Decentralized Data Acquisition Pipeline with Machine Learning For Side-Channel Information

by

Goksen Umut Guler

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Sciences
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Technological advancements and the COVID-19 pandemic caused an increase in the adoption of technologies, services, and computers. Public cloud services surged with a 17% increase, and adoption of software services such as online video conferencing tools increased for national and industrial actors. Subsequently, security became a crucial component due to increased adoption, connectivity, and cybersecurity risks of services and systems. The heightened interest from individuals, organizations and national actors in the security domain is not without cause, as security breaches caused by malicious actors surged in parallel. Security researchers and experts leverage their expertise to overcome threats by malicious actors.

The side-channel domain is an active research topic for security experts. Side-channel information is gathered from the involuntary leak of information from a system, which can represent a vulnerability for corporations and individuals alike. Security researchers and malicious actors have shown that they can use side-channel information to attack and protect systems. For instance, a malicious actor can attack a system by extracting secrets using side-channel information such as power consumption or electromagnetic emissions. In contrast, protection of a system to help detect malware and attacks against a system is also possible by using side-channels such as cache and power consumption.

Analyzing side-channel information is possible through different methodologies such as machine learning. Studies have shown that machine-learning models process side-channel information and help achieve the analysis goals with high accuracy and precision. However, machine-learning algorithms require large datasets, and in this case, this means a large number of samples from the used side-channels. The need for such datasets motivates this thesis to discuss the challenges and an approach to collecting large datasets of side-channel information from multiple systems.

The challenge of reliably capturing side-channel information for later analysis grows with the number of assessed targets, the number of channels, the sampling rate, and the resolution of each sample. Side-channel data acquisition relies on physical access to target systems, making it challenging to collect data from several devices. Thus, to enable machine learning models and a robust analysis process, side-channel data acquisition requires a scalable, decentralized, and consistent approach to collect data. To solve the scalability issue around collecting side-channel information from several systems, we propose a data pipeline architecture to collect side-channel information that fulfills quality attributes such as maintainability, reusability, reliability, and scalability.

Acknowledgements

In this thesis, the case-study section draws some conclusion from the implemented solutions by the Embedded Systems Group under Prof. Sebastian Fischmeister.

Parts of the case-study text is derived from the Electromechanical Emissions Tripwires (EET) project reports for Defence Research and Development Canada, which is written by part of the Embedded Systems Group and Palitronica Incorporation.

Dedication

I dedicate this thesis to my family, which has supported my journey in academia with unconditional love, respect and help. Also, I graciously thank my lover, who worked with me daily despite time zone differences while giving her best to support me emotionally.

Table of Contents

List of Figures

List of Tables

List of Abbrevations

ADC Analog-to-digital converter 61, 63

ADS-B Automatic Dependent Surveillance–Broadcast 1

CAN Controller Area Network 5

CIDS Collaborative Intrusion Detection System 59

CSV Comma Separated Values 18–20, 24

CVE Common Vulnerabilities and Exposures 1

DAC Data Acquisition Controller 14, 15

DUT Device Under Test 4, 12–14, 16–18, 35, 42, 46–50, 54

ECU Electronic control unit 5

EET Electromechanical Emissions Tripwires iv, viii, 58–67

FPGA Field-programmable gate array 61

IAM Identity and Access Management 25

IC Interceptor Controller 21

IDS Intrusion detection system 2, 59, 60

IHP Interceptor Heartbeat Packet 17, 18

IRF Interceptor Receiver & Forwarder 17, 18, 20, 21

IT Interceptor Transmitter 14, 15

JSON JavaScript Object Notation 18–20, 24

PCA Principal component analysis 6

PTP Precision Time Protocol 16, 21, 22, 58

SAAM Software Architecture Analysis Method 29–34, 36, 37, 45, 52, 56, 57, 62, 64, 65, 67, 68

TCP Transmission Control Protocol 28

UDP User Datagram Protocol 27, 28, 63

Chapter 1

Introduction

Computer systems are becoming more connected every day, and there is an increasing need for cybersecurity solutions to secure these systems. For instance, cars now include systems that enable autonomous driving and use navigation systems. Airplanes use Automatic Dependent Surveillance–Broadcast (ADS-B) systems and flight entertainment systems connected to a network. Ships report their positioning through satellite communications and provide the live status of their cargo. Power plants use advanced control systems that connect them to federal and provincial grid management systems. As systems get more connected, security risks also surge. Vulnerabilities can cause severe damage and incur costs to infrastructure and people.

The MITRE Corporation introduced a standard dictionary to track vulnerabilities called Common Vulnerabilities and Exposures (CVE) [1]. Security threats caused by the increase in connectivity are visible based on the expansion of CVE. Since 2016, CVE maintains a steady rise in vulnerabilities recorded [2]. Vulnerabilities such as CVE-2019-9977 allow malicious actors to alter the driving functions of a car [3]. CVE-2016-9361 can steer a ship off course [4], and CVE-2019-9019 enables malicious actors to use a buffer overflow to attack the entertainment system on Boeing 777-36N(ER) planes [5]. Each vulnerability can incur a high cost to lives and infrastructure. Individuals, governments, and organizations increasingly spend resources replacing, renewing, and hardening existing systems to mitigate the risks of cybersecurity issues [6].

Side-channel information leaks are another type of vulnerability that malicious actors and experts leverage for attacking and defending systems. The cause of the vulnerability is the involuntary information leak from a system. Analyzing side-channel information may yield secrets and details about a system, and machine learning is one of the many possible

approaches to analyzing side-channel information.

Side-channel information is rich data that lends itself to machine learning. Studies indicate that machine learning models can extract patterns from data with high accuracy [7]. For example, Lerman et al. show that Random Forest models apply to side-channel information with little context on a black-box setting [8]. Similarly, Benadjila et al. describe that transforming template attacks to a machine learning model is possible [9]. Backes et al. show another study that uses acoustic side-channel combined with machine learning models to extract information from printers [10].

Exploiting side-channel information for malicious purposes can hamper security, privacy and pose threats to individuals, organizations, and nations. Researchers show that cryptographic functions, personal computers, cloud servers, and mobile systems are vulnerable to side-channel information leaks [11][12][13][14]. These attacks pose a threat to national security, trade secrets, and privacy. Studies indicate that systems with no countermeasures against side-channel information leaks are often vulnerable to attacks. Literature and studies indicate that by exploiting side-channel information, it is possible to protect, validate and verify integrity on a wide array of systems, including but not limited to cloud systems, embedded systems, and safety-critical systems [15][16][17][18].

Using Intrusion detection system (IDS) is a common approach in the field to protect systems and IDSs which leverage side-channel information are getting more attention with studies showing that anomaly detection and analysis of side-channel information yields accurate results about a system [15]. A challenge with side-channel based IDS is that IDSs require collecting, cleaning and preparing data. These are repetitive tasks that every researcher needs to do. Thus, this thesis aims to provide a generalized framework for researchers using machine learning with side-channel data and researchers who aim to collect large amounts of side-channel data.

This thesis contributes to the literature by providing the following:

- A generalized approach to collect side-channel information from several systems, Chapter 3.
- A decentralized data pipeline architecture which provides reliable and secure access to the data, discussed in Chapter 3.
- An assessment of the data pipeline architecture in which the architecture's capabilities and provides considerations for future side-channel data acquisition systems, discussed in Chapter 4.

- A use-case based on the proposed architecture highlights the benefits and comparison to our findings in the pipeline assessment, discussed in Chapter 5.

Chapter 2

Background Information

To understand the data pipeline architecture understanding the processes and the terminology is a must. This section provides the necessary background information concerning the scope of the thesis and the architecture.

2.1 Side-Channel Information

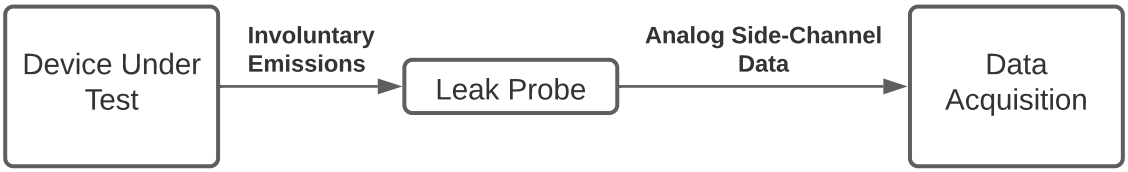
Side-channel information is involuntary information emissions from electronic components such as computers and embedded devices. Various emissions such as acoustic, power, and electromagnetic radiation exist and are labelled as side-channels.

To find and measure involuntary emissions, physical access to the target system is often a requirement. Typical side-channel data acquisition systems include a leakage probe and an acquisition device such as a digital oscilloscope to collect side-channel information [19]. Figure 2.1 shows an example side-channel acquisition system. For instance, the Device Under Test (DUT) can be a computer, the leak probe can be a shunt resistor, and the data acquisition component can be a digital oscilloscope.

Researchers show that using side-channel information makes it possible to attack a system [11][20]. Attacks include but are not limited to differential power analysis, timing attacks, and cache attacks [11][20][21].

Exploiting side-channel information to provide defensive measures is another use for side-channel information. Studies demonstrate that side-channel information is a viable way to monitor threats and verify run-time operations for a system. For example, Liu et al.

Figure 2.1: A high level view for a typical side-channel data acquisition system



show that it is possible to implement an authentication and verification layer for Controller Area Network (CAN) by characterizing voltage from Electronic control units (ECUs) [22]. Another protection system, called CloudRadar, uses cache-based side-channel information to identify anomalies in multi-tenant cloud systems [15].

Side-channel analysis often involves digital signal processing, statistical methods and machine learning. These methods contribute to understanding the patterns and information in side-channel data. Kocher et al. posit that to analyze data more efficiently, digital signal processing is an applicable method. CloudRadar also includes various preprocessing techniques and digital signal processing techniques to provide real-time protection to cloud systems [11][15].

2.2 Machine Learning

Machine learning can be described as "automatically recognizing patterns in data and employing extracted patterns to predict future patterns, and making decisions based on the patterns". In terms of knowledge and purpose, there are many intersections between machine-learning, statistics, and data mining [23]. Nevertheless, the tools needed to solve these problems often differ [24]. For instance, data mining can use rules extracted from the data, whereas a machine learning model extracts these patterns to formulate a problem.

Machine learning applies to various problems and includes an assemblage of broadly defined approaches to solving these problems. These approaches are called supervised learning, unsupervised learning, and reinforcement learning [23]. Typical machine learning problems include but are not limited to stock prediction, weather forecasting, malware analysis, spam filtering, threat identification, and anomalous behaviour detection [23].

Depending on the problem that machine learning is trying to solve, models can predict or classify data. The classification models categorize data points, whereas predictive models create data points based on historical data. For example, detecting fraudulent transactions with a machine learning model uses classification. For this problem, the model tries to categorize transactions based on the historical data into two categories which are safe and fraudulent. On the other hand, weather forecasting uses a predictive model. The predictive model tries to guess whether information for the future using historical weather data.

Machine learning algorithms produce models that can map data collected for the problem statement to provide outputs based on the mapping [25]. An essential step to building machine learning models is understanding underlying data and the problem.

Studies demonstrate that side-channel information analysis using machine learning is possible. Problems associated with side-channel information are solvable using supervised and unsupervised approaches [26]. Machine learning models can identify patterns and learn the characteristics of side-channel information. For example, machine learning models were able to identify malware in the 2019 study by Adnan et al. successfully [17].

Another prevalent name for supervised learning is classification or pattern recognition [27]. By utilizing labelled data, supervised learning seeks to match patterns in the data to labels. For example, identifying which emails are spam and not by user input is supervised learning.

Unsupervised learning implies finding patterns in data without any additional input than the data itself [27]. With this approach, the problem statement is usually broader, and often there is no prior information about patterns in the data. For instance, detecting fraud in online payment systems using log information generated by the users is an example problem that uses unsupervised learning.

