JOHANNES KEPLER
UNIVERSITY LINZ

Author
Joel Klimont, BSc
11923613

Submission
Institute of
Networks and Security

Thesis Supervisor
Assoz. Univ.-Prof. DI Mag. Dr.
Michael Sonntag

February 2024

Design and Implementation of a Data Recording System for Court-Admissible Forensic Evidence

Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

Critical infrastructure often goes unnoticed in everyday life. This is because we take electricity, water, and heating for granted these days. However, to ensure the stable operation of such utility services, regular inspections are required. Currently, these inspections are mostly performed by humans, who walk through the supply shafts and manually check for damages. The INFRASPEC project aims to automate parts of these tedious inspections. The goal is to create a robot that can assist the inspectors, by creating a detailed 3D map of the environment and detecting changes across scans from different inspections. In this master's thesis, a system is developed that can collect information about the robot's operational status and store that information in a "forensic data package". Security measures should ensure that bad actors cannot change data in the package without detection. The package also acts as a way to verify that an inspection has taken place and that it was performed correctly. To build such a forensic data package, YubiKeys and hash-chains are used. Each forensic data collection device has a YubiKey with a public-private key pair on it, which can be used to sign data, which guarantees the authenticity of the measurement. To ensure the integrity of the data, hash-chains are used. After the inspection is finished, the private keys on the YubiKeys are overwritten, and no new data can be added to the project, as the keys for signing new data entries are gone.

Users might want to further process and analyze the recorded data. As possibly vast amounts of data have been collected, users might want to offload the processing task into the cloud. However, as the data packages can contain confidential information about critical infrastructure, the cloud provider should not be able to gain access to the data. To still be able to process the data in the cloud, a demonstrator application has been developed that processes the forensic data in the cloud using the Intel Secure Guard Extension (SGX) technology.

Kurzfassung

Kritische Infrastrukturen bleiben im Alltag oft unbemerkt. Das liegt daran, dass wir Strom, Wasser und Heizung heutzutage als selbstverständlich ansehen. Um den stabilen Betrieb solcher Versorgungsdienste zu gewährleisten, sind jedoch regelmäßige Inspektionen erforderlich. Derzeit werden diese Inspektionen meist von Menschen durchgeführt, die durch die Versorgungsschächte gehen und manuell auf Schäden prüfen. Das Projekt INFRASPEC will einen Teil dieser mühsamen Inspektionen automatisieren. Ziel ist es, einen Roboter zu entwickeln, der die Inspektoren unterstützt, indem er eine detaillierte 3D-Karte der Umgebung erstellt und Veränderungen in den Scans verschiedener Inspektionen erkennt. In dieser Masterarbeit wird ein System entwickelt, das Informationen über den Betriebszustand des Roboters sammeln und in einem "forensischen Datenpaket" speichern kann. Sicherheitsmaßnahmen sollen sicherstellen, dass böswillige Akteure die Daten in diesem Paket nicht unbemerkt verändern können. Das Paket dient auch dazu, zu überprüfen, ob eine Inspektion stattgefunden hat und ob sie korrekt durchgeführt wurde. Um ein solches forensisches Datenpaket zu erstellen, werden YubiKeys und Hash-Chains verwendet. Jedes Gerät zur forensischen Datenerfassung verfügt über einen YubiKey mit einem öffentlich-privaten Schlüsselpaar, mit dem Daten signiert werden können, was die Authentizität der Messung garantiert. Um die Integrität der Daten zu gewährleisten, werden Hash-Chains verwendet. Nach Abschluss der Prüfung werden die privaten Schlüssel auf den YubiKeys überschrieben, und es können keine neuen Daten zum Projekt hinzugefügt werden, da die Schlüssel zum Signieren neuer Dateneinträge nicht mehr vorhanden sind.

Die Nutzer möchten die aufgezeichneten Daten möglicherweise weiterverarbeiten und analysieren. Da möglicherweise große Datenmengen gesammelt wurden, möchten die Nutzer die Verarbeitungsaufgaben möglicherweise in die Cloud verlagern. Da die Datenpakete jedoch vertrauliche Informationen über kritische Infrastrukturen enthalten können, sollte der Cloud-Anbieter nicht in der Lage sein, auf die Daten zuzugreifen. Um die Daten dennoch in der Cloud verarbeiten zu können, wurde eine Demonstrationsanwendung entwickelt, die die forensischen Daten in der Cloud unter Verwendung der Intel Secure Guard Extension (SGX) Technologie verarbeitet.

Acknowledgement

I would like to thank my thesis supervisor, Dr. Michael Sonntag for giving me the unique opportunity to work on this project. Without his support, this research would not have been possible. My thanks also goes to the entire INFRASPEC team, especially Stephan Schraml from AIT (Austrian Institute of Technology), who did a great job in coordinating all the efforts of the different institutions that worked on the project. I would also like to thank all the staff at INS (Institute of Networks and Security) for the supportive working environment and the infrastructure they provided. Finally, I would like to thank my parents, who gave me the opportunity to study at the JKU and supported me during that whole time.

Eidesstattliche Erklärung/ Affidavit

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

I declare in lieu of oath that I have written this master thesis independently and without outside help, that I have not used any sources or aids other than those indicated, and that I have marked the passages taken verbatim or in spirit as such.

Linz, 24.02.2024
Ort, Datum/ Place, date

Unterschrift/ Signature

Contents

Chapter 1

Introduction

1.1 Motivation

Underneath the streets and buildings of cities, a large network of supply shafts is located. While we walk above them daily, we usually don't notice their existence. However, should a problem occur in one of those shafts, we would observe it immediately. We depend on the electricity, water, gas, etc. that runs through those long and narrow corridors underground, and thus ensuring their proper function is an important task to the companies that are responsible for their operation. Regular inspection and maintenance are required to ensure the safe operation of the supply shafts. Otherwise, interruptions in the utility service could occur, or the leaking of dangerous substances could damage the surrounding environment. Currently, inspections are done by humans, that walk through the shafts and check for damages. For example, this includes damage to the supply lines (e.g. water or other substances leaking) and also damage to the shaft itself (structural integrity of the building). The goal of the INFRASPEC project is to support human inspectors and make the inspections faster and more efficient. By using a robot to drive through the supply shafts, the project aims to create a very detailed 3D map of the environment. These 3D maps can then automatically be compared to previously recorded versions and changes to the environment are shown to the inspectors. The software is also able to detect some anomalies automatically, such as sintering in the cement, and show them to the inspectors. Additionally, the robot also has an arm with various sensors at its disposal, as well as some other hazardous substance sensors, for example for gas detection.

The work performed in this master's thesis is part of the INFRASPEC project. The goal is to pack all the data collected by the robot, as well as other components in the INFRASPEC system, into a "forensic data package". This package should be unchangeable without detection and serve as proof that an inspection was done correctly, as it may be required by law that these inspections take place. Also, if there are damages to the supply line, the forensic package can serve as proof, that the robot did, or did not, cause those damages and whether it performed the inspection correctly or not.

1.2 Objectives and Approach

1.2.1 Collecting Forensic Evidence

The collection of the forensic evidence itself should occur without any intrusive changes to the whole system. For example, the robot uses a CAN bus to communicate with its motors and sensors. The data that is sent over the CAN bus should be recorded and included in the forensic data package. The recording of the data, should not have any impact on the operation of the robot or require any change in the robot's software. To fulfill these requirements, the forensic component on the robot has a USB2CAN device attached to it, which is connected to the CAN bus and records all received messages. The robot also uses ROS2 (robot operating system) to communicate via Ethernet with its various other components, like the 3D camera or the robot's arm. The forensic component on the robot can be configured to listen to certain ROS topics and record the published messages of the robot's components.

1.2.2 Storing Forensic Evidence

Another objective is the secure storage of the forensic evidence. After the data has been collected by the robot or the other forensic components in the system, the data should be stored in such a way, that it cannot be modified after the inspection without the changes being detectable. To that end, the forensic system employs two different techniques to guarantee the authenticity and integrity of the data. Firstly, each forensic component has a YubiKey attached to it. The base-station (which is the forensic component collecting all the data), must initialize these keys and generate a public-private key pair on the YubiKey. The public key is saved in the database and the private key is only stored on the YubiKey itself, where it cannot be extracted. During the data collection, the forensic components then use their YubiKey to sign the messages before sending them to the base-station. The base-station can validate the signatures, using the stored public key of the data source. Secondly, to ensure data integrity and that no data is missing, hash-chains are being used. Each forensic component updates its respective hash-chain when new data is collected and the base-station maintains a database of all hash-chain entries, which is used for data validation and then merges new data entries into a "main-hash-chain'.

1.2.3 Securely Processing Forensic Data in the Cloud

After recording and securely storing the data, the last step is to process and analyze the data. As sometimes huge amounts of data are collected, users might want to process these data

packages in the cloud. However, some of the collected data is considered confidential and should not be accessible to other entities (like the cloud provider). For example, some supply shafts are considered critical infrastructure and their layout should remain confidential. To sum up, the data should be processed in the cloud, but the cloud provider should not be able to have access to the data. To accomplish this task, the SGX (secure guard extension) technology from Intel is used. A prototype has been implemented, which is capable of securely processing a point cloud, without the cloud provider having access to the data in transit or while it is in use.

Chapter 2

System Design

2.1 Outline

As mentioned in section 1.2 the goal is to develop a system that is capable of collecting reliable forensic evidence for later examination. This automatic evidence collection has to happen while the monitored system is ”live”. The ”monitored system” is the robot and the individual components that it carries, which are used to survey the collector shaft, as well as the base-station. ”Live forensics” refers to the technique of collecting forensic data, while the system is running.[5] This is the opposite of ”static analysis”, where the surveyed system is shut down while being analyzed. The forensic system needs to be able to capture the data, store it securely, and enable the user to process the data in a secure environment. There are three main components to the whole system: data collection service(s), data storage service, and data processing service.

2.2 System under Examination

To create a system to monitor the application and to collect forensic evidence from the robot and its components, the area where the system will be used and how it will be used, must be analyzed first. The system is capable of surveying ”collector shafts”, as seen in Figure 2.1 and Figure 2.2. These ”collector shafts”, or ”supply shafts” are usually underground tunnels in which electricity, gas, water, cooling or heating pipes, and other necessary utilities are connected to buildings. In Figure 2.1 we see a relatively wide and bright supply shaft. In this shaft, a person can walk without being restricted by the height or the width of the shaft. Also, the shaft is well-lit, which is important for human workers, and even more so for the robot. However, the environments in which the human workers and the robot have to operate aren’t always hospitable. in Figure 2.2 we can see an example of a much narrower supply

shaft. This shaft is also not as well-lit as the first one, posing a challenge for digital cameras. Additionally, there is not much space for moving around and tall persons would have a problem standing up straight. Especially, the width of the shaft and the obstacles near the ground pose a challenge for the robot. It could easily get stuck on the metal support holding up the pipes on the right side of the shaft or bump into the utility lines running on both sides.

Figure 2.1: Example of a collector shaft.

Figure 2.2: Example of a collector shaft.

2.2.1 Use-Cases

The shaft seen in Figure 2.1 would be a typical environment where the system could be tested, evaluated, and eventually deployed in the future. Currently, humans are tasked with surveying and regularly checking these supply shafts. This means manually walking through kilometers of tunnels and checking for potential damage on the supply lines or the shaft itself. As these supply lines can carry gas or high-voltage electricity, this task is not without its dangers. In the future, the INFRASPEC system could help in these tedious surveys. The goal of the INFRASPEC project is to create a robot that can create a detailed 3D map of the whole environment for automatic and manual inspection, using its onboard sensors.

Improve Surveying

In the future, the system could be deployed in a tunnel, manually driven around by an operator, and then the recorded data could be analyzed and stored for later reference. This allows for a much more fine-grained and partly automatic inspection, compared to the task of manually surveying the shafts. Some parts of the inspection will happen automatically. For example, the automatic detection of sintering of concrete using AI image analysis, as well as the detection of changes to the environment. Recordings from the past are compared with new recordings and even small changes to the environment can be detected and are displayed to the human inspectors for analysis.

Prove Correct Operation

Together with the forensic part of the system the question of who is at fault in case of damage to the supply shafts can be resolved quickly. The forensic system records all the commands the robot has been given and can significantly simplify the reconstruction of accidents. Apart from the correct operator, the operator can also prove that they did perform the survey, as regular inspections may be required by law.

2.3 INFRASPEC - Components

The INFRASPEC system consists of multiple different hardware and software components. In this section, the components for surveying the supply shafts are explained in detail. In section 2.4 the components and details about the forensic system for data recording are explained.

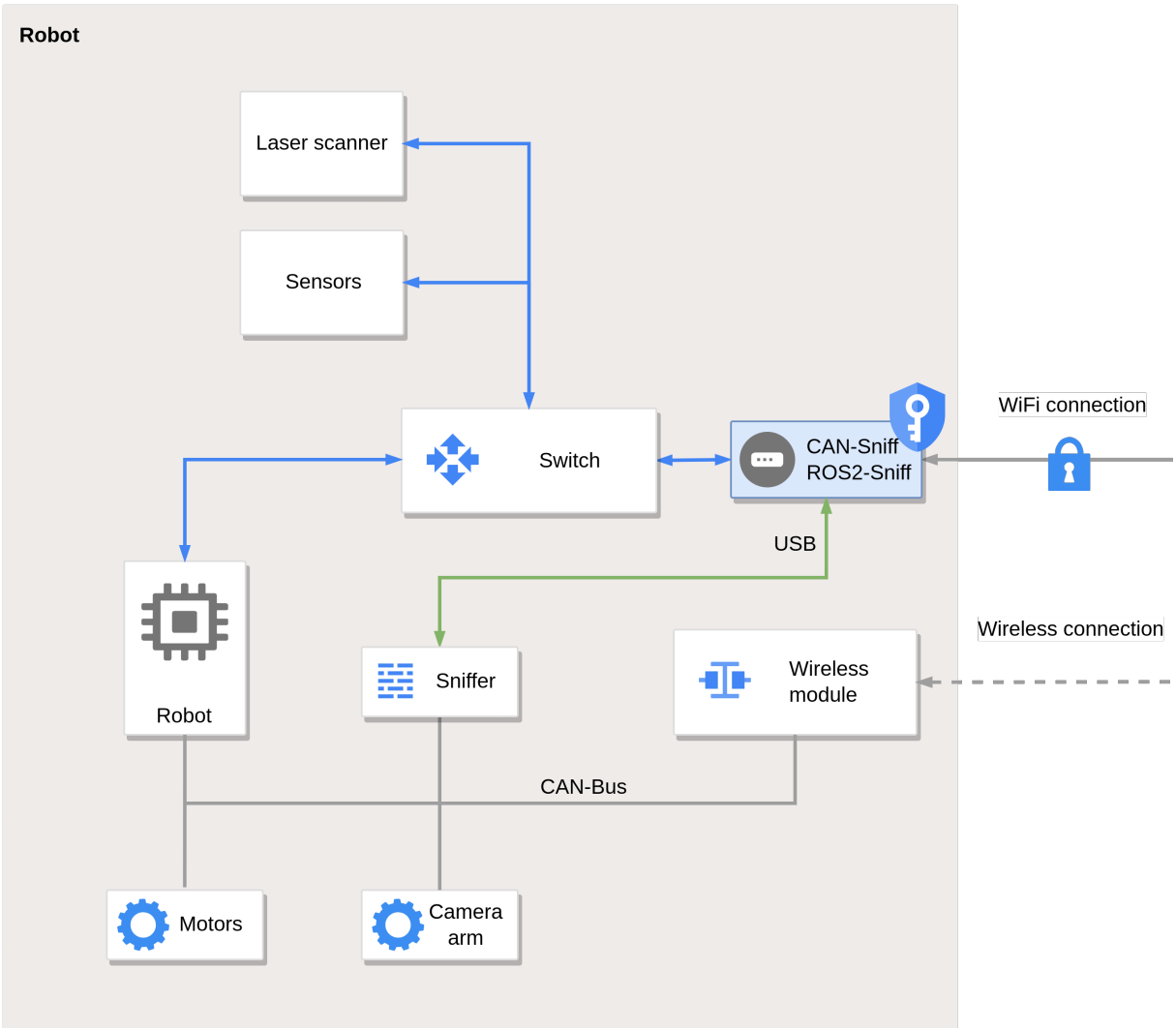

Figure 2.3: Overview of the robot's components (for complete system see Figure 2.7).

In Figure 2.3 a schematic overview of the robot's components and their interactions with each other can be seen. The robot has two connections to the control/ base-station. One wireless connection that the robot uses to transfer its sensor data and that the forensic system also uses and another wireless connection for the robot control. Furthermore, multiple sensors, that are using ROS2 [46], are attached to the robot and connected to its network (see "Sensors" and "Switch" in Figure 2.3 for example). Another important part of the robot is the "CAN bus". This bus is used to control the robot's vehicular components (see "CAN bus" in Figure 2.3).

2.3.1 Robot

The most important part of the system is the robot itself. It can be seen in Figure 2.4, without any of the other components and sensors mounted on it. The robot was developed by Rosenbauer and can navigate the narrow corridors of a supply shaft. In this first iteration of the project, it is not envisaged that the robot will navigate autonomously. Instead, an operator will control the robot from a ”base-station”, see section 2.3.6.

Figure 2.4: Robot used to navigate the collector shafts.

2.3.2 Vz-400i (Laser Scanner)

The Vz-400i is a 3D laser scanning system by Riegl.[45] Its features include a scanning range of up to 800 meters with an accuracy of 5 millimeters. The sensor itself also has a gyroscope built in, which makes it possible to move the sensor after one measurement, then take another measurement at a different position, and then finally combine the maps using the position estimates from the gyroscope. Furthermore, the sensor is mobile, as it has a built-in battery, which can supply the sensor with electricity for up to 1 hour and 40 minutes. The battery can also be charged while in use, which means that the robot can charge the battery while the scanner is being used.

Figure 2.5: 3D scanner used to map the collector shafts.[45]

2.3.3 Robot Arm

The 3D laser scanner cannot scan everything from its position on the robot. For example, scanning areas behind a pipe or other objects is not possible. Take the pipes in Figure 2.1 as an example. The operator might want to inspect an important area behind the pipes (as it could be damaged etc.). For this purpose, the robot has an arm, on which a sensor can be mounted to ”look” behind the pipes. The Vz-400i is too heavy and large, and cannot be mounted on the arm due to its weight. Instead an ”Azure Kinect DK” 3D camera is mounted on the arm.[3] The azure kinect DK consists of a depth camera (1MP Time-of-flight), an RGB camera (12MP CMOS sensor rolling shutter) and an IMU (3D digital accelerometer and a 3D digital gyroscope).[4] Using this 3D camera, the operator can map parts of the collector shaft, that the Vz-400i cannot see from its position. Currently, the maps created by the Kinect camera are not merged into the 3D map the Vz-400i creates but are saved separately. This merging might be implemented in follow-up projects.

Figure 2.6: The mechanical arm of the robot, where a 3D camera can be mounted for fine-grained inspection.

2.3.4 IR-Camera

To measure the heat on surfaces, the robot uses a ”Hikvision DS-2TD3017T-3/V Thermographic Cube Camera”. This camera creates images with a resolution of 640x480 and measures temperatures in a range of -20°C up to 150°C. Using this camera, the individual supply lines of the collector shafts can be checked for overheating or damaged insolation.

2.3.5 ROS2 (Robot Operating System)

The robot, the 3D sensor, and the robot’s arm are controlled using the Robot Operating System 2 (ROS 2). ROS is an open-source software framework designed to facilitate the development of complex robotic systems. It serves as a collection of tools, libraries, and conventions that help researchers and developers build and manage robotic applications. ROS2 is an evolution of the original ROS, aiming to address its limitations and offer improved capabilities. ROS2 enables modular and distributed development, allowing developers to create independent software components called ”nodes” that can communicate with each other via a

messaging system. This communication is based on a publish-subscribe model, where nodes publish data on specific topics, and other nodes subscribe to those topics to receive the data they're interested in. With ROS2, developers can work with multiple programming languages, such as C++, Python, and others, making it more accessible to a diverse range of developers. The framework also includes tools for visualization, debugging, simulation, and testing.[46]

2.3.6 Base-Station

The whole system can be split spatially into two different spheres. The robot, which has been explained in detail before, and the base-station. The robot consists of the moving platform, its sensors, and its communication equipment. However, the robot is not autonomous and must be controlled by an operator at the base-station. The operator can control the robot's movements from there, see a camera stream, and also see a visualization of incoming data from the robot's sensors. Two different communication links exist from the base-station to the robot. One is the "control-link" to the robot, and the other is the "data-link". The "control" link currently uses a wireless connection and is used to navigate the robot. The "data" link is used to transfer the data from the sensors to the visualization component of the base-station. In this way, the 3D maps, thermal information, and more, can be transferred and displayed to the operator in real time. This information can then be used to decide which areas to survey next, or if some part of the environment needs to be further examined using the robot's arm. The data link is currently planned as a Wi-Fi connection, which poses a challenge, as wireless connections might be obstructed by thick walls in the underground shafts. Because of the possible bandwidth limitations of the wireless connection, the forensic component should use as little bandwidth as possible, so that it does not interfere with the system's normal operation. The 3D map created by the Vz-400i, as well as the 3D map from the Azure Kinect DK, are already very large and put considerable stress on the wireless link.

2.4 Forensic Evidence Collection System

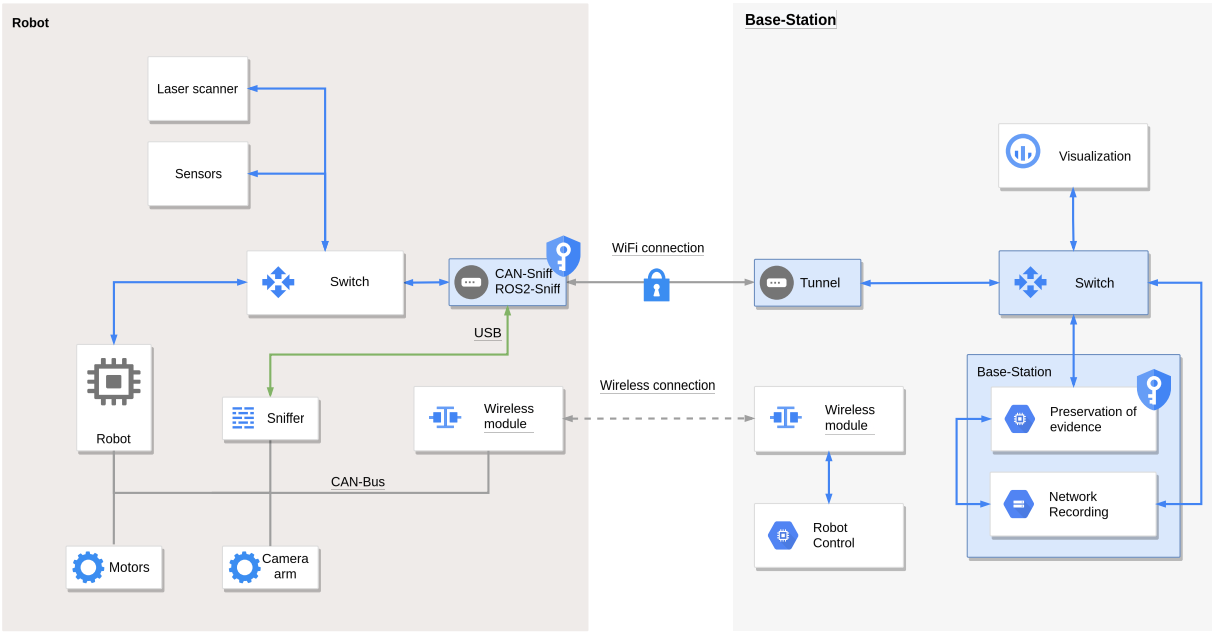

Figure 2.7: System overview including components for forensic data collection.

Until now the system for surveying the collector shafts has been described. Now an add-on to that system will be explained, which records the data and movements from the robot and stores them in a forensic record. This forensic record contains all the data that is necessary to reconstruct the movements and actions of the robot. This data record is cryptographically signed and timestamped so later modification would be detected. For more detail on the forensic data package see section 2.5.6.

2.4.1 Components

Robot Computer

To run the data collection software on the robot side, a computer is mounted onto the robot (see Figure 2.3, component: ”CAN-Sniff and ROS2-Sniff”). This computer is a Zotac ZBOX C Series CI547 nano. The required energy is supplied by the robot and it is also attached to the robot’s network and acts as its gateway to the base-station. This means that all the sensor data transferred over the network is sent using this component. This is why, apart from its forensic duties, it also creates an encrypted MACsec [48] tunnel to the base-station, to prevent bad actors from sniffing the data.

Base-Station

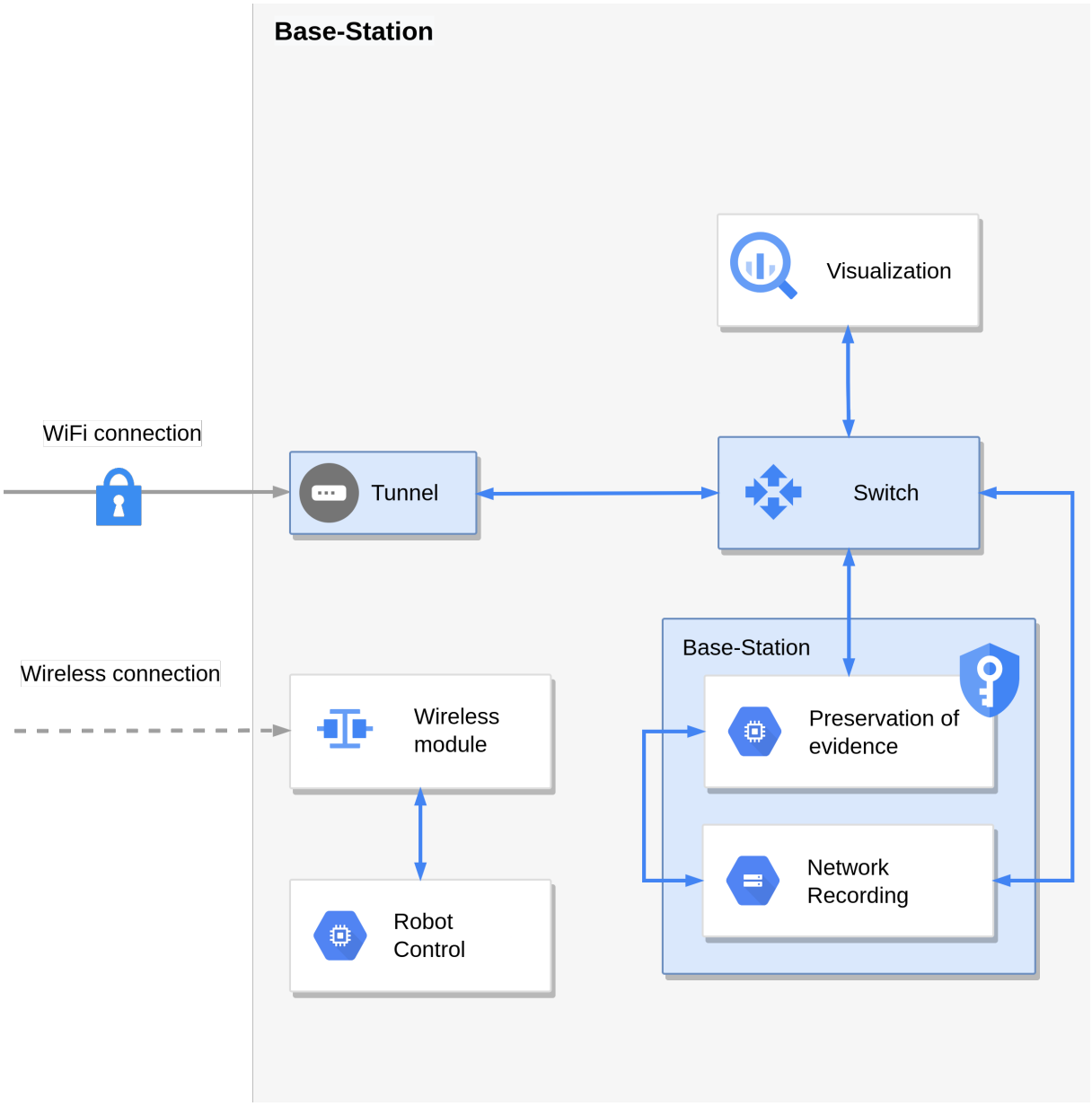

Figure 2.8: Detailed overview of the base-station components (for complete system see Figure 2.7).

