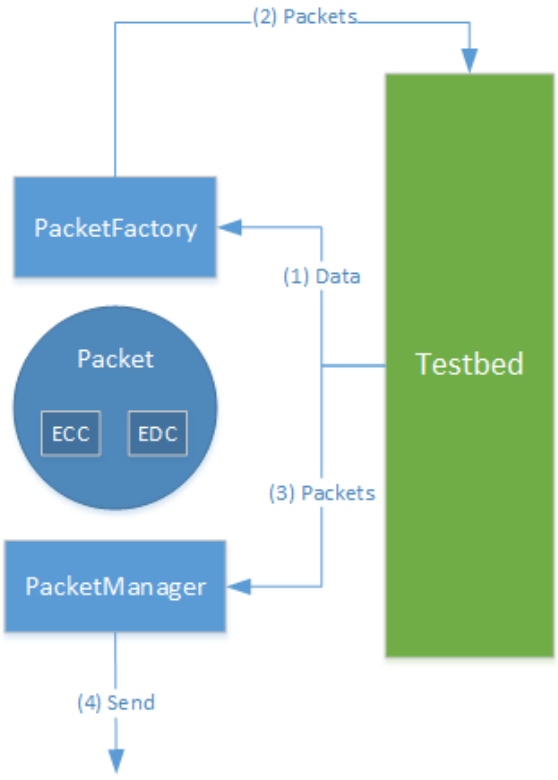


Figure 4.6: Module interactions in the packet layer for the sender.

Both EDC and ECC base classes are very similar. The main difference is, that EDC can not be decoded. As shown in Section 3.3.1, the EDC is calculated from the payload and appended to the end of the message. On receiving, the EDC is calculated again from the received payload and compared to the original EDC. If they are the same, the transmission was successful. So, for the EDC only one core function is needed:

```
virtual int generate(std::vector<bit_t> input,
                     std::vector<bit_t> &output) = 0;
```

In contrast, the ECC is generated by encoding the payload, which results in a completely new message instead of appending the code. Upon receiving an ECC, it has to be decoded. Therefore, two core functions are required:

```
virtual int encode(byte_t *input, size_t length, byte_t *output) = 0;

virtual int decode(byte_t *input, size_t length, byte_t *output) = 0;
```

For convenience, each ECC and EDC module has to implement a *check* function, that can be used to automate the validation process. The EDC version requires both, the received payload and the received EDC:

```
virtual int check(std::vector<bit_t> input,
                  std::vector<bit_t> edc_in) = 0;
```

The ECC version only requires the received payload. If it is able to decode the input to a valid message, the payload is valid:

```
virtual int check(byte_t *input, size_t length) = 0;
```

To be able to reserve the correct amount of memory for a packet, the framework needs to be able to calculate the size of the EDC and ECC. Therefore, each module has to implement the following function:

```
virtual int calcOutputSize(unsigned int inputLength) = 0;
```

Currently there is one implementation for each type of code. For the EDC, a Berger Code is used. The theory of this code is introduced in Section 3.3.1. This code is really simple to implement in C++. The main loop is implemented in one line using shift operations:

```
for (int i = 0; i < check_bits; i++) {
        output.push_back((bit_t) ((count & (1 << i)) >> i));
}
```

where *count* is the number of 1s in the binary representation of the input and *check_bits* is the number of output bits. The *check* function calculates the EDC again and compares it to the received EDC. If no EDC is required, there is also a class called *NoEDC*, which can be used to disable this module completely.

For the ECC, an Implementation of the Hadamard Code is used. Because this code is only used for very small payloads, e.g. the packet number, a static implementation is sufficient. For this, a set of lookup arrays are defined in the header file:

```
byte_t _H2[2] = {0b00, 0b01};
byte_t _H4[4] = {0b0000, 0b0011, 0b0101, 0b0110};
byte_t _H8[8] = {0b00000000, 0b00001111, 0b00110011, 0b00111100,
                 0b01010101, 0b01011010, 0b01100110, 0b01101001};
```

So, the supported input sizes are 1, 2 and 3 bit which result in a 2, 4, or 8-bit code. The encoding is a simple masking of the input, to prevent inputs that are to big, followed by a lookup. For the decoding, based on the input size, a lookup table is selected. Then, the hamming weight between the input and each of the entries in the lookup table is calculated. The entry with the lowest hamming weight distance is selected as the decoded counterpart. If the distance value is too big, too many errors were introduced and the received code cannot be decoded.

The next (pseudo) module in the packet layer is the definition of the packet itself. The structure of a packet was introduced in Section 3.3. To recap, the packet system is built upon a request-response protocol. There exist two different types of response packets: data and commands. The packet class defines some type constants:

```
static const int TYPE_DATA = 0;
static const int TYPE_CMD = 1;

static const int CMD_UP = 0;
static const int CMD_DOWN = 1;
static const int CMD_STOP = 2;
static const int CMD_REV = 3;
```

These constants are used during initialization of the packet. For each of the packet types, a separate constructor can be used by either using an array of data or an command constant:

```
Packet(std::vector<bit_t> data, int sqn, std::shared_ptr<EDC> _edc);

Packet(int cmd, int sqn, std::shared_ptr<EDC> _edc);
```

A sequence number is assigned to each packet, which is used to define the order of packets. Because this number is encoded using error correction codes, it is required to be as small as possible. In the current design a maximum of 3 bit is supported. Therefore the number can not be used as a unique identifier. If an array of data is provided to the constructor, it will create a data packet using that data as the payload. The EDC is calculated automatically. The data can be arbitrarily long, but it is advised to keep it at a sensible size, as bigger packets increase the probability of collisions and other errors. For command packets, one of the available command constants needs to be set as a parameter. These commands are used to initiate the dynamic packet scaling, introduced in Section 3.3.3 and for flow control (stop and reverse).

Another method that can be used to create data packets, is to first create an empty packet using the simple packet constructor

```
Packet(std::shared_ptr<EDC> _edc);
```

followed by calling

```
void fromBits(std::vector<bit_t> input, int scale);
```

to generate all packet fields from an input bit array. This is used, when receiving data, as this data is normally a bit array containing the entire packet. There is also a function to convert a packet to such a bit array:

```
void toBits(std::vector<bit_t> &output);
```

All other functions are either simple getter and setter or are used to perform various checks for elements, like packet type, correct sequence number, packet validity, etc. Internally, the packet holds all content parts in separate arrays:

```
std::vector<bit_t> _data_bits;
std::vector<bit_t> _sqn_bits;
std::vector<bit_t> _edc_bits;
```

Because most of the time, the data should be sent in multiple packets, a packet factory class was implemented. Using this class, all of the data can be passed in the constructor

```
PacketFactory(unsigned char *data_in, size_t length,
                              std::shared_ptr<EDC> edc);
```

and is stored internally. If more data needs to added, it can be appended afterwards using the *appendData* function. Now, all of the packets can be generated using the factory methods

```
int getNextPacket(int sqn, Packet &ret);
```

```
int getCommandPacket(int cmd, int sqn, Packet &ret);
```

The factory only generates new packets as they are requested and stores the rest of the data in a raw format. It also keeps track of which data was already sent and what data is up next. This is important, because the size of the packets can be changed on the fly using the *scaleUp* and *scaleDown* commands.

The last module in the packet layer is the packet manager. This module manages the scaling and all of the checks for the packets and implements functions that are used to unpack payloads. It uses the packet factory to generate new packets as they are required and sends and receives them using the functions provided by the attack layer. Most of its functions are just used as a proxy for lower-level functions, like *waitForRequest*, *request*, *send* and *receive*.