2.2.1 High Dimensional Data

Machine-learning applications that utilize data with high dimensions create challenges to researchers and industry alike. Having a high number of dimensions is deemed a curse [28][29]. Researchers attest that it is possible to lift this curse to a particular degree with dimension reduction techniques [29][30][31]. Techniques such as principal component analysis and manifold learning can be proffered as examples of dimension reduction techniques [32][33]. One challenge of these approaches is that it usually requires an understanding of data to a certain degree. For instance, techniques such as Principal component analysis (PCA) solely apply to linear data. Another challenge is that dimension reduction techniques may remove patterns or other valuable information while processing data [34].

In the context of side-channel information, gathered data is often time-series or frequency data. Both time-series and frequency data usually have high dimensionality and large samples with high resolution. Oscilloscope or similar acquisition devices often collect side-channel information, and these devices may collect a high number of samples for several purposes [19]. One notable reason for having high sampling rates is to preserve the resolution of the data, which contributes to noise reduction [35]. For example, Agrawal et al. show that collecting electromagnetic emissions at higher sampling rates enables new opportunities to craft new attacks [20]. Noise reduction is not the only benefit of having high-resolution data. Studies also indicate that having data with high resolution may help machine-learning models extract patterns more efficiently [36].

2.2.2 Large Datasets

Machine learning models can solve more complex problems and learn more with larger datasets [37][36]. With increased connectivity and the adoption of systems, it is possible to construct large datasets for machine learning [38][36]. Computer vision-related problems and machine learning approaches are well-studied examples of using large datasets to create more robust machine learning models [39]. For example, datasets such as UCF101, CIFAR-10, Kinetics-700 include large amounts of data curated to solve computer vision problems. Data sizes vary between datasets; however, they contain a large number of samples with metadata available. For instance CIFAR-10 includes 60 000 images that are 32x32, UCF101 includes 13 320 video snippets that cover 101 different classes of human actions, and Kinetics-700 includes snippets from approximately 650 000 videos with high resolution [40][41][42]. The curation of the datasets enables researchers to test different machine learning models without requiring further data acquisition, making it feasible to test different approaches requiring different sizes of datasets. Nevertheless, large datasets have a common problem which is scalability. As datasets grow in size, manipulating and managing datasets require more computational resources to conduct machine learning. Thus, scalable solutions for curating large datasets are a must.

Unlike most systems, such as personal computers, collecting side-channel information and data from embedded systems poses challenges in scalability due to the need for physical access [19]. Collecting data from many systems may impact the scalability of data acquisition methods, making it harder to create curated and large datasets. For instance, OpenAI left the research space for robotics due to the lack of data and ability to collect new datasets with relative ease [43].

Chapter 3

Architecture

Architectures can link abstract goals for a project with a system design [44]. Abstract objectives, however, can be explained differently or understood differently by individuals; this would lead to individuals conjecturing a variant of the project, goal and terminology. An architecture with formal definitions will remove abstract aspects of the architectural design that can cause confusion [44]. A system is a combination of structures. Thus we can say that architecture defines a standard vocabulary for structures within a system [45].

Structures divide into system design categories; these are static modules, dynamic components, and organizational structures. Each structure represents a different aspect of an architecture; static modules often represent the foundation of the architecture and involve technical details about the core functionality; dynamic components explicate the interchangeable parts in an architecture, which usually consists of business and technical information. The organizational structures mainly capture the business-related perspective of architecture.

Defining an architecture around the set of structures we described also requires formalizing the requirements for architecture. Requirements can be technical and non-technical, informing the designers about the core functionality and the business logic. Studies show that capturing the requirements can be a decisive factor for the success of an architecture [44].

In this chapter, we define our requirements for a decentralized data acquisition pipeline. We formalize requirements, assumptions, and design choices within the requirement section while considering our case study and other studies. As mentioned earlier, our architectural design aims to collect side-channel information from many systems. Thus, the requirements include items concerning side-channel information. Nonetheless, we note that our

approach may prove helpful in other fields by adjusting the provided requirements to curate a generalized approach.

Requirements define architectural functionality by describing expected features. To formalize the functionalities, a functional block diagram visually describes the structure. The visual representation of the functionality help lay the groundwork required for explaining components within the diagram. The explanations define the interactions between different components.

With the functionality of the architecture and architectural components defined, we provide an example deployment scheme for our architecture. We describe the example deployment through a deployment diagram and give information about deploying the architectural components.

3.1 Requirements

In system design, defining or capturing requirements is considered to be one of the significant steps [44]. To capture the requirements, we follow the standard approach of splitting them into functional and non-functional requirements [46]. Functional requirements specify how a system behaves and the use-cases of a system. Non-functional requirements, on the other hand, defines how a system should perform the use-cases and the functionality. For example, uploading data after collection is a functional requirement, whereas uploading data after 1 second the data collection occurs is a non-functional requirement.

Both functional and non-functional requirements describe how and what the architecture does and contain business and technical details. As technical and business requirements often collide, the architecture design needs to compensate by making compromises. For example, encryption in a data pipeline, a functional requirement, can reduce the pipeline's throughput. In contrast, a non-functional requirement may specify the need to operate on a specific throughput and latency.

Our motivation is to minimize the effects caused by compromises and design a scalable decentralized data acquisition system for side-channel information. For this purpose, we define our requirements from the business and technical scope of the architecture. The terminology between requirements may differ based on the scope of a requirement—for instance, business requirements related to management, budget and timelines regarding the project. In comparison, technological requirements relate to the project's technical aspect and differ based on the domain. Establishing a common terminology between both

technical and business requirements prevents confusion caused by different terminologies between scopes.

3.1.1 Terminology

The terminology required to describe the requirements and features of architecture differs from scope to scope. Thus, we first define a common language to capture and formalize the terminology used to describe the requirements.

- **Enrichment** refers to the process of adding contextual information to another information bit.
- **Storage** refers to the system which stores data.
- **System** refers to the Decentralized Data Acquisition Pipeline with Machine Learning for Side-Channel Information and encapsulates all the components of the architecture and the architecture.
- **Endpoint** refers to the device from which data is being gathered.
- **Channels** refers to data sources such as event data and continuous time-series data.
- **Hot Storage** refers to data sources such as event data and continuous time-series data.
- **Cold Storage** refers to data sources such as event data and continuous time-series data.

3.1.2 Functional Requirements

Functional requirements set the standard functions and behaviours of a system [47]. As behaviours and system functions capture what the system is capable of, they also provide insight with regards to the system use-cases [48]. Use-cases capture actions of the system and reflect technical details regarding the functionality of architecture. The technical details provide the level of detail required to develop architecture. In this case, the primary use case of the proposed architecture is to collect and store data.

To make the requirements accessible, we assign a number prepended with the abbreviation FR, functional requirement and then further detail the requirements with short descriptions.

FR-1 *The system shall support capturing data from multiple endpoints and channels.*

FR-2 *The system shall support query-able centralized access to hot and cold data.*

FR-3 *The system shall provide mechanisms to control and manage the data acquisition process and data storage.*

FR-4 *The system shall support attaching meta information to data.*

FR-5 *The system shall provide a callback mechanism at each stage of the data acquisition process.*

FR-6 *The system shall support synchronizing data from multiple endpoints and channels.*

FR-7 *The system shall provide access to performance metrics of the pipeline and storage.*

FR-8 *The system shall provide role-based access control to data and management functions.*

FR-9 *The system shall support seamless updates of components.*

3.1.3 Non-Functional Requirements

Only by understanding non-functional and functional requirements designers and developers can implement the structures within an architecture [47][49], Section 3.1.2 provides the functional requirements for the architecture and this section provides non-functional requirements. Non-functional requirements of a system focus on items concerning the performance of a system [47]. In contrast to the functional requirements, non-functional requirements do not focus on behaviours of the system [48]. Non-functional requirements show importance as they underline the limitations for performance, scalability, reusability, accessibility and maintainability [47].

To define the properties of our architecture, we follow the same format we follow in Section 3.1.2 and assign numbers to the requirements prepended with NFR, standing for non-functional requirements. The list of non-functional-requirements is as follows.

NFR-1 *Query mechanism should provide at least 10 seconds of data.*

NFR-2 *The system shall support capturing a minimum of two channels simultaneously.*

NFR-3 *The system shall provide a lifetime policy of 24 hours for hot storage.*

NFR-4 *The system shall provide a lifetime policy of a minimum of 30 days for cold storage.*

NFR-5 *The callbacks from the system shall trigger within 3 minutes.*

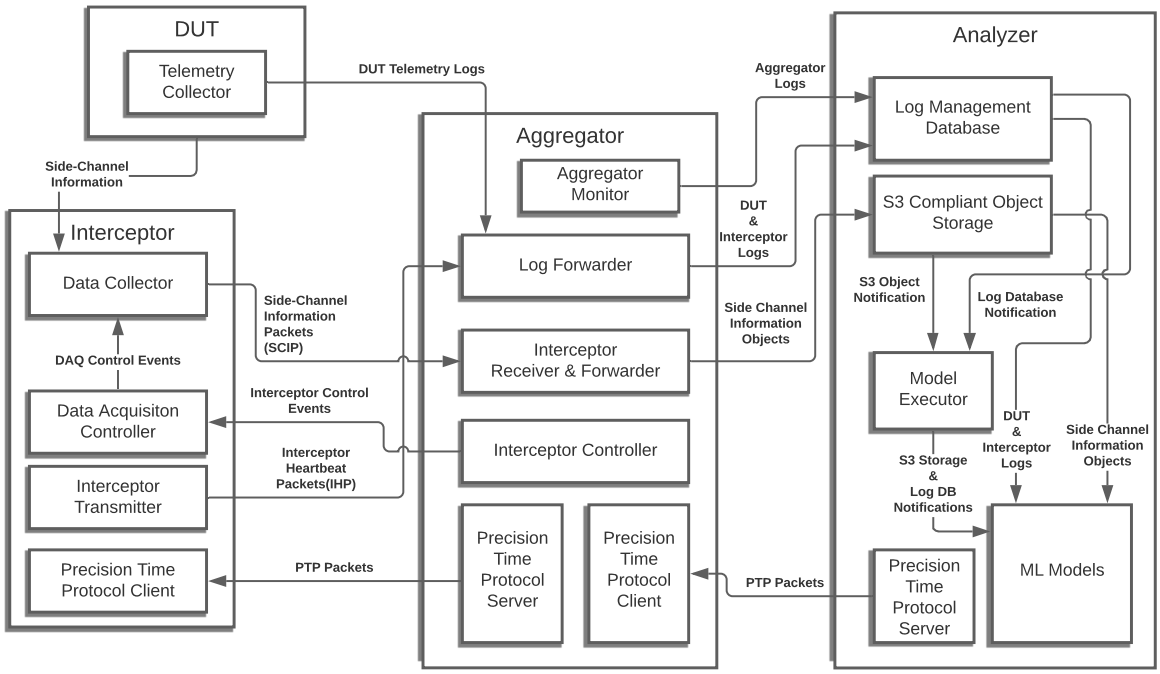
3.2 Functional View

Formalization of structures within an architecture mitigates risks that may occur during the implementation phase of an architecture [44]. Formalization removes abstractions for the system as developers can interpret abstract information differently. Thus, formalizing an architecture involves creating detailed technical diagrams, as diagrams provide the means to formalize the structures and technical details without requiring abstract descriptions. For example, a functional block diagram describes the structural mapping and technical information necessary to implement an architecture. By doing so, diagrams allow developers to reach a mutual understanding of the architecture.

This section provides a functional block diagram to show the structure of the architecture we propose by reducing and removing any abstraction involved. Each item in the architecture, denoted as a box, in Figure 3.1 refers to a structure or, in other words, a component. The lines between the boxes represent the interaction and the information transferred between the components. The architecture contains four main components. These are DUT, Interceptor, Aggregator, and Analyzer. Figure 3.1 shows the architecture by defining these components, provided with a description of each component with details such as use cases. Each component introduces different functionalities to the architecture based on the requirements defined.

Figure 3.1 shows that the architecture consists of four significant components called DUT, Interceptor, Aggregator and Analyzer. DUT in this case refers to the target device for side-channel information collection. The Interceptor is the component responsible for collecting side-channel information. The Aggregator includes the functionality of gathering side-channel information from Interceptor(s) and enriching data. The Analyzer component is responsible for the side-channel information analysis.