The base-station component of the forensic system can be seen in Figure 2.8 and Figure 2.7, as the big blue box labeled ”Base-Station”. This software component runs on its separate computer and is responsible for verifying and collecting all data, that the other forensic com-

ponents send to it. Two different self-developed software components run on it. Firstly the "preservation of evidence" and secondly the "network recording", as can be seen in Figure 2.7.

The software setup on the base-station uses docker together with docker-compose, to manage the docker containers. The docker-compose file starts several different services, which are necessary to run the forensic system:

- RabbitMQ [18]: It is an open-source message broker software that enables communication and data transfer between different components or systems in a distributed environment. It uses a message queueing system to ensure asynchronous and reliable message delivery. RabbitMQ is widely used in various industries, including software development, microservices architecture, and cloud computing, to enable efficient and decoupled communication between different parts of an application.
- PostgreSQL [41]: It is an open-source relational database management system (RDBMS) that allows users to efficiently store, manage, and retrieve structured data. It is known for its reliability, data integrity, and extensive features, including support for complex queries, indexing, and transactions. PostgreSQL is widely used in various applications, from web development to data warehousing, due to its robust and extensible nature, making it a popular choice for systems that require a robust and easy-to-use database solution.
- Adminer [1]: It is a lightweight, open-source database management tool that provides a user-friendly web-based interface for managing various database systems, including MySQL, PostgreSQL, and more. It allows users to perform database tasks such as querying, editing, and managing data, tables, and user accounts through a convenient and simple web interface, eliminating the need for a dedicated database client. This component is only enabled for debugging purposes and is not included in the productive deployment of the system.
- Redis [17]: It is an open-source, high-performance, in-memory data store that is often referred to as a "data structure server". It is designed for fast data retrieval and storage, making it well-suited for use cases requiring low-latency access to frequently updated data, such as caching and real-time analytics. Redis is known for its support of various data types, including strings, lists, sets, and more. It is widely used in modern web development and microservices architectures to improve application speed and responsiveness.

Additionally, several services were specially developed for this project:

- "netcap": This service can be seen in Figure 2.7 as "Network Recording". This service is responsible for recording all network activity that is transferred over the "data link" (WiFi Connection). The software then pushes that information to the "webserver" component (part of "Preservation of evidence" seen in Figure 2.7). For this component to work, the

port on the switch is passing through all data and not only data addressed to the computer running the forensic software.

- ”signserv”: This service is part of the ”Preservation of evidence” component seen in Figure 2.7. It is responsible for accessing the base station’s YubiKey and also for initializing all YubiKeys during the setup. It listens to a messaging queue on the RabbitMQ component and verifies and signs data that is put into the queue before saving it to the database.
- ”webserver”: This component is the heart of the whole forensic application. It receives data sent to it by the other components, processes them, and stores them in the database. The user can see live information about the incoming data in the web interface and create signed exports after all data is collected.

YubiKey

Figure 2.9: YubiKey 4C

A YubiKey is a compact, hardware-based security token designed to enhance the authentication and access control mechanisms in computer systems. It serves as a multifactor authentication device, generating one-time passwords (OTP) or public-key-based authentication codes to verify the identity of users and protect against unauthorized access or for signing data. YubiKeys have gained popularity for their robust security features, ease of use, and compatibility with a wide range of applications, making them a valuable asset in modern cybersecurity strategies.[62]

CAN Bus Sniffer

Figure 2.10: USB to CAN device from InnoMaker.[29]

To record messages from the CAN bus (see Figure 2.3), a CAN bus reader is attached to the forensic recording system (see section 2.4.1) on the robot. The CAN bus reader is a simple ”plug & play” USB to CAN bus converter. The device can read a CAN bus at a baud rate from 20 kbps to 1 mpbs.[29] On linux the USB2CAN device can be accessed using ”SocketCAN”.[50] SocketCAN consists of multiple open source CAN drivers. For more information about the CAN bus and how messages are read from it, see section 3.3.1.

ROS2 Message Collector

As mentioned in section 2.3.5 the robot uses the ROS2 framework to access some of its sensors. The Vz-400i (see section 2.3.2) uses ROS2 to publish its recorded point cloud messages. These messages also need to be recorded and included in the forensic data package. The ”ROS2 Message Collector” is a component directly located on the robot and is running on the same computer that is also used for all other forensic evidence collection on the robot (see section 2.4.1). In the configuration of this component, the user can specify which topics should be captured. The captured data is then collected and signed, just like the data recorded on the CAN bus.

Network Traffic Collector

The network traffic collector resides on the base-station (see Figure 2.8, component: ”Network Recording”). This component records all the data sent over the network and saves it directly to the ”preservation of evidence” component.

2.4.2 Interaction between Components

For the individual components to work correctly, some interaction and coordination between them is necessary. The base-station is the main source of control commands from the user. Most of the configuration is stored on the base-station. The forensic components query their respective configuration values from the base-station during their startup. Some settings, like the IP address of the base-station, must be manually configured at the forensic component.

CAN Bus Sniffer to Base-Station

The CAN bus sniffer is located on the robot and listens to the CAN bus on the robot. The CAN bus reader (see section 2.4.1) is a USB device that enables a normal PC to read and write to a CAN bus. As the forensic component does not write anything to the CAN bus, only the reading feature of the USB2CAN adapter is used. To read the CAN bus the USB2CAN device needs to know the bitrate of the CAN bus. This configuration value can be set in two places, either as an environment variable to the robot’s Docker container or as a ”config” value on the base-station. The base-station has a database table that contains key-value pairs of configuration parameters. These configuration values can be set in the web interface by the user and will take precedence over the environment variables. When the docker container, in which the CAN bus sniffer runs, is started, it queries the current configuration value from the base-station and then uses that bitrate if the value is set. If it is not set, it will use the value provided by the environment variable in its container.

The data from the CAN bus sniffer is put into a RabbitMQ message queue running on the robot. The robot component then processes that data, creates a hash-chain, signs it, and sends it over to the base-station. This whole process is covered in detail in section 2.5.6.

ROS2 Message Collector to Base-Station

The only interaction with the base-station is the command when to start or stop recording new data. This component is not configured over the base-station, but rather using a ”con-

fig.py" file, where the user can specify the topics to record, as well as define callbacks to data formatters. For more information on ROS2 and the data recording process, see section 3.3.2.

Network Traffic Collector to Base-Station

The network traffic collector is located on the base-station. This component runs in its own Docker container and gets its configuration directly from the base-station. The traffic sniffer checks which network interfaces it can access and then sends that information over to the base-station. The base-station presents the options to the user and the user can then choose one of the interfaces that the component should use to record the data. After that, the component will start recording the network data and then immediately submit that data to the base-station.

2.5 Preservation of Forensic Evidence

As explained in section 2.4.1 the most important part of the forensic system on the base-station is the component responsible for collecting and storing the forensic evidence. It was mentioned, that the Docker service called "webserver" handles this task together with the "signserv" service. Both of these components work together using the RabbitMQ service, as they have multiple messaging queues for communication between them.

2.5.1 Security Requirements

In [33] several properties are defined such that "digital evidence" can be considered trustworthy. According to their definition, digital evidence consists of the actual data (for example a photograph), as well as additional information about the device on which the data was recorded. Furthermore, they define two properties that must hold for a device to be considered "forensically ready".

- The device is physically protected to prevent tampering, and the data record is securely linked to the device's identity, status, and relevant parameters, with protection levels tailored to the scenario and data type.
- The integrity of the data record remains unaltered since its creation, ensuring that no modifications have occurred.

To fulfill these requirements, several security precautions are taken. First of all the data records must be tied to the recording device's identity. This is ensured using the YubiKeys

(see section 2.5.2). The YubiKeys are used to tie the data records generated by a device to that specific device. This is done using the public and private keys of the hardware token. The public key for the device is known to the base-station and the private key is only stored on the physical YubiKey, from where it cannot be extracted. The YubiKey is then plugged into the forensic device and the device can use the YubiKey to sign its recorded messages thus tying the messages to the signatures from one specific YubiKey. To fulfill the second requirement a hash-chain is created. Every new data record is attached to that hash-chain and the resulting hash-chain value is signed by the YubiKey. This way we can ensure that if data is missing or has it has been modified, these changes can be successfully detected when examining the data. Subsequently, this also means that we can trust the data if no changes have been made and the signatures can be verified.

2.5.2 YubiKeys

For the whole forensic system to work, at least two YubiKeys must be used to establish trust between the components and to create the signatures required for the forensic data package. The two YubiKeys are displayed in Figure 2.7, as the small blue keys next to the ”CAN-Sniff” and ”ROS2-Sniff” component on the robot and the ”Preservation of evidence” component on the base-station side. During the setup (see section 2.5.5) the base-station creates a public-private key pair on each of the YubiKeys and saves the public keys in its database. One YubiKey is plugged into the forensic component on the robot (”CAN-Sniff” and ”ROS2-Sniff”) so that it can sign recorded data before transmitting it over to the base-station. This way we can ensure, that only data coming from a trusted entity is included in our forensic data package.

Important to note here is, that a private key cannot be extracted from a YubiKey. As we generate the public-private key pair directly on the YubiKey, we only have the public key available and there is no chance, that the private key could be leaked, as it is securely stored on the YubiKey. The signing on the robot (and base-station) also uses the YubiKey hardware and the private key is not accessible, even to the person holding the YubiKey. This means the base-station component can be sure, that the entity signing data with the trusted YubiKey, must be the one physically holding the YubiKey.

2.5.3 Signing Service

The signing service, or short ”signserv”, is responsible for setting up the YubiKeys and for signing messages using the YubiKey, while the system is in its operational state. The different states of the system are listed in section 2.5.5.

The reason why the signing service is its component and not directly integrated into the "web-server" component, is purely an implementation necessity and not a specific design choice. The technical problem is the access of the YubiKey from within the docker container. As a USB device, the YubiKey cannot easily be accessed from within the container. One possible solution for this would be to run the docker container with the "–privileged" flag. According to the docker documentation, this lifts all restrictions placed on the container by the device cgroup controller[16]. So essentially, the docker container can do the same as a program directly running on the host system. It's easy to see why this flag should be avoided for security reasons, as this set of privileges would be much more far-reaching than what is required. Another way, which is already more restrictive, is to only mount the */dev* directory from the host into the docker container. This works because in Linux *"everything is a file" (and if not, it must be a process)*. Under the */dev* directory "special files", like USB devices are located. For example, a plugged-in USB camera could be located under */dev/video* and programs would then read or write to that file to receive the video stream or change settings on the camera. The same applies to the YubiKey. Once it is plugged in, the device can be found under */dev/hidraw1* and */dev/hidraw2*[1]. These "hidraw" devices are USB devices, which can be accessed via the HID (Human Interface Device) protocol.[35] The current solution used in the software implementation is that the whole */dev* folder gets mounted into the docker container. This is done because it cannot be assured that the YubiKey is always located under */dev/hidraw1* and */dev/hidraw2*. This can be shown with a simple test:

1. Run bash command: `ls /dev | grep hidraw`
   output: hidraw0
2. Plug in the YubiKey
3. Run bash command: `ls /dev | grep hidraw`
   output: hidraw0, hidraw1, hidraw2
4. YubiKey can be accessed using the files */dev/hidraw1* and */dev/hidraw2*

At the end of the first example, the YubiKey can be accessed using the special files */dev/hidraw1* and */dev/hidraw2*. If both of those files were mounted into the docker container, they could be accessed from inside the container, even without running the container in "privileged" mode.

1. Run bash command: `ls /dev | grep hidraw`
   output: hidraw0
2. Plug in a generic USB-Keyboard
3. Run bash command: `ls /dev | grep hidraw`
   output: hidraw0, hidraw1

[1] Numbers may vary depending on other hardware attached to the system.

4. Plug in one YubiKey
5. Run bash command: `ls /dev | grep hidraw`
   output: hidraw0, hidraw1, hidraw2, hidraw3
6. YubiKey can be accessed using the files */dev/hidraw2* and */dev/hidraw3*

At the end of this second example, the YubiKey can be accessed using the special files */dev/hidraw2* and */dev/hidraw3*. So if we would statically specify to mount the special files */dev/hidraw1* and */dev/hidraw2* in our *docker-compose.yaml* file, the container would not be able to access the YubiKey. So if the configuration works or doesn't work, would depend on the setup of the host system and thus the solution of statically mounting the hidraw files does not work well. The "fallback" option is to directly mount */dev* into the docker container. This is the solution in the current implementation, however, it is still not ideal, as the container can now access all devices located under */dev*.

2.5.4 Forensic Base-Station

General Design

The docker container called "webserver" is the central service of the whole forensic evidence collection system. It can be seen in Figure 2.8 in the blue box labeled "base-station". It collects all the data from all the other forensic components and also handles the network recording. In Figure 2.11 a simple diagram of the data flow in the forensic base-station can be seen. A more detailed overview including some information about the implementation can be seen in Figure 2.13.

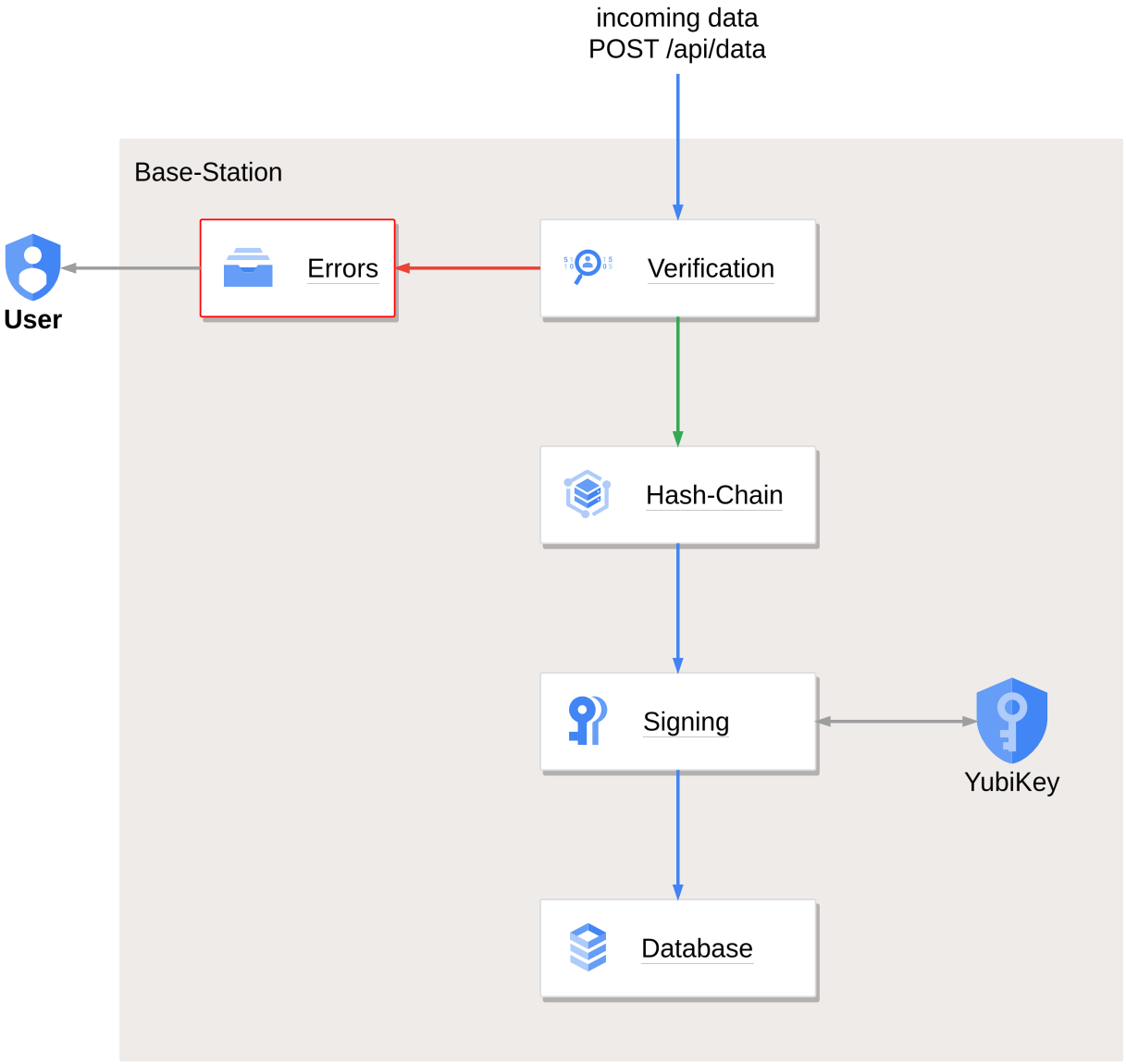

Figure 2.11: Data flow of incoming forensic data on the base-station.

In Figure 2.11 we can see there are essentially three steps necessary so that at the end of this process the data is stored securely. The first step is to verify the data. This includes a check if the signatures from the forensic component sending the data are valid and trusted, as well as a check of the hash-chain (see section 2.5.6). The second step is the securing of the data for later storage. This means that the base-station updates its hash-chain and also signs the new hash-chain value using its own YubiKey. After this process, we end up with a data record, that is signed by the forensic component which recorded it, is added to the base-stations hash-

chain, and is also signed by the base-station. This data record can then be saved into the base-station's database.

It is important to keep the security requirements from section 2.5.1 in mind. The requirements laid out there and in [33] boil down to the CIAAN principle:

- **Confidentiality**: This first point can be fulfilled by encrypting the traffic from and to the forensic components using HTTPS. In the current implementation and the subsequent tests, HTTPS was not used yet for simplicity. However, it is easily possible to switch to HTTPS by simply providing self-signed certificates to the web servers and then marking these certificates as trusted by the clients. An additional layer of encryption is the MACsec network encryption between the robot and the base-station. The data enters the encrypted tunnel when being transferred from the robot to the base station. During the wireless transmission, the data is sent encrypted and cannot be read by bad actors listening to the wireless signals. When arriving at the base-station, the MACsec tunnel is exited and the data is decrypted again.
- **Integrity**: This requirement is fulfilled by using hash-chains which include all recorded data. If data is changed or removed the verification process will detect these changes.
- **Availability**: The availability of the forensic system is assured by having a locally controlled network. The whole data recording happens in its network, where the user has to take precautions to ensure, that no bad actor gains access to the network to run a DoS (denial of service) attack. Also, after the recording is finished, the forensic data can be exported into a single SQLite3 database, which can be saved locally on any computer. All further investigations can be made using that exported data and do not rely on any online service (except the processing in the cloud using SGX, which is described in detail in chapter 4).
- **Authenticity**: This property is assured by using digital signatures. Each forensic component has a physical YubiKey attached. These YubiKeys are initialized by the base-station and a public and private key pair is generated on the YubiKey itself. The public key is saved on the base-station and the private key cannot be extracted from the YubiKey. The forensic components then use their YubiKey to sign its recorded data. When the data is sent over to the base-station, it can use the saved public key to verify the sent data.
- **Non-repudiation**:
  - The hash-chains also provide security against **replay attacks** (see section 2.5.6).
  - The hashes attached to each message in combination with the signatures also protect against **message tempering**.

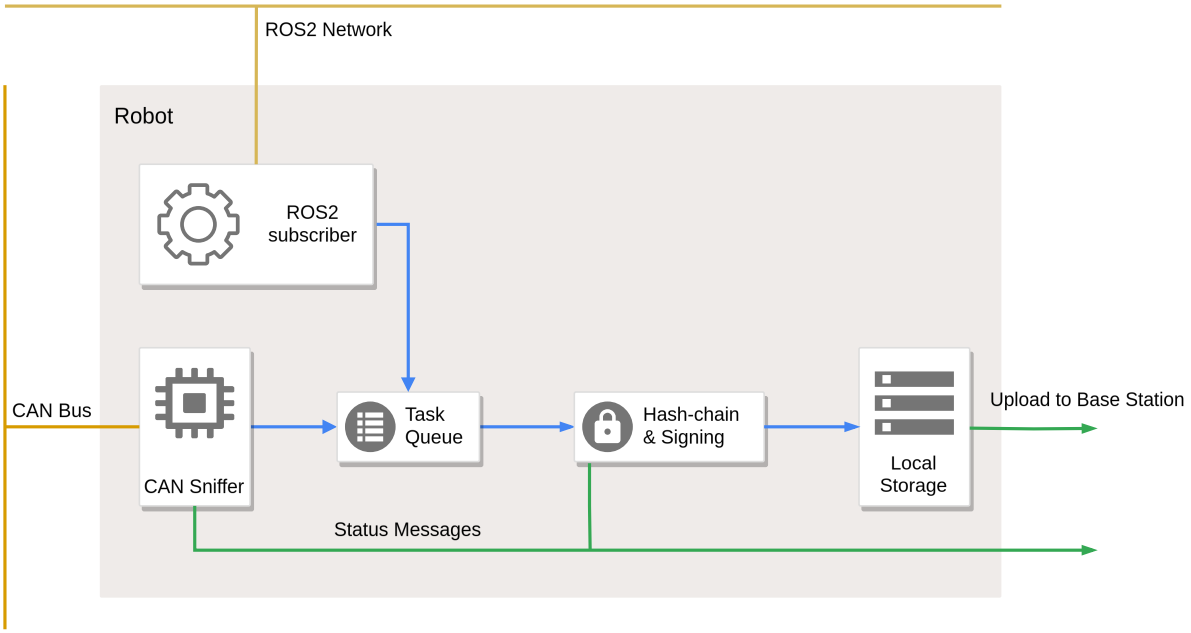

Figure 2.12: Data flow including implementation details of the robot.

Implementation Overview

Following from the basic design laid out in Figure 2.11, we can derive a concrete implementation by supplementing each step with one of our components from section 2.4.1. The incoming data from the robot (see Figure 2.12) is processed step by step by the components of the base-station, as can be seen in Figure 2.13.

All base-station components are docker containers, that can be started using a docker-compose file. When the services are running, the webserver listens on port 80 and new data can be sent to it. Once new data arrives from one of the other forensic components (for example the one on the robot), the data gets put into a messaging queue of the RabbitMQ service (see "Task Queue" in Figure 2.13). The "signsserv" component (see section 2.5.3) waits for new messages on the queue and as soon as a new message is published on the queue, starts processing it. Firstly, it performs the necessary verification of the signatures and the "sub-hash-chain" that is attached to the data record. This corresponds to the step "Verification" from Figure 2.11. If the verification fails, either because the signatures are invalid or the last hash-chain record does not match the signed one, an error message is saved to the database and displayed to the user in the web interface. If the verification is successful the "main-hash-chain" of the base-station is updated (see section 2.5.6). This corresponds to the block "Hash-Chain" in Figure 2.13. After the new hash-chain value has been calculated, the signserv components sign the value using its YubiKey. Additionally, if there is an internet connection,

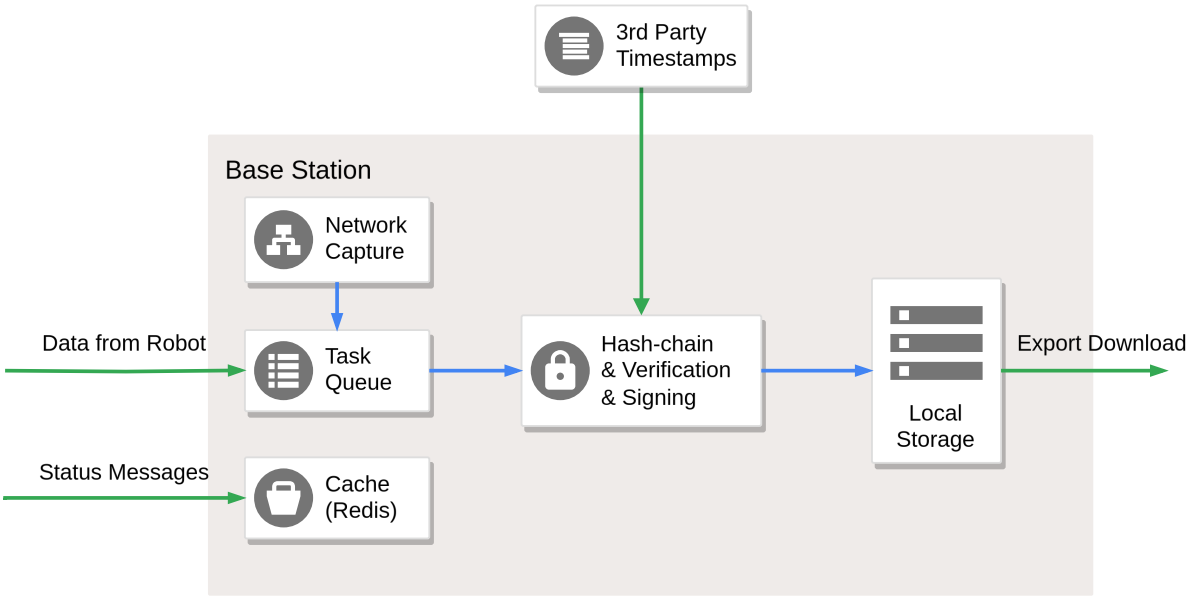

Figure 2.13: Data flow including implementation details on the base-station.

a third-party timestamp is added as an additional signature of the hash-chain. These signing operations correspond to the block "Signing" in Figure 2.13. Finally, the resulting data record is saved into the Postgres database, where it can later be exported for further processing.

In addition to the data records, the forensic components can also send status messages to the webserver. These messages are saved temporarily in a Redis database and get automatically deleted after a few minutes, or replaced with a newer message. For example, the robot continually sends a heartbeat message to the base-station, which includes its local system time. This is useful to check that the robot's timezone is set to the base-station's time zone, as otherwise, the timestamps of the data records won't match. The status messages also contain important information for the user. For example, it includes how much data the forensic component on the robot has gathered and can be sent to sent to the base-station, as well as several other metrics.

User Interaction

To use the forensic base-station, a web interface is available to the user. First, the user must set up a project (see section 2.5.5). Each project needs at least two YubiKeys. One that is constantly plugged into the base-station and one that is plugged into the robot (see blue key symbols in Figure 2.7). Additional YubiKeys for other data sources can be added if required. During the setup, public-private key pairs are generated on the YubiKeys. The public key is then saved in the project's database at the base-station and the private key remains on the

YubiKey, where it cannot be extracted. Then the user is instructed to plug in the robot's YubiKey on the forensic component that is attached to the robot. After this setup, the project can be set to "active", which tells all forensic components, that they should now start gathering data for the project. Once the user finishes the inspection of the supply shaft, the user can start "finalizing" the project. Now the other components start sending their collected data to the base-station for verification and archiving.