To manage all of the settings and modules in the different layers, the testbed class is introduced. In this class each selectable module is assigned a constant:

```
static const int ECC_HADAMARD = 1;

static const int EDC_NOEDC = 0;
static const int EDC_BERGER = 1;

static const int HAL_I2C = 0;
static const int HAL_SPI = 1;

static const int SENSOR_LPS25H = 0;
static const int SENSOR_HTS221 = 1;

static const int ATTACK_UNUSEDREG = 0;
static const int ATTACK_TOGGLESET = 1;
static const int ATTACK_READFLAGS = 2;
```

For each of the different settings, a getter and setter is defined. Additionally, this class implements the *runTest* function:

```
int runTest(bool send);
```

that is used to initiate a test. It expects a flag that toggles send/receive. For the test, different states are defined for both the sender and receiver. Figure 4.8 shows all of the states in a graph, for better visualization.

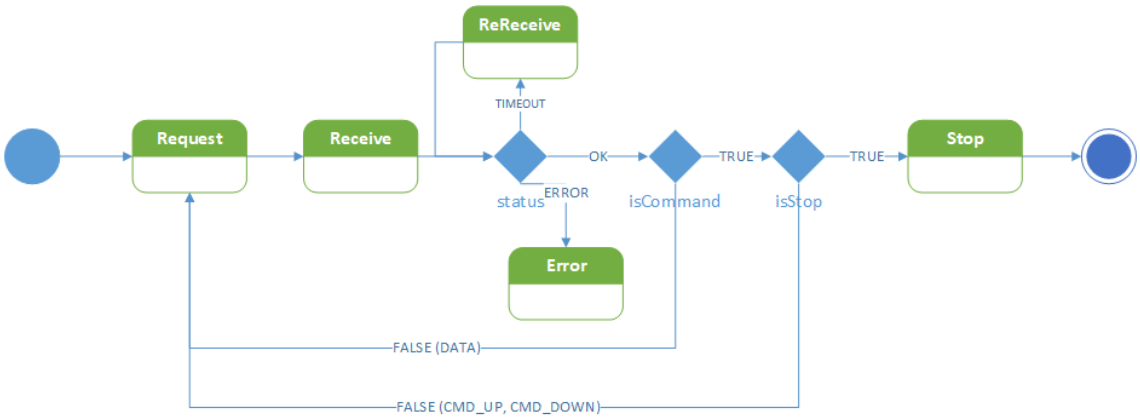


Figure 4.7: State diagram for the sender.

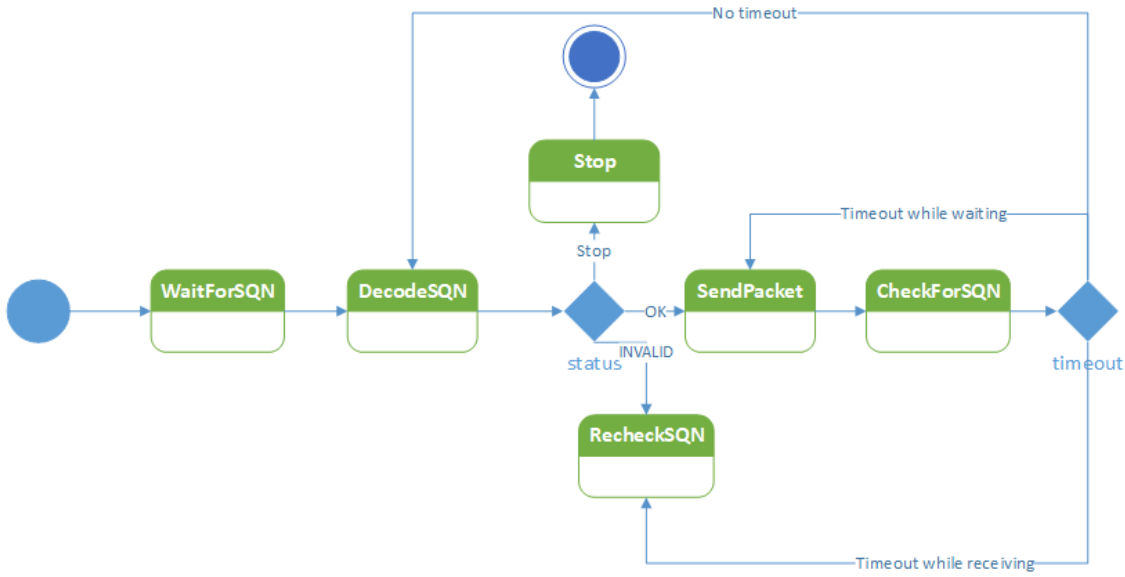


Figure 4.8: State diagram for the receiver.

4.2 Hardware

4.2.1 Texas Instruments

There are several hardware products from Texas Instruments that are relevant for this thesis. Because of there wide usage and actual availability, the products listed in Table 4.1 were chosen as good candidates for further evaluation.

Product Number	Product Name
MSP430FR5969	LaunchPad Development Kit
BOOSTXL-SENSORS	Sensors BoosterPack
TIDC-CC2650STK-SENSORTAG	SimpleLink™ multi-standard CC2650 SensorTag™ kit reference design
CC-DEVPACK-DEBUG	SimpleLink SensorTag Debugger DevPack

Table 4.1: Selected products from TI

4.2.1.1 MSP430 LaunchPad and Sensors BoosterPack

The MSP430 Launchpad was the first product that was taken into consideration, because it is relatively cheap and is often used in a wide variety of applications. For this thesis, the main requirement for the development platform was the compatibility with many different sensors. Like most microcontrollers, the MSP430 has built-in hardware support for I2C and SPI buses. These buses are the de facto standard when it comes to sensor interfaces, so this platform is compatible with most available sensors.

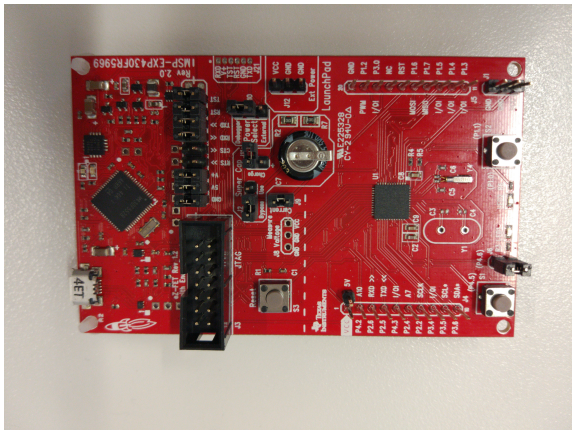


Figure 4.9: MSP430 dev board

Name	Manufacturer	"Hybrid"	Type(s)
OPT3001	TI	N	Ambient light
TMP007	TI	N	Infrared temperature
BMI160	Bosch	Y	Accelerometer, gyroscope
BMM150	Bosch	N	Magnetometer
BME280	Bosch	Y	Pressure, Ambient temperature, humidity

Table 4.2: TI BoosterPack extension board and its application

For the software development, TI offers a specialized integrated development environment (IDE) called Code Composer Studio [27]. This IDE includes a resource explorer, where users can search and download examples, templates, and documentation for TI products. To simplify application development involving sensors, Texas Instruments offers the so-called "Sensors BoosterPack" (see Table 4.1). This product is a small extension board containing 5 different sensors shown in Table 4.2. It is designed to connect with TI's Launchpads without modifications via the two 20pin headers. Although most of the sensors support I2C and SPI interfaces, the extension board only connects to their I2C interfaces. Therefore the SPI bus cannot be used. The extension board and its application can be seen in Figure 4.10.