Figure 3.1: Functional Block Diagram



Below we describe each component and the functionality which these components encapsulate by describing the sub-components as shown in Figure 3.1.

3.2.1 Device Under Test (DUT)

Figure 3.1 shows the structure of the DUT component within the architecture. DUT is the target system from which side-channel information gets collected.

Enriching side-channel information by using meta-information is a method to provide analysts with the means to correlate and differentiate the data, as required by **FR-4**. To facilitate this functionality the DUT uses the Telemetry Collector sub-component within the DUT to send meta-information to the Aggregator component.

Telemetry Collector

Telemetry Collector collects meta-information from the DUT, as required by **FR-4**. By providing meta-information, Telemetry Collector enables attaching meta-information to side-channel data. After collecting meta-information, the Telemetry Collector relays them to the Aggregator. The Aggregator component uses meta-information to enrich the side-channel data, enabling different machine learning approaches, such as supervised learning. Having enriched data also allows correlating the data. For instance, classification problems can use the meta-information as labels by correlating data based on DUT to train a supervised model.

3.2.2 Interceptor

Interceptor component is responsible for the data collection from a DUT, including meta-information from the DUT and side-channel information. The Interceptor contains three major functions. These functions are; data acquisition, data transmission, and controlling the data acquisition.

Data Collector

Data Collector interacts with single or multiple leak probes to collect side-channel information, as required by **FR-1**. The Interceptor is the initial component that interacts with the data. Side-channel information collection often takes place by using physical probes targeting the DUT. Data Collector can also utilize digitally available leak probes and collect data from them by interacting with the DUT. Data Collector then forwards the data for Interceptor Transmitter (IT) transmission. Before transmitting the information, intermediary steps such as buffering, filtering, and preparing data for machine learning are possible tasks for the Data Collector as well as the IT. Depending on the side-channel source and the representation of data, implementation of pre-processing and analysis may vary. Thus, to generalize the approach, the architecture design considers implementations concerning the side-channel leak probe and pre-processing as implementation details that can change due to end-user requirements.

Data Acquisition Controller (DAC)

Data Acquisition Controller (DAC) is the main controller of all operations within the Interceptor component, as required by **FR-3**. It provides an interface to configure the

data acquisition process. The communication that takes place, as shown in Figure 3.1, is through network packets that we call Interceptor Control Events. These events are, in essence, commands which change the internal configuration and setup of an Interceptor. Commands change, configure, disable, and enable any sub-component within the Interceptor. Changing sampling rate, enabling or disabling transmission, changing settings regarding the probe, and changing resource utilization of the Interceptor can be given as examples to DAC commands.

Interceptor Transmitter (IT)

The IT is a sub-component of the Interceptor, which transmits data to the Aggregator component as shown in Figure 3.1 and a requirement for implementing **FR-1**, **FR-2**, **FR-4**, and **FR-7**. The communication layer between an Aggregator and an Interceptor relies on IT. Communication takes place via network packets that include side-channel information and meta-information.

The architecture collects data from many systems, and due to the number of systems involved, monitoring helps maintain the pipeline and ensure reliable data. Monitoring contributes to having a scalable and maintainable approach to collect data by providing insight about the deployed hardware, software, and data [50]. The Interceptor enables monitoring by sending packets to an Aggregator instance as shown in Figure 3.1. These packets are Interceptor Heartbeat Packets, a term that covers all metrics and internal logs acquired from Interceptors themselves. Metrics and internal logs show the configuration of the Interceptor and metrics such as throughput, errors, and latency logs. As logs and metrics include details about the data and the pipeline, they can also serve as meta-information to provide means to differentiate experiments. By monitoring these logs, it is possible to provide statistics about the pipeline's state. Correlating telemetry information from Interceptors can serve as a means to provide labels to the side-channel data.

The Interceptors collect data at high sampling rates to preserve the resolution, which increases the size of data transfers consuming large amounts of bandwidth. To preserve more bandwidth and allow higher sampling rates on the side-channel collection IT reduces data sizes by compressing and hashing certain types of logs and metrics. Hashed information includes certain bits of internal logs, such as the configuration of the Interceptor. Configuration of an Interceptor can be extensive in size due to the information required to configure the data acquisition process. IT uses compression algorithms such as Snappy, Zlib, and Gzip to further reduce the size of the metrics and logs by compressing them.

Hashing and deploying a large number of systems require a versioning system to keep

track of configurations. For this purpose, Analyzer stores configurations and their respective hashes before deploying the configuration to an Interceptor. The Log Management Database is the main component that stores the Interceptor configurations and their respective hashes. Provided the query mechanisms of the Log Management Database, it is possible to use hashes from Interceptors to fetch the related configuration for an Interceptor from Log Management Database.

Precision Time Protocol (PTP) Client

The analysis uses meta-information to enrich the side-channel information. As the acquisition methods and sampling rates differ between side-channel data and meta-information, there is a need for synchronization between the meta-information and data, as required by **FR-6**. To facilitate time synchronization, the Interceptor makes use of Precision Time Protocol (PTP).

PTP is a time synchronization protocol defined by the IEEE 1588 standard series. Synchronization of time between multiple systems takes place via the help of network packets. The architecture implements PTP based on IEEE 1588-2019 [51]. PTP server is the main synchronization point and distributes the necessary network packets to facilitate time synchronization. At the same time, PTP clients make use of the network packets to synchronize the time to the time of the PTP server.

The time-synchronized between Aggregators and Interceptors allows precise timestamps in the logs, meta-information and the side-channel information. An example use case that shows the usefulness of time synchronization is the correlation between data from multiple DUTs.

3.2.3 Aggregator

The Aggregator is a component that handles the transfer layer between Interceptors and the Analyzer. Side-channel information often consists of millions of samples; multiple Interceptors in a network will transmit copious amounts of data. As machine learning and analysis often employ large computational resources, deploying high-resource hardware for several Interceptors is not scalable due to the costs attached. The Aggregator acts as an intermediary layer of hardware and software to help reduce costs. The structure of the Aggregator is displayed in Figure 3.1. Each instance of an Aggregator represents a node, a term commonly used within the distributed systems domain [52]. Below we describe the

functionality of sub-components of the Aggregator and provide information on use-cases regarding an Aggregator node.

Aggregator Monitor

The Aggregator Monitor is responsible for monitoring sub-components and providing meta-information, and metrics from sub-components to the Analyzer, as required by **FR-7**. A crucial reason behind monitoring Aggregators is to ensure that all operations within the Aggregator are uninterrupted and the data is reliable. The Aggregator Monitor serves a critical role in scaling the architecture by collecting necessary metrics to form a decision to scale the resources for the pipeline [53]. Metrics such as throughput and bandwidth usage are example metrics that can help form a decision. Due to the number of systems from which the data pipeline collects data, a monitoring system also provides the necessary tools and metrics regarding the data. In return, monitoring the data ensures that the data is reliable. For instance, users can acknowledge that the pipeline is running at full capacity, meaning no data was lost during that period.

Log Forwarder

Log Forwarder sub-component relays data from multiple Interceptors to the Interceptor Receiver & Forwarder (IRF) and the Analyzer, which is a requirement specified by **FR-4**, and **FR-7**. Received data consists of different types, Interceptor Heartbeat Packet (IHP) and DUT metrics, logs and meta-information. The IRF uses metrics, logs, and meta-information to tag and enrich side-channel information.

The Analyzer stores files as objects in an S3 Storage and uses tags and the available meta-information from the objects attached in the Aggregator. In a way, Log Forwarder provides data necessary to provide context to the side-channel information as well as metrics and logs to monitor the system.

Interceptor Receiver & Forwarder (IRF)

The IRF sub-component is responsible for collecting side-channel data from several Interceptors that have access to the Aggregator instance, as required by **FR-1**, and **FR-2**. The communication layer consists of network packets between Interceptors and Aggregators. IRF receives the packets and decodes them into two different types of packets. These packets are IHP and Side-Channel Information Packets.

Interceptor Heartbeat Packet (IHP) Definition we provided on section 3.2.2 was that each IHP provides telemetry information regarding the Interceptor sending the IHP and metadata regarding the data being collected. IRF first decodes IHPs and then stores the log information. We define the decoding process as operations that do not alter or modify the raw data.

Side-Channel Information Packet (SCIP) Side-Channel Information Packets contain side-channel data and compromises most of the traffic between an Interceptor and an Aggregator. IRF minimizes the number of processing done on the side-channel information to maintain throughput to ensure that the Aggregator does not require many resources. The IRF processes SCIPs in the order of adding meta-information to data, decoding, and compression of data. The decoding process is the same process we define in Section 3.2.3.

Adding meta-information requires some assumption about the IHP packets received, as the interval may vary compared to SCIPs. Thus, we act with the assumption that the last received IHP is accurate for the given side-channel information. Metrics, log and meta-information, should be sent from DUT at the same rate with SCIPs to provide more resolution if needed.

The IRF converts the decoded SCIPs with additional meta information to a file to store them in the Analyzer. The file can use various file formats depending on the requirements of the user. For example, some users may prefer Comma Separated Values (CSV) format, and the others may prefer JavaScript Object Notation (JSON) and so on. However, to fully utilize the architecture and features, the architecture fully integrates with a handful of file formats. This chapter provides a comparison table for a set of file formats to show which file formats fully integrate with the architecture.

The Aggregator is the only communication point to the Analyzer, thus to preserve bandwidth, the Aggregator implements methods such as compression during file generation. File creation and file formatting occur in volatile memory, reducing the number of I/O operations to preserve throughput. During this process, the IRF compresses data, reducing the file sizes. Reduction in the data sizes allows preserving Aggregator bandwidth, which in return allows more.

The IRF streams the files to the side-channel data storage system. The architecture uses S3 Object Storage as the side-channel data storage system, as shown in Figure 3.1. However, alternative solutions and file systems do not provide the same set of tools as an S3 Object Storage. Section 3.2.4 provides a list of alternatives and discusses them to the S3 Object Storage to detail the design decision behind using S3 Object Storage.

Different file-formats support different use-cases and need to satisfy the architecture requirements. For example, retrieving the file in chunks, query support with the storage system is a requirement specified in Section 3.1. The set of file-formats provided in Table 3.1 relates to big data-related problems or commonly used file formats for machine learning and analysis. Thus, the majority of the file-formats meet the requirements for analysis involving large amounts of data. Table 3.1 shows the comparison of file-formats based on three different categories. These categories are S3 query support, compression support and chunk support. To provide further details about the formats, we offer a short description of our evaluation for each file format.

File Format	**S3 Query Support**	**Compression**	**Chunk**
Raw Binary	-	-	-
CSV	+	GZIP or BZIP2	+[1]
JSON	+	GZIP or BZIP2	-
Apache Parquet	+	GZIP or Snappy	-
Apache ORC	-	Zlib or Snappy	-

[1]Only when the file is not compressed.

Table 3.1: File format evaluation

Raw Binary The format holds the information as a binary blob, without including out-of-the-box mechanisms that exist in other file formats we evaluate. It requires additions to be made to the components to enable features such as compression as there is no out-of-the-box implementation readily available, as shown in Figure 3.1. S3 query systems support for file-formats differ based on the implementation. In our case, all the alternatives we consider in Table 3.3 indicate that there is no query system support for raw binary files. With the number of changes and the lack of support on multiple features, we consider raw binary file format unsuitable for our architecture.

Comma Separated Values(CSV) CSV format is a human-readable text-based file format that separates data through commas and line breaks [54]. Architecture requires additions to the components to use compression. S3 query does support querying files in CSV format, uncompressed or compressed. However, only two compression methods

are supported when used with an S3 query system. Implementing special readers allow splitting CSV files into chunks. However, to query compressed data, it must be read in full. Features such as compression are not available out of the box. Feature compatibility wise CSV meets the majority of the requirements except splitting the files into chunks while using compression. Nonetheless, we consider CSV an alternative file format that requires additions to the existing structure of our architecture to meet the requirements fully.