Data Collection

1. The data arrives on an endpoint at the webservers API running on the base-station (see "Data from Robot", Figure 2.13). As soon as the data arrives it is checked if there is a project currently accumulating data. If no such project exists, an HTTP 400 error is sent back, with the message *"No project is currently accumulating data"*. Otherwise, the data is put into a RabbitMQ queue for verification.
2. The "signserv" service listens for incoming messages on that queue and is responsible for verification of the incoming data. The verification itself is described in section 2.5.6. If the verification is successful, the data is saved in the project's database.
3. During the saving process, the hash-chains on the base-station are updated, and the resulting values are signed by the base-station. For more information on the data storing process see section 2.5.7
4. Additionally to the verification, every few seconds a 3rd party timestamp is attached to the data. This is one of the mechanics that ensure, that data cannot be changed after it is saved on the base-station.

2.5.5 Setup

As mentioned in section 2.5.4, the user creates "projects" on the base-station, which correspond to one inspection. Whenever the user wants to create a new forensic recording, they first create a project, then initialize it and after the data has been gathered "finalize" it. After this process is complete, the user can export the data into a SQLite3 database and archive it. The correct setup of the project is essential. For ease of use, the whole setup process is described in the web interface step by step and must be followed by the user.

Creation

After the project has been given a name and a description, the software puts the project in the "creation" state. Multiple projects can be in this state at once, but no data can be added

to the project yet. When clicking on the project page, the user is shown a dialog, that reads ”Initialize Project”, see Figure 2.14. If the button is clicked, the initialization process of the project will start. Note that, only one project can be in the state ”initialize” at the same time, to avoid conflicts when plugging in the necessary hardware (namely the YubiKeys).

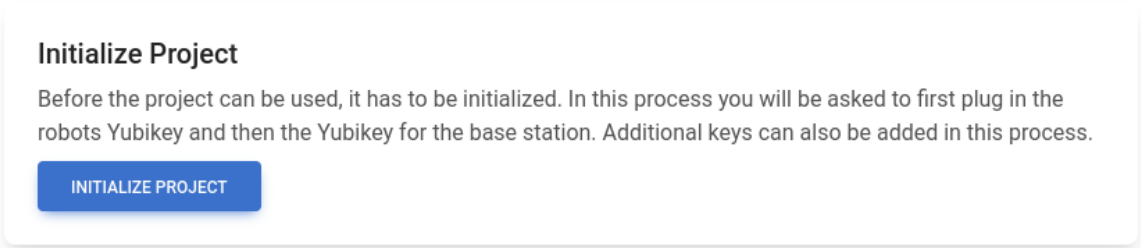

Figure 2.14: Initialize project

Initialize

Once the project is in the initialization state, it will ask the user to plug in the first YubiKey, which will be attached to the robot, as can be seen in Figure 2.15. The software waits until the YubiKey is detected and then displays the next message.

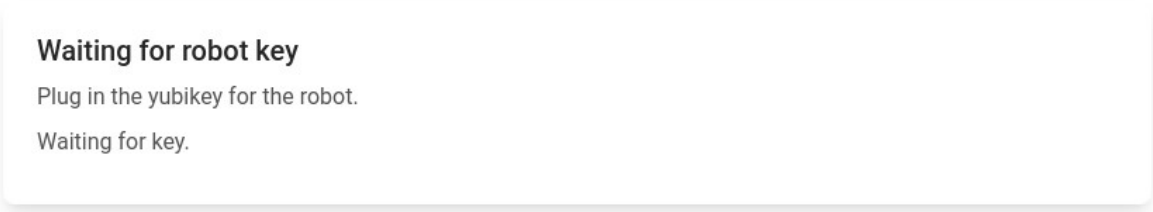

Figure 2.15: Waiting for the robot key

The next message simply informs the user, that the YubiKey was detected and the public-private key pair is now being generated inside the YubiKey. After the key is created, the public key is saved in the project’s database.

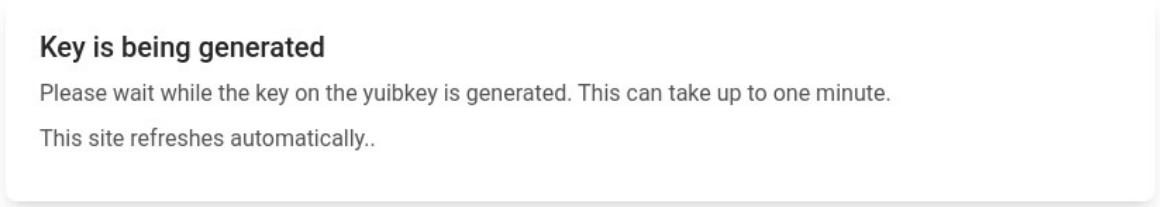

Figure 2.16: The key is being generated

After the robot's YubiKey is initialized, the user is prompted to remove the YubiKey from the base-station. Then the user is asked to plug in the base-station's YubiKey and the process is repeated. After the base-station's YubiKey is initialized, the user can choose to add more trusted forensic components and in turn initialize their YubiKeys.

Active

After the initialization phase, the project is in the "active" state. In this state, the forensic components can send their data to the base-station, where it will be displayed to the user. Additionally, the user can see a small "stats" window, which displays the number of error messages, as well as the amount of data the robot has gathered.

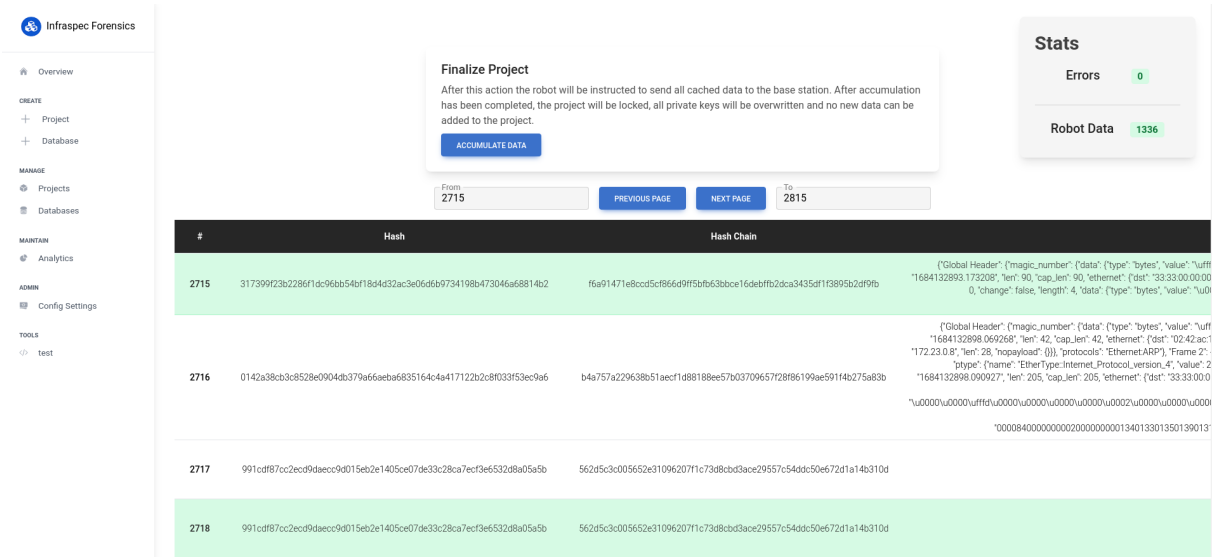

Figure 2.17: Project Overview

When the user decides that the inspection is finished, the user can click on "Accumulate Data".

This will tell all the forensic components to stop gathering new data and send over the collected data. A progress bar is then shown to the user to display how much data has already been sent over, and how much is still missing.

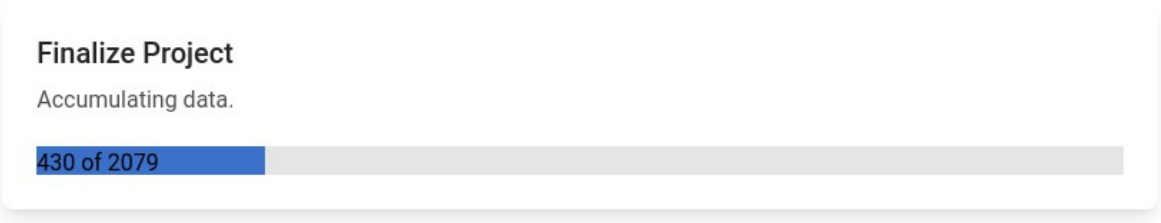

Figure 2.18: Accumulating Data

Finalize

Once all the data has been sent over, the project is automatically set to "locked". This means that no more data can be added to the project and that all forensic components have been instructed to overwrite their private keys on their YubiKey. To create an export of the data, the user can simply click on "Prepare Export" and the project's data will be converted into a SQLite3 database. Once this process is completed, the user can click "Download Export" and will receive a ZIP file which includes the SQLite3 database and a script to verify the integrity of the data in the export.

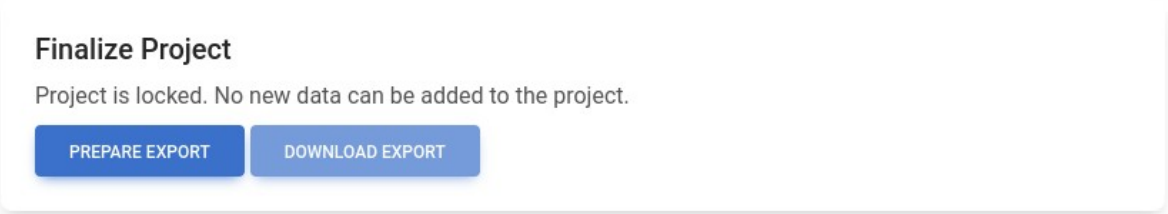

Figure 2.19: Finalize Project

2.5.6 Data Verification

One of the most important tasks of the base-station is to verify all incoming data from the various forensic components. As already explained before, the forensic components sign the data records they send to the base-station using their own YubiKey. The YubiKey holds a private-public key pair, which was generated directly on the YubiKey, while it was plugged into the

base-station, during the project initialization phase (see section 2.5.5). The forensic components then use the YubiKey to sign the tail of the hash-chain.

A hash-chain is thereby defined as a chain of multiple hash values that get added together repetitively, resulting in a new hash-chain "tail", each time a new data value is added to the chain. The starting value of the hash-chain will be the concatenation of the base64 representation of each public key that is used to sign hashes in the project. For simplicity, random starting values are chosen in the examples. The following example will illustrate how a hash-chain works using MD5 hash values[2] ("T" stands for the "tail" of the hash-chain and "D" for "data-point").

1. Choose a random start value: T1 = "wasd1234"
   md5(T1) = *1d5a51389eab7a976d3572e9db25fe78*
2. First data point D1 = "hello"
   md5(D1) = *5d41402abc4b2a76b9719d911017c592*
3. Update hash-chain (+ stands for string concatenation):
   md5(T1) + md5(D1) = *1d5a51389eab7a976d3572e9db25fe785d41402abc4b2a76b9719d911017c592*
   T2 = md5(md5(T1) + md5(D1)) = *5fdf411b626dff5e9c7d7e504bcba86e*
4. Next data point D2 = "world"
   md5(D2) = *7d793037a0760186574b0282f2f435e7*
5. Update hash-chain:
   md5(T2) + md5(D2) = *5fdf411b626dff5e9c7d7e504bcba86e7d793037a0760186574b0282f2f435e7*
   T3 = md5(md5(T2) + md5(D2)) = *c55946d78d8268cc710f7f88cc5ce1fd*

In the example, two data points were added to the hash-chain: "hello" and "world". To be able to verify the hash-chain, we need to save the two data values and the starting point of the hash-chain, as well as the last tail (T3) of the hash-chain. The verification process for this hash-chain is very simple. We just need to recompute the whole hash-chain and check if the recomputed last tail matches our saved tail:

1. Set T1 to the saved start value:
   md5(T1) = *1d5a51389eab7a976d3572e9db25fe78*
2. Compute hash of D1:
   md5(D1) = *5d41402abc4b2a76b9719d911017c592*
3. Update hash-chain:
   md5(T1) + md5(D1) = *1d5a51389eab7a976d3572e9db25fe785d41402abc4b2a76b9719d911017c592*
   T2 = md5(md5(T1) + md5(D1)) = *5fdf411b626dff5e9c7d7e504bcba86e*

[2] Please note that MD5 hashes are only used for demonstration purposes, as they are very short and can be displayed nicely. MD5 is considered insecure today and should not be used in productive systems.[51]

4. Compute hash of D2:
   md5(D2) = *7d793037a0760186574b0282f2f435e7*
5. Update hash-chain:
   md5(T2) + md5(D2) = *5fdf411b626dff5e9c7d7e504bcba86e7d793037a0760186574b0282f2f435e7*
   T3 = md5(md5(T2) + md5(D2)) = *c55946d78d8268cc710f7f88cc5ce1fd*

If we compare our recomputed T3 value with the saved T3 value, we can see that they match. If someone had changed the data, for example, replacing "hello" with "bye", the final value of the hash-chain would not match. But this alone is not enough to verify that no change has been made to the data. After all, an attacker could just change "hello" to "bye" and then recompute the hash-chain from there on. So we need some kind of mechanism to be sure, that the last tail was not modified. Here signatures come into play. An entity that we trust needs to sign the last entry. Now if someone would simply recompute the whole hash-chain, they would also need to redo the signature of the hash-chain tail, which should not be possible, as the attacker should not have access to the private key of the signing party. So to update the short algorithm presented above, the verification process now looks like this:

1. Verify the signature of the saved last hash-chain entry.
2. If it is invalid, the verification process ends immediately, as the provided tail value cannot be trusted.
3. Otherwise, start computing the hash-chain from the initial value T1.
4. Update the hash-chain with all data entries until the final tail value is calculated.
5. Then compare the recomputed tail value with the signed one.
6. If they match, the data has not been tampered with.
7. Otherwise, changes were made to the data.

There are two different kinds of hash-chains in the system. The "sub-hash-chains" and the "main-hash-chain".

Sub-Hash-Chains

The sub-hash-chains are the hash-chains that are created by the individual forensic components. Each time they record a new data point, they calculate the hash value for that data and then update their locally stored hash-chain value. When sending the data to the base-station, the forensic components include their current sub-hash-chain tail with the data and signs that hash using their YubiKey.

Main-Hash-Chain

The main-hash-chain is the hash-chain, that is created on the base-station. It unifies all sub-hash-chain values from the different components and merges them into one hash-chain. The tail of the main-hash-chain is signed using the base-station's YubiKey. Additionally, if there is internet available, the base-station adds a 3rd party timestamp to the hash-chain value.

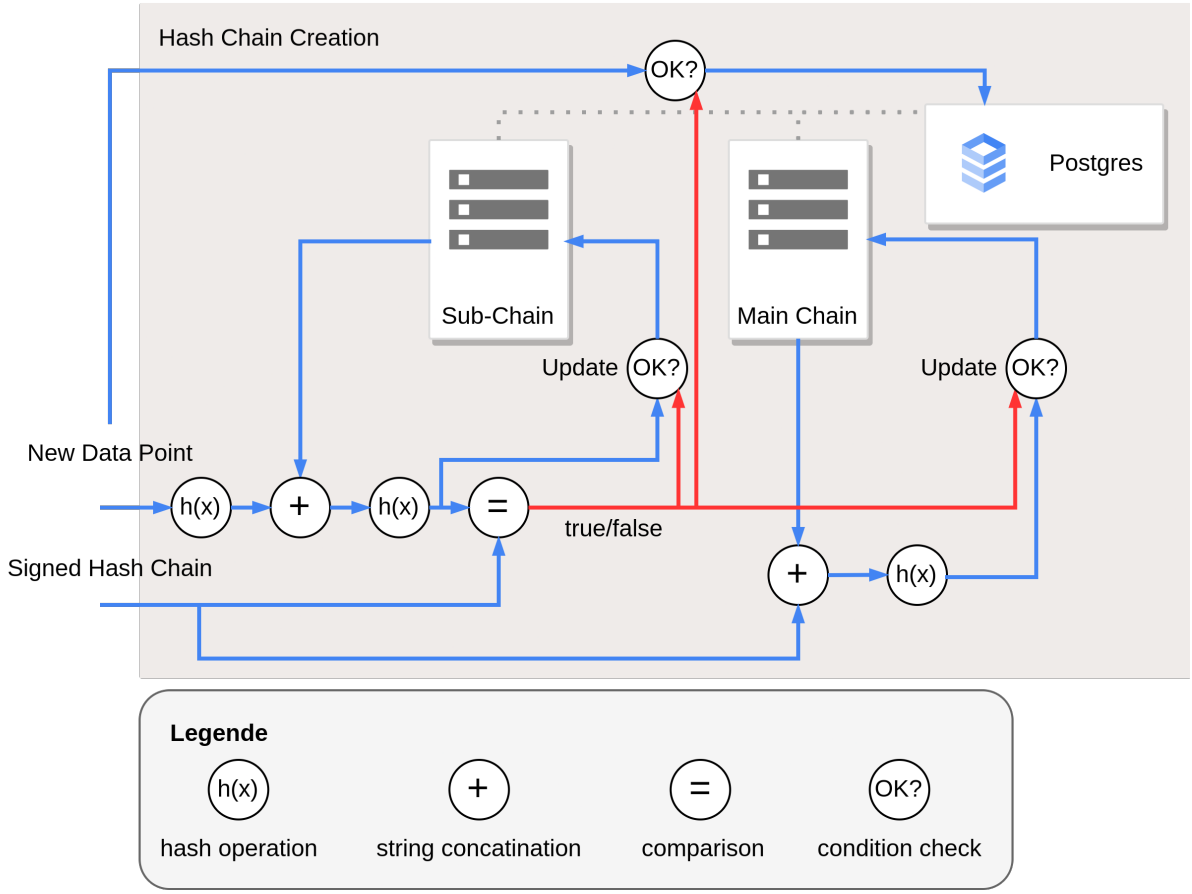

Figure 2.20: Verification of sub-hash-chain and updating of main-hash-chain.

In Figure 2.20 the process of verifying and merging the sub-hash-chain into the main-hash-chain is visualized. The process of verification and updating the main hash-chain works as follows:

1. First a new data point arrives at the base-station. This data point includes the actual data, the tail of the sub-hash-chain, as well as a signature for the tail.
2. The base-station queries the public key of the forensic component that sends the data from its database and then uses that key to verify the signature.

- If the signature does not check out, the base-station rejects the data and displays an error message to the user.

3. The hash value of the new data point is calculated (see h(x) next to the label "New Data Point" in Figure 2.20).
4. The hash value of the new data is concatenated with the last known tail of the sub-hash-chain for the forensic component that sent it (see left "+" operation).
5. The resulting string of that operation is hashed (see second leftmost h(x) operation).
6. The calculated sub-hash-chain value is compared with the signed one.
   - If they don't match, the sub-chain and the main-chain are not updated and the data entry is not saved to the database. Additionally, an error is displayed to the user.
7. If the signed tail of the sub-hash-chain matches the computed one, then the sub-hash-chain value in the base-stations database can be updated.
8. Then the main-hash-chain is updated. The computed sub-hash-chain value is concatenated to the current main hash-chain value (see right "+" operation).
9. The resulting string of that operation is hashed (see second rightmost h(x) operation).
10. Finally, the resulting hash value is saved into the base-station's database.

One question that might arise is, why the forensic component sends the sub-hash-chains tail and not simply a signed hash of the data. After all the signature verifies that the data has not been tampered with and that this data entry can be trusted. While this is correct and the sent data value could be trusted and saved in the database, we lose some of the useful properties the hash-chain technique gives us. This includes protection against denial of message attacks. For example, a bad actor lets us record all kinds of data but if we record something that he does not want us to see, he drops the package in transit and the base-station never gets the data and thus it won't show up in the forensic report later. The hash-chain protects against such attacks. If the attacker drops a data package, we would immediately notice that the sub-hash-chain tails do not match if we do the verification for the next data point. It is important that when the project is set to accumulate all data from the forensic components, the base-station expects one last signed entry of the sub hash-chain tail of the respective forensic component. This is done to ensure that the base-station has actually received all the data from the forensic components and that a bad actor did not just drop all data packages after letting the system run for some time.

Data Batches

In section 2.5.6 the process of verifying the sub-hash-chain and updating the main-hash-chain was explained. However, for performance reasons, the above model was slightly altered to facilitate the use of data batches. One problem is, that the signing process on the YubiKey is quite slow and limits us in the amount of messages that we can sign per second. To facilitate a high number of data points being transferred to the base-station, while keeping the amount of messages low, the forensic components always send their data in batches. The batch size per message can be configured by the user and defaults to ten messages. This also leads to a slightly modified verification process and a divergence from the algorithm presented above, as some additional steps need to be performed for verification:

1. The data, containing multiple individual entries, arrives and the signature of the sub-hash-chain tail is verified.
2. Then the sub-hash-chain value is retrieved, from the forensic component that sent us the data, from the database of the base-station and stored in a buffer.
3. Previously, in step three we computed the hash of the individual data point. Now we calculate the hash of the first datapoint in the batch.
4. The hash value of the first datapoint is concatenated with the sub-hash-chain tail and the resulting value replaces the previously stored sub-hash-chain value in the buffer.
5. Then the resulting string is hashed.
6. Now instead of directly comparing the current sub-hash-chain value with the signed one that the forensic component sent us, we save the sub-chain tail in the buffer and jump back to step ”3.” where the next data point is processed.
7. If all data points have been processed, we finally compare the recomputed sub-hash-chain tail stored in the buffer with the one the forensic component sent us. If they match we can just continue normally like in the other algorithm and start saving the data and update the main-hash-chain.

Full Example

Using the architecture shown in Figure 2.20 a full example (including the batches) is given.

Robot:
First, we examine the process of the ”sub-hash-chain” creation on the robot side. Note that MD5 hashes are used here again. They are not used because they are secure in any way (as

already discussed above), but simply because they are very short and thus nicer to display than longer SHA256 or SHA512 hash values.

Table 2.1: Example of how data is stored on the robot side, along with the creation of the hash-chain.

Data	Hash	Hash-Chain
12345	827ccb0eea8a706c4c34a16891f84e7b	827ccb0eea8a706c4c34a16891f84e7b
hello	5d41402abc4b2a76b9719d911017c592	ddf9c2336b3759423d068728bdc122e7
world	7d793037a0760186574b0282f2f435e7	08be48c3a386a7d0c02aa5faa1cfe662
test1	5a105e8b9d40e1329780d62ea2265d8a	f8d497c445501920890470 56b8b5d30e

12345 : The first entry in the table is not a real data entry. It is simply the random starting value of the hash-chain. This is also why the columns "Hash" and "Hash-Chain" match.

hello : The second entry now contains valid data. As before, we simply build a new hash-chain entry by calculating the hash value of the data and then appending that value to the hash-chain:

a) h("hello") = "5d41402abc4b2a76b9719d911017c592"

b) Lookup previous hash-chain tail: 827ccb0eea8a706c4c34a16891f84e7b

c) Calculate the new hash-chain tail, by concatenating the new data hash to the tail and hashing the resulting string:
h("827ccb0eea8a706c4c34a16891f84e7b5d41402abc4b2a76b9719d911017c592") = "ddf9c2336b3759423d068728bdc122e7"

world : The process for this data entry is the same as for "hello", only this time we take the new tail that we calculated in the previous step.

test1 : The same applies for "test1".

Robot - Data Transmission

When the user wants to "finalize" the project, all forensic components are instructed to stop gathering new data and to send all their collected data over to the base-station. The sending process for the robot is the same as for any other forensic component.

Looking back at Table 2.1, we can see that there are three data entries (the first entry is a random starting value for the hash-chain, which is given to the robot by the base-station)[3]. As

[3]Note that in the real application, the starting value of the hash-chain is not random, but is a hash value that is derived from the public keys used in the project.

already mentioned in section 2.5.6 the data entries are not transferred one by one, but rather as batches of data. As the whole table in this example is rather small, it will be transferred as one batch. This means all three data entries will be transferred in one request.

The forensic components communicate with the base-station via the HTTP protocol. The base-station has multiple API endpoints that can be used to transfer the current status of the component, error messages, or other metrics that the user can view in the web interface. The forensic components can upload their gathered data at an API endpoint called */api/data*. Using a *POST* request, the components can upload their stored data using this endpoint. The data structure, which the endpoint expects in the *POST* request, can be seen in Listing 2.1.

```
class Data(BaseModel):
data: list[dict] = None
signature: str = None
last_entry: Optional[str] = None
last_entry_signature: Optional[str] = None
hash_chain_name: str = None
```

Listing 2.1: Data model for base-station API endpoint: *POST/api/data*

This data model corresponds to how the JSON body of the *POST* request should look. If a forensic component sends data that does not conform to the required model, the HTTP status code ”422 Unprocessable Entity” is returned.

- **data**: The ”data” entry contains a list of all data records. The base model can be seen in Listing 2.2. Each record contains the actual data, a hash of the data, the hash-chain value, and a timestamp. Note that this is not a third-party timestamp, but simply a timestamp that uses the robot’s system time and is measured when the data is recorded.
  - **data**: In the ”data” field the actual data of the record is stored.
  - **hash**: A hash value of the ”data” field.
  - **hash_chain**: The hash-chain tail after this data record was added to the hash-chain.
  - **timestamp**: The timestamp field corresponds to the system time at the point in time when the data record was measured.
- **signature**: In this entry, the signature from the YubiKey is stored. The value that is signed is always the hash-chain entry of the last data record in the data list.

- **last_entry**: The base-station must know when the last data batch arrives. This is because otherwise a bad actor could deny the last message and the base-station would not get the ”real” end of the hash-chain. Also, it is a requirement that each forensic component, must eventually send a data batch where the ”last_entry” field is present and set to a random value. Otherwise, the base-station cannot validate the whole hash-chain from the first to the last entry and won’t know if data is missing. The user cannot finalize and export the project until the last entry of every forensic component has been sent. If the last entry package has been dropped by a bad actor the forensic component will continually try to send it again.
- **last_entry_signature**: This field contains a signature of the hash value of the concatenation of the final hash-chain tail and the hash of the random value in the ”last_entry” field.
- **hash_chain_name**: The base-station stores the hash-chain tail of each sub-hash-chain. These hash-chains are given a name during the initialization process when initializing the individual YubiKeys.

```
class RobotData(BaseModel):
    data = TextField()
    hash = TextField()
    hash_chain = TextField()
    timestamp = TextField()
```

Listing 2.2: Data model for the robots data entries

For sending the example data from Table 2.1 over to the base-station, the JSON object would look like in Listing 2.3. Note that in the example the ”signature” field is not a valid signature and only shows which value would be signed by the YubiKey.

```
{
    "data":[
    {
        "data":"hello",
        "hash":"5d41402abc4b2a76b9719d911017c592",
        "hash_chain":"ddf9c2336b3759423d068728bdc122e7",
        "timestamp":"1702899443"
    },
    {
        "data":"world",
        "hash":"7d793037a0760186574b0282f2f435e7",
        "hash_chain":"08be48c3a386a7d0c02aa5faa1cfe662",
        "timestamp":"1702899448"
    },
    {
        "data":"test1",
        "hash":"5a105e8b9d40e1329780d62ea2265d8a",
        "hash_chain":"f8d497c44550192089047056b8b5d30e",
        "timestamp":"1702899498"
    }
    ],
    "signature":"SIGN(f8d497c44550192089047056b8b5d30e)",
    "last_entry":"RANDOM_VALUE",
    "last_entry_signature":"SIGN(h(f8d497c44550192089047056b8b5d30e +
    h(RANDOM_VALUE)))",
    "hash_chain_name":"robot"
}
```

Listing 2.3: JSON body of POST request to */api/data* with the example data from Table 2.1

In Listing 2.3 the JSON object for the POST request to */api/data* can be seen. The single data entries are simply put into a list in the order they were recorded. The first entry in Table 2.1 is ignored, as it is only the starting value for the hash-chain, which is already known by the base-station. Also, note that the ”hash_chain_name” value is set to ”robot”. This is important because the base-station keeps track of all the individual hash-chains on the component and when submitting the data needs to ”catch up” with its stored tail of the sub-hash-chain. There is exactly one hash-chain per YubiKey and their names are given by the user when ini-

tializing the project. The base-station only accepts the signed data from one specific YubiKey for one specific hash-chain. For example, the robot cannot sign messages for another hash-chain but the ”robot” hash-chain.