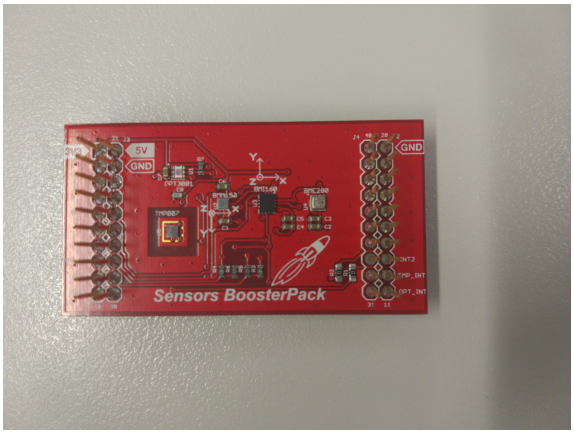
Other pins connected are:

- 3.3v power rail
- 5v power rail
- Ground
- Accelerometer interrupt
- Magnetometer interrupt
- Gyroscope interrupt
- Infrared temperature interrupt
- Ambient light interrupt

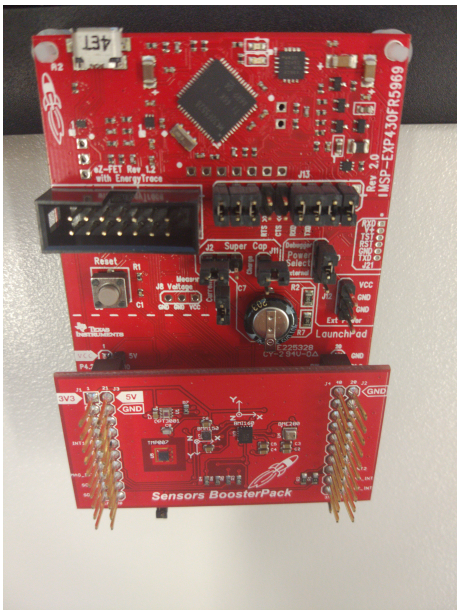
After some experimentation, it was not possible to compile RTOS for the MSP430 development board due to compilation errors. To save time, it was decided to move on, to the very similar but more recent CC2650 SensorTag. Most of the information discussed in this subsection also applies to the next subsection and vice versa. So, if the compilation would be successful, the introduced covert channel should also work on the MSP430 development board.

4.2.1.2 CC2650 SensorTag

The CC2650 SensorTag is another product from Texas Instruments. This tiny device is designed for Internet of Things applications and therefore comes with Bluetooth low energy



(a) TI Sensors BoosterPack



(b) Sensors BoosterPack installed on MSP430 Launchpad

Figure 4.10: Available sensors on the TI BoosterPack extension board

Name	Manufacturer	"Hybrid"	Type(s)
OPT3001	TI	N	Ambient light
HDC1000YPA	TI	Y	humidity, temperature
MPU-9250	TI	Y	Accelerometer, gyroscope
BME280	Bosch	Y	Pressure, Ambient temperature, humidity
SPH0641LU4H	Knowles	N	Microphone

Table 4.3: Available sensors on the CC2650 SensorTag

and 10 different sensors. Some sensors are exactly the same as on the Sensors BoosterPack, for example the ambient light sensor OPT3001 and the environmental sensor BME280. Table 4.3 lists all available sensors.

4.2.2 Raspberry Pi

The Raspberry Pi is one of the most famous single-board computers. It is available in different versions, ranging from the Zero with a very small profile to the most common standard "Pi Model B". For each Generation, a smaller incremental update is released, which can be identified by the "+" suffix. For this thesis, the Raspberry Pi 3B+ was chosen, as is was widely available and the actual computing power and memory are sufficient.

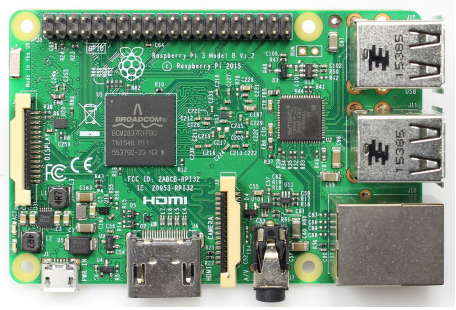


Figure 4.11: Raspberry Pi 3B+ [59]

This model comes with the following core specifications:

- Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- 1GB LPDDR2 SDRAM
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN
- Extended 40-pin GPIO header
- 4 USB 2.0 ports
- 5V/2.5A DC power input

For the operating system, Raspbian was chosen because it is a Debian version specially developed for the Raspberry, which ensures high compatibility. The installation is straight forward using the provided "NOOBS" installer [14].

To be able to test the developed code efficiently, a "Sense HAT" add-on board [45] was used. This board incorporates multiple different sensors that are already connected to an I2C interface. Additional to the hardware sensors listed in Table 4.4, a LED Matrix, a Joystick, and a small 32kbit EEPROM are included.

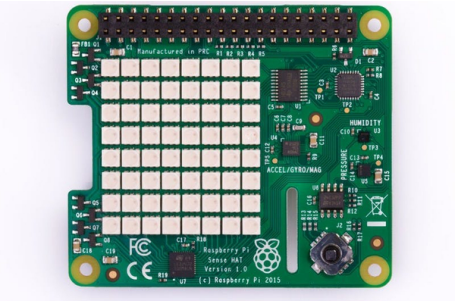


Figure 4.12: Raspberry Sense HAT [59]

Name	Manufacturer	"Hybrid"	Type(s)
HTS221	STMicroelectronics	Y	Humidity, Temperature
LPS25H	STMicroelectronics	N	Barometric Pressure
LSM9DS1	STMicroelectronics	Y	Accelerometer, Gyroscope, Magnetometer

Table 4.4: Hardware sensors on the Raspberry Sense HAT

To attach the board to the Raspberry, it simply has to be connected to the 40-pin GPIO header. All of the pins are predefined so no additional setup steps are needed. One thing to note, the Raspberry Pi 3B+ implements two I2C interfaces and before the sensors can be accessed, the correct interface has to be selected. The developers provide a python library which implements most of the sensor functionality, but does not allow direct access to the sensor. As mentioned in the theory discussions, it is good practice to restrict access, but there are cases where a user needs more control over the settings of a sensor for example. In these cases, the library may be too restrictive and other access methods can be used, like the dev interface provided by Linux. For the upcoming evaluations, the covert channel entities use the dev interface, while the entities simulating users use the python library. This is the simplest way to allow quick changes to the user behavior, while still being able to control the channel with precision from the sender and receiver side. Figure 4.13 shows an overview of the test setup.

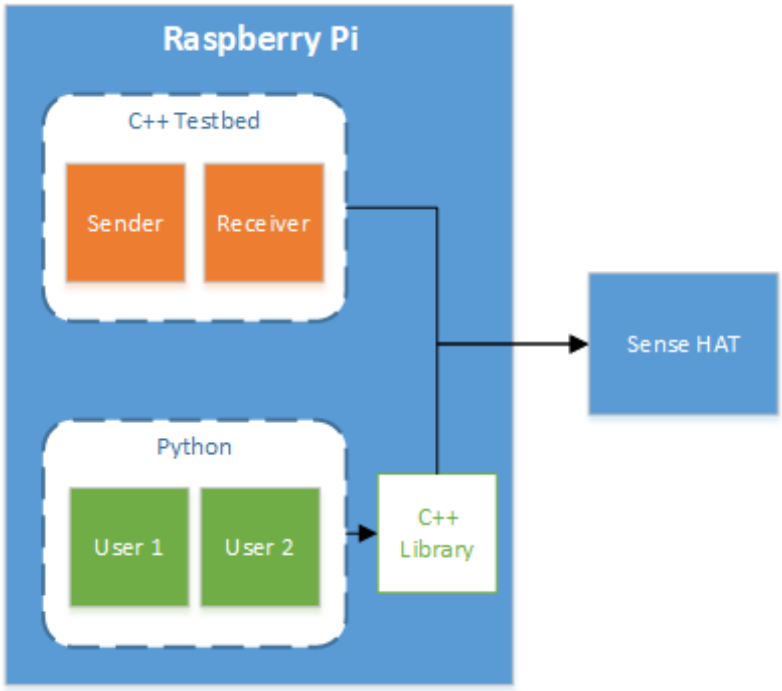


Figure 4.13: Raspberry Pi Testsetup

4.3 Evaluation

In this section, different aspects of the covert channel designs, introduced in Chapter 3, are evaluated. For the Raspberry Pi platform, the testbed developed during this thesis is used. More information about the implementation of this testbed can be found in Section 4.1. The evaluation on the CC2650 SensorTag and the Android smartphone required separate implementations, as the programming language or environment is vastly different from a standard Linux C++ environment. All three systems were evaluated in terms of covert channel throughput, which is shown in detail in Section 4.3.1. To show the purpose of the adaptive packet size, the error code detection and correction, all of these systems were evaluated using the testbed implementation. The results are shown in Section 4.3.2 and 4.3.3.