JavaScript Object Notation(JSON) JSON format is a human-readable text-based file format [55]. Compression support is the same with CSV format. To be compatible with the S3 query system, compression of JSON format can be done via either GZIP or BZIP2 methods. When the data is uncompressed S3 query system functions as intended and allows users to query data. Splitting JSON files is not possible without loading the data in full. Therefore we consider JSON format to fail the evaluation criteria about splitting a file into chunks.

Apache Parquet Apache Parquet format is a columnar data format designed for big data with non-lossy compression [56]. The format supports several compression methods such as GZIP, Snappy, LZ4 [56]. Nonetheless, Apache Parquet support depends on the implementation of the S3 Object Storage. For our evaluation, we use the S3 Object Storage implementations we list in Table 3.3. These S3 Object Storage implementations support Apache Parquet when files use GZIP or Snappy algorithms to compress data on the column level. Regardless of the compression, the S3 query system can split Parquet files into chunks and fetch them without fully loading the file. Thus, making the Apache Parquet format a viable solution for the Architecture.

Apache ORC Apache ORC is a type aware columnar format designed by Apache with support for large streaming reads [57]. Apache ORC format supports compression with various methods and can split files into chunks without loading them fully; however, S3 query support for Apache ORC is non-existent. Nevertheless, due to a lack of S3 query support, the Apache ORC format does not satisfy the requirements.

Based on the comparison in Table 3.1 Apache Parquet format satisfy the requirements of the architecture. Thus, it is a viable option to use as the file format for the architecture.

Uploading is a network-dependent operation that involves risks such as losing network packets. Thus, the architecture considers potential data loss during implementation to include fail-safe mechanisms to ensure data reliability. The IRF provides a fail-safe mechanism and saves the data locally in the case that upload fails, followed by an exponential

back-off retry strategy to upload the data. As the exponential back-off strategy will cause IRF to accumulate resources, after a certain threshold of time reached data IRF deletes the locally available data. Without a deletion policy, data can take up the limited resources available in an Aggregator, causing losses on the incoming new data.

Interceptor Controller

Interceptor Controller (IC) is the sub-component responsible for sending commands to Interceptors connected to an Aggregator, as required by **FR-3**. IC provides capabilities that allow remotely configuring and interacting with an Interceptor while removing the need to interact with an Interceptor physically as such IC allows the users of our architecture to change the configuration of the Interceptor from a remote location. If the side-channel source is digitally available, the controller can be utilized to organize data collection and overall maintenance of the Interceptor remotely.

The communication takes place between the Interceptor and the Aggregator as shown in Figure 3.1 through Interceptor Control Events we defined earlier in Section 3.2.2. With the communication line represents, our architecture provides enhancement over the data. Arranging data collection with different parameters is possible remotely without requiring changes in the physical setup and the positioning of a leak probe. For machine learning and analysis, keeping experiment settings the same is crucial due to data quality concerns. Within the side-channel domain, having the probe simultaneously without any physical changes ensures consistent leak source measurements. Having the measures consistent helps reduce the noise as filtering out noise becomes less challenging. Thus, by limiting the physical movement of the probe by having remote access to data acquisition system configuration and actions, IC enhances data qualities that are well-defined in the big data community [58].

Precision Time Protocol (PTP) Server

We follow the same description provided in Section and adhere to the standard IEEE 1588-2019 [51]. Hence, PTP Server in Aggregators is responsible for keeping all the connected interceptors in a time-synchronized state.

We underlined the benefits of having PTP previously to expand on it; time synchronization provides consistency for the data as the timestamps stay consistent between the Aggregator and the Interceptor. As we see in Figure 3.1 the additional meta information

from Interceptor and Aggregator itself is also available to provide contextual information to data. PTP server is responsible for keeping all the connected interceptors in a time-synchronized state; by doing so, information transfer between the Interceptor and Aggregators stays consistent in terms of time. Having a time synchronization also helps analysis as side-channel information objects sent to S3 Storage include timestamps. Due to the synchronous time, some machine learning problems also become less challenging; an example problem that requires synchronous time is determining patterns over multiple interceptors. Thus, time synchronization allows us to enrich and add contextual information to data via available meta-information for a given time frame with accuracy and the limitations of the PTP as described in IEEE 1588-2019 [51].

3.2.4 Analyzer

The Analyzer component of the proposed architecture provides a ground for analysis and machine learning functionalities within the described pipeline. The Analyzer can deploy using cloud services and bare metal. Regardless of the deployment type, the Analyzer functions as a data lake, given the S3 Object Storage system deployment within the Analyzer as shown in Figure 3.1. To provide contextual information and enrich data, the Analyzer also contains a Log Management Database. Utilizing this database, it is also possible to accumulate metrics regarding the pipeline itself. Thus, providing means to monitor the overall architecture.

In Analyzer, the machine learning models and analysis steps get triggered via event notifications from both S3 Object Storage and Log Management Database. Thus, we introduce Model Executor, a sub-component that oversees available machine learning models and analysis processes.

As the S3 Object Storage and Log Management Database pipeline consistently provides contextual information, we consider machine learning models developed on the data to be reusable later with similarly represented side-channel data without defining machine learning model architectures from scratch. Analysis processes are also transferable to again expressed side-channel data within this context. The architecture requires machine learning models and analysis sub-components to be pluggable to provide this transferability without further efforts. By having machine learning models pluggable, the architecture enables other learning techniques such as ensemble learning.

S3 Object Storage

Side-channel information is often high dimensional and high-frequency data, as we described earlier. Storing such data requires a storage system that can scale and adhere to the requirements such as **FR-2**. With the developments in the cloud domain and big data domain, many storage systems have been developed. The decision on using S3 object storage was the result of an evaluation of the storage systems compared to the requirements of the architecture defined earlier in Section 3.1. Thus, To explain our decision about the S3 Object Storage, we provide the following evaluation shown in Table 3.2.

System Name	Distributed	Query Engine	Meta Information	Versioning
S3 Object Storage	+	+	+	+
Apache Hadoop	+	+	+	-
NFS	+	-	-	-
SMB	+	-	-	-

Table 3.2: Scenario Interaction based on Major Components

As we show in Table 3.2, both Apache Hadoop and S3 Object Storage meet the requirements specified without requiring additional tools. However, in the architecture, we make use of S3 Object Storage. The reason we prefer S3 Object Storage is that it includes an additional feature which is versioning. Considering future iterations and use-cases, having an extra feature such as versioning would provide flexibility to users. The flexibility is desirable as the architecture aims to reduce the implementation-related efforts. To explain our decision, we further detail the advantages of S3 Object Storage.

S3 Object Storage has access control systems embedded, which provide authentication and permission management with granular settings as required by **FR-8**. These features allow conducting data management-related tasks in an organized manner. For example, analysis and machine learning-related software can use read-only access, whereas data management software can use write-only access to specific objects. Using policies to access data also reduces other risks such as accidental modifications to the data, contributing significantly to data reliability.

The versioning feature is extra to the requirements and provides flexibility for future use-cases. It allows organizing data for analysis and machine learning models by reducing

the computations. Utilizing the data versioning analysis process can generate alternative data representations, marking them as versions without losing the link to the original raw data. For example, each pre-processing step can store data as a version with the callback system of the architecture, providing the means to test various machine learning models against a different version of data without repeating pre-processing.

A standard interface of objects creates a common interface to process and analyze data. Similarities in the data representation for side-channel information make it possible to implement machine learning models once and test against different experiments or side-channel sources.

Another advantage of S3 Object Storage is that it supports streaming data on upload and download, meaning a file can grow in size while also downloading the data. Some machine learning approaches utilize this feature more than other such as models which use online training.

With the S3 Object Storage query mechanism, analysis processes and machine learning models have access to data filtering capabilities without additional tools required. S3 Object Storage provides access to contextual information, and the ability to fetch this information without manual look-ups offers flexibility to both machine learning and data management. Because data is in large volumes, the ability to query files directly without knowing the contents provides reliable access to filtering and accessing data.

We also note that the S3 Object Storage query system has fewer features than traditional RDBMSs in terms of capabilities. Nevertheless, the S3 Object Storage query system can support various functions that reduce, select and filter data. S3 Object Storage implementations in the industry have query mechanisms for certain file formats such as CSV [54], JSON [55], Apache Parquet [56] as mentioned earlier. Considering the evaluation in Table 3.1, alternatives to Parquet exist but require further changes to the architecture.

S3 Object Storage has many implementations available; most of the implementations match in features; nonetheless, for our architecture, we provide available alternatives that meet our requirements and provide properties such as licensing and deployment type for them. We show possible options in Table 3.3 which provides an insight into the deployment of the S3 Object Storage.

Log Management Database

The data pipeline collects meta and metric information regarding data and the pipeline itself per requirements **FR-1**, **FR-4**, and **FR-7**. A storage solution is necessary to store

Storage System	Deployment	Query Engine	License
AWS S3	Cloud	CSV, JSON, Parquet	Proprietary
MinIO	On Premises	CSV, JSON, Parquet	AGPLv3
Ceph	On Premises	CSV, JSON, Parquet	LGPLv2.1

Table 3.3: S3 Implementation Comparison

meta and metric information. Log Management Database exists for this purpose; it helps facilitate a storage environment. With the amount of data going through the pipeline, a scalable third-party solution meets the non-functional requirements is preferable.

Model Executor

The Model Executor facilitates the callback mechanism within the Analyzer, as required by **FR-5**. The Analyzer receives data from many Interceptors and makes use of callbacks to automate the analysis process. The Model Executor is the sub-component that implements the callback mechanism,

The data intake is significant, and the Model Executor reduces the tasks in the Analyzer to smaller chunks allowing techniques such as batch processing to analyze data. With limited resources using this sub-component, managing the Analyzer resources becomes less challenging.

3.2.5 Machine Learning Models

Machine learning models provide outputs of the analysis in the architecture and answers to problems trying to be solved, such as the effect of cryptographic operations on CPU utilization. The architecture does not limit the defined set of problems and allows analysis processes and machine learning models to answer most problems without additional changes in the system. As machine learning also ties to data quality, we describe the properties or functionalities within the architecture that preserve data reliability and quality. For instance, object versioning, tagging, and Identity and Access Management (IAM)

in the S3 Object Storage ensure users access data only they are associated with limited rights. In this case, write permissions are an example of the limitations users have. The architecture limits users to only read data without writing permissions to ensure that raw data is non-modified and preserved. Having policies and permissions also helps organize machine learning models as data access policies can be determined per model.

The out-of-the-box features of the S3 system also help provision data with often required features such as querying and filtering, as required by **FR-2**. Out-of-the-box features allow the data scientist to apply machine learning models instead of additional tooling requirements in the data pipeline between models and the storage. Using the data storage system as mentioned, models can be applied to different data in terms of properties. The sampling rate can be tested without requiring the analysis code to be written from scratch.

By leveraging existing data storage systems, organizing analysis and machine learning models require less implementation effort. For this purpose, the architecture standardizes machine learning tools and reusable software components. Reducing the resources to build machine learning modelling techniques can be written once code-wise. Reduction in effort to build machine learning model makes it possible to apply machine learning models with no additional requirements if the data representation is the same or similar form. Thus, it is possible to reuse already implemented approaches with the same or different side-channel sources without modifying the process itself. Machine learning model training, testing and validating models is possible without changing many components within the architecture. By utilizing the versioning system on the S3 storage system, pre-processing steps can be applied without modifications to the existing tools when the data representation is similar. Tools can be applied re-applied to create versions of new types of raw data. Using pre-processing, as mentioned, also makes it possible to apply machine learning models with different pre-processing technique combinations. In essence, our proposed architecture provides a flexible testing environment to side-channel sources with identical or similar representation.