Base-Station - Data Verification

After the data has been sent to the base-station, it must be verified before being stored in the database. The verification process that can be seen in figure Figure 2.20 needs to recompute the hash-chain to verify it. The base-station has an entry in a table called ”hash-chains”, where the tail of each sub-hash-chain for each forensic component of the system is stored. In the field ”hash_chain_name” the value ”robot” is given, so the base-station would look at the tail of that sub-hash-chain in its database and then start the verification process:

1. Lookup stored hash-chain value for the sub-hash-chain ”robot”.
   Last known value is: 827ccb0eea8a706c4c34a16891f84e7b
2. Take the first data entry in the hash-chain and start computing the hash-chain:
   h(”hello”) = ”5d41402abc4b2a76b9719d911017c592”
3. Calculate the new hash-chain tail, by concatenating the new data hash to the tail and hashing the resulting string:
   h(”827ccb0eea8a706c4c34a16891f84e7b5d41402abc4b2a76b9719d911017c592”) = ”ddf9c2336b3759423d068728bdc122e7”
4. Take the second data entry and continue:
   h(”world”) = ”7d793037a0760186574b0282f2f435e7”
5. Calculate the new hash-chain tail, by concatenating the new data hash to the tail and hashing the resulting string:
   h(”ddf9c2336b3759423d068728bdc122e77d793037a0760186574b0282f2f435e7”) = ”08be48c3a386a7d0c02aa5faa1cfe662”
6. Take the third data entry and continue:
   h(”test1”) = ”5a105e8b9d40e1329780d62ea2265d8a”
7. Calculate the new hash-chain tail, by concatenating the new data hash to the tail and hashing the resulting string:
   h(”08be48c3a386a7d0c02aa5faa1cfe6625a105e8b9d40e1329780d62ea2265d8a”) = ”f8d497c4455019208904705 6b8b5d30e”
8. Verify the signature of the sent hash-chain tail and compare it with the computed one.
   - If the signature can be successfully verified and the signed hash-chain tail matches the computed one. The verification of the data is successful.

- If the signature cannot be verified or the hash-chain tails do not match, the verification process is unsuccessful. The data is not stored in the database and the user is informed of the problem. The data is not deleted but attached to the error report shown to the user.

9. If the verification is successful, the main-hash-chain is also updated. The main-hash-chain spans across all different sub-hash-chains, as can be seen in figure Figure 2.20.[4] Calculate the new main-hash-chain tail, by concatenating the new sub-hash-chain tail to the main hash-chain tail and hashing the resulting string: h("7ddf32e17a6ac5ce04a8ecbf782ca509f8d497c4455019208904705 6b8b5d30e") = "f225f964374062afb86c1e828a151c4b"
10. After this new main-hash-chain tail has been computed, the base-station signs the value using its attached YubiKey.
11. Finally, the main-hash-chain tail and the sub-hash-chain tail are updated in the database. Also, all the data records are inserted into the database.

It's important to note that the sent hashes and intermediate hash-chain values are not used during the verification process, but that everything is recomputed from the data. This is because the software cannot trust the sent hash values or the intermediate hash values, as a bad actor could simply change the data and leave the hash values as they are. This way the base-station would not notice that a change occurred. This data can be omitted from the sent data batch and only the data fields and the last hash-chain entry need to be present in the data list. However, in the current software implementation, these values are still sent for debugging purposes.

2.5.7 Data Storage

Both on the base-station and the robot side a Postgres database is used to store all persistent data. The Postgres instance is started together with the other Docker containers using docker-compose. The containers can access the database using the name of the Postgres container (by default "db") and the password (by default "pass"). These options, as well as multiple other ones, can be modified by the user. It is recommended that the default password is changed, however, it is not a requirement, as the database can only be accessed from within the Docker network. This means a bad actor would already need to be able to execute docker containers locally on the machine, which would likely also mean that the attacker could simply read the .env file with the password inside (or attach to the docker container and find out the password by checking the ENV variables) and gain access to the database using the password.

[4] Note that in this example, the main hash-chain tail is simply the MD5 hash value of the word "random", which would be the starting value for the hash-chain.

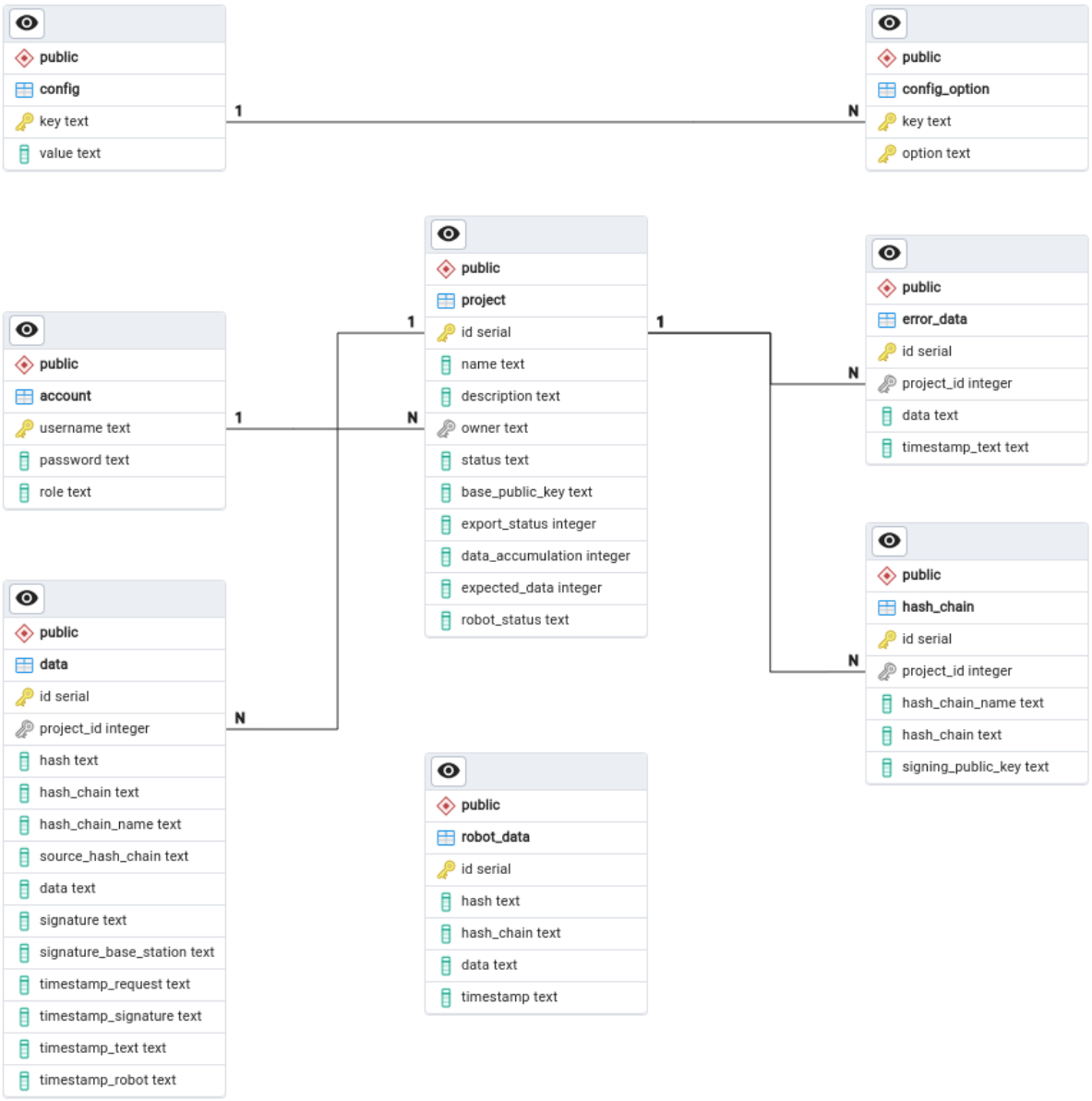

Figure 2.21: ERD (entity relationship diagram) of the Postgres database(s).

Base-Station - Project Table

The ”project” table on the base-station component holds all the information about a single project. When data is sent to the base-station it is always associated with the currently ”active” project (see section 2.5.5 for a description of the project states).

- **"id"**: This is a unique serial number identifying the project.
- **"name"**: The name of the project is given by the user during the initialization process (see section 2.5.5).
- **"description"**: Along with the name, the user can also include a description of the project.
- **"owner"**: This foreign key links the project to an account in the web interface.
- **"status"**: This text field specifies the current status of the project. (The project states are described in section 2.5.5)
- **"base_public_key"**: Public key of the base-station's YubiKey, which is generated during the project's initialization process. (Note that the robot's public key is stored in the table "hash_chain", see section 2.5.7)
- **"export_status"**: This is an integer value between 0 and 100. If the value is 0, this means the project has not been exported yet. If the value is between 1 and 99, the export is in progress (this value gets updated periodically during the export process and indicates the progress of the export). If the value is 100 the project is fully exported and the export can be downloaded by the user.
- **"data_accumulation"**: This is an integer value between 0 and 100. If the value is 0, this means that the project is not accumulating data yet. If the value is between 1 and 99, the accumulation is in progress (this value gets updated periodically during the accumulation process and indicates the progress of the data accumulation). If the value is 100 the project is finished accumulating data and is locked.
- **"expected_data"**: This value is gathered as debug information from the forensic components and is used to calculate the progress for the "data_accumulation" value. However, it is only used for displaying the progress to the user and the base-station does not expect this value to be accurate. The data accumulation is only finished, when all the different forensic component send their final data batch (a data batch with the "last_entry" set to a random value).
- **"robot_status"**: In this text field the status instruction for the robot is saved for the project. For example, one instruction is "send data", which means that the robot must send all the data that it has collected to the project.

Opposed to the public key of the base-station, the public key of the robot and the other forensic components, are stored in the "Hash-Chain Table", see section 2.5.7.

Base-Station - Data Table

In the data table, the base-station stores all of its recorded data. After the verification process, the data entries are inserted into the table.

- **"id"**: This is a unique serial number identifying the data entry.
- **"project_id"**: This foreign key associates the data record with a project.
- **"hash"**: Hash of the data entry as it was received by the robot.
- **"main_hash_chain"**: Main-hash-chain entry, available if it was computed during the verification process, otherwise it will be null.
- **"sub_hash_chain_name"**: Name of the forensic component from which the data was sent to the base-station. This also corresponds to the name of the sub-hash-chain entry in the hash-chain table.
- **"sub_hash_chain"**: Value of the sub-hash-chain.
- **"data"**: The actual data of the record.
- **"signature_base_station"**: The signature of the base-station for the "main_hash_chain" value (if it was computed, otherwise it will be null).
- **"signature"**: The signature of the robot for the "sub_hash_chain" value (if it was computed, otherwise it will be null).
- **"timestamp_request"**: Base64 encoded .tsq (**T**ime **S**tamp Re**q**uest) file of the 3rd party timestamp request. If a third-party timestamp is not added to the data record, this value will be null.
- **"timestamp_signature"**: Base64 encoded .tsr (**T**ime **S**tamp **R**esponse) file of the 3rd party timestamp request. If a third-party timestamp was not added to the data record, this value will be null.
- **"timestamp_text"**: Parsed timestamp from the .tsr file. (This value is buffered because otherwise, the .tsr file would need to be parsed and read every time the user views the data records in the web interface).
- **"timestamp_robot"**: This is the timestamp the robot attaches to the data.

Base-Station - Hash-Chain Table

- **"id"**: This is a unique serial number identifying the hash-chain entry.
- **"project_id"**: This foreign key associates the data record with a project.
- **"hash_chain_name"**: Name of the sub-hash-chain. The name is given to the hash-chain by the user when adding the data source and initializing its corresponding YubiKey.
- **"hash_chain"**: The actual tail of the sub-hash-chain.
- **"signing_public_key"**: The public key of the YubiKey that is trusted to sign the data from the forensic component.

Other Tables

Other tables on the base-station are:

- **account**: Used for account management. Accounts have a name, a hashed password, and a role. Currently, the only supported role is "admin", which has access to all projects and all settings. The software will prompt the user to create an admin account the first time it is started. The user must then supply a username as well as a password for the account. Currently, only the admin account can be created on the platform.
- **error_data**: This table contains all the data records that failed the verification process. The data can be viewed by the user in the web interface. It contains the data record, along with some additional data like the IP address of the device that sent the data, the project ID the data belongs to, etc.
- **config**: This table is used to store config parameters that the user can change in the web interface.
- **config_option**: The config option table is for config entries that only support a few specific values. In this case, the user can select one of the config options for the config entry. For example, the "netcap" (network capturing) component reads a config value on which interface it should capture the data. For this purpose, the component inserts all available interfaces in the config option table and the user can choose one in the web interface.

Robot - Data

The robot is the forensic component provided by default to the project. As already mentioned, more trusted YubiKeys, and thus data sources, can be added to a project. These additional forensic components can be developed by the user and new data recording methods can be added to the project. The robot component can be seen as a guideline on how to implement these other components. In the "data" table, the robot stores all its recorded data, before sending it over to the base-station component.

- **"id"**: This is a unique serial number identifying the hash-chain entry.
- **"hash"**: This is the hash value of the data. The data hash is computed as soon as the data is recorded.
- **"hash_chain"**: This is the sub-hash-chain value for the data entry. It is computed directly before the data is inserted into the database.
- **"data"**: The actual data of the record (for example, CAN bus or ROS2 message).
- **"timestamp"**: A timestamp of when the data was recorded. This is not a third-party timestamp, but simply the robot's system time at the moment the data was recorded.

2.6 Test

To evaluate the software and test how much data the implementation can handle, example-data was prepared and fed to the individual forensic components. The test data includes 50.000 random CAN bus messages and a 4 GiB ros2 bag recording that was played back to capture it again. The point cloud used for testing can be seen in Figure 2.22. It was recorded using a Zed2i camera, the details of the bag file are listed in Listing 2.4. The final recording of the test map has 6647184 points and was recorded at an accuracy of 0.02 meters. In the 66 messages, in which the map is published in the bag file, the size of the map gradually increases (as the Zed2i camera was moved around to map the environment), until it reaches its full size in the final message.

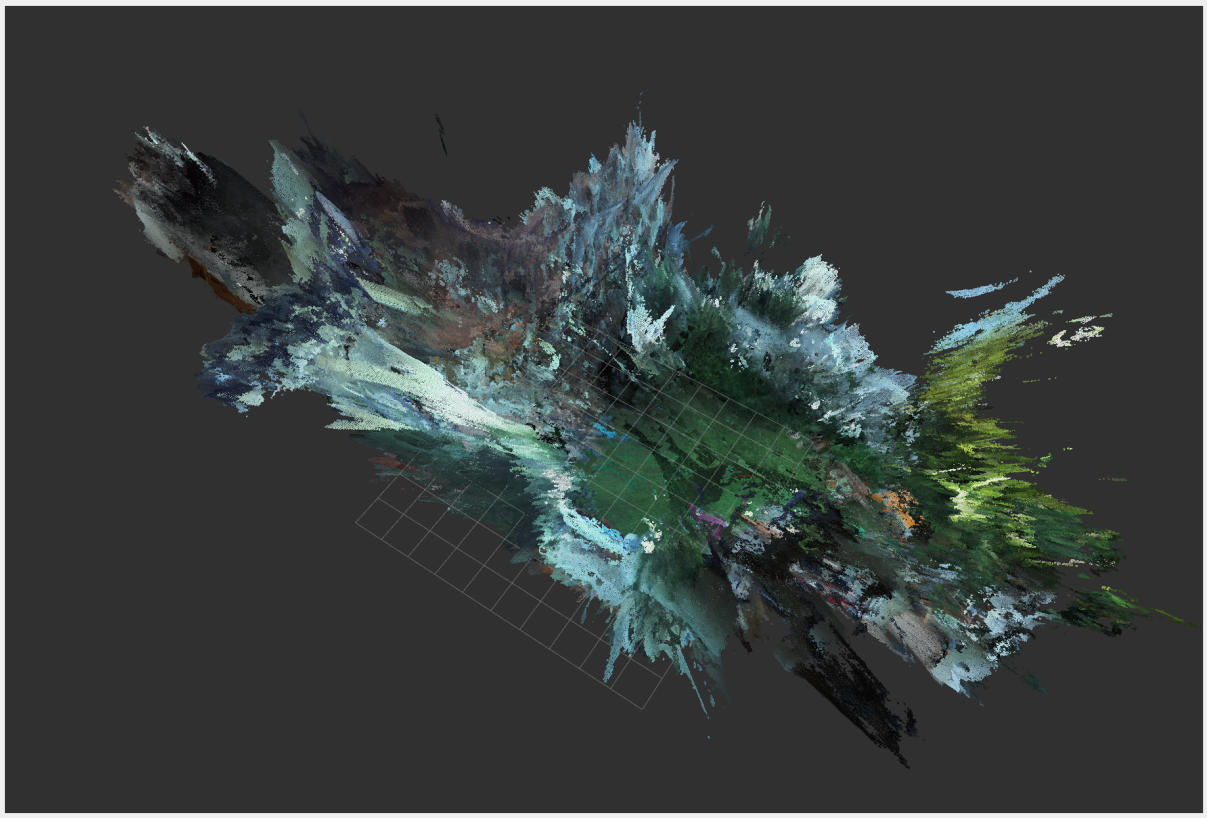

Figure 2.22: Point cloud used for testing.

In the test setup, the base-station and an additional computer running the robot's software were connected over an Ethernet connection. Additionally, the CAN bus reader was plugged into the computer running the robot's software. To write messages on the CAN bus, a second USB2CAN device was plugged into a third machine, which was used to generate the random CAN bus messages. The test was configured to take place over a time span of 20 minutes. The ros2 bag file was played back four times during that period, and the computer writing the messages to the CAN bus was configured to write 42 random messages per second[5]. After 20 minutes of the simulated "inspection", the project was finalized in the base-station's web interface.

[5] $42 * 20 * 60 = 50400$ messages

```
> ros2 bag info rosbag2_2023_07_02-19_09_58_HD2K_15_002

Files:             rosbag2_2023_07_02-19_09_58_0.db3
Bag size:          4.2 GiB
Storage id:        sqlite3
Duration:          272.632s
Start:             Jul  2 2023 19:10:07.713 (1688317807.713)
End:               Jul  2 2023 19:14:40.346 (1688318080.346)
Messages:          81547
    Topic information: Topic: /zed2i/zed_node/path_map |
        Type: nav_msgs/msg/Path |
        Count: 6100 | Serialization Format: cdr
    Topic: /zed2i/zed_node/mapping/fused_cloud |
        Type: sensor_msgs/msg/PointCloud2 |
        Count: 66 | Serialization Format: cdr
    Topic: /zed2i/zed_node/left/image_rect_gray/compressed |
        Type: sensor_msgs/msg/CompressedImage |
        Count: 3990 | Serialization Format: cdr
    Topic: /zed2i/zed_node/left/camera_info |
        Type: sensor_msgs/msg/CameraInfo |
        Count: 7983 | Serialization Format: cdr
    Topic: /zed2i/zed_node/pose_with_covariance |
        Type: geometry_msgs/msg/PoseWithCovarianceStamped |
        Count: 3993 | Serialization Format: cdr
    Topic: /zed2i/robot_description | Type: std_msgs/msg/String |
        Count: 1 | Serialization Format: cdr
    Topic: /tf | Type: tf2_msgs/msg/TFMessage |
        Count: 59414 | Serialization Format: cdr
```

Listing 2.4: Output of the "ros2 bag info" command when analyzing the test bag file.

2.6.1 Observations

During the test, the following observations were made:

- The CAN bus reader was able to correctly record all messages, and the robot was able to process them (e.g. compute sub-hash-chain, save incoming data into the database, etc.).

- The ros2 message collector was able to correctly capture all messages published on the "/zed2i/zed_node/mapping/fused_cloud" topic.[6]
- After the 20-minute recording period, the project was finalized, and the robot started sending all recorded messages to the base-station.
  - The only observed bottleneck is the speed with which the YuibKey can sign messages. In the current implementation, the YubiKey can sign one message every 300 milliseconds. However, this is not a problem since the robot does not sign every single recorded message, but rather sends them in batches over to the base-station. That means that only each batch needs to be signed. The size of the batches can be configured by the user and set to ten messages by default. For such a huge recording, a batch of only ten messages is too small, and the software will adjust the value automatically (if the user has not manually specified one).
  - In this test, the robot and the base-station were connected using an Ethernet cable, so transferring the data was not a bottleneck. If Wi-Fi connections are used, the lower bandwidth can have an impact on the speed of the data export from the robot to the base-station. In the test scenario, a total of about 20 GiB have been transferred.
  - No bottleneck was observed on the base-station during the verification of the hash-chain and the signatures.

[6] In Listing 2.4 it can be seen that on this topic the point cloud is published.

Chapter 3

Data Collection

3.1 Forensic Readiness of Data

The ”forensic readiness”[53] of data indicates how useful data is for forensic analysis. When incidents happen, it usually takes significantly more time to sort through all the data and figure out what happened, than for a bad actor to perform an attack. While it is easy to collect data, it is not easy to collect (only) useful data. For instance, when collecting the network traffic to a server that is under attack, most of the traffic could be genuine and only a few packages may contain malicious data. Thus, it is imperative, that the collected data is of good quality and useful to the investigator.[40] Several steps can be taken to ensure a good quality of the collected data, for example, the time of the recording devices should be synchronized across the network. Furthermore, data should be time-stamped, so that an investigator can reconstruct the timeline of an attack. These are only a few mechanics that ensure the collection of useful forensic evidence, a comprehensive list can be found in [53].

3.2 Data Sources

In the current implementation, three different data sources are supported. Additional data sources can be added by the user, who has to add additional YubiKeys during the projects' initialization process, for reference see section 2.5.5. This YubiKey can then be used for signing messages before sending them over to the base-station. The implemented forensic components that serve as data sources can be used as reference for the implementation of additional ones. These additional components could then gather other kinds of data and add them to the forensic recording.

3.2.1 CAN Bus

The CAN bus (Controller Area Network bus) is a robust and widely used communication protocol primarily employed in the automotive industry for connecting various electronic control units (ECUs) within a vehicle. The primary purpose of the CAN bus is to enable communication among different components in a vehicle without the need for a host computer. It allows various electronic systems to exchange information and control signals in a fast and efficient manner. In the INFRASPEC project, the robot's motors are controlled via a CAN bus.[9][50]

One important aspect of each CAN bus is the "baud rate". The baud rate specifies the speed at which new symbols arrive at the bus. For classical CAN, this speed is usually 1 mbit/s. And for CAN FD it is 5 mbit/s. The CANF FD (Controller Area Network Flexible Data-Rate) is an extension of the CAN bus protocol. It was designed to fulfill higher bandwidth requirements, as cars got more complex and required more data to be transferred between the engine units. It also supports larger message frames and more data can be transferred in one single message. Another limiting factor of a CAN network is the maximum cable length. A cable of a CAN bus should not be longer than 500 meters (which will result in a maximum baud rate of 125 kbit/s). For comparison, at a cable length of 40 meters, CAN still support the whole 1 mbit/s. Another requirement for a CAN bus is that is terminated with a 120 Ohm resistor.[9]

Messages

Similar to messages sent over Ethernet, CAN frames consist of a header, the data, and a footer. A CAN frame always starts with a "0", which tells other nodes in the CAN network that the device sending the 0 wants to send a CAN message. Then an "frame identity", which is 11 bits in size, is put into the message. After the ID, the "remote transmission request" field specifies if the node wants to send new data to the other nodes, or if it requests data to be sent to it. Inside the next field, called "control", the size of the data section is specified. Also located in the control field is the identifier extension bit. Behind the control field, the data field is located. A CRC (cyclic redundancy check) field is located behind the data, which ensures the integrity of the data, similar to how it is done in Ethernet packages. The second last field in the frame is the "ACK" slot. This is used for nodes to signal that they have acknowledged and received the data correctly. Finally, the "EOF" (End of Frame) field, signals the end of the CAN bus frame.[9]

3.2.2 ROS2 Messages

ROS stands for "Robot Operating System" and is a "publisher and subscriber" based message broking system, as was explained in section 2.3.5. Its main application is the interoperability

of individual components of a robot. The software to operate single components (for example, the mechanical arm on the INFRASPEC robot) is implemented using ROS2 ”nodes”. These nodes can then publish or subscribe to specific topics. For example, one topic could be called ”/arm/pos”, where the node would publish the current position of the robot’s arm. Another node can then ”subscribe” to that topic and will get all the messages that were published on said topic passed to a callback function. The data format, declaring what format can be published on the topic, must be specified when the topic is created. The message format is specified by the ”ROS2 messages”. ROS2 messages are defined in ”.msg” files and usually consist of multiple lines with ”field type” and ”field name” pairs.[46]

```
1 string name
2 int32 age
```

Listing 3.1: Example of a ROS2 .msg file.

A simple example for a .msg file can be seen in Listing 3.1. In the message, a field called ”name” with the data type ”string” and a field called ”age” with the data type ”int32” is defined. When the ROS2 node is compiled, the messages are automatically converted into usable classes for either Python or C++. This is another advantage of ROS2, as it supports both programming languages and the messages can easily be interchanged between nodes written in Python or C++.

As already stated, nodes can subscribe to or publish messages on topics. This means that usually components continually publish recorded sensor data and subscribe to topics for input from the operator. For example, the robot’s arm would subscribe to a ROS2 topic where it would get a 3D position that the arm should move to. Subsequently, it would publish the motor ticks it recorded from its attached hardware. All of this information is relevant for forensic data recording, as the movements of the robot’s arm could potentially damage other components in the supply shaft. If the user moves the arm inappropriately, that information must be recorded and be available in the forensic data package for later analysis.