4.3.1 Throughput

The throughput is an important property of a (covert) channel. It not only can be used to calculate how long it will take to send a payload over the channel from the sender to the receiver, but also can be used to estimate the viability of the covert channel. If the throughput is very low, and the sender and/or receiver applications can only run for a short amount of time before being terminated, it is nearly impossible to send a meaningful amount of data over the channel.

For this evaluation, a payload with a known size was sent over the covert channel. The time from which the data transmission was started until it was fully received by the receiver was measured and is shown in Table 4.5. It is obvious, that the first two covert channel designs are way faster as they are only limited by the bus speed. Another simple observation is that the unused register attack is approximately seven times faster as the configuration bits attack as seven payload bits per transaction can be transmitted instead of just one.

Payload size	Channel Type	Time (min:sec)
4kb	Unused Register	0:14
4kb	Configuration Bits	1:36
4kb	Triggering Sensors	52:49

Table 4.5: Results for the transmission times using the Raspberry Pi 3B+

In Figure 4.14, a comparison between different platforms is shown for all three attack types. The third attack was only implemented using the testbed, therefore no values for the CC2650 SensorTag are available. For the Android part, aa OnePlus 5 was used to perform some throughput evaluations, which are not as accurate as the others. The 30bit/s are a rough estimate. There is a notable difference between the CC2650 and the Pi. This is, because of the synchronization needed for Linux systems. As TI-RTOS implements a predictive scheduler, the two processes that communicate over the covert channel are automatically in sync. This results in a throughput roughly two times of the Pi.

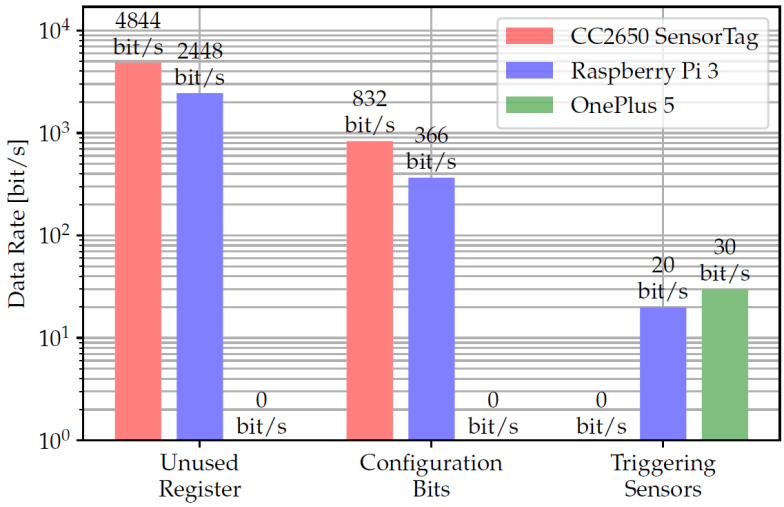


Figure 4.14: Throughput comparison between different platforms [56]

The throughput graph uses a logarithmic scale because of the big differences between the different attack methods. While the unused register attack is able to reach up to 2448bit/s, the configuration bits attack reaches about a seventh, or 366bit/s. The triggering sensors attack is by far the slowest with a throughput of 20bit/s. This value highly depends on the speed of the sensor, as only one bit can be sent per measurement cycle.

4.3.2 Adaptive Packet Size

To be able to respond to errors introduced by concurrent user accesses, the packet size was designed to be adaptive. For the evaluation, a payload with a fixed size was transmitted using different packet sizes. Because user access times are related to the number of retransmissions, the test was performed using different time values ranging from 1s to 20s. The packet sizes 11, 20, 37 and 70bit were chosen because of the Berger error detection code size thresholds shown in Table 4.6. The results show a steady downward trend for the transfer time when the packet size is increased. The drawback is visible in the 1s access time bars in Figure 4.16. The time values increase significantly for packets of 37bits and above. Even bigger packets with 70+bits, have trouble sending even a single packet which leads to a much higher retransmission count and thus, transfer time.

EDC size	Payload size (max)	Packet size (max)
3bit	5bit	11bit
4bit	13bit	20bit
5bit	29bit	37bit
6bit	61bit	70bit

Table 4.6: Resulting maximum packet sizes based on berger code size

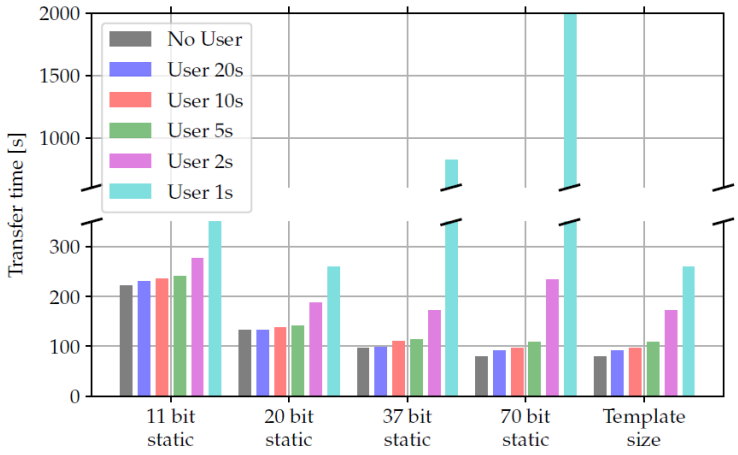


Figure 4.15: Transfer time differences for various packet sizes and user access timings [56]

For the evaluation of the dynamic packet size, three tests were performed. Two of the tests used a static packet size of 37bit and 11bit and the third test used a dynamic size. 30 seconds into each test, a user starts polling the sensor in a 1-second interval. As shown in Figure 4.16, the 37bit static test failed to transmit any data, as soon as the user starts polling. The 11bit static struggles a bit, but is able to continue the transmission. Because an 11bit packet contains only 5bit of data, approximately 55% are overhead in contrast to 22% for the 37bit packet. Therefore, the smaller packets need to transfer more, which slows down the transmission. The dynamic packet approach combines the best of both approaches, i.e., the higher throughput of larger packets and the ability of smaller packets to transmit data when a lot of noise is generated by the user. When the user starts polling after 30 seconds, the test-bed recognizes the failed transmissions and starts to scale down the packet size. After 60 seconds, the packets are small enough to be successfully transmitted again.