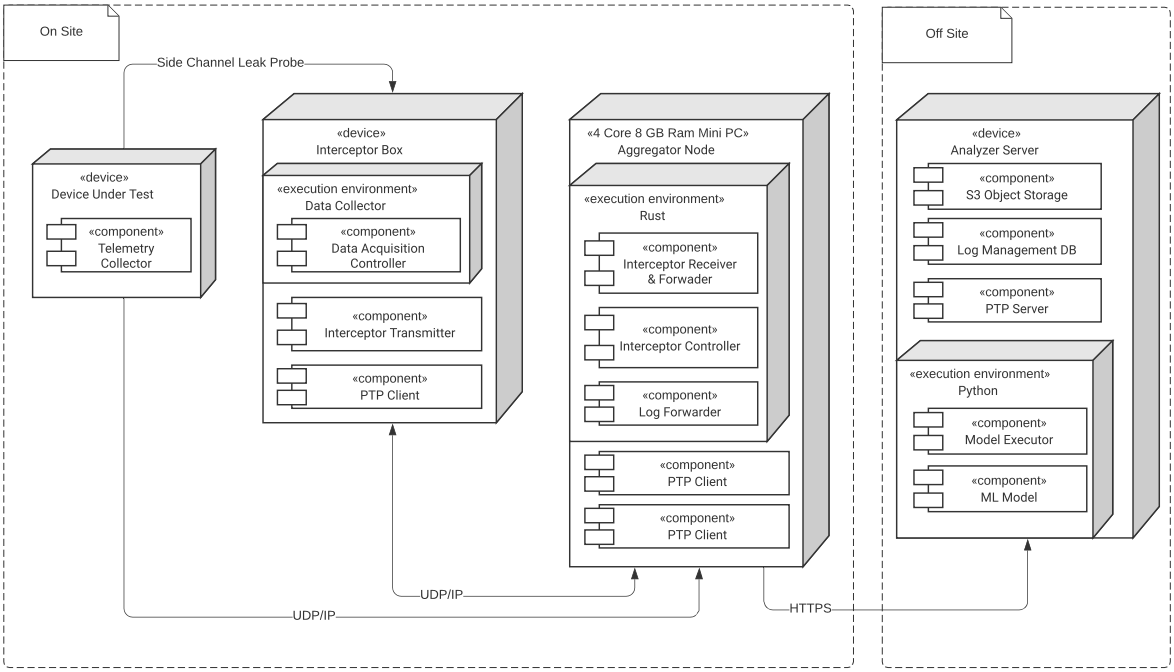
3.3 Deployment View

Deployment view or, in other words, deployment diagram of architecture provides means to translate the abstract architecture to real-world use cases [59]. By providing a complete instance of the architecture, we understand the way architecture deploys and works in reality. We estimate the hardware requirements, and a guideline on the deployment allows users to enable the pipeline for production use. Thus, reducing the implementation

risks of the architecture and formally documenting the requirements to deploy the architecture [44] [59]. We use the term operational requirements to describe requirements to deploy and use the pipeline. Operational requirements of architecture can be the cost of the components, hardware requirements, network requirements and other details.

Deployment diagrams in a distributed setting also help determine the critical points in architecture; by critical points, we refer to the critical components critical to a system's operation. In the cases these components fail, the process will cease impacting the use. As distributed systems deploy across different physical locations, identifying critical points and managing them may help reduce the physical access required to update, fix or use these systems. As such, in Figure 3.2 we provide the locations of the major components.

Figure 3.2: Deployment Diagram



As seen in Figure 3.2 Interceptors and Aggregators locate on the same site within a network where they have access to each other. Their interactions and data transfer are over User Datagram Protocol (UDP). We note that alternatives to UDP exist and apply to our architecture. The key reason to use UDP is that it is less resource-intensive compared to

the Transmission Control Protocol (TCP), given high sampling rates and high throughput requirements, we chose UDP to gain more throughput. On the other hand, we consider Analyzer fully remote to the other components in the system. Analyzer functionality mimics micro-services architecture. Analyzer deployment can be done through many cloud services, cloud servers, or a server.

Regarding hardware requirements, we know that factors such as sampling rate and the side-channel source affect these requirements. As we provide a generalized instance of the architecture, we do not make any assumptions for the hardware resource requirements. In our case study on Chapter 5, we show how we determine the hardware requirements based on the required throughput for the pipeline.

Chapter 4

Software Architecture Analysis Method (SAAM)

A software architecture consists of a structure and a set of requirements for a system. Understanding the structure and requirements allow developers to build systems that have quality attributes such as maintainability. Quality attributes are traits that contribute to the architecture's quality, just like the functionality of the architecture. Understanding whether architecture is high quality or not involves assessing the architecture. The architecture must show that it meets the functionality and has a set of quality attributes during the assessment. These attributes differ from architecture to architecture, depending on the structure and functionality. Nevertheless, the assessment requires a consistent methodology. Software Architecture Analysis Method (SAAM) process is a software architecture evaluation method that helps evaluate architectures based on a set of scenarios. This chapter provides an evaluation of the architecture proposed in Chapter 3 based on the SAAM process under different scenarios assessing whether the architecture has specific quality attributes as part of its traits.

4.1 Introduction to Software Architecture Analysis Method (SAAM)

The SAAM process is an architectural analysis and evaluation method proposed by Kazman et al. in 1994. Architectures require structured definitions that capture all features of the system to analyze an architecture. Kazman et al., 1994, describe that functionality,

structure, and allocation entail perspectives required for understanding the architecture. Evaluating the architecture by using SAAM is possible once all the perspectives are established [60].

Kazman et al. describe a set of activities of the SAAM process to evaluate and analyze an architecture. These activities consist of the following steps:

1. Characterize a canonical functional partitioning for the domain.
2. Map the functional partitioning onto the architecture's structural decomposition.
3. Choose a set of quality attributes with which to assess the architecture.
4. Choose a set of concrete tasks which test the desired quality attributes.
5. Evaluate the degree to which each architecture provides support for each task.

Chapter 3 defines the structure and the functional partitioning of our proposed architecture through diagrams, requirements and functionality descriptions for components in the architecture. As such, for the SAAM process, we consider Step 1 and Step 2 to be completed through the analysis shown in Chapter 3. For Step 3 of SAAM, we define a set of quality attributes that can be used to evaluate the proposed architecture. We describe quality attributes under Section 4.3.1

4.2 Motivation & Goal

Different views can represent an architecture, and we describe the following three here: the architecture's functional mapping to a domain, the architecture's structure and fitting functional mappings to the architecture's structure [60][44]. Software engineers can use these views to design and display an architecture, as we did in Chapter 3.

Functional views provide an insight into the system's functions and behaviour. The insight gained from functional views is valuable for understanding the architecture; however, it is not enough to deem an architecture good or bad. The quality of architecture is different from providing a functional view [44].

Architectures consider functionality and quality attributes that show the extent of the success of the architecture. Quality attributes describe the extent to which the architecture

is successful about certain traits. Maintainability is an example trait that requires the necessary functionality and traits to be in place to operate and maintain a data pipeline.

Quality attributes define a system's worth, and developers should pay attention to the quality attributes of the architecture during the design and implementation. In reality, developers often consider functionality first and then consider quality attributes, which ends up with architectures that lack desired quality attributes. The leading cause of this is cost and time constraints, forcing developers and designers to choose between functionality and quality attributes.

Confirming whether architecture has specific quality attributes while meeting the functionality requirements require an assessment. Qualities such as portability, maintainability, and scalability must also be considered to design and implement functional and high-quality software. Studies label architectures *good*, if the system implemented from the architecture meets all the required quality attributes [61].

Focusing solely on the functionality of a system causes developers and designers to overlook quality measures of architecture, with factors such as organizational pressure, stress and time constraints affecting developers. Developers and designers do not have an incentive to make design changes on architectures as long as the system is feature/functionality-wise complete [62][63][64][65]. As such, systems are often subject to redesign due to the lack of the quality attributes of the architecture rather than a lack of functionality [44]. Nowadays, most critical software systems and their architectures involve architecture evaluations when making design choices. When the assessment is inadequate or non-existent, the outcome can be a bad architectural design which can be unpleasant, costly, and fatal depending on the type of software. Examples such as Pentium FDIV Defect [66], EDS Child Support System Defect [67], Boeing 737 Max 8 MCAS Defect [68] suggest that it is imperative to evaluate architectural decisions and design.

Architectural evaluations and overviews provide the capability to designer teams to catch costly mistakes early on. In this case, we evaluate the proposed architecture post-design and try to determine the shortcomings. By doing so, determining whether the proposed architecture is *good* is possible. Thus, the architecture requires an analysis that considers a set of quality attributes often seen in data pipelines. For this purpose, we use SAAM to assess our architecture's overall quality and the components within the architecture.

4.3 Analysis

The definition provided in Section 4.1 captures all of the required steps for SAAM. In this section, we proceed in the same order as the steps defined. Then we determine the quality attributes for the proposed architecture. By describing selected quality attributes, we proceed to create scenarios as described in Step-4 of SAAM for the architecture.

Chapter 3 provides the functional mapping definition of the architecture and completes Step-1 and Step-2 of the SAAM process. Step-3 of SAAM requires us to define a set of quality attributes to evaluate the architecture. Thus, the following section provides a set of quality attributes.

4.3.1 Quality Attributes for SAAM

Quality attributes for a data pipeline consist of two primary subjects, data and the pipeline itself [45][69]. To understand which quality attributes to evaluate for the architecture, using standards is a standard practice in the industry and research environments. Thus, we define our quality attributes by reviewing software architectures that involve data pipelines, software evaluation methodologies, and software quality to refine a set of quality attributes based on our review [70][71][72][58][73]. ISO/IEC 25010:2011 describes quality attributes for software architectures in general. Quality attributes such as functional suitability, reliability, performance efficiency, use-ability, security, compatibility, maintainability, and portability describe the quality of a software architecture per ISO/IEC 25010:2011 [74]. We select a subset of attributes based on the ISO/IEC 25010:2011 and combine the attributes with others based on our literature review. We define the quality attributes as maintainability, scalability, reliability and reusability per SAAM. We provide a short description of each attribute and why we think they are relevant to our architecture.

Maintainability

Maintainability considers actions taken to preserve the system [75]. Tasks such as data management and pipeline monitoring are relevant to maintainability. Upgrading several systems connected in the data pipeline is another item that impacts maintainability. As such assessing maintainability is a major quality attribute that requires an assessment.

Reusability

Reusability refers to the action of recycling the existing software components or architecture to meet the requirements of different use-cases [76]. In the side-channel information domain, there can be many side-channel sources. Thus, to understand whether our architecture provides a general architecture for all side-channel sources, we consider reusability a primary quality attribute that must be assessed.

Reliability

Software systems and architectures are subject to errors in design and implementation. Reliability means that a software system or architecture is error-prone or fail-safe during operation [77]. Reliability is a crucial quality attribute for architecture as failures can prevent expected functionality [76]. In a data pipeline, reliability affects the operation and the data [78]. For example, failures can cause loss and modifications to the data.

Scalability

The scalability attribute shows whether architecture can be used with the increasing workload without making any changes to the architecture except adding more resources to the architecture [79]. Scalability is a crucial attribute for the future success of a system as large amounts of data stream through the data pipeline from multiple systems. The scalability requirement of the architecture draws similarities to a distributed system [80].

4.3.2 Scenarios

SAAM is a scenario-based evaluation method requiring different scenarios that captures a variety of possible actions with the system. Scenario definitions take the data acquisition process and data itself into consideration. For instance, from the data perspective, scenarios consider tasks such as data labelling, data cleaning, data sharing, and data querying into consideration [78]. Data acquisition process-wise scenarios incorporate data management, pipeline configuration, and software maintenance.

As part of the SAAM we define 25 scenarios. This section lists, provides short descriptions for the scenarios and show all 25 scenarios in Table 4.1 with labels as defined in Section 4.1, and the changes required based on the descriptions of the scenarios as defined by SAAM [81].

SAAM requires stakeholders contribute to the scenarios and the architectural design. The architecture separates stakeholders into four groups of users within the system. The groups are *Data Managers*, *Service Managers*, *Analysts* and *Data Collection Managers*.

Each group of users has their responsibilities in the use of the architecture. *Data Managers* is responsible for managing storage systems and making data available to other user groups. *Service Managers* monitor the pipeline and maintains the running services. *Analysts* analyze data to create machine learning models and draw conclusions from exploring the data. *Data Collection Managers* are responsible for coordinating other groups and deciding how the data is provided with metadata for different experiments or data collection scenarios.

Below we categorize the scenarios based on the predominantly affected quality attribute while providing a short description for each scenario.