QoS Policy

The QoS (Quality of Service) policies in ROS2 pose an additional challenge for recording all the data that we need. Usually, not all messages are passed to the node which subscribes to the topic. For example, point cloud messages can be very large and are usually published continually every few seconds. If the subscribing node is still receiving a message, while a newer one is already available, the node cannot keep up with processing the messages. Depending

on the QoS policy, the next message will either get saved into a buffer and the node will receive it immediately after finishing processing the current message, or it will be discarded. In the latter case, the node would not get to see the message. This is a problem for our forensic data recording, as we need to capture all the messages and want to avoid losing potentially important data at all costs. In ROS2, a variety of different QoS policies exist.[43]

- ”History”: Specifies how many messages should be buffered. This is especially important for the case described above, where a node temporarily cannot keep up with processing the incoming message.
  - ”Keep last”: This option will only store the last message. Messages can be lost if a node cannot process the message fast enough.
  - ”Keep all”: This will store all incoming messages. However, a message can still be lost, if the resource limits of the underlying middleware are exhausted.
- ”Depth”: This specifies a ”Queue size”, which is the amount of ”last messages’ to keep. It is only used if the ”History” policy is set to ”Keep last”. For example, it can be specified to keep the last five messages in a buffer.
- ”Reliability”: This policy specifies if the messages should be delivered, even if the network is under high load.
  - ”Best effort”: This option will potentially drop messages if the network is under stress and the message can’t immediately be sent.
  - ”Reliable”: This option guarantees that the message will eventually be delivered to the node, even if the sending must be retried multiple times.
- ”Durability”: Specifies how the messages are saved.
  - ”Transient local”: The publisher will be responsible for persisting messages.
  - ”Volatile”: No messages will be persisted.
- ”Deadline”: Using the parameter ”duration” this policy sets the expected maximum amount of time between two messages which are published on the same topic.
- ”Lifespan”: Using the parameter ”duration” this policy sets a maximum time limit for the delivery of a message. If the message cannot be delivered within the specified deadline, it will be dropped.
- ”Liveliness”: This pertains to the health or ”liveliness” of a node in the network.
  - ”Automatic”: When using this option, the ROS2 system will consider the publishing node to be alive for the ”Lease Duration”, each time it publishes a new message.

    - ”Manual by topic”: The node will be considered alive for the ”Lease Duration”, each time it publishes a message on a specific topic. This is useful for nodes that periodically publish messages on a specific topic (for example, a sensor publishing recorded data every second).
- ”Lease Duration”: Specifies the duration for which a node is considered alive after some event has happened.

As we want to collect as much information as possible, the QoS policy must be configured to reflect that. A possible QoS policy set on the subscriber node which collects as many messages as possible could look as follows:

- History = ”Keep all”: This will lead to all incoming messages being saved in a buffer, even if the subscribing node is still busy processing the last message.
- Depth: This does not have to be set, as all messages are kept.
- Reliability = ”Reliable”: This will tell the messaging middleware to keep trying to deliver the message, even if the network is under high load.
- Durability = ”Transient local”: Using this option, messages can be persisted, as opposed to setting it to volatile. However, if the publisher is set to ”Volatile” this QoS policy will lead to a problem, as those two options are not compatible with each other and no messages will be sent at all. If possible, the publisher must also be set to ”Transient local” for the messages to be persisted properly. If this is not possible, we must set the value to ”volatile” and keep in mind that some messages could be lost.
- Deadline: This value can be left at the default value for a good estimate that fits most use cases.
- Lifespan: This value can be left at the default value for a good estimate that fits most use cases.
- Liveliness = ”Automatic”. This way, the publishing node will be considered alive, if it publishes any message on any topic.
- Lease Duration: This value can be left at the default value for a good estimate that fits most use cases.

3.2.3 Network

The network is the third data source that the current software implementation uses. The local network is an important source of information, as all components of the whole system

are connected to it. The base-station and the robot use the network for communication. For example, sensor data from the robot is sent over to the base-station and the base-station issues commands to the robot. These commands are usually ROS2 messages, which are also captured using the ”ROS2 message collector”, for reference see section 3.3.2. However, the recorded network data can also contain additional information, for example on other active devices in the network. The network recording can also be used to gain insight if a bad actor was present in the system or if there were any other anomalies.[7][58]

To record the network data, the Wireshark package sniffer is used. Package sniffers work by setting the network card of the computer into ”promiscuous mode”, which will result in the network card accepting all packages, even those meant for other computers. Then the network sniffer parses the package and stores the information. When inspecting the stored information, the user can see all the data that is contained in the package. For example, the source IP, the destination IP, etc., and of course the actual data of the package. However, the data section of the package could be encrypted if a secure protocol was used. But information like the source or destination IP is still visible, and this metadata alone can already be valuable information for later analysis.[2]

3.3 Collection Methods

3.3.1 CAN Bus

To record the data on the CAN bus, a device is used, which can attach to the bus and read all messages that are transmitted on it. For simplicity, a ”USB2CAN” device was chosen. Once the device is attached to a CAN bus, the reader can be plugged into a USB port and used for reading or writing to the CAN bus.[29] The CAN bus typically has three wires that need to be connected to the CAN bus reader:

- **CAN_RX Cable**: Used for data reception, incoming data signals are accepted on this cable.
- **CAN_TX Cable**: Used for data transmission, this cable enables the sending of data signals.
- **GND Cable**: Serving as the ground connection, this cable establishes the reference point for electrical potential.

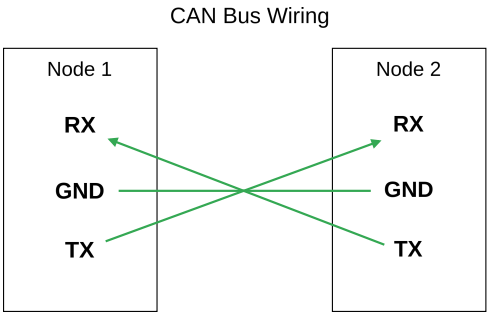

Figure 3.1: Wiring of a CAN bus connecting two nodes.

In Figure 3.1 the wiring of the CAN bus connecting two USB2CAN devices from Figure 3.2 can be seen. Note that the RX and TX connection must ”cross” each other, as the receiver at the one end must be wired to the transmitter of the other node.

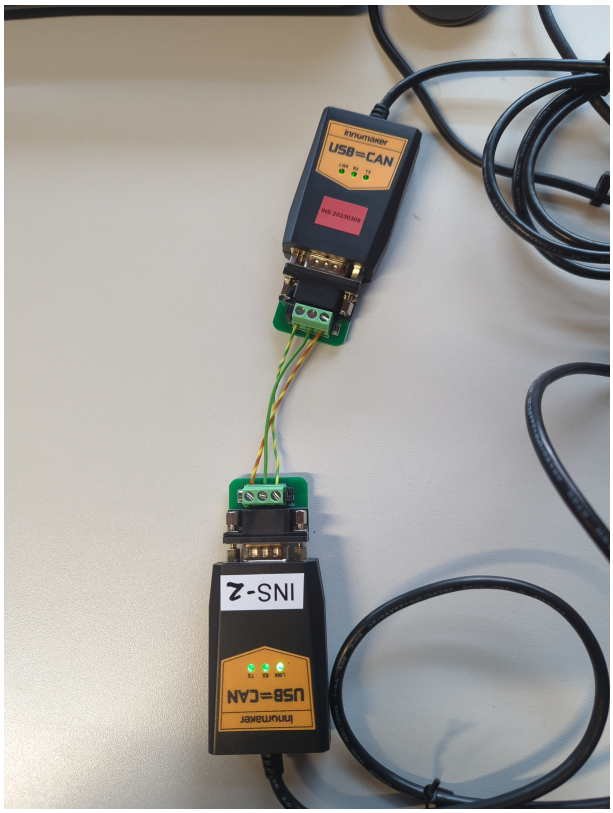

Figure 3.2: CAN bus connection between two USB2CAN devices.

Testing the CAN bus

To implement the collection and to test the data collection for the CAN bus, a simple test CAN bus was created. It can be seen in Figure 3.2 and consists of two USB2CAN devices that are wired together using copper cables. The CAN bus is extremely short, and the cables have a simple isolation so no errors or crosstalk is to be expected. To use the CAN bus, both devices must be configured:

1. The devices should be plugged into a computer that is running an up-to-date Linux version. After they have been wired up and plugged in using USB, the "Link" LED should light up green.
2. In the next step, the CAN devices must be brought up using the "IP" command from the "iproute2" package.[1]
3. To bring up the first device, the user must issue the following command using the terminal:
   `ip link set can0 up type can bitrate 1000000`
   This sets up a link with the name "can0" using a bitrate (baud rate) of 1,000,000.
4. Then we can set up the second device using the command:
   `ip link set can1 up type can bitrate 1000000`
5. If the setup was successful, the RX and TX LEDs of both devices should now light up green.

Both of the USB2CAN devices should now be set up correctly and can communicate with each other. One important thing to note is that, for this test case, it does not matter which device will be "can0" and which one "can1". We are essentially communicating between two USB ports using the CAN protocol. To test the sending and receiving of messages, we can use the following commands, which should be present on most Linux systems:

1. We choose one device that will just listen and print out all the incoming messages. For example, we choose the "can0" device as the listener. To print all incoming data on the "can0" RX line, we can use the "cansniffer" command from the "can-utils" package[2].
   `cansniffer can0`
   Now the terminal should show all the messages that are received by the "can0" interface.
2. To test if the CAN bus works and if the sniffer will print the sent messages, a message can be sent using the "cansend" command. Just as the "cansniffer" command, this one also requires the "can-utils" package to be installed.

[1] Should already be installed on most Debian and Ubuntu derivatives.
[2] Can be installed on most Linux distributions by installing the "can-utils" package. For example, this has been tested under Debian and Ubuntu: `sudo apt install can-utils`.

```
cansend can1 123#DEADBEEF
```

When issuing this command, the TX line of the ”can1” interface should write out the specified message. The ”can frame” that we sent in the example above, has the frame ID ”123” and carries the data ”DEADBEEF”. The data, as well as the ID, is written in hexadecimal and are separated by a ”#” character.

3. At the moment we press enter and issue the send command from above, the TX LED of the ”can1” device, as well as the RX LED of the ”can0” device should blink briefly. This means that some message was transmitted over that cable.
4. Looking back at the receiving side, if everything is working correctly, the cansniffer program should print out the sent message.

CAN bus Sniffer Implementation

The implementation of the CAN bus sniffer in the INFRASPEC software is very straightforward. It runs on the robot in its Docker container and publishes read messages onto a RabbitMQ messaging queue. The only configuration parameters of this container need the name of the ”CAN_INTERFACE” and the ”CAN_BAUDRATE”. Taking the example setup from above, we could set the environment variable CAN_INTERFACE to ”can0” and the CAN_BAUDRATE to 1000000. This way the program listens to all incoming messages on the ”can0” interface and then publishes them onto a messaging queue for the signing service. The signing service runs on the robot and builds the hash-chain, signs, stores, and eventually sends over the messages to the base-station, as already described in detail in chapter 2. To read the messages from the CAN bus, the Python library ”python-can” is used.[42] This library supports reading and writing CAN frames to a Linux CAN socket.

```
1 with can.Bus(interface="socketcan", channel=CAN_INTERFACE, bitrate=CAN_BAUDRATE)
  ↪  as bus:
2   while crawl_data:
3       msg = bus.recv(timeout=1)
4       if msg is None:
5           continue
6
7       with RabbitMQConn() as conn:
8           channel = conn.channel()
9           channel.basic_publish(exchange="",
            ↪  routing_key=rabbitmq.Queues.RobotDataCollect.value,
            ↪  body=str(list(msg.data)).encode())
```

Listing 3.2: Very simple implementation of a CAN bus sniffer using the USB2CAN device [29].

The code in Listing 3.2 is a simple implementation of a CAN bus sniffer in Python. In line one, the ”with” statement is used to create a CAN bus reader using the ”socketcan” protocol. The ”socketcan” protocol is a collection of open-source frameworks that implement the use of a variety of CAN bus devices in a standardized way. After the interface has been set up, a loop is executed which runs as long as the sniffer should record new data. In that loop a new message is received (line three) then it is checked if the received message is valid (lines four to five) and finally the message is converted to a string format and published on a RabbitMQ messaging queue, where the signing service on the robot takes over and handles the signing, hash-chain creation, etc.

3.3.2 ROS2 Message Collector

For collecting the ROS2 messages, the software supports two different approaches. As already explained in section 3.2.2, ROS2 uses a publisher and subscriber model. To record as many ROS2 messages as possible, we need a node that subscribes to all the relevant topics and then records the messages sent on the topics. In ROS2, there is a tool that already implements that use case, called ”ros2 bag”. However, this tool will simply create one single file, which contains all the data, but some users might want to have the individual ROS2 messages in the database together with the CAN bus messages and all the other recorded data. For this use case, a configurable ROS2 node has been developed, that listens to the topics and then processes the individual messages, just like the CAN bus messages are processed. When the ROS2

node is used in conjunction with the robot, it can sign the messages, create the sub-hash-chain, etc.

The ROS2 message collector could theoretically be located anywhere in the network, which means that it could also be located on the base-station. However, the robot is a more convenient place to run this component. If the ROS2 message collector were located on the base-station side, all the sensor data from the robot would be sent over the network to the base-station. This causes two problems, firstly it causes additional unnecessary network load while the INFRASPEC system is in operation, and secondly, it will cause the ROS2 message to be included twice in the forensic data package, as the network sniffer also collects the sent data. However, on the robot side, the ROS2 messages can be sent directly from the sensors to the recording component and no additional traffic is caused on the main data link between the robot and the base-station. For further reference, see Figure 2.7.

Using Bag Files

When using the ”bag file” approach, a bash script is provided that handles the starting of the ROS2 bag file recorder and will also sign and send the bag file over to the base-station, so that it is included in the forensic data package. The user must only specify which topics should be recorded, the rest is automatically handled by the provided bash script. The ”bag files” are files that are like records of all ROS2 traffic on the topics. They can even be ”played back” by running ”ros2 bag play FILENAME”, which will start a ROS2 node that publishes all recorded messages on their respective topic at the same time intervals as they were recorded. So for example, the recorded positions of the robot’s arm could be ”played back” and viewed in a ROS2 visualization tool like ”rviz” and the examiners would precisely see how the arm has moved. Even more interesting could be the recording of the commands that were sent to the arm. These could also be played back and used to let the arm move again just like it did before and then see how it behaves in the real world when those commands are issued.[46]

As already mentioned in section 3.2.2, we must override the default QoS policy, to ensure that the subscribing node receives all messages. This is also true for the ROS2 bag file recorder, which is essentially nothing more than a ROS2 node that subscribes to the specified topics and then writes the ROS2 messages to a file format from which they can be read again at a later time. The configuration file to override the default QoS policy can be seen in Listing 3.3. The file is automatically passed to the ROS2 bag file recorder by the provided bash script.[46]

```
history: keep_all
reliability: reliable
durability: transient_local
liveliness: automatic
```

Listing 3.3: QoS override file passed to the ROS2 bag recorder.

When the user wants to finalize the project on the base-station, the ROS2 bag file creator is stopped. Then the file is encoded in base64, put into the robot's RabbitMQ messaging queue and the signing component handles the rest.

Manual ROS2 Subscriber Node

If the user wishes that the individual ROS2 messages should be parsed and included in the forensic data just like the CAN bus messages, a specially written ROS2 node can fulfill that use-case. The ROS2 message collector starts as a ROS2 node on the robot and records messages that were published using ROS2. The user can specify which topics should be recorded and can provide additional formatters for the collected data. For example, the recorded point clouds can be converted from the ROS2 message "sensor_msgs/PointCloud2" to a .pcl file and can even be viewed directly in the base-station's web interface, once they are transferred over. The topics this service records are specified in a Python file that acts as a configuration file:

```
from infra_bridge import processors

CONFIG = [
    {
        "topic": "test",
        "processor": processors.StringProcessor
    },
    {
        "topic": "/mapping/fused_cloud",
        "processor": processors.PointCloud2Processor
    }
]
```

Listing 3.4: Configuration file for the ROS2 bridge component.

Using the example configuration, from Listing 3.4, the ROS2 message collector will listen to a topic called ”test”, where simple string messages can be published and another topic, where a 3D camera will publish the 3D point cloud that it records. The users must supply the ”processors” (apart from the string and point cloud processor, as they are implemented by default) themselves. The processors simply convert the ROS2 data into a string format and then push the data onto the robot’s RabbitMQ queue, where the signing component takes over.

The QoS policy is directly specified in the Python code of the ROS2 node. This is done when creating the message callback, as the QoS policy can be passed to the subscriber creation function.

3.3.3 Network Traffic Collector

The task of the network traffic collector is to monitor one specific network interface and record all transmitted packages on that interface. One popular choice for a package sniffer is ”Wireshark”.[61] The software can record all the network traffic that the network card receives and display it to the user. The network traffic collector does not need a graphical interface for the user. Conveniently, Wireshark can be used without a GUI, by directly using the underlying program that records the traffic. This program is called ”dumpcap” and it can be started from the command line.

The network traffic collector starts the dumpcap program using the subprocess module of Python. The arguments passed to the dumpcap program can be seen in Listing 3.5 and configure the process to write out all recorded network data to ”.pcap” files at an increment of five seconds. Another thread checks if new files have been written and if it finds a new file, it encodes it in base64 and then puts it into the appropriate RabbitMQ messaging queue on the base-station. The base-station then handles the hash-chain creation, signing, etc.

```
1 dumpcap -i {interface} -w /tmp/dumpcap.pcap --ring-buffer duration:5 -P
```

Listing 3.5: Starting the dumpcap program in ring buffer mode.

This will result in the network traffic being split into multiple ”.pcap” files, of which each one contains five seconds worth of data. Users that want to analyze the network traffic, will find it very inconvenient that the traffic is split into so many different files. For ease of use, a script is provided that can merge the ”.pcap” files into one single file again, which can then easily be inspected using wireguard. The Python script is shipped together with the exported data and the verification script.

3.4 Visualization of Data

Different types of data require different techniques for visualization. In Figure 3.3 the web page showing an overview of the project's collected data can be seen. The forensic components can include information about the type of the sent data, for example, the CAN bus data is tagged as "CAN" data. When the web server renders the web page, it checks if a plugin is present to render the data type. If a plugin is found, the data is passed to the plugin, and the HTML content visualizing the data is returned. Plugins are also used to add support for visualizing more complex data types. For example, in the implementation, a plugin exists that can visualize point clouds. The plugin returns an HTML button that reads "View Pointcloud", and can be clicked by the user to view a 3D representation of the data directly in their browser. To visualize the data in the browser, the "three.js" library is used.[55]

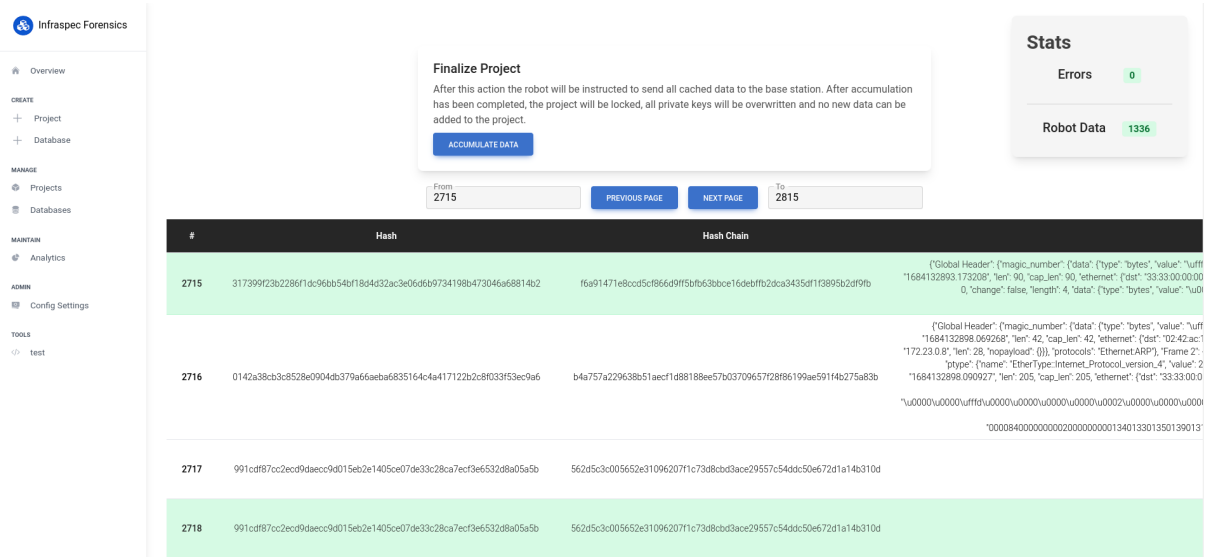

Figure 3.3: Project overview in the web interface.

Chapter 4

Data Processing

4.1 Outline

In the previous chapters the collection of data, as well as a data format for secure storage, were explained. In chapter 2, the design of the forensic system has been discussed and the creation of the "forensic data package" has been shown in detail. The exported forensic data package itself is a Sqlite3 database, that can be queried with common tools. Users might want to automatically analyze this data package. For example, they could perform an analysis of the recorded network traffic using such methods as mentioned in [7] and [58] and which were already discussed in section 3.2.3. The analysis of the data may require a long time or a lot of processing power, which would be an incentive to process the data in the cloud. However, this would immediately spark concerns, as the data that is sent to the cloud provider could be recorded or copied. Even if the cloud provider can be trusted, there might be a bad actor hiding in their network without their knowledge, and we would put our transferred data at risk of being captured by that bad actor. As the data that is recorded by the forensic software includes a lot of information about components of critical infrastructure, the recorded data must be kept confidential. This leaves us with two requirements, firstly the data should be processed in the cloud and secondly, the cloud provider must be unable to gain any insight into the data that is being processed. To fulfill this requirement of secure computation in the cloud, Intel secure guard extensions (SGX) are used.

4.2 SGX - Software Guard Extensions

The SGX technology from Intel is an extension to the x86 architecture to enable the user to create applications that run in "secure enclaves". SGX is a hardware technology that is directly built into (some) Intel CPUs.[60] As such, SGX is a CPU functionality and can protect

the enclave from usual attacks on the OS, the hypervisor, corrupt drivers, and even the BIOS. The technology provides several useful security features to the user:[11]

- Encrypted memory: The program memory of the "trusted" part of an SGX application is encrypted and cannot be accessed by any other application or even by hardware sniffers on the memory bus, as the data is only decrypted inside the CPU. The key for encryption and decryption is also only stored inside the CPU.
- Remote Attestation: When a user runs an enclave program in a server center, they can verify that exactly the program they sent to the cloud provider is executed using the remote attestation procedure. This is necessary to verify that the cloud provider cannot access the memory and that SGX is working properly.
- Sealed Storage: The data and code that make up the enclave program are not secrets. Enclave applications can be reverse-engineered, their data sections inspected, and their code decompiled. If confidential information should be included in the enclave application itself that information must be "sealed". SGX has a sealing process in which the data can be encrypted and only accessed by the enclave program.

Combining all the functionalities listed above, the user can execute a program on a cloud platform without the cloud provider gaining any knowledge as to which data is processed. Furthermore, the user can verify that the program they sent is also the one running in the cloud environment and that the cloud provider did not alter the program in any way.

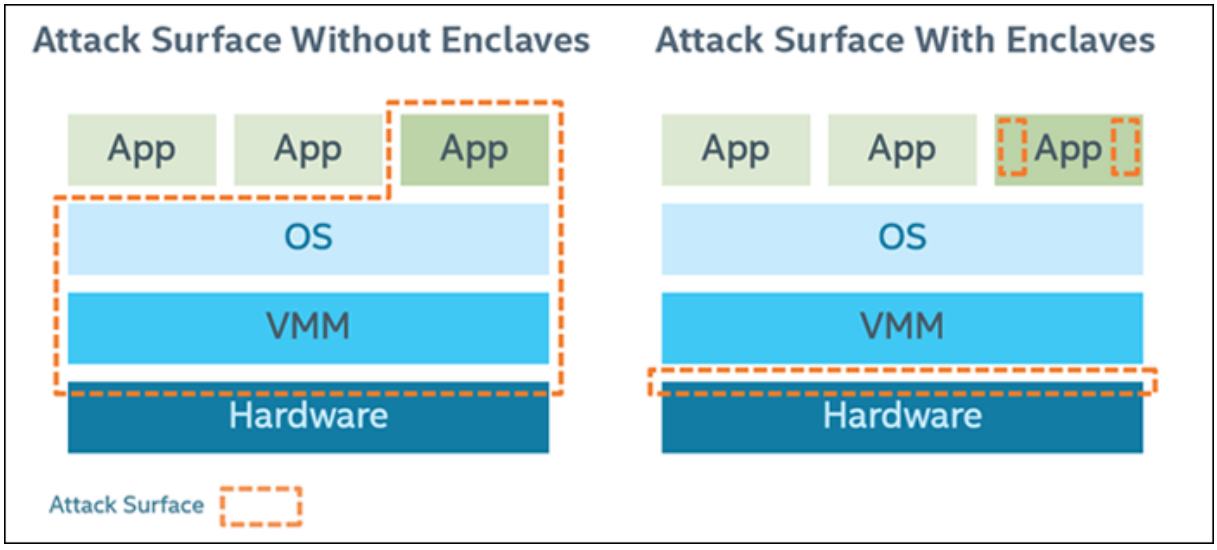

Figure 4.1: Attack surface or normal applications vs applications that utilize SGX enclaves.[34][Overview]

Apart from keeping data and code secure, another primary goal is to reduce the attack sur-

face. Keeping the attack surface as small as possible will reduce the amount of possible attack vectors and increase overall security. In Figure 4.1 the attack surface of a normal application is compared with one using SGX enclaves. The normal application has a larger attack surface, as it relies on the security of the whole application itself, the OS, the hypervisor, and the hardware. If one of these parts contains an exploitable security vulnerability, the application itself is likely also compromised. But this large attack surface does not only include actual attacks. If a bad actor has access to the OS, they could manipulate the application on that level. When running an enclave application, the sensitive parts of the application will be protected and will be running in a secure environment, where it does not matter if the OS or hypervisor is compromised, as the application is executed in a secure environment that even the OS or hypervisor has no access to. The only remaining attack surfaces are parts of the application and the hardware itself. In SGX, the application is split into ”trusted” and ”untrusted” code. The trusted part is always securely executed in the enclave, meaning that its memory content is encrypted, etc. The untrusted part is running just like a normal application, and the two parts can interact with each other with so-called ”E-Calls” and ”O-Calls”. In the E-Calls, the untrusted part ”calls” a function in the trusted part, and for the O-Calls, it is the exact opposite. If there is a bug in the enclave application itself, for example, if a signature validation performed in the untrusted part of the application is trusted, the whole application can still be compromised. The attacks against the hardware remain, even when using enclaves. As the secure environment relies on the hardware to be secure, an attacker who can tamper with the CPU at the time of manufacturing will be able to modify security measures or install a backdoor. However, the ”root of trust” must be somewhere, and the hardware layer is already at the bottom of the hierarchy, as presented in Figure 4.1. For further reference on the root of trust, see section 4.3.2.