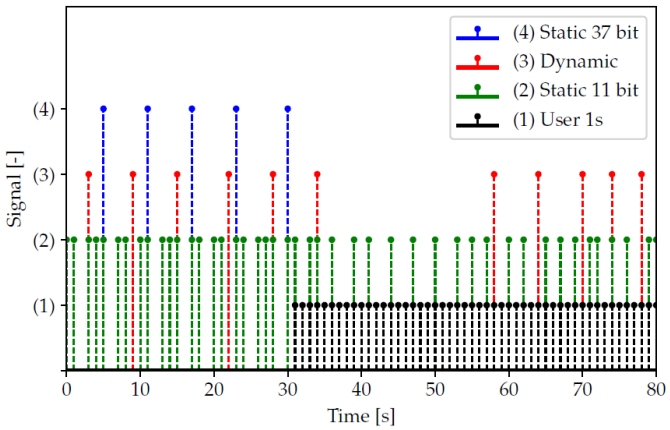
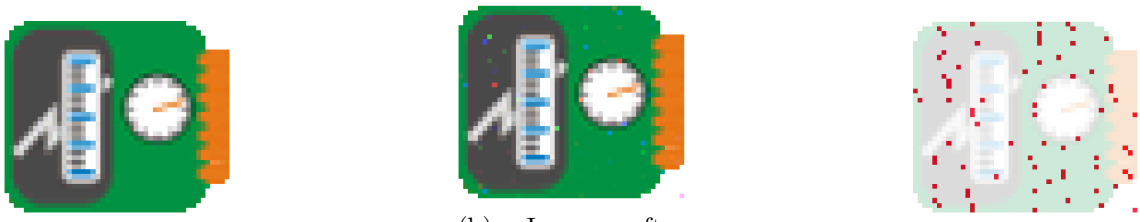


Figure 4.16: Comparison between static and dynamic packets sizes with noise [56]

4.3.3 Error Correction & Error Detection

In Section 3.3.2 and 3.3.1 error correction and error detection were introduced to the packet system. The codes are used to detect errors, which are then handled by the packet systems retransmissions. If too many retransmission occurs, the transmission is aborted, as the channel is too noisy to transmit data reliably. For the evaluation, a small 32x32 image was transmitted over a covert channel. The original image is shown in Figure 4.17a. Noise is generated using two users who access the sensor at random times. Both the error detection and the error correction are disabled, which means bit-flips in the payload are not detected. If a bit-flip occurs in the part where the sequence number is stored, the packet system still performs a retransmission because of the out of order sequence number. Figure 4.17c shows the difference between the input and the output image. A lot of pixel errors were introduced by the user interferences, but the image is still recognizable.



(a) Original image

(b) Image after transmission

(c) Pixel errors

Figure 4.17: Errors resulting in a transmission without EDC and ECC enabled [56]

For command packets on the other hand, the problems are a lot more noticeable. Because command packets are not sent often and are very short in relative, the error count is a lot lower as for data packets. The problem arises if a bit-flip occurs in the command part of the command packet, which leads to misinterpretation by the receiver. As both parties need to keep track of the packet scale independently, these scales have to be in sync for any communication to be possible. If the command is misinterpreted by the receiver, the scaling will drift and is no longer in sync. Currently there is no algorithm implemented to detect scale drifts, so if this is the case, the communication is not possible anymore.

4.3.4 Managed-Access

The following part outlines the evaluation for both Android-based covert channels, which were introduced in Section 3.2. Two separate applications were developed, one receiver and one sender. Upon start, the receiver registers a callback for a sensor event using a very high interval time. This ensures that other applications are able to lower the interval time when registering for the same sensor event. The sender implements a simple UI with one button. When the button is pressed, the sender starts the transmission of the payload using the specified covert channel. Each time the callback of the receiver is called, both the sensor event timestamp and the current system time are logged. The log is saved to the device and later transferred to a computer for post-processing. As already mentioned

in the design section, the system timestamp can only be used as an internal reference value as it is highly fluctuating based on system load.

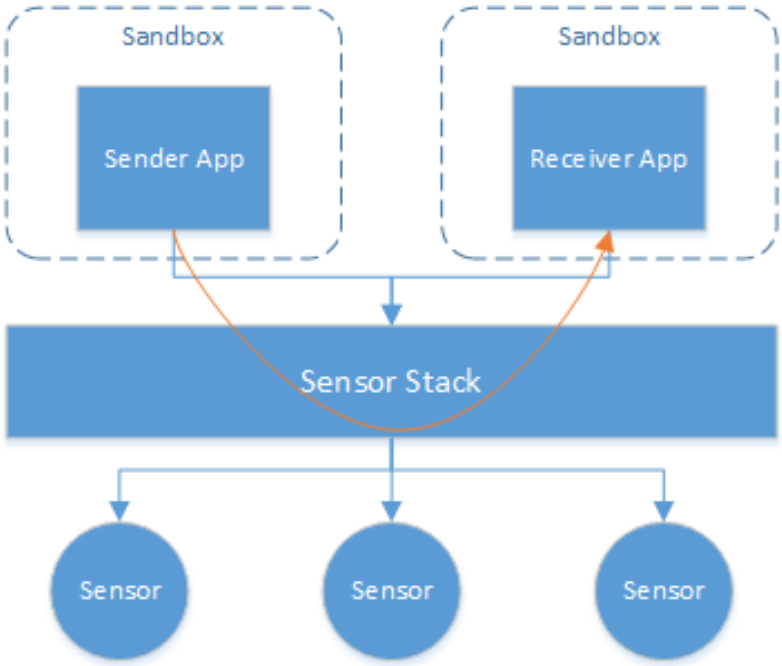


Figure 4.18: Android test setup

In post-processing, the interval between the timestamps is calculated. The respective graph can be seen in Figure 4.19. For this test, an interval value of 20ms is used to transmit a 0 bit and a value of 40ms is used for a 1 bit. The transmission is only slightly susceptible to system load as the sensor timestamps are used instead of the system time. These sensor timestamps are generated when the result is read from the sensor by the sensor stacks' C++ implementation, which runs in near real-time.
In the center of the graph in Figure 4.19, after the transmission of the second 1 bit, a slight interference is visible. If these interferences disturb the decoding process, multiple adjustments are possible. For example, the distance between the intervals could be increased to generate a more aggressive pattern (5ms and 100ms). Another parameter that could be changed is the number of events per payload bit, which would increase the possibility for the channel to stabilize between interval changes.

While performing the evaluation of the first covert channel design, it was discovered that all (or most) browsers immediately subscribe to sensor events using the fastest possible reporting speed. This renders the previous channel infeasible, as no further changes are possible until the browser is closed. After some experimentation, a workaround was found which ended in the second covert channel design, introduced in Section 3.2. Figure 4.20 shows the transmission of another test payload using the new design. The blue line represents the interval timings logged by the receiver each time the callback is called. The

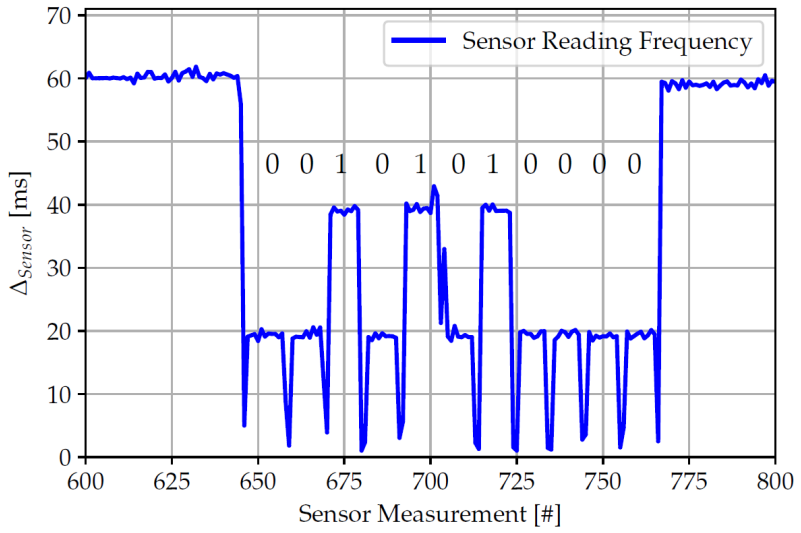


Figure 4.19: Received transmission of a test patter over the covert channel [56]

red dots are timestamps logged by the sender, each time it registers a new listener for the sensor events. It can be seen that both logs correlate and each time a new listener is registered, an anomaly in the intervals can be detected. Using different delays between subscriptions, data can be encoded, e.g., 10 intervals for a 0 bit and 50 intervals for a 1 bit. This channel design highly depends on the used hardware, is very unstable and therefore only presented as a concept.