Maintainability Scenarios

M-1 ***Provide access to side-channel data based on role***
Data Managers need to ensure that data is not modifiable by other groups and enforce that only the pipeline has to write access to the S3 storage, preserving maintainability.

M-2 ***Label data during collection***
Data Collection Managers need a way to categorize the set of data collection scenarios or experiments. Analysts can use the labels to utilize supervised machine learning models.

M-3 ***Send Control Events to Interceptors through Analyzer***
Service Managers need to manage *Interceptors*, when a large number of *Interceptors* and *Aggregators* are deployed. This is meant to manage all or a set of *Interceptors* directly from an *Analyzer* without any intermediary helps maintain ease-of-use.

M-4 ***View Interceptor telemetry data***
Service Managers and Analysts need to view information from *Interceptor* to the label to provide context to data and monitor *Interceptor* status.

M-5 ***Collect different data collection scenarios at the same time.***
Data Collection Managers may want to parallelize data collection across different locations and collect different types of data.

M-6 ***Migrate to a different storage system that is not S3 compliant***
Data Managers and Data Collection Managers may prefer to use another storage solution or swap the S3 storage for a newer solution.

Reusability Scenarios

RES-1 ***Use a different file format other than Apache Parquet.***
Data Managers and Analysts may want to use another file format based on their experience or replace Apache Parquet with a newer file format.

RES-2 ***Store different versions of data without additional structuring.***
Data Managers can manage one structure vs many, and Analysts can organize processed data for machine learning models.

RES-3 ***Apply pre-processing steps to data upon arrival to the Analyze.r***
Analysts can use different pre-processing steps before feeding data to a machine learning model. Applying pre-processing steps that Analysts determine on arrival to *Analyzer* provides data that can be directly fed into machine learning models.

RES-4 **Use the Interceptor to capture a different side-channel information.**
The architecture should not focus on a single side-channel source but in-fact should allow different side-channel sources.

Reliability Scenarios

REL-1 ***Change settings and configuration of a DUT***
As analysis relies on consistency, having a way to control the configuration of DUT helps collect data reliably.

REL-2 **The data pipeline continues to function when the Aggregator fails to operate**
The Aggregator can fail and stop responding within the pipeline. The pipeline in this scenario will lose data from the Interceptors connected to the Aggregator. Thus, the system needs to ensure data is not lost.

REL-3 ***Synchronize time between Aggregators***
The pipeline collects data from multiple systems and multiple data sources such as

meta-information and side-channel information. Analysis requires the data sources to be in sync in order to provide meaningful results. Thus, synchronizing time allows the pipeline to collect reliable data. Analysts, Data Collection Managers and Data Managers may require reviews on the raw data by looking at the contextual information.

REL-4 ***Recover from an Aggregator crash***
Aggregator crashing can cause data losses as Interceptors will continue transmitting data. Recovering from a crash on the Aggregator instance, the system can include tools that ensure that services are restarted, such as the IRF sub-component.

REL-5 ***Use redundancy for Aggregator***
Many Interceptors connect to a single Aggregator node instance. In the case of a failure, if the traffic passes through a redundant instance, the pipeline will reduce the risk of losing data in the case of a failure occurs.

Scalability Scenarios

S-1 ***Collect multiple channels at the same time***
Different data sources exist, and Data Collection Managers can decide to collect multiple sources simultaneously, which impacts the scalability of the pipeline as more information will be transferred and collected.

S-2 ***Execute a callback after data transmission on Interceptor***
The system can require callbacks to be executed right after an event such as data transmission. For example, Interceptor can execute a callback that adjusts the leak probe after transmitting data.

S-3 ***Use a cloud-based S3 storage***
Data Managers may opt to use cloud-based S3 storage instead of maintaining S3 storage themselves.

S-4 ***Run multiple instances of Analyzer at the same time.***
Analysis and machine learning often resource extensive. Having the capability to run multiple Analyzer nodes is a possible method in this case to increase resources.

After defining and describing scenarios rest of the SAAM steps are applicable to evaluate the system. The evaluation takes place over a set of quality attributes determined by stakeholders. Today, standards define the quality attributes that architecture should have,

such as defined in ISO 25010 [74]. The analysis uses the following set of quality attributes to assess the architecture. The set consists of maintainability, reusability, upgradability and modifiability of the architecture.

4.3.3 Evaluating Component-Scenario Interactions

Understanding the interactions between components and scenarios require an assessment. Scenarios need to be assessed to determine whether they are *directly* applicable to the architecture or *indirectly* applicable with a set of changes being required [81]. Thus, scenarios identifies as *direct* or *indirect* per SAAM. Provided scenarios with their respective identity further assessment of indirect scenarios is a requirement. Table 4.1 defines these changes and specifies the identity assigned to the scenarios similar to the case study provided by Kazman et al., 1996 [81].

Table 4.1: Scenario Evaluation for Proposed Architecture

	Scenario	**Direct & Indirect**	**Required Changes**	**Affected Component(s)**
M-1	Provide access to side-channel data based on role	Direct	-	-
M-2	Label data during collection.	Direct	-	-

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
M-3	Send Control Events to Interceptors through Analyzer	Indirect	Configuration is done through the Aggregator with the current structure of the architecture. Implementing a relay which relays the commands from the Analyzer to Interceptors is a solution.	Aggregator
M-4	View Interceptor telemetry data.	Direct	-	-
M-5	Collect different data collection scenarios at the same time.	Direct	-	-

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
M-6	Migrate to a different storage system that is not S3 compliant	Indirect	A new query system needs to be added, metadata and tagging needs to be done with additional files, databases or supported feature set of the new storage system. Machine learning models require changes in how the data is queried.	Aggregator, Analyzer

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
RES-1	Use a different file format other than Apache Parquet	Indirect	Aggregator needs to decode information to the new format, requiring changes on the upload phase of the data. Query system may or may not be supported with the new file format assuming worst case scenario a new query system needs to be written for the new file format if S3 does not support querying the new file format. Machine learning models and analysis related software requires changes related to the queries used within.	Aggregator, Analyzer

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
RES-2	Store different versions of data without additional structuring.	Direct	-	-
RES-3	Apply preprocessing steps to data upon arrival to Analyzer.	Direct	-	-
RES-4	Use the Interceptor to capture a different side-channel information.	Direct	-	-

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
REL-1	Change settings, configuration of a DUT.	Indirect	A controller should be implemented on DUT. Adding the controller allows the system to reliably determine the state and adjust the state of the DUT. As analysis and datasets rely on consistency, having a way to control the configuration of DUT helps collect data reliably.	DUT, Interceptor
REL-2	View Interceptor telemetry data.	Direct	-	-
REL-3	Synchronize time between Aggregators.	Direct	-	-

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
REL-4	Recover from an Aggregator crash.	Indirect	Requires additions to the Aggregator which can automatically rerun sub-components in the case of a failure	Aggregator

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
REL-5	Use redundancy on Aggregator and Analyzer.	Indirect	In the case of an Analyzer failure data pipeline halts transmitting data completely and an Aggregator failure will cause loss of data coming from the connected Interceptors, the architecture needs to ensure redundancy mechanisms exists to prevent data losses. For this purpose Interceptor should implement a failure detection mechanisms to alternate the transmission to another Aggregator instance.	Interceptor, Aggregator, Analyzer -

Continued on next page

Table 4.1 – continued from previous page

	Scenario	Direct & Indirect	Required Changes	Affected Component(s)
S-1	Collect multiple channels at the same time.	Direct	-	-
S-2	Execute a callback after data transmission on Interceptor	Direct	-	-
S-3	Use a cloud based S3 storage.	Direct	-	-
S-4	Run multiple instances of Analyzer at the same time.	Indirect	Running multiple instances of Analyzer requires a clustering setup for the storage and callback mechanisms need to rely on an event mechanism that can work in a cluster environment.	Analyzer

Section 4.1 defined SAAM as a method that evaluates the interactions between scenarios and the architectural modules or components. As with Kazman et al., 1996 we use a similar representation of the interactions and present them via a table. With Table 4.1 we show which scenarios are supported without requiring changes versus which scenarios require minor or significant changes. SAAM helps provide insight on possible future cases and decisions that may happen after the deployment of an architecture. When we review Kazman et al., 1996 we also see similar findings based on a Scenario Evaluation Table.

We can identify the interaction between scenarios and components by reviewing the sce-

narios and required changes in Table 4.1. These interactions indicate possible features for the architecture, which may be implemented in the future. Having information about future cases provides insight for architecture development because, as technology progresses, tools can be replaced by newer alternatives that can meet the requirements.

By clearly establishing our proposed architecture's capabilities, we prepare the underlying work of defining points that an alternate may replace in the future. This will help reduce the risks around the development and implementation of the architecture while capturing flexible parts of the architecture.

Component	**Number of Changes**
Analyzer	4
Aggregator	5
Interceptor	2
Device Under Test (DUT)	1

Table 4.2: Scenario Interaction based on Major Components

Based on Table 4.1 and Table 4.2, we can see that *the Analyzer* component requires four changes followed by *the Aggregator* with five changes based on the scenarios we crafted. On the other hand *Interceptor* has two changes, and *DUT* has a single change. Interceptor and DUT when compared to the other components, do not require as many changes indicating less complexity when the structural definition provided in Chapter 3 is considered. On the other hand, analyzing the changes needed in Table 4.1 and several changes required in Table 4.2 we can conclude that the *Analyzer* and the *Aggregator* are responsible for more complex operations within the architecture. The structural mapping, sub-components and functionality described in Chapter 3 also align with this conclusion. To illustrate the effects of the analysis on quality attributes in detail, we provide our analysis separately for each major component within this section.

Device Under Test (DUT)

DUT as we describe in Chapter 3 is the hardware device from which the pipeline captures side-channel information from. As shown in Chapter 3 the pipeline collects side-channel and telemetry information from the DUT. The interactio

By having a layer of interaction between DUT's software and our architecture, we can provide an analysis of the DUT component regarding scenarios that affect how the side-channel information gets collected.

Maintainability As we collect telemetry logs from a DUT, we have the means to monitor and assess activities. Possessing such information provides the means to have to oversee the data being collected. Thus, by using the logs, maintaining the status of the system is possible. However, without any control mechanism over the DUT, it removes any possibility to intervene in the device. Hardware control mechanisms like Power over Ethernet switches may contribute towards maintainability; however, a lack of control over the software will persist without any means to control the DUT.

Side-channel data acquisition happens independent of the DUT with the help of a probe; thus, in the case that an issue arises over the DUT, issues will pollute the data collection. As query mechanisms that can filter out data based on telemetry logs exist, the data can be filtered to remove the polluted data; however, having such a case will increase the analysis and machine learning modelling efforts.

Due to the lack of control mechanisms on the DUT required changes affect the maintainability negatively on the component and architectural level. To provide a reason, considering our architecture's decentralized and the large number of systems connected to the pipeline, lacking any control over multiple systems prevents pipeline and data maintainability throughout the system.

Our findings suggest a possible remedy to reduce the impact of problems that may exist on data. However, they introduce additional complexity, resource, and time costs to the pipeline's machine learning and analysis parts.

Reusability Reusability of the architecture relies on whether components within the architecture are reusable as the DUT is the source of telemetry data, and the targeted system for side-channel reusability of DUT significantly impacts the reusability of the architecture. Side-channels can vary, and data can be collected from any electronics that leak information, and many side-channel sources exist. In this context, swapping the DUT with another does not impact the pipeline regarding how the data is transferred. However, it may affect the representation of its data. As described in Chapter 3, there are two possible conditions if the data representation does not change and machine learning models are reusable.

In comparison, different representations require analysis and machine learning modelling to be repeated. Nonetheless, existing models and techniques can be modified to a

certain degree by adjusting the query mechanisms. Depending on the perspective, we can argue that the data with different representations present another analysis problem. Thus, applying analysis and machine learning will be required no matter what the pipeline offers. As such, we deem that DUT does not reduce the architecture's reusability and preserve its reusability.