4.2.1 Use Cases

Blue-ray

”PowerDVD” by CyberLink is a Windows application that can be used to play UHD Blu-ray discs on computers. It is the only software that is licensed to play UHD Blu-rays, as all other players use dedicated hardware. To play Blu-rays, the proprietary ”Advanced Access Content System (AACS) 2” is used that acts as a DRM (Digital Rights Management) software that should prevent the user from copying the data on the disk. The goal of the PowerDVD application is to hide the AACS2 keys from the user, as some of these keys are used to decrypt the Blu-ray content. If a user obtained the keys, they would be able to play back any Blu-ray on any computer and also copy their content. In [57] the authors found a flaw in the PowerDVD implementation and abused an old insecure Intel processor to form a ”rogue Quoting Enclave”. Usually, platforms that are known to contain a flaw in their SGX implementation are

marked as insecure, and during the attestation process other programs can notice that they are communicating with a potentially insecure enclave. However, the PowerDVDs implementation ignored the ”GROUP_OUT_OF_DATE” status during the attestation procedure and the authors were able to extract the SGX attestation keys using the ”Foreshadow attack”[56]. By deliberately using an unpatched Intel Core i7-6820HQ and constructing a rouge Quoting Enclave to generate the desired quotes for an attack, they were able to contact CyberLink’s AACS2 provisioning service and get the AACS2 key material. Using that key material, they were able to play a UHD Blu-ray on Linux using VLC without any SGX enclaves being required for the playback.

The extraction of the keys was only possible because the PowerDVD application did not reject an SGX implementation that was known to have security issues. So this was not an issue with SGX itself, but with the PowerDVD application ignoring a warning during the attestation process. However, it must be noted that it is at least ”indirectly” a problem with SGX, to be more precise a problem with their update strategy. To fix known SGX vulnerabilities, users must perform a BIOS update and update the microcode of their CPU. Performing these updates is already too hard for some users, as the update process is different across motherboard brands and requires some manual work. For example, downloading the correct BIOS version for the correct motherboard, then saving that binary onto a USB stick, pressing the correct button during the boot process, etc. This is not what the ”typical” user usually does, and from general experience it follows that not every PC user knows the exact model and brand name of their motherboard. However, even if the user can make those updates, the BIOS updates are often rolled out very late, taking up to 100 days or more since the discovery of the vulnerability.[57] During the time spent waiting for a BIOS update, users would be left unable to play their Blu-rays. This is why PowerDVD did not enforce a strict policy of only trusting secure SGX implementations, and their application disregarded the SGX warning. In the near future, PC users won’t be able to play Blu-ray discs anymore using PowerDVD, as the newer consumer Intel CPUs (starting from the 11th generation) don’t support SGX anymore.[14][60]

TTP - Trusted Third Party

One application for SGX, which is presented in [31], is to utilize SGX to create a ”Trusted Third Party” (TTP). A TTP is an entity that is trusted by all participants in a system. However, the single participants are generally distrusting each other and do not want to reveal their secrets to the other participants. If a TTP is present, each participant can simply send their data to the TTP and process it there. In [31] a concrete implementation for a trusted third party is provided by using the SGX technology.

4.3 Confidential Computing

4.3.1 Definitions

The ”Confidential Computing Consortium” (CCC) defines ”Confidential Computing” as the protection of data in use (= data being processed) by hardware-based attested Trusted Execution Environments.[10] A ”Trusted Execution Environment” (TEE) is an environment that ensures that unauthorized entities cannot...

- modify the data inside the TEE. **(Data Integrity)**
- view the data inside the TEE. **(Data Confidentiality)**
- modify the code running inside the TEE. **(Code Integrity)**

An ”unauthorized entity” could be another program, the user, the operating system, etc. Essentially, anyone or anything that should not be able to access or modify the data or the code should not be able to do so.

The CCC also defines several additional attributes that TEEs can have. This includes ”Code Confidentiality”, which means that in addition to preventing unauthorized entities from modifying the code, they are also unable to access it. This is useful if the algorithm processing the data inside the TEE should be kept confidential. However, this shouldn’t result in developers turning to ”security by obscurity”[38] and implementing flawed algorithms in the hope that the flaws won’t be discovered due to the code being a secret. A better use case for this property is to keep algorithms confidential that are considered intellectual property and should not be copied by unauthorized entities. Another such property is the ”Attestability”, which is described as a process in which another party can verify the integrity of the TEE. During that check, a signed ”evidence” is created by hardware, which checks the current state of the TEE and verifies that the three attributes listed above can be fulfilled.

4.3.2 Hardware vs Software TEEs

The CCC argues that ”security is only as strong as the layers below it”[10][Section 2.2 first sentence]. This is due to the layered architecture of computers. The lowest layer is the hardware, then comes the operating system with its device drivers, and then the user’s applications (at least in a very simplified architecture). If the operating system is compromised, so are the user’s applications, as they depend on the integrity of the operating system. In reverse, this means that if we want to trust the application of a user (to process secrets, etc.), we must also trust the operating system and the hardware. That would involve a lot of components that

we need to trust, as the operating system usually also includes drivers that come from different vendors, etc. This shows that to guarantee confidential computing, we need to have a very small ”trusted computing base”. As the bottom layer, the hardware is ideal for this task, as it has only minimal dependencies on other components.

4.3.3 CCC Threat Model

The threat model defined by the CCC includes five threat vectors that are in-scope (meaning that a TEE should protect against them) and three that are out-of-scope.[10][Section 5.2] The in-scope threat vectors are:

- Software attacks: This threat vector includes all attacks against the software, the BIOS, the firmware, etc. As already mentioned in section 4.3.2, these sorts of attacks are prevented by placing the TEE in the hardware layer.
- Protocol attacks: The main point of concern here is flaws in the attestation protocol. If the attestation is flawed, a third party could trust a compromised TEE and send confidential information to the TEE, which could then be extracted. This is similar to the attack described in section 4.2.1.
- Cryptographic attacks: These attacks include concerns about possible vulnerabilities in cryptographic algorithms, as well as the ongoing increase in computational power, which poses a threat to some encryption algorithms. Also, new emerging technologies, such as quantum computing, pose a threat to conventional cryptography algorithms. Mitigating these sorts of threats is especially challenging. As discussed in section 4.3.2 the TEE is located on the hardware layer. Changing ”hardwired” encryption schemes can’t easily be done.
- Basic physical attacks: A basic physical attack would be to read the bus connection between the CPU and the memory banks. Reading the contents that are transferred between these two components should not yield secret information. For example, this kind of attack can be successful against a ”Trusted Platform Module” (TPM 2.0).[15][54] The TPM can be used to store secret keys for disk encryption. On boot, the TPM will send the secret key to the CPU to decrypt the disk. This transfer can be sniffed, by listening on the correct bus on the motherboard and recording the key, which would give the adversary the ability to decrypt the disk on another device.
- Basic upstream supply-chain attacks: The TEEs should also protect against ”simple” supply chain attacks. For example, a supply chain attack could be to include debugging ports that allow access to secret data or violate any of the three principles listed in section 4.3.1.

Some other attacks are considered out-of-scope as protecting against them is currently hardly possible, or it is simply a limitation of current TEE implementations:

- Upstream hardware supply-chain attacks: In this sort of attack, an adversary has the power to modify the CPU or influence the generation of a secret key at the time the CPU is being manufactured. It makes sense to exclude these attack vectors, as somewhere the "root of trust" must be placed. As discussed in section 4.3.2, the hardware is the lowest layer in the architecture and thus gives the smallest attack surface. However, this does not mean that we need to trust no one. If the hardware manufacturer builds in a deliberate backdoor or leaves debug registers in the final product, the TEE will still be compromised. As such, we need to trust the hardware manufacturer to design a secure chip.
- Sophisticated physical attacks: If an adversary has long-lasting physical access to the device and is capable of using chip scraping techniques or other very intrusive logical analyzers, the attacks are considered out of scope. These kinds of attacks could become a problem in the future, especially when thinking about the PowerDVD example that was discussed in section 4.2.1. In that use case, a TEE was used to obtain a secret decryption key from a third party. The third-party only wants to send that key to attested TEEs where it can be sure that the three principles listed in section 4.3.1 are assured. However, the adversary has full control over the hardware of the TEE, which receives the key and could use that power to extract secret information from the TEE. This also enables them to use side-channel attacks and possibly gain insight into the data being processed. The CCC threat model mentions side-channel attacks separately and also discusses them as a major problem.[10][Section 5.3] However, the mitigation that is proposed only mentions that the TEE vendors and application developers should prevent side-channel attacks from being possible. For example, numerous side-channel attacks have recently been discovered in SGX.[39][59]
- Availability attacks: Denial-of-service attacks are currently not considered within the scope of attacks that current TEE implementations must protect against.

4.4 SGX Enclaves

The "enclave" is the part of the SGX system in which private data is securely processed. An SGX application can consist of multiple enclaves that perform different tasks and computations. The application is always split into two different components: the trusted component and the untrusted component. The trusted component is the enclave or enclaves, and data is secured while being processed in those components. This means that the RAM content is protected, and the data is encrypted while it is not processed inside the CPU. Additionally, integrity checks are also performed to detect changes to the memory. The trusted components

should always be designed "as simple and small as possible" and to only do one thing, but do that thing well. This design directive can not only be applied to enclaves but is also a general philosophy which, for example, the Linux kernel also follows.[20] Having the trusted part as small as possible is not only important for security but also for performance. Computation and memory access inside the enclave can be a lot slower than those in the untrusted part. Thus, only computations that have to be performed in the trusted part should be performed in the trusted part. Any additional code that is run inside the trusted part, but does not necessarily have to be run there, may slow down the application considerably.[11][34]

4.4.1 Example Application

Take the following application as an example: The application is part of a messaging APP and should process text by running it through a secret AI model to detect certain phrases in it. The AI model is considered a secret; it should not be revealed to the client and is sent to the client once each time the application is started. The client inputs the text directly on their computer, and it is not considered a secret. However, the result of the AI model is considered a secret, as otherwise the user could drop the message and prevent its delivery if the AI model outputs an unwanted result. From that description, it follows that there will be a server, which has a copy of the secret AI model, and a client, which processes text messages and uses the AI model. When the client application is started, the AI model needs to be downloaded. As the model should be kept secret, that AI model can only reside in the trusted part of the application. The download of the model could work as follows:

1. Enclave for downloading the AI model is started on the client.
2. Enclave sends an HTTP request to the server indicating that it wants to download the model.
3. The server performs remote attestation (see section 4.5.2) on the client's enclave to ensure that it sends the secret AI model to a genuine enclave.
4. The server sends the AI model to the enclave (using an attested TLS channel).

Note that all data transfers are encrypted and can only be read by the enclave itself. The client computer cannot simply listen to the network traffic and record the AI model that way. The encrypted channel is established during the remote attestation procedure. The untrusted part of this SGX application contains all the necessary interactions with the networking devices. The enclave itself cannot directly interact with the network card, but still relies on the operating system and the drivers to communicate with that hardware device. So these networking parts would be considered "untrusted" and the enclave would issue an "O-Call" to the functions outside the enclave. Once the enclave has the AI model downloaded, it can start processing messages from the user. When the client inputs new text into the chat application, an

”E-Call” is made to a function inside the enclave. This way, the enclave is ”entered” and the text will be processed securely inside the enclave using the downloaded AI model. As the outcome is considered a secret, it should be encrypted and signed by the enclave and then sent along with the message text. Additionally, the message should include a hash of the message content, so that a verifier can detect if the message content was altered after being analyzed by the AI model. These two hashes are combined, and the resulting value is signed by the enclave.[1] When a new message arrives at the server, it is processed in the following way:

1. Verify the signature of the whole message (signed message hash).
2. Check the message hash.
3. Decrypt the result of the AI model.
4. Either block the message or forward it, depending on the result of the AI model.

This way, the server can trust the result from the client without the need to verify the text on their own. This process would still work if the text messages were end-to-end encrypted for another client, and the server should be unable to read the message. Then the enclave (which still must see the plain text to process it) would simply include the hash of the encrypted message instead.

Note that the data input by the user comes from an untrusted part of the application. The operating system handles the text input and the interaction with the keyboard, the speech-to-text input, the touchscreen input, etc. So the input of the user cannot be ”trusted” in any way.

4.4.2 O-Calls and E-Calls

To enter and exit the enclave, so-called ”O-Calls” and ”E-Calls” are needed. ”O-Calls” are made from inside the enclave to the ”outside”, which means from trusted code to untrusted code. The ”E-Calls” are doing exactly the reverse, meaning that untrusted code is calling trusted code inside the enclave. When an ”E-Call” is performed, all input parameters and pointers to shared memory are passed to the enclave. Contrary to that, when performing an ”O-Call”, the application cannot simply pass pointers to data that resides in the enclave application, as they are protected and cannot be read from the untrusted part. When passing data with an O-Call, the enclave must first copy the data contents into the application’s memory, as they can’t be accessed otherwise.[11][34]

A very simple and common ”O-Call” and ”E-Call” combination is one where the untrusted application calls a function inside the enclave and then the enclaves ”returns” the result by making an ”O-Call” back to the untrusted part.

[1] The hashes could be combined by concatenating them and then hashing the result.

1. Untrusted code calls the function ”calc(a, b)” inside the enclave (E-Call). The variables ”a” and ”b” are pointers to shared memory and can be accessed by the enclave.
2. The enclave reads ”a” and ”b” from the memory and saves them in local variables (stack).
3. The enclave then computes ”a + b + S”, where ”S” is a secret value only the enclave knows. Then it saves the result in the local variable ”c”.
4. The enclave copies the variable ”c” into the unprotected memory.
5. Then the enclave makes an ”O-Call” so that the untrusted application can continue. (Like a ”return” from the trusted function)

Note that this example enclave application would not protect the ”secret variable S”. While ”S” is not known to the untrusted component and only processed within the trusted component, it is still easily retrievable. By setting ”a” and ”b” to 0, the function would return $0+0+S$, which means that the secret is easily leaked without modifying the enclave code or breaking any of the security grantees of SGX.

4.5 SGX Attestation

Attestation is essentially the process of one party verifying the other party’s TCB (trusted computing base). The TCB includes everything from the enclave code and data to the security flags being used. SGX generates a cryptographic log of the build activities, which includes the previously listed values. This log is being used to create the TCB value. This value represents an ”Enclave Identity”, which is a 256-bit hash of the aforementioned log. In the SGX implementation, this measurement is called ”MRENCLAVE”.[34][52]

For the attestation process to work, two private keys are stored inside the CPU. These are called the ”device root keys”. The ”root provisioning key” (RPK) is randomly generated at an ”Intel Key Generation Facility”. This key needs to be known to Intel for the attestation protocol to work; for this reason, Intel keeps a database of all the keys generated at those facilities. The second key is the ”root sealing key” (RSK), which is generated inside the CPU during production and is not known to Intel.[34]

The device root keys can’t be used by the enclaves directly. They have special SGX instructions available to get key derivatives, but not the actual keys. If someone were able to extract the root provisioning key and the root seal key, they would be able to ”fake” attestation. This would be a very bad situation, as this would mean that anyone who obtains a copy of the keys could fake attestation, and no one could trust the process anymore. For this reason, Intel has created a ”full reset” function, that applies specific transformations to the root provisioning key, after the SGX update fixing the vulnerability has been installed. As can be seen in

Figure 4.2 the "security version" can be incremented to apply transformations to the original root provisioning key. To get the newest security version, users have to update their firmware, microcode, etc. on their SGX systems. As the security version is also part of the report, auditors would not trust a system that reports an old security version. Such a "full reset" was necessary due to CVE-2022-0005, which allowed attackers to extract secrets from SGX systems using JTAG functionalities.[12][30]

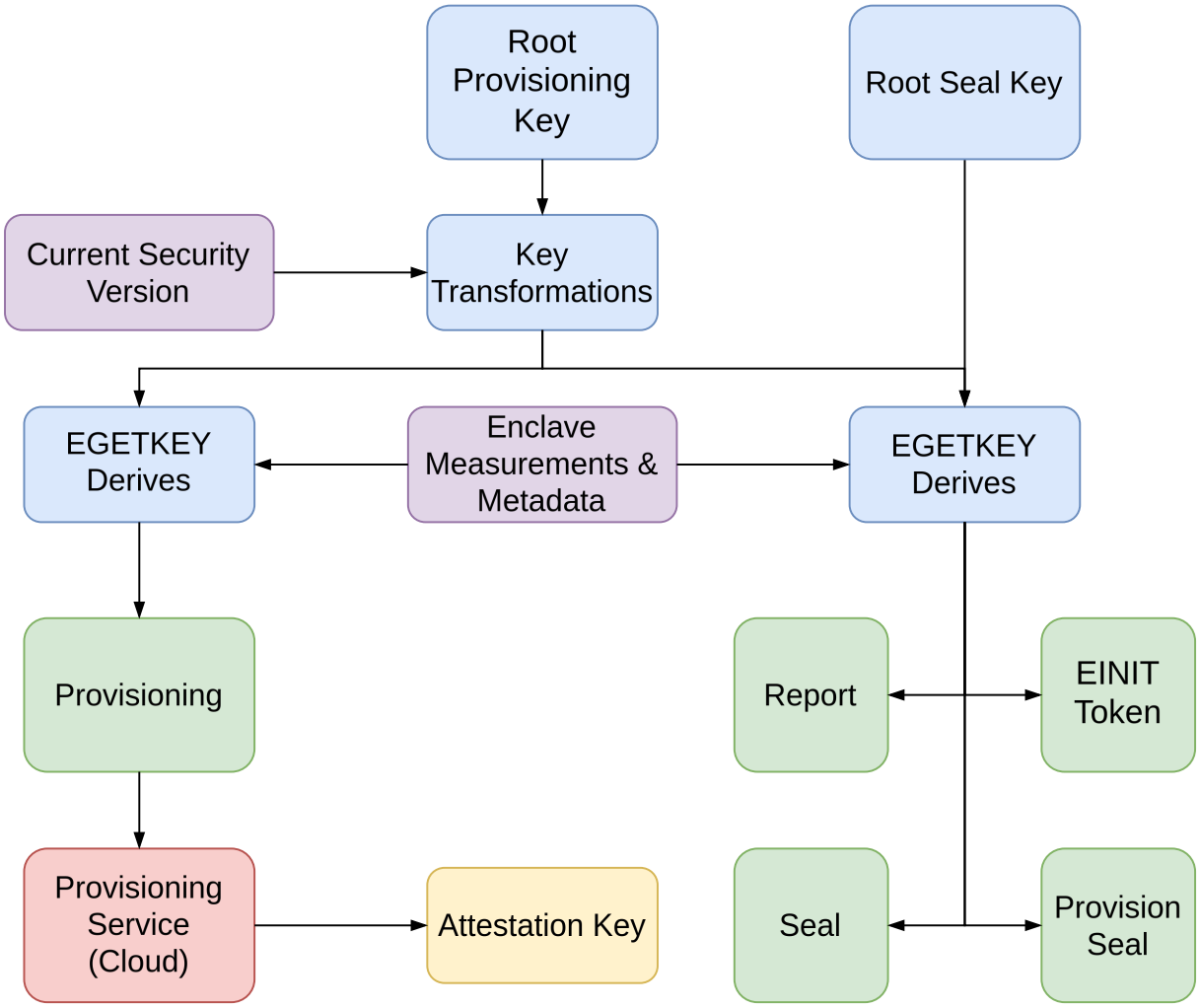

Figure 4.2: Device root keys and keys being derived from them for attestation.[34][Attestation]

4.5.1 Local-Attestation

Using local attestation, two SGX enclaves can verify that they are running on the same platform. This process can also be used to establish a secure connection between the two enclaves. In Figure 4.3, two enclaves are present in the system. The enclaves called ”A” and ”B” want to prove to each other, that they are running on the same TCB platform and at the same time want to establish a trusted channel (e.g. an encrypted channel) between them. In the first message, B sends A its MRENCLAVE value (see message 1 in Figure 4.3). Using the MRENCLAVE value, enclave A calls EREPORT giving B’s MRENCALVE value as input. This results in a signed report being generated for B’s MRENCLAVE, which binds it to the TCB of A. Additionally, A can begin a Diffie-Helman key exchange and attach the necessary data in the report’s data section. This report is then sent over to enclave B (see message 2 in Figure 4.3). B gets the REPORT KEY using the SGX instruction EGETKEY. Using that key, B tries to verify the report sent by A. As the report key is platform-specific, the verification will only be successful if A and B are on the same platform. B has now verified that A is on the same platform. For A to verify that B is on the same platform, B now uses A’s MRENCLAVE value (which was included in message 2), to create a report using the EREPORT instruction and sends it over to A (see message 3 in Figure 4.3). To verify the report from B, enclave A can then use the same steps as when B verified the report of A.

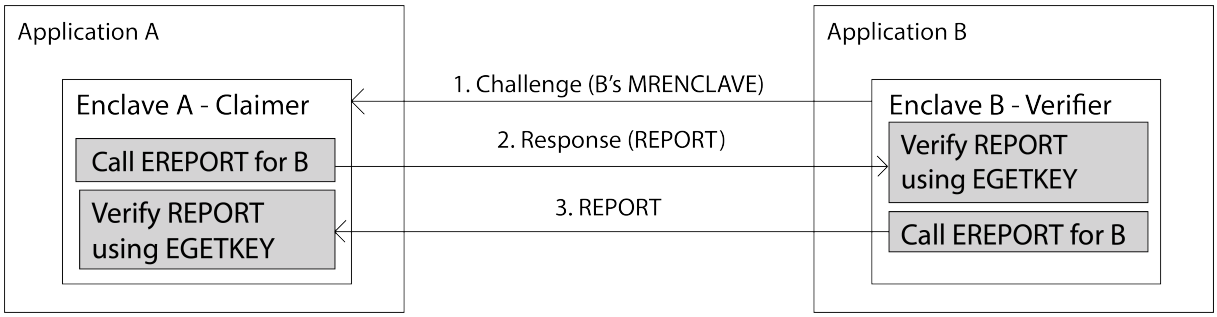

Figure 4.3: Local attestation of SGX enclaves running on the same platform.[34][Attestation]

4.5.2 Remote-Attestation

In remote attestation, an enclave attests a certain TCB to a remote party. For this process, an ”architectural enclave” is used. These types of enclaves are written by Intel and included in SGX by default. The ”quoting enclave” is the architectural enclave used for remote attestation. First, the user enclave sends its report to the QE (quoting enclave), which verifies the report. Then the QE signs the report with the private EPID (enhanced private ID) key and converts the result into a ”quote”. The remote attestation party can then verify the quote using the

EPID public key and check if the MRENCLAVE value matches the expected result. In this way a remote party can verify that a certain enclave is running inside a TEE.

4.6 SGX Security

4.6.1 Memory Limits

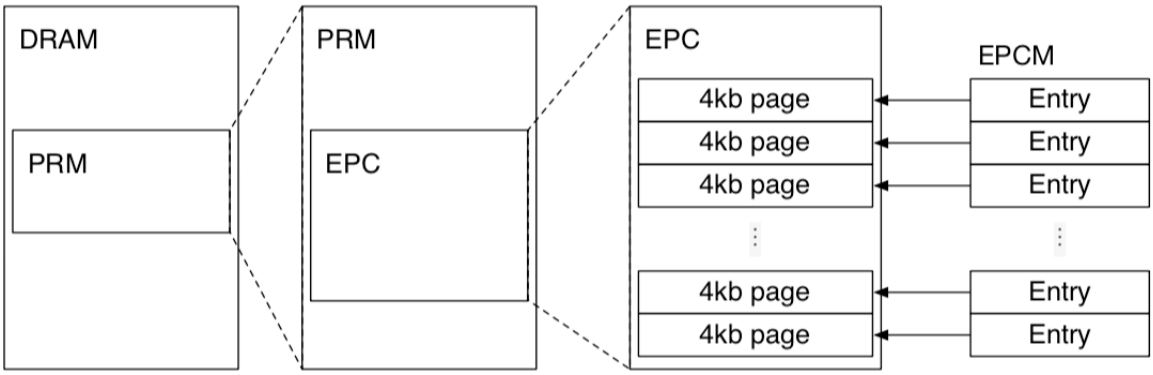

Figure 4.4: "Processor Reserved Memory" (PRM) used by the enclaves.[34][Enclave]

In SGX the DRAM is not trusted, as bad actors could read or modify the contents of the DRAM using specialized (hardware) tools. Also, the bus in between the CPU and the DRAM could be recorded, and secrets stored in memory would be revealed. SGX uses a special section of the DRAM, called the PRM (process reserved memory), where the EPC (enclave page cache) is located. In the EPC, the enclaves store their data and code. When CPUs access memory, they first look into their internal caches (L1, L2 etc.), and if the requested section of memory is not present there, they use the MC (memory controller) to retrieve the memory section from DRAM. In SGX, the MC is used in extension with the MEE (memory encryption engine). When accessing the special PRM memory section, the MEE supplements the MC's functionality by adding a encryption and an integrity check to the memory access. As the MEE resides directly on the CPU, this means that the data from the DRAM is encrypted until it is inside the physical bounds of the CPU. The downside of this model is that the PRM section (on current machines) has a maximum size of only 128 MB.[28]

This means that enclaves can use a maximum of 128 MB of RAM for code and data. However, the Linux SGX driver supports paging and can store the encrypted memory pages in normal DRAM. However, as pages need to be swapped out from DRAM back to PRM when being ac-

cessed, this will slow down applications, as memory access will take significantly longer than when staying under the 128 MB RAM limit and only using the PRM section.[49][32]

4.6.2 Exploits and Vulnerabilities

Although SGX provides a very secure environment, it is not without flaws. Over the years since SGX was released there have been numerous discoveries of attacks that can leak data, manipulate the control flow etc. One such attack on SGX was already mentioned in section 4.5. As it was possible to extract information from enclaves using the JTAG interface, a ”full reset” of the SGX system was necessary, and the security version number was incremented. Most flaws can usually be patched using BIOS updates; however, this is very inconvenient to normal users, which was one problem of the PowerDVD application (see section 4.2.1).

ÆPIC Leak

One such recently discovered flaw in which bad actors (with admin access to the machine) can leak data from inside an enclave is CVE-2022-21233.[13] The authors discovered an ”architectural CPU vulnerability” in which the Advanced Programmable Interrupt Controller (APIC) could be abused to read data transferred between the L2 and the ”last-level cache”. This cache can also include enclave data. Their implementation of the attack can leak memory from enclaves at a rate of 334.8 b/s and a success rate of 92.2%. In combination with two other attacks, they were able to extract Intel SGX sealing keys, as well as remote attestation keys.[8]

Malware Guard Extension

”Malware Guard Extension” is the first SGX malware, that uses the TEE to conceal cache attacks. The authors developed a software-based side-channel attack, in which one enclave (running the malware) attacks co-located enclaves running on the same machine. The attack performed was a ”Prime+Probe” cache side-channel attack, and they were able to extract 96% of an RSA private key from a single trace using the wide-spread mbedTLS library.[36] Extracting a full private key took five minutes, and the process of extracting it was automated. Running malware inside SGX enclaves can make detection and analyzing the malware harder for antivirus programs and vendors.[47]

Other Attacks

Most attacks on SGX are ”side-channel” attacks. Intel itself admitted that SGX does not defend against side-channel attacks based on power statistics, branch timing, cache miss statis-

tics, and page access via page tables.[59][SGX side-channel hazards] In side-channel attacks, the attacker does not directly target the SGX system but rather tries to infer information about an enclave from a "side-channel", for example, the amount of power that the computer draws. By measuring the total power usage of the computer and also measuring how much power a specific operation uses, bad actors can interpret the power usage and determine what operations were executed. As a defense against side-channel attacks, multiple fixes were proposed. One such defense is called "Varys"[39], which can protect unmodified SGX enclaves from cache timing and page table side-channel attacks, but intrudes an overhead of 15%. "Varys" prevents an attacker from accessing shared CPU resources by "reserving" one CPU core for the enclave application.[39][57]

While many attacks exist, that does not mean that SGX is entirely "broken". Most attacks can be patched using firmware updates and many attacks cannot be exploited easily. With proper isolation in cloud environments and restricted access to the machine running SGX will still be secure against attacks "from the outside". However, the threat model of SGX states that nothing but the CPU itself should be trusted. This is not fulfilled, as the numerous side-channel attacks could be exploited by the cloud provider running the machine, as the provider has physical access.