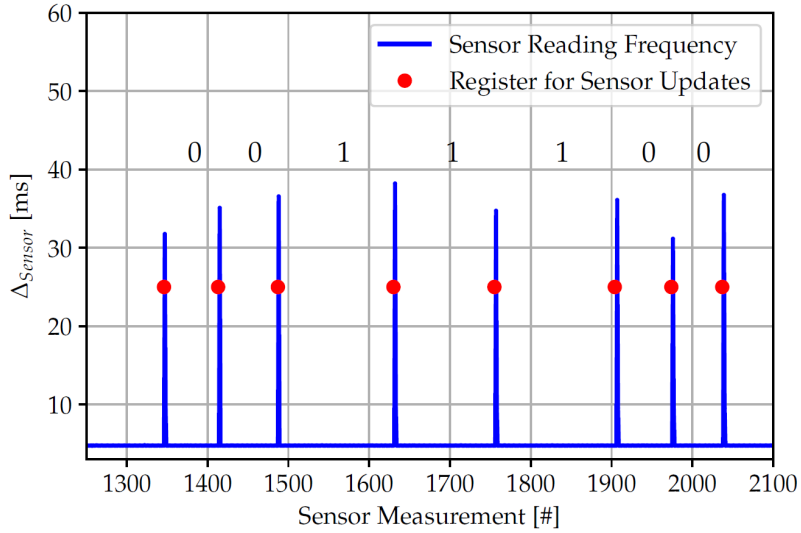


Figure 4.20: Correlation between sensor reading frequency and registering a sensor event listener [56]

Chapter 5

Conclusion and Future Work

In this chapter, first a summary of the contributions is given in Section 5.1, followed by a few selected topics that are subject to future work in Section 5.2.

5.1 Conclusion

This thesis introduced some new designs for sensor-based covert channels, ranging from simple attacks targeting sensor registers to more sophisticated attacks, that are able to abuse even abstracted architectures like Androids' sensor stack. One of the biggest problems in designing covert channels is the high susceptibility to noise. In the case of sensor-based covert channels, each user access introduces noise which has to be detected to mitigate errors. Therefore, a packet system was introduced which includes a request-response protocol for packet transmission. Each packet integrates either an error correction- or an error detection code, depending on the payload size. The implemented request-response protocol ensures the correct order of the packets and using this method, data can be transmitted even over noisy channels. A testbed was designed and implemented that follows a layered abstraction architecture. Each of the four layers abstracts one part of the system: the hardware interface, the sensor, the attack implementation, and the communication flow. This enables the possibility for unified testing of an attack on different target sensors. The layers consist of separate modules, which are interchangeable and extendable. A management class is used to configure the selected modules for each layer. Using the testbed, it was shown that a covert channel between two processes can be successfully established using the proposed methods. The data rate ranged from 4,8kbit/s to 20bit/s depending on the complexity of the design. Besides a throughput comparison between the designs, also two different platforms were evaluated. A Raspberry Pi 3B+ running a Linux distribution was used as a baseline and for tests on a low power platform, the CC2650 SensorTag was used. The evaluation concluded that very simple designs are able to run at nearly the data rate limit of the hardware interface, while more sophisticated and less obvious attacks come with a much lower throughput.

5.2 Future Work

In this section, some of the ideas for future work are explained.

Porting to other RTOSs: For evaluation purposes, the CC2650 SensorTag running the Texas Instruments TI-RTOS was used. As explained in Section 3.1.1.1, there are a few other popular operating systems available that support the CC2650. As these operating systems can be vastly different in relevant parts of the implementation such as scheduling, hardware abstraction, etc., it would be interesting to test the proposed covered channel designs on these operating systems.

Support for additional Hardware Interfaces: For the most part of this thesis, only I2C was considered. As nearly every sensor supports both I2C and SPI, the next logical step would be to implement the SPI specification to be able to compare it to I2C in terms of throughput and viability. Further, there are other systems such as CAN bus or Ethernet-based access, that add an additional layer which would be interesting to investigate. CAN bus systems are often used in vehicles, which would make a good target.

Support for additional Error-Detection- And Error-Correction-Codes: Besides the already considered algorithms, there are many more additional error-detection- and error-correction-codes which are more efficient in terms of generated overhead. As they are way more sophisticated, the implementation of those codes is also a lot more work. For this thesis it was important to just implement a rudimentary version of each code to be able to evaluate the channel designs reliably. In the future it will be interesting to compare the efficiency of the implementation to other codes, like turbo-codes for example.

Templating implementation: Most accesses to sensors are either very sporadic or follow a specific pattern. For example a process might poll a temperature sensor once a day or it might request new accelerometer readings every 10ms. By listening to the usage of the sensor the attacker might be able to recognize the pattern. This enables new methods for packet scaling and timeouts.

iOS Devices: Apples' iOS is the mobile operating system with the second-highest market share worldwide, at around 28.8%. It is very different from Android and it would be interesting to research how sensors are handled on iOS. There are some points, that make the development for iOS more complicated than for Android. Apple decided to keep iOS development exclusive to its own ecosystem, therefore a computer running macOS and Xcode is needed. This is why the focus of this thesis was on Android and iOS will be subject to future work.

Appendix A

Appendix

A.1 I2C

I2C was developed by NXP Semiconductors in 1982 [50]. Nowadays, it is the de facto standard for communication between ICs. The bus consists of only two wires and a very simple circuit which is important for the integration in small low power hardware, such as sensors. One of the two wires is used as a serial clock line (SCL) while the other wire, the serial data line (SDA), is used to transmit data. For each bus multiple devices can be connected which are enumerated using software-defined addresses. The bus communication follows a master/slave principle. A basic example is shown in Figure A.1 where a Raspberry Pi (master) can access multiple different sensors (slaves).

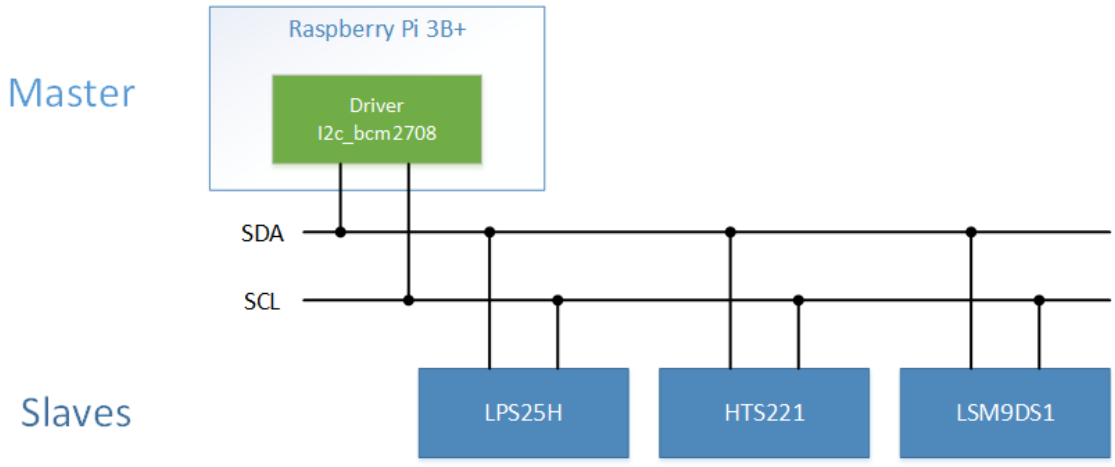


Figure A.1: Example I2C setup

To increase compatability, the I2C standard defines different bidirectional modes that determine the maximum throughput:

- **Standard-mode:** up to 100kbit/s
- **Fast-mode:** up to 400kbit/s
- **Fast-mode Plus:** up to 1Mbit/s
- **High-speed mode:** up to 3.4Mbit/s

The specification also defines multi-master support, which is not relevant in this thesis as its focus is inter-process side channels, which does not entail inter-device side channels. All of the communications have to be initiated by the master, while the slaves should only send a response when the master request data from them. At the beginning of each transaction, the master sends a START condition which is defined by a HIGH to LOW transition on the SDA line while SCL is HIGH. Then, data is sent by using either a HIGH or LOW state on the SDA line while the SCL is HIGH. It is important, that transitions on the SDA line are only allowed while the SCL is LOW. Otherwise, the data is invalid. For each clock cycle one bit of data can be transferred. At the end of a transaction the master sends a STOP condition to signal the bus availability. This is very important for multi-master setups. In Figure A.2 two example transfers, including the START and END conditions, are shown.