Reliability Reliability requires monitoring capabilities in a system to underline that the data pipeline operates as intended with reliable data flowing for analysis. Telemetry logs provide the means to monitor the status of the system. By using logs, one can determine whether issues are present within the DUT. The ability to monitor the DUT allows the assessment of whether any issues impact side-channel information being collected, and by filtering out the parts with issues, the overall reliability of the data can be preserved. As the data flow will not be interrupted if the DUT faces problems, our architecture provides solutions to issues that may arise. Hence, we can claim that DUT does not reduce the reusability of the architecture; in fact, it contributes to the reliability of the data by introducing telemetry logs.

Scalability The DUT does not communicate with any component in the architecture other than the Aggregator. The communication layer in between the Interceptor and the Aggregator transfers telemetry logs to the Aggregator. As such, the number of DUT does not affect the capability of the DUT as a component. However, with the increasing number of DUTs, DUT 's are limited based on the bandwidth available to the Aggregators, meaning a single Aggregator instance can support a finite amount of DUTs. We do not provide a definite number within our analysis as the information being transferred can change based on the requirements for the particular task.

Interceptor

The Interceptor component consists of several sub-components to capture side-channel information from a target DUT and manage the data acquisition systems that capture the side-channel information. It transmits telemetry information regarding the data and the data to the Aggregator; in essence, the data pipeline starts from the Interceptor as the primary purpose of the pipeline is to collect side-channel information and run the machine learning model's analysis process.

The scenarios we listed suggest that the Interceptor component requires *two* changes in total. The nature of both changes indicates that they are additive changes, meaning

existing sub-components does not require any changes to them in such scenarios. Based on these findings, we assess the quality attributes while considering the changes needed.

Maintainability The Interceptor's maintainability in the pipeline is a significant factor and a quality attribute, as the data reliability and, by extent, analysis quality depends on the Interceptor. The Interceptor provides a surface to configure the Interceptor's underlying sub-components through an Aggregator and an Interceptor interface. Having a configurable layer, in this case, helps users to maintain the Interceptors to fit their needs and requirements. Thus, we can claim that the existing structure of the architecture increases maintainability by exposing an interface to configure Interceptors. When we examine the changes, we see that Scenario **REL-1** relates to the configurability of a DUT, as the change suggests that the lack of this functionality reduces the overall maintainability in the pipeline. The other change is coming from Scenario **REL-5**, which implements a mechanism on the Interceptor to ensure data transmission happens to a healthy Aggregator instance.

Mentioned two changes contribute towards making the architecture more maintainable. Both changes being additive also reduces the complexity of implementing these changes. Thus it is advisable to improve the architecture with the two changes caused by the scenarios.

Reusability Scenarios in Table 4.1 that require changes on the Interceptor do not affect the system's reusability as they are both additive changes and do not impact the data being collected. In order to provide an analysis, we examine our structure again and provide an assessment based on two different perspectives. The first being the Interceptor reusability concerning capturing various side-channel sources, and the second perspective is collecting the same side-channel source, targeting different DUTs.

When we assess the reusability of the Interceptor for different side-channel sources, we can say that reusability depends on the channel and not the Interceptor. As described earlier in Chapter 3, the data collector encapsulates the process of capturing a side-channel source via a leakage probe. Therefore different side-channels can have different leak probes and by extent data. These changes depend on the non-functional requirements of the architecture and can change. Whereas with the second perspective, which is capturing the same side-channel source from multiple DUTs, the data acquisition system will not differ. Both perspectives indicate that the Interceptor meets the reusability quality attribute without any changes to any sub-components or components, excluding the changes caused by non-functional requirements.

Reliability The Interceptor is responsible for side-channel data collection and works as the primary source of data in the pipeline. Thus, the reliability of the Interceptor affects the pipeline in full. Meaning, both data and operations will be affected in the cases where Interceptors fail. For example, corruption of the data within the Interceptor will result in unusable data.

The architecture design and our analysis assume the leak probe works as intended and collects data accurately at all times. As such, we evaluate changes regarding the sub-components of the Interceptor only. The changes due to scenarios are additive, which do not require any breaking changes to the component and sub-components. Hence they can be implemented without any impact on the data, which preserves the quality of the data and the overall functionality of the Interceptor. As such, the Interceptor changes do not affect the reliability negatively, and changes increase the reliability, albeit in a minor way. Overall, the Interceptor is a reliable component as the changes which may affect the data are limited based on the scenarios, and their effect on the reliability is minor.

Scalability A single Interceptor is responsible for collecting data from a single DUT and transmitting it to a single Aggregator instance. The changes required by the scenarios do not contribute or affect the scalability of the Interceptors. Based on our structure, we can claim that any number of Interceptors are deployable with the limitation of Aggregator bandwidth. As the Aggregator bandwidth is limited, a single instance of an Aggregator will support a finite amount of Interceptor. Given the nature of high-frequency data and high sampling in the side-channel domain, the number of Interceptors for a single Aggregator instance can quickly decrease due to the increased bandwidth usage, indicating a cost increase that reduces the overall scalability of the pipeline and the Interceptors. Mitigating this problem and increasing scalability is possible by compressing the traffic between an Interceptor and an Aggregator. Hence, we can say that the Interceptor component's scalability also relies on the Aggregator scalability. Overall, the architecture shows that the Interceptor is a scalable component.

Analyzer

Analyzer component encapsulates vital operations within the architecture, such as analysis and data storage. Thus, the required changes are expected to rise with the number of future use-cases increasing. Regarding the functionality of the architecture, most of the required functionality is met through the Analyzer sub-components. Our scenarios indicate that there are four major changes to these sub-components of the Analyzer. When we review

Table 4.1, scenarios from the Table cover fundamental design changes and possible additive changes. With the majority of the sub-components that provide key functionality such as storing data and logs being third-party, the changes regarding scenarios help us assess the limitations of the third-party system in our context. Thus, while providing our analysis for each quality attribute, we also underline the critical limitations around the third-party software used within the architecture.

Maintainability The summary of changes indicates that several changes are required for the possible scenarios that are listed. Based on the features, there are no callback mechanisms that notify the users of failures within the pipeline, which leads to degraded maintainability of the overall system, and critical functionalities on the Analyzer become more prone to issues. As the Analyzer functions as the primary analysis component, it is essential to note that the lack of flexibility around maintaining other components also affects the Analyzer. As the reliance on third-party software is prevalent in the Analyzer, there is a cost to maintaining the third-party software. Thus, we note that third-party tools increase the required knowledge to maintain the Analyzer while reducing implementation costs.

Analyzer maintainability covers large parts of the architecture maintainability, as distributed components, such as Interceptors and Aggregators, monitor the Analyzer. The maintainability quality attribute of a system indicates the prevention of any failures throughout the operation. Thus, in a failure, Analyzer will prevent any data from entering the storage sub-components. We note that any maintainability factor within the pipeline ceases to exist in the cases where the Analyzer fails to work—as such, monitoring the pipeline provides the users with the status of the pipeline. Maintainability-wise, there are no control mechanisms exist that do not require user input.

Our findings indicate that the Analyzer component lacks control and fail-safe mechanisms that can function without user input to maintain the operations on the pipeline. For instance, users can log in to the Analyzer and start recovering services manually. However, this falls short as users require an alert from the logs to know the Analyzer failed.

Utilizing existing logs from components provides a vector to monitor the architecture. Introducing the changes suggested by the scenarios and above will help control all architecture components while monitoring the Analyzer component. Based on our analysis, changes regarding the maintainability require additive changes without any changes to the existing structure. As such, we are implementing suggested mechanisms that will increase the maintainability of the overall architecture.

Reusability We consider reusability under different aspects within the system. With requirements for users changing due to environment and costs, we first assess the reusability of the setup, including the sub-components and changes required for them to be replaced by other alternatives. Then we assess the reusability based on the side-channel being collected.

Changes suggested by the scenarios in Table 4.1 shows us that the structural changes which replace the Analyzer's existing sub-components introduce several breaking changes, which may span across the pipeline. In terms of breaking changes, we consider changes that require further modifications to the pipeline, other sub-components and the interactions between them as breaking changes and alterations spanning multiple components. Hence, we can say that the Analyzer is less reusable when we replace sub-components with non-compatible other technologies. For example, the usage of S3 Object Storage provides necessary features compared to a traditional database. If this sub-component is changed, modifications to the Aggregator and the Analyzer and their sub-components will be required. Our analysis and example, SAAM show us that Analyzer sub-components couple tightly. While the tight coupling reduces the reusability, we consider it a trade-off to increase the reliability of these sub-components.

The other aspect we consider is the reusability of Analyzer when different side-channel sources are being collected. The scenarios indicate that our system allows any type of side-channel source to be collected without changes to the Analyzer except the analysis and machine learning processes. Chapter 3 defined this condition as expected because, with a new data representation, data will be unique, requiring further analysis on it. Thus, Analyzer is a reusable component.

Reliability The Analyzer includes both third-party solutions and user-created solutions as sub-components. In terms of reliability, we assume that all third-party solutions in the architecture are reliable. Thus, we analyze the reliability based on required changes for non-third-party solutions.

Analyzer functionality breaks into three categories: storing information and triggering analysis with incoming data and analysis process. These functionalities are critical to the pipeline due to the overall goal of the pipeline, which is to create outputs based on incoming data from Interceptors. From these categories, storing information contributes and significantly decides the overall pipeline's reliability and the Analyzer. Having reliability issues for storing information leads to permanent loss of data and prevents the completing actions under the other two categories, such as triggering the analysis or providing decisions/outcomes from machine learning models or analysis models. In contrast, any failures

occurring due to actions belonging to the other two categories are recoverable, as the data would be available retroactively. As such, we consider information storing sub-components to be decisive factors for the reliability of both the Analyzer and the architecture.

Using third-party software developed by large organizations and large communities reduces reliability issues and helps free developers implement complex code bases. Another advantage of using third-party software is that the design and architecture are often mature. In the case of the two categories and sub-components under them, users must ensure the pipeline's reliability by monitoring these sub-components and ensuring their runtime occurs without problems. When we consider the changes related to the Analyzer and our earlier analysis concerning maintainability, we consider the monitoring capabilities of the Analyzer to be lacking. As the pipeline reliability also ties to the monitorability as we describe, for the system's reliability, we consider that our architecture lacks any sub-components to monitor the system coherently and provide any metrics that can assist. This helps assure the reliability of the pipeline.

Scalability The description of changes shows that the Analyzer modifications have limited flexibility in the modifiability of defined sub-components. Replacing sub-components in the Analyzer often requires breaking changes or significant changes to the Analyzer. Unsurprisingly, these are expected as Analyzer sub-components are third-party tools, excluding machine learning and pre-processing sub-components. The primary reason for third-party software usage is to provide out-of-the-box features that meet the requirements set in Section 3.1. As mentioned, third-party software reduces flexibility and adds dependencies to the system. However, it reduces maintenance costs and increases reliability, making it a viable solution for a data pipeline as reliability is often an essential requirement. Having a reliable data pipeline also ensures that the data to be analyzed is consistent, which reduces the time and effort spent on machine learning modelling and analysis. Overall our architecture trades flexibility in exchange for maintainability and reliability. It becomes a feasible solution as object storage has its architecture, involving a major component out of the shelf, increasing our proposed architecture's maintainability and upgradeability. We can also conclude that additional sub-components can be added to the architecture without requiring significant changes to the system itself.

Aggregator

The Aggregator component is responsible for gathering data from many Interceptors and uploading them to the storage sub-components under the Analyzer, as described in Chapter 3. It also acts as a way to change the configuration of Interceptors and issue commands.

The pipeline, Aggregator receives telemetry information from DUTs, Interceptors and itself via the pipeline. By using this information, it is conceivable to enrich the side-channel information from the Interceptors and provide contextual information to the data. Doing so provides the means to any process related to the analysis to use the contextual information to label and filter data. Hence, it contributes to the data collection and analysis, which are major functionalities of our data pipeline. When we analyze the scenarios and required changes in Table 4.1, we see that the changes often include additive changes to the component, without breaking or requiring changes to the underlying sub-components while adding new sub-components. Based on this information, we assess the quality attributes and consider these changes to the Aggregator.