4.7 SGX Frameworks

Several different frameworks exist for SGX. These enable the developers to write software applications that use the SGX hardware in many different languages. The frameworks vary in which language or operating system they support, as well as in their complexity to develop applications.

4.7.1 Intel SGX

This framework includes the "Intel SGX SDK", "Intel SGX driver", and "Intel SGX Platform Software (PSW)".[25] It is developed directly by Intel and is available for development on Linux operating systems. Currently, Ubuntu 20.04, 22.04, and Debian 10, as well as some other distributions, are supported. Note that at the time of writing this, the "stable" Debian version is 12, and version 10 will be EOL (End of Life) soon. This framework is arguably one of the more complex ones and is not a good way to get started developing SGX applications. For example, the remote attestation example has 25,000 lines of code and has been archived on GitHub since January 2023.[26]

4.7.2 Asylo

This framework was developed by Google and uses the "bazel" build system. Asylo aims to provide a backend that can run secure applications on different TEEs. The goal is for the TEE technology to be interchangeable, with Asylo acting as the proxy between the application and the TEE. So an Asylo application could be started on a machine with an Intel CPU and run using SGX, but it would also work (in the future) with an AMD CPU and then use AMD TrustZone. However, despite the note in their GitHub repository, that Asylo is being actively developed, the software is neither actively developed nor being maintained. Currently, when trying to build the project, it fails with various errors. Firstly, only an outdated Bazel version is supported, as there have been breaking changes in the newer versions. The second problem is how dependencies are managed in Bazel builds. Essentially, dependency versions are pinned and need to be manually updated by the developers. So, for example, the dependency "boringssl" (Google's fork of OpenSSL) is specified by directly linking to the GitHub repository in the bazel build files. However, the version linked to it cannot be built with GCC 12 and should be upgraded. Unfortunately, when updating the dependency by specifying a newer version, the build also fails because of other changes in boringssl. Another option is to use an older GCC version. This can be done using the following script to execute bazel: `CC=/usr/bin/gcc-10 CCX=/usr/bin/g++-10 bazelisk "$@"` [2] However, even when building the project with an outdated GCC version, the simulation mode works (which does not use the SGX hardware, but only simulates it), but the hardware mode, unfortunately, does not.[24]

4.7.3 Openenclave

The aim of the Openenclave framework is similar to the one of Asylo. Unlike asylo however, it is still actively being developed and maintained. Developing programs for Openenclave is more similar to developing applications for the Intel SGX SDK. For example, they also use .edl (enclave definition language) files to specify the trusted and untrusted functions.[27]

4.7.4 Mystikos

This framework can be used to run Docker containers in a trusted environment using SGX. This framework uses Openenclave for remote attestation and starting the enclave. This significantly simplifies the building of SGX applications, as the developer can directly start by creating a Docker container and does not have to worry about trusted and untrusted components. The framework implements a small kernel that handles most syscalls (for networking,

[2] Note that the tool "bazelisk" will automatically download and use the version of Bazel that has been specified in the projects ".bazelversion" file.

threads, etc.) and processes them directly inside the enclave if possible. This architecture can be seen in Figure 4.5. For remote attestation and to use other features of the TEE, the developer must create a ”TEE-aware application”. This means that the application ”knows” that it runs inside an enclave and can perform such tasks as remote attestation and other SGX-specific functionalities.[21]

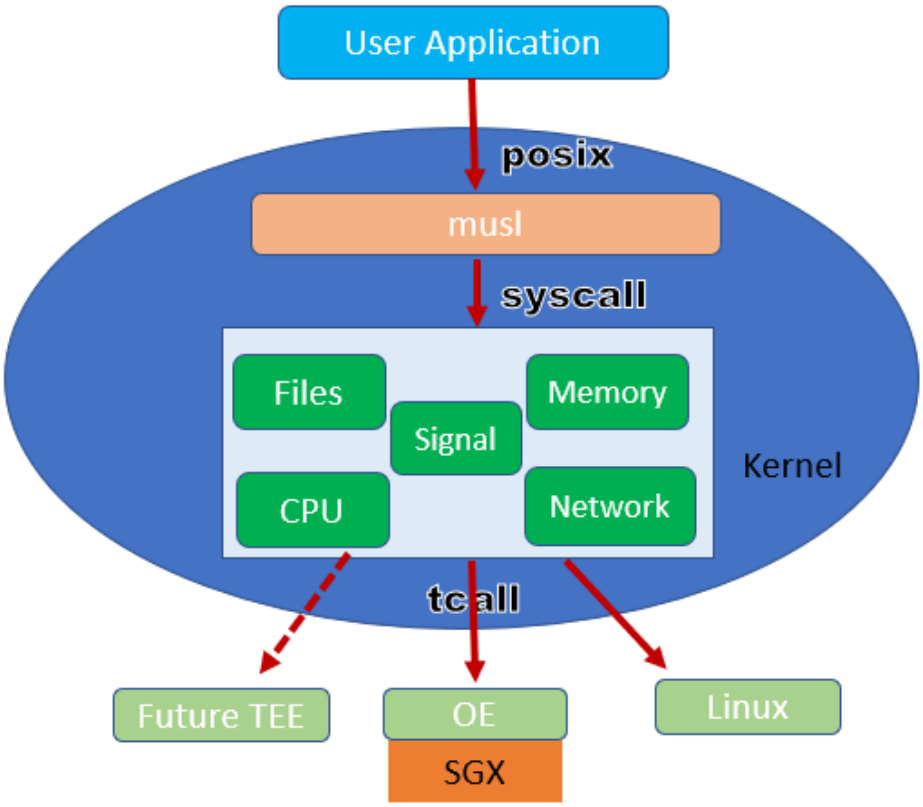

Figure 4.5: Architecture of mystikos.[21]

4.7.5 Edgeless RT

This framework is also built on top of Openenclave and further simplifies the development of enclave applications. Edgeless RT provides support for C++ 17, pthread, std::thread, and overall better comparability with libstdc++. It also adds support for other programming languages, such as Rust or GO.[22]

4.7.6 EGo

”EGo” is an SGX framework that uses Edgeless RT and further simplifies the development of secure applications. It provides an adapted compiler, called ”ego-go” and a CLI tool called ”ego”. Using the CLI tool, applications can easily be built, signed, and executed.[23] EGo was chosen for the implementation of this project because of its simplicity and good documentation. The remote attestation exampled was the only one that worked ”out of the box” without any changes.

Example

The example application written in Go, seen in Listing 4.1 simply prints the string ”Hello World” ten times.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func say_hello() {
8      for i := 0; i < 10; i++ {
9          fmt.Println("Hello World")
10     }
11 }
12
13 func main() {
14     say()
15 }
```

Listing 4.1: Example ”Hello World” application.

- To run this go program in an enclave, it must be compiled using the ”ego-go” compiler. This can be done using the following command:
  `ego-go build`
- Then the application needs to be signed:
  `ego sign helloworld`

- And finally, the application can be started:
  `ego run helloworld`

If the platform that the application is started on does not support Intel SGX, the application can also be started in simulation mode: `OE_SIMULATION=1 ego run helloworld`

```
[erthost] loading enclave ...
[erthost] entering enclave ...
[ego] starting application ...
Hello World
Hello World
Hello World
Hello World
.
.
.
```

Listing 4.2: The output of the application, shown in Listing 4.1, after being compiled, signed, and then executed using the "ego" tools.

4.8 Processing Forensic Data in the Cloud

As already mentioned in section 4.1, the goal is to create an SGX application that can securely process the recorded forensic data in the cloud. The application should be a simple demonstrator, that can be further extended to process all kinds of forensic data. The implementation should be able to securely process a point cloud or a portion of a point cloud map. For demonstration purposes, the point cloud will simply be normalized to values between -1 and 1. The input for the application should be sent over an "Attested TLS channel", that is established during the remote attestation procedure, where the client (holding the data) audits the enclave (on the server) before sending the data directly to the enclave. After the data has been processed by the enclave, it should be sent back over the same secure, attested TLS connection.

4.8.1 Security Requirements

The security requirements for the whole application are similar to the ones already described in section 4.3.1 (Confidential Computing). The most important task is to keep the data strictly

confidential. That means, neither the cloud provider nor anyone listening on the communication link between the client and the cloud should be able to modify the data (data integrity) or view the data (data confidentiality). Furthermore, they should also not be able to extract the data while it is being processed or manipulate how the data is being processed (code integrity). However, the code of the application itself is not considered a secret. By default, the code of an enclave is not secret and can be disassembled before the enclave is fully initialized. This is not a security vulnerability but this was a design decision by Intel.[52] However, if the application developer wants the enclave code to be a secret, they can use SGXElide which enables enclave code secrecy via self-modification at runtime.[6]

4.8.2 Implementation

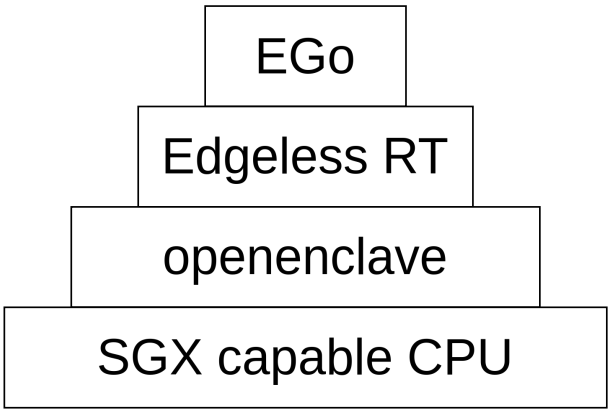

Figure 4.6: Dependencies of the SGX libraries used.

The demonstrator application uses the EGo framework, which is an extension of the Edgeless RT library, as can be seen in Figure 4.6. The application can be compiled using the ”ego-go” compiler (a modified go compiler) on Ubuntu 20.04 and 22.04. Furthermore, the system must have an SGX-capable Intel CPU installed if the software should be running in ”hardware mode” with proper remote attestation. The software can also be run in ”simulation mode” on non-SGX-capable CPUs. The remote attestation ”Data Center Attestation Primitives” (DCAP) procedure requires that a ”Provisioning Certification Caching Service” (PCCS) is running.[44] Intel specifies that the PCCS service should be running on a different machine from the one that is executing the SGX workloads.[37] However, for this demonstrator, everything (the client, the server (SGX), and the PCCS) will be running on the same machine.

To run a PCCS service, an API key from Intel is required.[37] It can be obtained by registering an account with Intel and filling out a form. For running the service, the EGo developers provide a

convenient Docker container. When starting the Docker container, the API key must be passed to the container via an environment variable.

```
docker run -e APIKEY=YOUR_KEY -p 8081:8081 --name pccs -d ghcr.io/edgelesssys/pccs
```

Listing 4.3: Starting the PCCS Docker container provided by the EGo developers.[44]

For the SGX applications to recognize the PCCS service, two values need to be changed in the configuration file */etc/sgx_default_qcnl.conf* (see Appendix A). If the file does not exist at all, it must be created first. The following configuration values should be changed:

- *"pccs_url"*: Needs to be set to the local instance of the PCCS service.
  For example: *https://localhost:8081/sgx/certification/v4/*
- *"use_secure_cert"*: Needs to be set to "false", as the PCCS service is only started locally and no valid certificate is provided by it.

Architecture

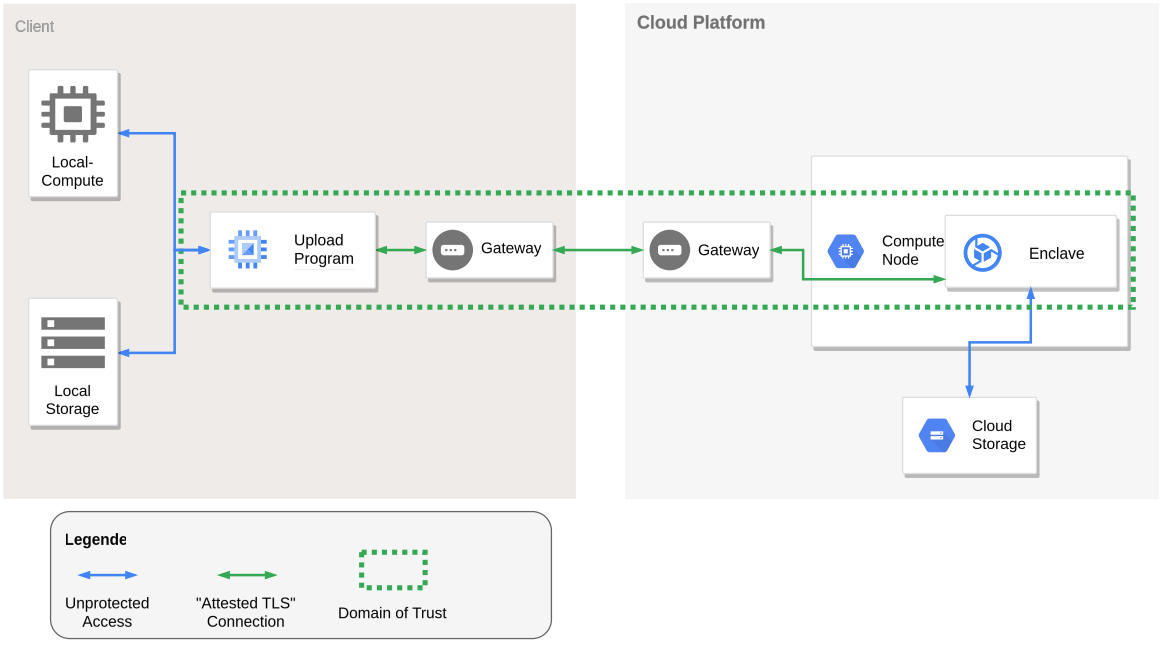

Figure 4.7: TLS connection into an attested enclave running in the cloud.

Figure 4.7 shows the intended architecture of the demonstrator. A client, which holds the forensic data, and a server running in the cloud, which processes the secret data. The only ”trusted” component is the client itself, and the whole cloud environment is considered ”untrusted”. This means that the data must not be accessible to the cloud provider and also should not be accessible to anyone listening on the communication link between the client and the server. To send the data into a TEE, the client will perform the remote attestation of the data processing enclave started on the server. After the enclave has been audited, the client trusts the application and sends the data over an ”attested TLS” channel to the enclave. The data sent over the attested TLS connection can only be decrypted by the enclave itself, which in turn runs in a TEE. Note that the ”Domain of Trust” in Figure 4.7 extends only over the TLS connection and the enclave, but not the whole cloud server. Only the enclave itself has been audited and can be trusted; other components, like the ”Cloud Storage” seen in Figure 4.7, cannot be trusted. Those components operate out of bounds from the enclave, and data written to that device could be read by the cloud provider. If the enclave has to write data to the device, it would have to write encrypted data to it.

The demonstrator could be set up as shown in Figure 4.7, but for simplicity, all services will run on the same machine. The components communicate with each other using the loopback interface.

Server

The server enclave runs an HTTP server and has an endpoint called ”*/pointcloud/normalize*”. The client can send a secret point cloud via the attested TLS connection, and the HTTP response will return the normalized point cloud. The point cloud is simply a list of multiple points, as defined in Listing 4.4.

```
type Point struct {
    X float64 `json:"x"`
    Y float64 `json:"y"`
    Z float64 `json:"z"`
}
```

Listing 4.4: Definition of the ”Point” structure.

When the HTTP endpoint receives a point cloud from the client, the function `normalizePoints(points []Point)` will be called. It takes a reference to the list of points as input and performs the normalization on that reference, instead of copying the list and then

returning a copy where the normalization is applied. This must be done due to the strict memory limits of SGX applications, as already explained in section 4.6.1. The algorithm seen in Listing 4.5 then looks for the minimum and maximum X, Y, and Z values and scales the values accordingly. The result is a normalized point cloud where all values are between -1 and 1. This algorithm was chosen because it's a small and simple example, where the server uses all the data sent to it and the result can be sent back to the client.

```go
func normalizePoints(points []Point) {
    min_x := math.Inf(1)
    min_y := math.Inf(1)
    min_z := math.Inf(1)
    max_x := math.Inf(-1)
    max_y := math.Inf(-1)
    max_z := math.Inf(-1)

    // find the minimum and maximum values for X, Y, and Z
    for _, item := range points {
        if item.X < min_x {
            min_x = item.X
        }
        if item.Y < min_y {
            min_y = item.Y
        }
        if item.Z < min_z {
            min_z = item.Z
        }
        if item.X > max_x {
            max_x = item.X
        }
        if item.Y > max_y {
            max_y = item.Y
        }
        if item.Z > max_z {
            max_z = item.Z
        }
    }

    // normalize the points
    for i := range points {
        points[i].X = (points[i].X - min_x) / (max_x - min_x)
        points[i].Y = (points[i].Y - min_y) / (max_y - min_y)
        points[i].Z = (points[i].Z - min_z) / (max_z - min_z)
    }
}
```

Listing 4.5: Simple algorithm for normalizing point clouds.

To receive the point cloud from the client, an HTTP server is started in the enclave. The function `normalizePointcloud(w http.ResponseWriter, r *http.Request)` is called by the server, when a

point cloud is sent to it. The normalization function can be seen in Listing 4.6. It supports two different HTTP requests. If a ”GET” request is sent, the server simply replies with an example point cloud. If the client sends a ”POST” request, the server expects a JSON object of a point cloud in the request body. It then parses that JSON and saves it into a list of ”Point” objects; see Listing 4.4. Then the function to normalize the points, seen in Listing 4.5, is called. Finally, the normalized list of points is marshaled into a JSON object and then returned in the HTTP response.

```
func normalizePointcloud(w http.ResponseWriter, r *http.Request) {
    var inputPoints []Point

    switch r.Method {
        case "GET":
            test_pointcloud := []Point{
                {1, 2, 3},
                {2, 2, 2},
            }
            j, _ := json.Marshal(test_pointcloud)
            w.Write(j)
        case "POST":

            // decode the JSON request body into the inputPoints slice
            d := json.NewDecoder(r.Body)
            err := d.Decode(&inputPoints)
            if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
            }

            // normalize the pointcloud
            normalizePoints(inputPoints)

            // return the modified pointcloud as JSON
            j, err := json.Marshal(inputPoints)
            if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
            }
            w.Write(j)
        default:
            w.WriteHeader(http.StatusMethodNotAllowed)
            fmt.Fprintf(w, "Method not allowed")
    }
}
```

Listing 4.6: Callback function for the "*/pointcloud/normalize*" HTTP endpoint.

To initialize the server, the enclave first generates a certificate (see Listing 4.7), which will be used for the attested TLS connection. Then it generates a report (line 5 in Listing 4.8), which includes the hash of the certificate and can be used to attest the application. Then the HTTP

routes are created:

- The "*/cert*" route is being used to query the generated certificate.
- The "*/report*" route is used to query the enclave report that will be used for remote attestation.
- The "*/pointcloud/normalize*" route is used by the client to process a point cloud.

After the routes have been specified, a "tls.Config" is created, which holds the previously generated certificate and private key. The TLS config is then passed to the HTTP server, which starts accepting connections on port 8000 and uses the certificate and private key for encryption.

```
func createCertificate() ([]byte, crypto.PrivateKey) {
    template := &x509.Certificate{
        SerialNumber: &big.Int{},
        Subject:      pkix.Name{CommonName: "localhost"},
        NotAfter:     time.Now().Add(time.Hour),
        DNSNames:     []string{"localhost"},
    }
    priv, _ := rsa.GenerateKey(rand.Reader, 2048)
    cert, _ := x509.CreateCertificate(rand.Reader, template, template, &priv.PublicKey, priv)
    return cert, priv
}
```

Listing 4.7: Function for creating a certificate.[19]

```go
func main() {
    // create certificate and a report that includes the certificate's hash.
    cert, priv := createCertificate()
    hash := sha256.Sum256(cert)
    report, err := enclave.GetRemoteReport(hash[:])
    if err != nil {
        fmt.Println(err)
    }

    // create HTTPS server routes

    // returns the server certificate
    http.HandleFunc("/cert", func(w http.ResponseWriter, r *http.Request) { w.Write(cert) })

    // returns the enclave report
    http.HandleFunc("/report", func(w http.ResponseWriter, r *http.Request) { w.Write(report) })

    // normalize a pointcloud
    http.HandleFunc("/pointcloud/normalize", normalizePointcloud)

    tlsCfg := tls.Config{
        Certificates: []tls.Certificate{
            {
                Certificate: [][]byte{cert},
                PrivateKey:  priv,
            },
        },
    }

    server := http.Server{Addr: "0.0.0.0:8080", TLSConfig: &tlsCfg}

    fmt.Println("listening ...")
    err = server.ListenAndServeTLS("", "")
    fmt.Println(err)
}
```

Listing 4.8: Initializing the HTTP server in the enclave.[19]

Note that in the current implementation, everyone with the server address and the right port can connect to the web server. This means that anyone could use the *"/pointcloud/noraml-ize"* endpoint, even without performing remote attestation; they would only need to trust the certificate manually, as it is self-signed. This would open the door to DoS (denial of service)

attacks, as a bad actor could just upload random point clouds over and over again and fill the memory of the application. As already discussed in section 4.6.1, once the PRM limit of 128 MB is reached, the application will become significantly slower.[32] To fix this problem, in a real-world application, the client first needs to authenticate with the server. One practical solution would be to include a public key in the enclave application. The server would then send the client a random value (after the attestation has been completed), which they sign and send back to the server for verification.

Client

In the demonstrator application, the client has secret forensic data in the form of a point cloud. The client wants to utilize a cloud application to normalize this point cloud but without the risk of the data being leaked to the cloud provider or other entities. Before sending the secret information to the cloud, the client must perform the attestation and verify that the enclave is running correctly and has not been modified. This is done via ”remote attestation”; see section 4.5.2 for more details.

In Listing 4.9 the main function of the client can be seen. It starts by parsing the input arguments, like the server address, and then continues to perform remote attestation on the server. Similar to the server, the ”tls.Config” must be initialized. The client simply queries the certificate from the server over HTTP (line 16 in Listing 4.9). Then the client gets the ”report” from the server, which is used for attestation. The function `verifyReport(reportBytes, certBytes, signer []byte)`, seen in Listing 4.10, takes the report, the certificate and the signer ID as input. Using the EGo tool, the ID can be derived from the public key used by the signer: `ego signerid public.pem`. Inside the ”verifyReport” function, the client then calls the EGo function `eclient.VerifyRemoteReport`, which checks the report. If the function returns an error, for example, if it is determined that the TCB (trusted computing base) level is invalid (which means that the enclave is running on an outdated or insecure SGX platform), the verification will fail.[3] Previously, the certificate of the web server running in the enclave was simply downloaded using HTTP, which is possibly insecure. For that reason, the server has included a hash of the valid certificate in the report’s data section. As we can trust the report (if it’s valid), we can also trust the hash provided in the report. If the hash of the certificate queried over HTTP matches the hash provided in the report, the client can be sure that the certificate was generated securely inside the enclave. Other checks include checking if the signer ID matches, if debug mode is set to false etc.

[3]Note that in this example the TCB level is ignored, as it caused problems on the test machine. This warning can usually be fixed by performing a BIOS update if one is available, which is not the case for the test machine. Please keep in mind that ignoring the TCB level is not a good idea, as it defeats the whole purpose of a TEE. If the audited system is running a version of the SGX with known security issues, that system should not be trusted. This is essentially what made the Blue-ray hack discussed in section 4.2.1 possible.

If the checks are passed, the certificate can be used to securely transmit the demo point cloud to the web server running in the enclave. To marshal the point cloud into the JSON format and send it in a PORT request to the HTTP *"/pointcloud/normalize*, the function `normalizePointCloudOnServer(pointcloud []Point, tlsConfig *tls.Config, url string, debug_print bool)` is called. If everything works accordingly, the server returns the normalized point cloud in the response body.

```
func main() {
    signerArg := flag.String("s", "", "signer ID")
    serverAddr := flag.String("a", "localhost:8080", "server address")
    flag.Parse()
    signer, err := hex.DecodeString(*signerArg)
    if err != nil {
        panic(err)
    }
    if len(signer) == 0 {
        flag.Usage()
        return
    }

    url := "https://" + *serverAddr
    tlsConfig := &tls.Config{InsecureSkipVerify: true}
    certBytes := httpGet(tlsConfig, url+"/cert")
    reportBytes := httpGet(tlsConfig, url+"/report")
    if err := verifyReport(reportBytes, certBytes, signer); err != nil {
        panic(err)
    }

    cert, _ := x509.ParseCertificate(certBytes)
    tlsConfig = &tls.Config{RootCAs: x509.NewCertPool(), ServerName: "localhost"}
    tlsConfig.RootCAs.AddCert(cert)
    test_pointcloud := []Point{{100, 100, 100}, {0, 0, 0}}

    fmt.Printf("Sending pointcloud with len: %d\n", len(test_pointcloud))
    if err := normalizePointCloudOnServer(test_pointcloud, tlsConfig, url +
↪   "/pointcloud/normalize", false); err != nil {
        fmt.Printf("Error: %v\n", err)
    }
}
```

Listing 4.9: Perform remote attestation on the server and process an example point cloud in the enclave.[19]

```go
func verifyReport(reportBytes, certBytes, signer []byte) error {
    report, err := eclient.VerifyRemoteReport(reportBytes)
    if err == attestation.ErrTCBLevelInvalid {
        fmt.Printf("Warning: TCB level is invalid: %v\n%v\n", report.TCBStatus,
    ↪ tcbstatus.Explain(report.TCBStatus))
    } else if err != nil {
        return err
    }

    hash := sha256.Sum256(certBytes)
    if !bytes.Equal(report.Data[:len(hash)], hash[:]) {
        return errors.New("report data does not match the certificate's hash")
    }

    // either verify the UniqueID or the tuple (SignerID, ProductID, SecurityVersion, Debug).

    if report.SecurityVersion < 2 {
        return errors.New("invalid security version")
    }
    if binary.LittleEndian.Uint16(report.ProductID) != 1234 {
        return errors.New("invalid product")
    }
    if !bytes.Equal(report.SignerID, signer) {
        return errors.New("invalid signer")
    }

    // for production debug should be: report.Debug == false
    if report.Debug == true {
        fmt.Println("Warning: Debug is set to true in report!")
    }

    return nil
}
```

Listing 4.10: Verify the report sent by the server during the remote attestation procedure.[19]

```go
func normalizePointCloudOnServer(pointcloud []Point, tlsConfig *tls.Config, url string,
    debug_print bool) error {
    // serialize the point cloud to JSON
    jsonData, err := json.Marshal(pointcloud)
    if err != nil {
        return err
    }

    // create an HTTP client with the provided TLS configuration
    client := &http.Client{
        Transport: &http.Transport{
            TLSClientConfig: tlsConfig,
        },
    }

    // send a POST request to the specified URL with the JSON data in the body
    resp, err := client.Post(url, "application/json", bytes.NewBuffer(jsonData))
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    // check the response status code
    if resp.StatusCode != http.StatusOK {
        return fmt.Errorf("Server returned non-OK status code: %d", resp.StatusCode)
    }

    // read the response body
    responseBody, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return err
    }

    // unmarshal the response JSON into a slice of Point
    var modifiedPointCloud []Point
    if err := json.Unmarshal(responseBody, &modifiedPointCloud); err != nil {
        return err
    }

    if (debug_print) {
        // print the modified point cloud
        fmt.Println("Modified Point Cloud:")
        for _, point := range modifiedPointCloud {
            fmt.Printf("X: %f, Y: %f, Z: %f\n", point.X, point.Y, point.Z)
        }
    }

    return nil
}
```

Listing 4.11: Sending a secret point cloud to the server for normalization.

4.8.3 Running the Demonstrator

To run the demonstrator, either Ubuntu 20.04 or 22.04 are required. Those are the Ubuntu versions that are currently supported by "EGo".[23] Furthermore, a CPU with SGX capabilities is required. To find out if a CPU supports SGX, the "cpuid" command can be used. This command is provided by the package "cpuid" on most Linux distributions.