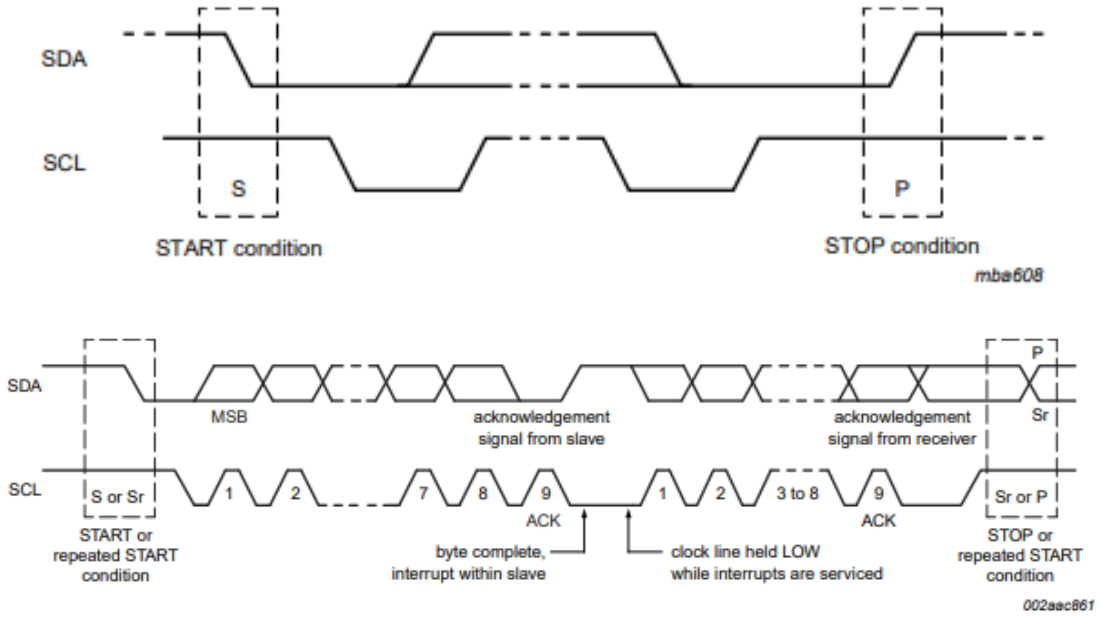
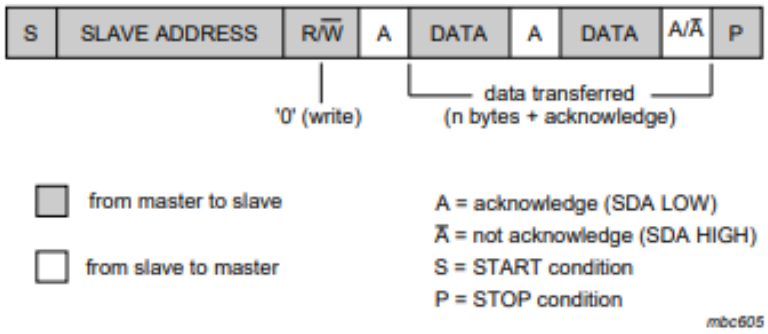


Figure A.2: I2C START and STOP conditions [50]

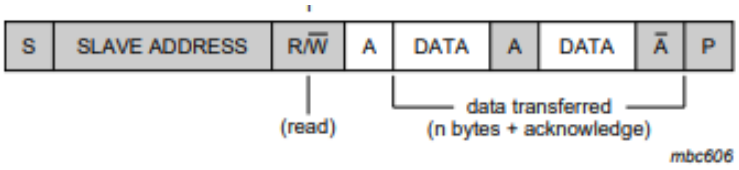
After every byte the receiver has to send an acknowledge (ACK) to let the sender know if the transmission was successful. This is defined as follows: "The transmitter releases the

SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse." If the SDA line stays HIGH it will be interpreted as a not acknowledge (NACK), which leads to either a restart of the transaction or it is aborted by the master.

For the transmission, there are three defined data transfer formats. Figure A.3a shows the structure of a write transaction. First, the master initiates the transmission by sending a START condition followed by the slave address and the read-write flag set to 0. After the ACK is received the master starts sending bytes of data which each has to be acknowledged by the receiver. After the last byte is sent successfully, the master terminates the connection by sending a STOP condition. The read transaction is very similar and is shown in Figure A.3b. Instead of 0, the read-write flag is set to 1 and after the ACK, the slave starts sending bytes until the request is fulfilled. The third data transfer format is a combination of read and write.



(a) Write transaction



(b) Read transaction

Figure A.3: Structure of an I2C transaction [50]

Because most slaves have multiple data registers that can be accessed, the read-only transfer is rarely used. Instead the combined format is used to first send the register address in write mode followed by receiving the requested data in read mode. Some examples for the LPS25H are shown in Table A.1 to A.4, where the slave address is shortened to SAD and the register address is shortened to SUB. The other abbreviations are defined in Figure A.3a.

Master	S	SAD	W		SUB		DATA		P
Slave				A		A		A	

Table A.1: Transfer when master is writing one byte to slave [46]

Master	S	SAD	W		SUB		DATA		DATA		P
Slave				A		A		A		A	

Table A.2: Transfer when master is writing multiple bytes to slave [46]

Master	S	SAD	W		SUB		S	SAD	R			NA	P
Slave				A		A				A	DATA		

Table A.3: Transfer when master is receiving (reading) one byte of data from slave [46]

Master	S	SAD	W		SUB		S	SAD	R			A		NA	P
Slave				A		A				A	DATA		DATA		

Table A.4: Transfer when master is receiving (reading) multiple bytes of data from slave [46]

Bibliography

[1] Kamran Ahsan and Deepa Kundur. "Practical data hiding in TCP/IP". In: *Proc. Workshop on Multimedia Security at ACM Multimedia*. Vol. 2. 7. 2002, pp. 1–8.

[2] Adam J Aviv et al. "Practicality of accelerometer side channels on smartphones". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, pp. 41–50.

[3] J.M. Berger. "A note on error detection codes for asymmetric channels". In: *Information and Control* 4.1 (1961), pp. 68–73. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(61)80037-5. URL: http://www.sciencedirect.com/science/article/pii/S0019995861800375.

[4] Bootlin. *Elixir Cross Referencer - Linux Source, I2C Driver*. 2020. URL: https://elixir.bootlin.com/linux/latest/source/drivers/i2c (visited on 2020).

[5] Iliya Bouyukliev and David B Jaffe. "Optimal binary linear codes of dimension at most seven". In: *Discrete Mathematics* 226.1-3 (2001), pp. 51–70.

[6] Thomas Byrne. *Driver for the LPS25H Air Pressure / Temperature Sensor*. 2020. URL: https://github.com/ersatzavian/LPS25H (visited on 2020).

[7] Qingping Chi et al. "A reconfigurable smart sensor interface for industrial WSN in IoT environment". In: *IEEE transactions on industrial informatics* 10.2 (2014), pp. 1417–1425.

[8] Contiki. *The Contiki Operating System*. 2020. URL: http://www.contiki-os.org/ (visited on 2020).

[9] *Crosstool-NG*. 2020. URL: https://crosstool-ng.github.io/ (visited on 2020).

[10] Manik Lal Das. "Two-factor user authentication in wireless sensor networks". In: *IEEE transactions on wireless communications* 8.3 (2009), pp. 1086–1090.

[11] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. "Modeling cyber–physical systems". In: *Proceedings of the IEEE* 100.1 (2011), pp. 13–28.

[12] Luke Deshotels. "Inaudible sound as a covert channel in mobile devices". In: *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)*. 2014.

[13] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the eighth symposium on usable privacy and security*. 2012, pp. 1–14.

[14] RASPBERRY PI FOUNDATION. *NOOBS – New Out Of the Box Software*. 2020. URL: https://www.raspberrypi.org/downloads/noobs/ (visited on 2020).

[15] Charles V. Freiman. "Optimal Error Detection Codes for Completely Asymmetric Binary Channels". In: *Information and Control* 5 (1962), pp. 64–71.

[16] Lilia Frikha, Zouheir Trabelsi, and Wassim El-Hajj. "Implementation of a Covert Channel in the 802.11 Header". In: *2008 International Wireless Communications and Mobile Computing Conference*. IEEE. 2008, pp. 594–599.

[17] Daniel Genkin et al. "ECDSA key extraction from mobile devices via nonintrusive physical side channels". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1626–1638.

[18] C. Gray Girling. "Covert Channels in LAN". In: *IEEE Transactions on software engineering* 2 (1987), pp. 292–296.

[19] Google. *Android Developer Guide - Permissions overview*. 2020. URL: `https://developer.android.com/guide/topics/permissions/overview` (visited on 2020).

[20] Google. *Android Developer Guide - Sensors Overview*. 2020. URL: `https://developer.android.com/guide/topics/sensors/sensors_overview` (visited on 2020).

[21] Google. *Android Things*. 2020. URL: `https://developer.android.com/things` (visited on 2020).

[22] Google. *Application Sandbox*. 2020. URL: `https://source.android.com/security/app-sandbox` (visited on 2020).

[23] Daniel Gruss et al. "Flush+ Flush: a fast and stealthy cache attack". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 279–299.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache games–bringing access-based cache attacks on AES to practice". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 490–505.

[25] University of Southern California Information Sciences Institute. *INTERNET PROTOCOL - PROTOCOL SPECIFICATION*. 1981. URL: `https://tools.ietf.org/html/rfc791` (visited on 2020).

[26] Texas Instruments. 2020. URL: `http://www.ti.com/` (visited on 2020).

[27] Texas Instruments. *Code Composer Studio (CCS) Integrated Development Environment (IDE)*. 2020. URL: `http://www.ti.com/tool/ccstudio` (visited on 2020).

[28] Texas Instruments. *OPT3001 Ambient Light Sensor (ALS)*. 2014/2017. URL: `http://www.ti.com/lit/ds/sbos681c/sbos681c.pdf` (visited on 2020).

[29] Texas Instruments. *TI-RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU)*. 2020. URL: `http://www.ti.com/tool/TI-RTOS-MCU` (visited on 2020).

[30] Jetbrains. *CLion*. 2020. URL: `https://www.jetbrains.com/de-de/clion/` (visited on 2020).

[31] Chris Karlof and David Wagner. "Secure routing in wireless sensor networks: Attacks and countermeasures". In: *Ad hoc networks* 1.2-3 (2003), pp. 293–315.

[32] Chong Hee Kim and Jean-Jacques Quisquater. "How can we overcome both side channel analysis and fault attacks on RSA-CRT?" In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. IEEE. 2007, pp. 21–29.

[33] Tim Kosse. *FileZilla® - the free FTP solution*. 2020. URL: `https://filezilla-project.org/` (visited on 2020).

[34] Cheolhyeon Kwon, Weiyi Liu, and Inseok Hwang. "Security analysis for cyber-physical systems against stealthy deception attacks". In: *2013 American control conference*. IEEE. 2013, pp. 3344–3349.

[35] Butler W Lampson. "A note on the confinement problem". In: *Communications of the ACM* 16.10 (1973), pp. 613–615.

[36] Linux. *Linux Manual - IOCTL(2)*. 2020. URL: `http://man7.org/linux/man-pages/man2/ioctl.2.html` (visited on 2020).

[37] Jake Longo et al. "SoC it to EM: electromagnetic side-channel attacks on a complex system-on-chip". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2015, pp. 620–640.

[38] Real Time Engineers Ltd. *FreeRTOS^{TM}: Real-time operating system for microcontrollers*. 2020. URL: `https://www.freertos.org/` (visited on 2020).

[39] Chao Luo et al. "Side-channel power analysis of a GPU AES implementation". In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE. 2015, pp. 281–288.

[40] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Vol. 16. Elsevier, 1977.

[41] Clémentine Maurice et al. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. Vol. 17. 2017, pp. 8–11.

[42] Anh Nguyen et al. "A lightweight and inexpensive in-ear sensing system for automatic whole-night sleep stage monitoring". In: *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. 2016, pp. 230–244.

[43] Peter Pessl et al. "{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 565–581.

[44] Astro Pi. *RTIMULib - a versatile C++ and Python 9-dof, 10-dof and 11-dof IMU library*. 2020. URL: `https://github.com/RPi-Distro/RTIMULib` (visited on 2020).

[45] Astro Pi. *Sense HAT*. 2020. URL: `https://github.com/astro-pi/python-sense-hat` (visited on 2020).

[46] Pololu. *LPS25H Pressure/Altitude Sensor Carrier with Voltage Regulator*. 2020. URL: `https://www.pololu.com/product/2724` (visited on 2020).

[47] Qualcomm. *Snapdragon 865 5G Mobile Platform*. 2020. URL: `https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform` (visited on 2020).

[48] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. "Security and privacy challenges in industrial internet of things". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6.

[49] Helpnet Security. *Number of connected devices reached 22 billion, where is the revenue?* 2020. URL: https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/ (visited on 2020).

[50] NXP Semiconductors. *UM10204: I2C-bus specification and user manual*. 1982/2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (visited on 2020).

[51] Jaydip Sen. "A survey on wireless sensor network security". In: *arXiv preprint arXiv:1011.1529* (2010).

[52] Raphael Spreitzer. "Pin skimming: Exploiting the ambient-light sensor in mobile devices". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014, pp. 51–62.

[53] Animesh Srivastava et al. "CamForensics: Understanding Visual Privacy Leaks in the Wild". In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. SenSys '17. Delft, Netherlands: Association for Computing Machinery, 2017. ISBN: 9781450354592. DOI: 10.1145/3131672.3131683. URL: https://doi.org/10.1145/3131672.3131683.

[54] Statcounter. *Mobile Operating System Market Share Worldwide: Q1 2016 - Q1 2020*. 2020. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide/%5C#quarterly-201601-202001 (visited on 2020).

[55] Hui Suo et al. "Security in the internet of things: a review". In: *2012 international conference on computer science and electronics engineering*. Vol. 3. IEEE. 2012, pp. 648–651.

[56] Thomas Ulz et al. "Sensing Danger: Exploiting Sensors to Build Covert Channels". In: *ICISSP*. 2019.

[57] VMware. *VMware vSphere - Memory Sharing*. 2020. URL: https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.resmgmt.doc/GUID-FEAC3A43-C57E-49A2-8303-B06DBC9054C5.html (visited on 2020).

[58] Yong Wang, Garhan Attebury, and Byrav Ramamurthy. "A survey of security issues in wireless sensor networks". In: (2006).

[59] Wikipedia. *Raspberry Pi*. 2020. URL: https://de.wikipedia.org/wiki/Raspberry_Pi (visited on 2020).

[60] Jidong Xiao et al. "Security implications of memory deduplication in a virtualized environment". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2013, pp. 1–12.

[61] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[62] Xun Yi et al. "Privacy protection for wireless medical sensor data". In: *IEEE transactions on dependable and secure computing* 13.3 (2015), pp. 369–380.

[63] Younis A Younis et al. "A new prime and probe cache side-channel attack for cloud computing". In: *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE. 2015, pp. 1718–1724.

[64] Zephyr. 2020. URL: `https://www.zephyrproject.org/` (visited on 2020).