```
>> cpuid | grep SGX
SGX: Software Guard Extensions supported = true
SGX_LC: SGX launch config supported      = true
Software Guard Extensions (SGX) capability (0x12/0):
SGX1 supported                           = true
SGX2 supported                           = false
SGX ENCLV E*VIRTCHILD, ESETCONTEXT       = false
SGX ENCLS ETRACKC, ERDINFO, ELDBC, ELDUC = false
SGX attributes: ECREATE SECS.ATTRIBUTES (0x12/1):
SGX Enclave Page Cache (EPC) enumeration (0x12/0x2):
SGX Enclave Page Cache (EPC) enumeration (0x12/0x3):
```

Listing 4.12: Checking if the installed CPU (Intel Core i9-10885H) has SGX support, by running the "cpuid" command and filtering for the word "SGX".

In Listing 4.12 the output of "cpuid" for the CPU Intel Core i9-10885H can be seen. Important are the first three lines that end with "true", as those are the requirements for EGo program to run.[23]

After verifying that the CPU has SGX capabilities, the PCCS must be started, as already described in Listing 4.3. After that, the applications can be built and started.

Simulation Mode

SGX also supports a "Simulation Mode", in which the program is not run using an enclave, but some features of SGX are "simulated". This should only be used for debugging and testing purposes and is useful on machines that have no SGX support. However, as the demonstrator application depends on SGX's report functionality, running the enclave in simulation mode is not possible. If the server is started in simulation mode, it will print a warning, as can be seen in Listing 4.13.

```
EGo v1.4.1 (8b99356398dd3bcb5f74e5194d20ce421f607404)
EGo v1.4.1 (8b99356398dd3bcb5f74e5194d20ce421f607404)
[erthost] running in simulation mode
[erthost] loading enclave ...
[erthost] entering enclave ...
[ego] starting application ...
ERROR: can't get report in simulation mode (oe_result_t=OE_UNSUPPORTED)
 ↪ [openenclave-src/enclave/sgx/report.c:oe_get_report_v2:190]
OE_UNSUPPORTED
listening ...
```

Listing 4.13: Running the server in simulation mode.

If the client is started, it will receive an empty report from the server and throw an error, as it can't verify that the server is running in a secure enclave.

Hardware Mode

On a CPU with SGX support, the server can be started by running the commands shown in Listing 4.14. This will build the server, sign the application, and then start it. If the server is started successfully, the output should look like in Listing 4.15.

```
mkdir build
ego-go build
ego sign server
mv server ./build
cp private.pem public.pem ./build
ego run build/server
```

Listing 4.14: Building and starting the server part of the demonstrator in hardware mode.

```
EGo v1.4.1 (8b99356398dd3bcb5f74e5194d20ce421f607404)
EGo v1.4.1 (8b99356398dd3bcb5f74e5194d20ce421f607404)
[erthost] loading enclave ...
[erthost] entering enclave ...
[ego] starting application ...
listening ...
```

Listing 4.15: Output of the server application.

After the server has been successfully started, the commands shown in Listing 4.16 can be used to build and start the client application. The ”public.pem” key is used to verify the signature of the server application, which was previously signed in Listing 4.14 using the private key of that key-pair.

```
mkdir build
CGO_CFLAGS=-I/opt/ego/include CGO_LDFLAGS=-L/opt/ego/lib go build client.go
mv client build/
cp public.pem build/
./build/client -s `ego signerid public.pem`
```

Listing 4.16: Building and starting the client in hardware mode.

In Listing 4.17 the output of the client application can be seen. One warning concerning the TCB level of the application is printed, but as already explained in section 4.8.2 this warning is currently ignored in the demonstrator. As the warning points out, the ”cpu_svn” (CPU Security Version Number) is outdated. This warning should not be ignored on productive systems and can be fixed as soon as a new BIOS update is available, that patches the CPU microcode to the newest SGX version. Otherwise, the remote attestation is successful, and the point cloud can be processed on the server. The point cloud that is sent to the server contains one point where x, y, and z are all 100 and one point where all are 0. The correct result is returned by the enclave and printed at the end of the program, as can be seen in Listing 4.17.

```
EGo v1.4.1 (8b99356398dd3bcb5f74e5194d20ce421f607404)
2024-01-30T10:50:06+0100.103368Z [(H)ERROR] tid(0x7fb1694ee740) | Invalid platform TCB level: OutOfDate (cpu_svn[0] = 0xb, pce_svn = 0xb) [/ertbuild/3rdparty/openenclave/openenclave-src/common/sgx/tcbinfo.c: oe_parse_tcb_info_json:1213]
2024-01-30T10:50:06+0100.107406Z [(H)ERROR] tid(0x7fb1694ee740) | Invalid platform TCB level: OutOfDate (cpu_svn[0] = 0xb, pce_svn = 0xb) [/ertbuild/3rdparty/openenclave/openenclave-src/common/sgx/tcbinfo.c: oe_parse_tcb_info_json:1213]
Warning: TCB level is invalid: OutOfDate
TCB level of SGX platform is outdated.
Sending pointcloud with len: 2
Modified Point Cloud:
X: 1.000000, Y: 1.000000, Z: 1.000000
X: 0.000000, Y: 0.000000, Z: 0.000000
```

Listing 4.17: Output of the client application.

Chapter 5

Conclusion and Outlook

5.1 Forensic Data Collection and Storage

This master thesis demonstrates a practical approach on how to collect, store, and securely process forensic data. One requirement is, that the collection and storage system must work in offline environments. To overcome this challenge and still be able to securely exchange information, YubiKeys are used to establish trust between the different parts of the system. During the initialization of a new project, the base-station (data collector) generates a public-private key pair directly on the YubiKey, without the private key leaving the physical boundaries of the YubiKey at any time. While the public key can be retrieved from the device and is saved in the base-station's database, the private key cannot be extracted from the YubiKey and remains a secret. When a forensic component has recorded data and wants to send it to the base-station, it can use the YubiKey to sign a hash-chain including the recorded data, and send the whole package over to the base-station. The base-station is then able to verify the signature by looking up the public key in its database and checking the validity of the signature.

The YubiKeys ensure the authenticity of the data. To ensure integrity and that no data was lost, or intentionally denied, multiple hash-chains are used. Each forensic component maintains a "sub-hash-chain", which is a hash-chain that includes all the data the forensic component collects. When sending over the data to the base-station, the component includes a signed sub-hash-chain value of the hash-chain's tail in the data package. On the base-station side, this signed value is then verified using the stored public key of that forensic component. To verify the integrity, the base-station also stores the sub-hash-chain of each forensic component. When a new data package arrives, the base-station "catches up" with the sent signed sub-hash-chain value by using its stored old tail value as a starting point and then repeatedly computes the new tail value for each measurement included in the data package, until it stops at the last value. If the computed tail matches the sent sub-hash-chain tail, no data has been lost or changed in the transmission from the forensic component to the base-station and the data is saved into the project's data table.

Finally, the collected data can be exported into one single SQLite3 database. Alongside this database, a few scripts are provided for the user. The most important one is the ”verification script”, which checks each signature, all hash-chains, as well as some additional properties that ensure that the export was not altered in any way. For example, one of the additional properties is that the last data entry (e.g. the hash-chain entry) must be signed by both the forensic component that sent it, and the base-station. Otherwise, a bad actor could add data to the export, as the necessary hash-chain entries can easily be computed, but the signatures cannot be computed without the private keys. Subsequently, this means that the YubiKeys should be overwritten as soon as the inspection is finished, and all data has been collected by the base-station. Deleting the keys will ensure that no one can add data to the package at a later time. This results in one currently unsolved problem. It’s not possible to sign a message that states that the private key on the YubiKey has been deleted, as the key for signing that message would not exist anymore if it was deleted securely. On the other hand, one approach could be to sign a message, that states that the next operation will be to delete the private key, but is this enough? No, because after sending that message, a bad actor could simply unplug the YubiKey and hinder the deletion of the private key. Other approaches, like protecting the YubiKey using a PIN, would not be sufficient, as the key would have to be known to the forensic component. A bad actor, that can gain access to the YubiKey (meaning physical access), could likely also gain access to the forensic component itself. Conventional solutions do not protect against these kinds of attacks, however, there is a possible solution to the problem, which is further discussed in section 5.3.1.

5.2 Confidential Computing

After collecting and recording the forensic data, users might want to analyze the recorded data in the cloud. However, as the data collected by the INFRASPEC project contains supply shafts which are considered critical infrastructure, the data has to be kept strictly confidential. This hinders the use of cloud services, as data sent to the server could potentially be read by the cloud provider. To ensure that the data stays confidential, a demonstrator was developed, which uses SGX to secure the data during transmission and processing.

5.2.1 SGX

The implemented demonstrator is split into two components. The client and the server. The client holds the forensic data that was previously collected, and the server is the component that should process it and is running in a distant cloud environment. After the client and the

server are started, the client performs remote attestation and can verify, that the server program has not been modified and is running in a proper SGX enclave. The cloud provider cannot access any data that the SGX enclave is processing, as the application's data in memory is encrypted and only decrypted directly inside the CPU. During the transmission from the client to the server the data is also encrypted using an attested TLS channel, which is established during the remote attestation process.

While SGX promises a secure computation environment, various attacks have breached its security guarantees and attackers were able to extract data, mostly using side-channel attacks, as was discussed in section 4.6.2. Apart from the security vulnerabilities, SGX is also impractical to use, as is evident when looking at one of the few SGX applications that were developed for clients, as was explained in section 4.2.1. Users will regularly need to update their BIOS, to keep the installed SGX version up to date, so that enclaves can be run securely. This is a burden to many users, as not every BIOS updates automatically and some are also rarely updated by their vendors. All in all, it's understandable why Intel has deprecated SGX on consumer CPUs and focused its efforts on the cloud environments and server CPUs. For the remote attestation to work, the client usually does not need an enclave and thus does not need to have SGX support. This is also the case for the demonstrator developed in this thesis.

5.3 Further Work

5.3.1 Solving the Key Deletion Problem using SGX

As already mentioned, the secure deletion of the private keys used for singing remains one potential security problem. If the private keys of one forensic component and the base-station are not deleted, a bad actor could use those keys to add or modify the forensic data. As the YubiKeys are physical devices, a bad actor could unplug them before the deletion process is finished and thus have access to the private key to sign new data. A future solution to this problem could be to utilize SGX enclaves for signing. The whole system could then work without YubiKeys and only use the enclaves to establish trust between the devices. The system could work as follows:

1. The base-station is started with a trusted component (enclave), that generates a public-private key pair. The private key remains inside the enclave at all times and should not be leaked outside the trusted component.
2. Then the other forensic components are started, which also generate a public-private key pair inside their enclaves.

3. The forensic components notify the base-station to perform mutual remote attestation and establish an attested TLS connection. After this step, each forensic component and the base-station have been attested and a secure connection from each forensic component to the base-station exists.
4. Then the forensic components can start gathering their data. All their data is directly passed to the enclave, which handles the creation of the hash-chain, as well as the signing and the transmission to the base-station.
5. After the inspection is complete, the base-station instructs the forensic components to delete their private key.
6. The forensic components then sign a message that states, that the next operation they will perform is the deletion of their private key.
7. After the base-station has received all those messages from each forensic component, the base-station signs the forensic package one last time and then deletes its private key.

This procedure seems very similar at first, however, it's the use of enclaves that solves the key deletion problem in this case. If in step 6 a YubiKey is used, a bad actor could simply unplug the key and prevent the key deletion. However, with enclaves, this is not a problem anymore. Firstly, the enclave protects the private key from being accessible by anyone but the enclave itself. This means that even a person with physical access could not extract the private key from the enclave's memory. Secondly, the enclave code cannot be modified to print out the key after sending the deletion message to the base station, as these changes would be immediately detected by the base-station during the remote attestation procedure. Finally, even if the key is not deleted and a bad actor shuts off the computer running the enclave (similar to unplugging the YubiKey before it is deleted), the private key would also be lost, as it's stored (encrypted) in memory, which deteriorates fast after the machine is turned off. But even if parts of the memory could be extracted using cold boot attacks, the memory content of the enclave would still be encrypted. Either way, the attacker is not able to extract the private key.

Bibliography

[1] 2023. Adminer - Database management in a single PHP file — adminer.org. https://www.adminer.org/. [Accessed 18-12-2023]. (2023).

[2] S. Ansari, S.G. Rajeev, and H.S. Chandrashekar. 2003. Packet sniffing: a brief introduction. *IEEE Potentials*, 21, 5, 17–19. DOI: 10.1109/MP.2002.1166620.

[3] 2023. Azure Kinect DK. Retrieved 10/24/2023 from https://azure.microsoft.com/de-de/products/kinect-dk.

[4] 2018. Azure Kinect DK Fact Sheet. Retrieved 10/24/2023 from https://news.microsoft.com/wp-content/uploads/prod/2019/06/Factsheet-Azure-Kinect-DK.pdf.

[5] Muhammad Shamraiz Bashir and MNA Khan. 2013. Triage in live digital forensic analysis. *International journal of Forensic Computer Science*, 1, 35–44.

[6] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. 2018. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 75–86.

[7] Klaus Biß, Jörg Kippe, and Markus Karch. 2023. Geräteerkennung und -identifizierung in industriellen Netzen. *at - Automatisierungstechnik*, 71, 9, 726–735. DOI: doi:10.1515/auto-2023-0135. https://doi.org/10.1515/auto-2023-0135.

[8] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. {ÆPIC} Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3917–3934.

[9] 2022. CAN Bus Explained - A Simple Intro [2023] — csselectronics.com. https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial. [Accessed 11-12-2023]. (2022).

[10] Confidential Computing Consortium. 2022. A Technical Analysis of Confidential Computing. Technical report. [Accessed 27-12-2023].

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive*.

[12] 2022. CVE-2022-0005. Available from MITRE, CVE-ID CVE-2022-0005. (2022). Retrieved 01/09/2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0005.

[13] 2022. CVE-2022-21233. Available from MITRE, CVE-ID CVE-2022-21233. (2022). Retrieved 01/11/2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21233.

[14] 2023. CyberLink Support Center — cyberlink.com. https://www.cyberlink.com/support-center/faq/content?id=26690. [Accessed 27-12-2023]. (2023).

[15] Matheus Bichara de Assumpção, Marcelo Abdalla dos Reis, Marcos Roberto Marcondes, Pedro Monteiro da Silva Eleutério, and Victor Hugo Vieira. 2023. Forensic method for decrypting TPM-protected BitLocker volumes using Intel DCI. *Forensic Science International: Digital Investigation*, 44, 301514.

[16] 2023. docker run (docker container run). Retrieved 10/30/2023 from https://docs.docker.com/engine/reference/commandline/run/.

[17] 2023. Documentation — redis.io. https://redis.io/docs/. [Accessed 18-12-2023]. (2023).

[18] 2023. Documentation: Table of Contents 2014; RabbitMQ — rabbitmq.com. https://www.rabbitmq.com/documentation.html. [Accessed 18-12-2023]. (2023).

[19] 2023. ego/samples/remote_attestation at master edgelesssys/ego — github.com. https://github.com/edgelesssys/ego/tree/master/samples/remote_attestation. [Accessed 08-01-2024]. (2023).

[20] Mike Gancarz. 2001. *Unix philosophy*.

[21] 2023. GitHub - deislabs/mystikos: Tools and runtime for launching unmodified container images in Trusted Execution Environments — github.com. https://github.com/deislabs/mystikos. [Accessed 02-01-2024]. (2023).

[22] 2023. GitHub - edgelesssys/edgelessrt: Edgeless RT is an SDK and a runtime for Intel SGX. It combines top-notch Go support with simplicity, robustness and a small TCB. Developing confidential microservices has never been easier! C++17 and Rust (experimental) are also supported. — github.com. https://github.com/edgelesssys/edgelessrt. [Accessed 02-01-2024]. (2023).

[23] 2023. GitHub - edgelesssys/ego: EGo is an open-source SDK that enables you to develop your own confidential apps in the Go programming language. — github.com. https://github.com/edgelesssys/ego. [Accessed 02-01-2024]. (2023).

[24] 2022. GitHub - google/asylo: An open and flexible framework for developing enclave applications — github.com. https://github.com/google/asylo. [Accessed 02-01-2024]. (2022).

[25] 2023. GitHub - intel/linux-sgx: Intel SGX for Linux* — github.com. https://github.com/intel/linux-sgx. [Accessed 02-01-2024]. (2023).

[26] 2023. GitHub - intel/sgx-ra-sample — github.com. https://github.com/intel/sgx-ra-sample. [Accessed 02-01-2024]. (2023).

[27] 2023. GitHub - openenclave/openenclave: SDK for developing enclaves — github.com. https://github.com/openenclave/openenclave. [Accessed 02-01-2024]. (2023).

[28] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Paper 2016/204. https://eprint.iacr.org/2016/204. (2016). https://eprint.iacr.org/2016/204.

[29] INNO-MAKER. [n. d.] Innomaker USB2CAN Device UserManual. Version 1.8. Retrieved 10/31/2023 from https://github.com/INNO-MAKER/usb2can/blob/master/Document%EF%BC%88read%20me%20first%EF%BC%89/USB2CAN%20UserManual%20v.1.8.pdf.

[30] 2022. INTEL-SA-00614 — intel.com. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00614.html. [Accessed 09-01-2024]. (2022).

[31] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N Asokan, Andrew Simpson, and Robin Ankele. 2016. Exploring the use of Intel SGX for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, pp. 1–6.

[32] Sandeep Kumar, Abhisek Panda, and Smruti R Sarangi. 2022. A Comprehensive Benchmark Suite for Intel SGX. *arXiv preprint arXiv:2205.06415*.

[33] Nicolai Kuntze, Carsten Rudolph, Aaron Alva, Barbara Endicott-Popovsky, John Christiansen, and Thomas Kemmerich. 2012. On the creation of reliable digital evidence. In *Advances in Digital Forensics VIII: 8th IFIP WG 11.9 International Conference on Digital Forensics, Pretoria, South Africa, January 3-5, 2012, Revised Selected Papers 8*. Springer, pp. 3–17.

[34] Systems Software & Security Lab. 2022. Home — sgx101.gitbook.io. https://sgx101.gitbook.io/sgx101/. [Accessed 30-12-2023]. (2022).

[35] Linux Kernel Organization. [n. d.] hidraw.txt. Retrieved 10/30/2023 from https://www.kernel.org/doc/Documentation/hid/hidraw.txt.

[36] 2023. Mbed TLS — trustedfirmware.org. https://www.trustedfirmware.org/projects/mbed-tls/. [Accessed 11-01-2024]. (2023).

[37] John Mechalas. [n. d.] Intel® SGX DCAP Quick Install Guide. (). https://www.intel.com/content/www/us/en/developer/articles/guide/intel-software-guard-extensions-data-center-attestation-primitives-quick-install-guide.html.

[38] Rebecca T Mercuri and Peter G Neumann. 2003. Security by obscurity. *Communications of the ACM*, 46, 11, 160.

[39] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting {SGX} Enclaves from Practical {Side-Channel} Attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pp. 227–240.

[40] Mark M Pollitt. 2007. An ad hoc review of digital forensic models. In *Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'07)*. IEEE, pp. 43–54.

[41] 2023. PostgreSQL: Documentation — postgresql.org. https://www.postgresql.org/docs/. [Accessed 18-12-2023]. (2023).

[42] 2023. python-can 4.3.1 documentation — python-can.readthedocs.io. https://python-can.readthedocs.io/en/stable/. [Accessed 16-12-2023]. (2023).

[43] 2023. Quality of Service settings 2014; ROS 2 Documentation: Rolling documentation — docs.ros.org. https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html. [Accessed 16-12-2023]. (2023).

[44] 2023. Remote attestation | EGo — docs.edgeless.systems. https://docs.edgeless.systems/ego/reference/attest. [Accessed 07-01-2024]. (2023).

[45] 2023. RIEGL VZ-400i. Retrieved 10/24/2023 from http://www.riegl.com/nc/products/terrestrial-scanning/produktdetail/product/scanner/48/.

[46] 2023. ROS 2 Documentation 2014 ROS 2 Documentation: Humble documentation — docs.ros.org. https://docs.ros.org/en/humble/index.html. [Accessed 16-12-2023]. (2023).

[47] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2020. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity*, 3, 1–20.

[48] Mick Seaman. 2023. 802.1AE: MAC Security (MACsec) | — 1.ieee802.org. https://1.ieee802.org/security/802-1ae/. [Accessed 09-02-2024]. (2023).

[49] 2016. SGX protected memory limit in SGX — community.intel.com. https://community.intel.com/t5/Intel-Software-Guard-Extensions/SGX-protected-memory-limit-in-SGX/td-p/1068817. [Accessed 08-01-2024]. (2016).

[50] Michal Sojka, Pavel Píša, Martin Petera, Ondřej Špinka, and Zdeněk Hanzálek. 2010. A comparison of Linux CAN drivers and their applications. In *International Symposium on Industrial Embedded System (SIES)*. IEEE, pp. 18–27.

[51] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. 2008. MD5 considered harmful today, creating a rogue CA certificate. In *25th Annual Chaos Communication Congress* number CONF.

[52] 2015. Stanford Seminar - Intel Software Guard Extensions — youtube.com. https://www.youtube.com/watch?v=mPT_vJrlHlg. [Accessed 05-01-2024]. (2015).

[53] John Tan. 2001. Forensic readiness. *Cambridge, MA:@ Stake*, 1.

[54] Julien Oberson Thomas Dewaele. 2021. TPM sniffing; Sec Team Blog — blog.scrt.ch. https://blog.scrt.ch/2021/11/15/tpm-sniffing/. [Accessed 28-12-2023]. (2021).

[55] 2024. three.js manual — threejs.org. https://threejs.org/manual/. [Accessed 29-01-2024]. (2024).

[56] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 991–1008.

[57] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. 2022. Sok: Sgx. fail: How stuff get exposed. (2022).

[58] Radu Velea, Casian Ciobanu, Florina Gurzau, and Victor-Valeriu Patriciu. 2017. Feature Extraction and Visualization for Network PcapNg Traces. In *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pp. 311–316. DOI: 10.1109/CSCS.2017.49.

[59] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2421–2434.

[60] 2021. Will SGX be deprecated? Issue 760 intel/linux-sgx — github.com. https://github.com/intel/linux-sgx/issues/760. [Accessed 28-01-2024]. (2021).

[61] 2023. Wireshark Developer 2019;s Guide — wireshark.org. https://www.wireshark.org/docs/wsdg_html_chunked/. [Accessed 19-12-2023]. (2023).

[62] 2023. yubico. Retrieved 10/30/2023 from https://www.yubico.com/.

Appendix A

sgx_default_qcnl.conf File

```
{
    // *** ATTENTION : This file is in JSON format so the keys are case sensitive. Don't change them.

    //PCCS server address
    "pccs_url": "https://localhost:8081/sgx/certification/v4/"

    // To accept insecure HTTPS certificate, set this option to false
    ,"use_secure_cert": false

    // You can use the Intel PCS or another PCCS to get quote verification collateral.  Retrieval of PCK
    // Certificates will always use the PCCS described in pccs_url.  When collateral_service is not defined, both
    // PCK Certs and verification collateral will be retrieved using pccs_url
    //,"collateral_service": "https://api.trustedservices.intel.com/sgx/certification/v4/"

    // If you use a PCCS service to get the quote verification collateral, you can specify which PCCS API version is to be used.
    // The legacy 3.0 API will return CRLs in HEX encoded DER format and the sgx_ql_qve_collateral_t.version will be set to 3.0, while
    // the new 3.1 API will return raw DER format and the sgx_ql_qve_collateral_t.version will be set to 3.1. The pccs_api_version
    // setting is ignored if collateral_service is set to the Intel PCS. In this case, the pccs_api_version is forced to be 3.1
    // internally.  Currently, only values of 3.0 and 3.1 are valid.  Note, if you set this to 3.1, the PCCS use to retrieve
    // verification collateral must support the new 3.1 APIs.
    //,"pccs_api_version": "3.1"
    // Maximum retry times for QCNL. If RETRY is not defined or set to 0, no retry will be performed.
```

```
    // It will first wait one second and then for all forthcoming retries it will double the waiting time.
    // By using retry_delay you disable this exponential backoff algorithm
    ,"retry_times": 6

    // Sleep this amount of seconds before each retry when a transfer has failed with a transient error
    ,"retry_delay": 10

    // If local_pck_url is defined, the QCNL will try to retrieve PCK cert chain from local_pck_url first,
    // and failover to pccs_url as in legacy mode.
    //,"local_pck_url": "http://localhost:8081/sgx/certification/v4/"

    // If local_pck_url is not defined, set pck_cache_expire_hours to a none-zero value will enable local cache.
    // The PCK certificates will be cached in memory and then to the disk drive.
    // ===== Important: Once the local cache files are created, currently there is no other way to clean them other
    //                  than to delete them manually, or wait for them to expire after "pck_cache_expire_hours" hours.
    //                  To delete the cache files manually, go to these foders:
    //                       Linux : $AZDCAP_CACHE, $XDG_CACHE_HOME, $HOME, $TMPDIR, /tmp/
    //                       Windows : $AZDCAP_CACHE, $LOCALAPPDATA\..\..\LocalLow
    //                  If there is a folder called .dcap-qcnl, delete it. Restart the service after all cache
    //                  folders were deleted. The same method applies to "verify_collateral_cache_expire_hours"
    ,"pck_cache_expire_hours": 168

    // To set cache expire time for quote verification collateral in hours
    // See the above comment for pck_cache_expire_hours for more information on the local cache.
    ,"verify_collateral_cache_expire_hours": 168

    // When the "local_cache_only" parameter is set to true, the QPL/QCNL will exclusively use PCK certificates
    // from local cache files and will not request any PCK certificates from service providers, whether local or remote.
    // To ensure that the PCK cache is available for use, an administrator must pre-populate the cache folders with
    // the appropriate cache files. To generate these cache files for specific platforms, the administrator can use
    // the PCCS admin tool. Once the cache files are generated, the administrator must distribute them to each platform
```

```
    // that requires provisioning.
    ,"local_cache_only": false

    // You can add custom request headers and parameters to the get certificate API.
    // But the default PCCS implementation just ignores them.
    //,"custom_request_options" : {
        //  "get_cert" : {
            //    "headers": {
                //      "head1": "value1"
                //    },
            //    "params": {
                //      "param1": "value1",
                //      "param2": "value2"
                //    }
            //  }
        //}
}
```

Listing A.1: Modified configuration file ”/etc/sgx_default_qcnl.conf” which the demonstrator application requires for the remote attestation.[44]