Maintainability The Aggregator component acts as a relay for side-channel information, which is a controller for the pipeline by configuring the data acquisition system under the Interceptor. With this approach, Aggregator provides an interface to manage the Interceptors connected to it. When we review the required changes in Table 4.1, most scenarios require changes that span multiple components. For instance, **M-6** includes changes to both Aggregator and the Analyzer. We consider these types of changes to require more effort compared to the additive changes such as **M-3**.

Table 4.1 provides insight into which scenarios affect the maintainability directly and which do not. Based on the scenarios and the analysis, we can conclude that the lack of these changes hampers the user's ability to maintain the Aggregator and other components, potentially causing disruptions while operating the pipeline. We see that **M-3** and **M-6** predominantly affect the maintainability of the pipeline and the Aggregator. **M-3** requires additions to the architecture additions to architecture, whereas **M-6** introduces breaking changes which can end up altering multiple sub-components and components.

Based on the changes required to implement **M-3**, the architecture is missing a management interface to control all Interceptors at the same time. Users would need access to the Aggregators to which the interceptors are connected and then use the control interface to control the Interceptors. The required additions for the scenario suggest that a management interface must be added between the Aggregator and the Analyzer to enable the management of Interceptors and Aggregators. The interface would also support sending commands to any Interceptors connected to the Aggregator providing control over each component in the pipeline. The lack of this feature reduces the maintainability as the pipeline scales. Managing several Aggregators and Interceptors will become challenging due to the lack of a central control system.

M-6 changes are different in nature compared to the **M-3** as it includes changes that

break sub-components of the Aggregator and other components and consider it as a major change. Changing file systems might be a precondition for some users and may require a change depending on the requirements. However, when we assess this change, we see many components changing due to this change, such as the IRF sub-component. While maintainability-wise, users may prefer other file systems, the architecture is not flexible in the file system to provide an interchangeable structure.

Overall, maintainability wise both changes affect the architecture. Based on the analysis, we learnt that the file system of the architecture is not flexible and requires specific systems. For this reason, we can say that from a certain perspective, which considers swapping the file system, the maintainability of the architecture reduces. However, we do not recommend this change with the architecture as we consider selecting a specific storage system is justifiable and a trade-off to having a fully functioning pipeline. Controlling the components poses a challenge, as seen in the section, and we recommend implementing changes suggested by **M-3**, that can enable controlling from Analyzer directly utilizing the Aggregator as a relay.

Reusability Aggregators, described in Chapter 3, are distributed systems in nature with multiple instances within different or exact locations. In terms of data, no operations other than the enrichment of side-channel data happens on the Aggregator. Changes in Table 4.1 also show that the Aggregator is a component with high reusability due to its functionality. Some scenarios may reduce the overall reusability of the Aggregator. For instance, **RES-1** may end up requiring the query system for the architecture to be fully implemented from scratch without relying on third-party software. Overall we consider the changes relating to the reusability in Table 4.1 as items that do not contribute to the betterment of the reusability of the pipeline and the Aggregator.

Reliability Aggregators are critical points in the pipeline as all available information passes through them, including but not limited to telemetry information from all the components except the Analyzer, side-channel information and information about the pipeline. We see through the required changes and scenarios in Table 4.1 that the Aggregator tightly couples to the Analyzer sub-components. As the interaction defined between these sub-components are part of third-party software in the Analyzer, the reliability of these interactions relates to the overall reliability of third-party software being used. With Aggregator being a critical part of the pipeline, we remind that fault-tolerance and redundancy would play a massive role in improving reliability in the pipeline **REL-5** requires changes, particularly in line with these recommendations. Hence, we consider the lack of changes caused

by **REL-5** to reduce the Aggregator's reliability factors.

Scalability The Aggregator plays a vital role in scaling the system as a single Aggregator instance accepts connections from several Interceptors, and scenarios do not negatively affect the Aggregator's scalability. The scalability of the Aggregator mostly relates to the bandwidth requirements required to transfer data to the storage.

4.4 Lessons Learnt

With the analysis, we show the weaknesses and the strengths of our architecture with the help of SAAM. Our findings indicate that the proposed pipeline and components have improvements concerning maintainability and reliability by implementing the suggested changes based on the analysis.

Reliability-wise analysis shows that the lack of fault-tolerance in Aggregator and Analyzer can end up in data losses, with mechanisms to provide a fail-safe pipeline fell short. Overall the lack of fail-tolerance indicates that the architecture has reduced maintainability and reliability. A pipeline must ensure constant operation to prevent any data loss. For instance, critical functions such as storing data occur within the Analyzer, thus making Analyzer capable of recovering from a failure reduces the risk of losing information for long periods. Based on the SAAM analysis, the Aggregator is similar in terms of vulnerability against failures just as Analyzer, indicating that Aggregator is another vulnerable part of the data pipeline.

When analyzing Analyzer component SAAM scenarios showed that not having fail-safe mechanisms leads to a total loss of data and prevents the pipeline from functioning. Given the number of systems where data flows from having fail-safe mechanisms reduces the risk of usage. Having fail-safe mechanisms improves maintainability and reliability. Thus, the Analyzer component requires improvements to prevent any data losses with improving changes such as **REL-5**.

The Aggregator component is different from the Analyzer, and the failure causes data losses on all the Interceptors to connect to the Aggregator. As such, scenarios and their respective changes fell short due to a lack of functionality that can recover the system from failures and a lack of redundancy mechanisms to prevent data loss. Applying these improvement strategies will boost maintainability and reliability with a significant impact on reliability.

SAAM analysis and scenarios also indicate that the components are tightly coupled with each other, allowing little room for replacing existing sub-components on the Analyzer. As the sub-components we use are third-party software, we consider this limitation a trade-off to increase the reliability and robustness of the architecture. From this, we can assume that third-party solutions may indeed reduce the flexibility of approaches to architecture. However, we also note that the architecture has advantages with new sub-components entering the system. The architecture we propose uses the sub-components and structures them in an additive manner, meaning new sub-components can be added or removed without requiring complex changes. A lesson learnt from this conclusion is that by trading off some quality attributes, we can improve other quality attributes. In our case, we sacrificed modifiability and flexibility with our third-party usage in exchange for increased reliability.

Overall, SAAM provided helpful insight about possible improvements for our architecture and showed the pitfalls of our design. Consequently, improving the architecture is feasible with suggested changes in a different iteration for the architecture. As our sub-components operate in an additive manner, implementing the proposed or required changes can be trivial most of the time, as seen in our analysis.

Chapter 5

Case Study: EET

The architecture provided in Chapter 3 is an abstract model with no real-world implementation to compare our findings based on our analysis in Chapter 4. The EET project provides an example implementation of the architecture.

5.1 Overview of Project

EET is a technology that aims to protect personal computers, workstations and embedded systems by utilizing the involuntary emissions emitted by these systems called side-channel information. The side-channel information collection capable EET uses power consumption data and telemetry information from a system to detect attacks on the system.

The architecture described in Chapter 3 provides and meets the requirements of the EET with changes to the non-functional parts of the requirements. For instance, EET requires a certain number of systems to be monitored at the same time to detect ransomware within a set of systems.

EET preserves the architecture by making minor modifications to the architecture. For example, names of the components change in the EET architecture, and the PTP synchronization is different. The changes are albeit minor and do not impact the core functionality. Thus EET serves as an example of the architecture.

Systems monitored by EET have a physical probe attached to them to collect side-channel information. As EET aims to be as non-invasive as possible, leak probes do not alter or affect the protected system. Nevertheless, the technology requires physical access to the monitored system.

The technology leverages analysis processes and machine learning to identify attacks against a system and verify the system's integrity by analyzing ongoing activities. Feature-wise EET draws a similar picture to IDS systems but adds new side-channel data sources. EET requires side-channel information probes and a target device to collect data and is independent of other protection and detection systems, which means EET can serve as a complementary system to them.

Attack detection methods of EET rely on a series of machine learning models. EET uses machine learning models that can train online and offline, supported by the proposed architecture as described in Chapter 3 and Chapter 4. Machine learning model requirements for EET technology are similar to other detection systems. For instance IDSs and Collaborative Intrusion Detection Systems (CIDSs) evaluate available log information and other data sources such as network traffic data in near real-time, analyzing and identifying threats live [82][83]. On the other hand, EET analyzes side-channel information and log data, live, through a stream of information to identify and detect attacks. Thus, time-related metrics such as CPU consumption are of importance for the EET.

The key difference between EET and other detection systems such as IDS and CIDS is that EET utilizes logs and adds side-channel information as a data source. IDSs and CIDSs utilize telemetry, and log information from systems to identify and analyze the threat[84][85]. In contrast, EET utilizes not just telemetry and log information but also utilizes side-channel information to detect attacks against a system.

Monitoring and other features such as the analysis process show similarities between EET and other protection systems such as CIDS. CIDSs uses machine learning to detect and alert users by callback mechanisms. EET has the same capabilities and detects attacks on a system with machine learning and alerts users with callback mechanisms.

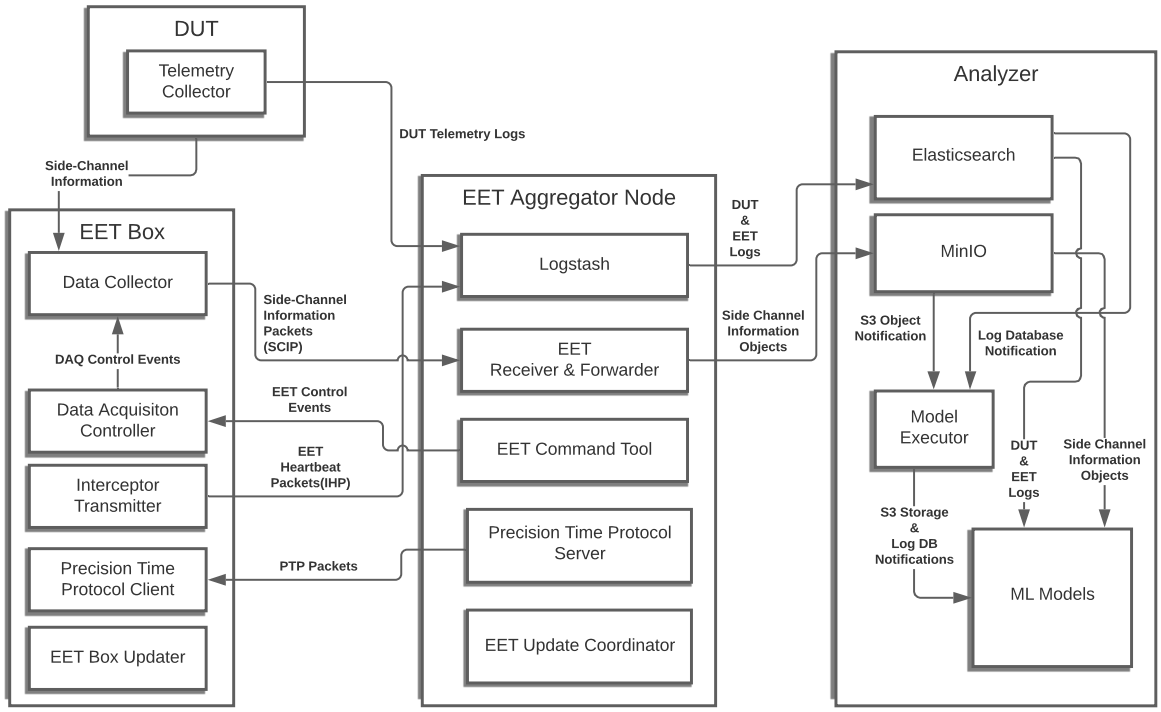
EET uses physical probes attached to the system to collect side-channel information. In a real-world setting, the number of systems to be monitored and their locations may vary. EET uses the proposed architecture to facilitate data collection in a decentralized and scalable approach.

5.2 Functional View

EET uses the same functionality we propose in our architecture and utilizes the same components under different names. As EET deploys to multiple networks, it also includes changes concerning interactions between components. The terminology difference between the proposed architecture and EET architecture is visible in the functional block diagram

Figure 5.1. To establish a common understanding, each component within EET architecture is mapped to a component in the proposed architecture.

Figure 5.1: Functional Block Diagram of EET

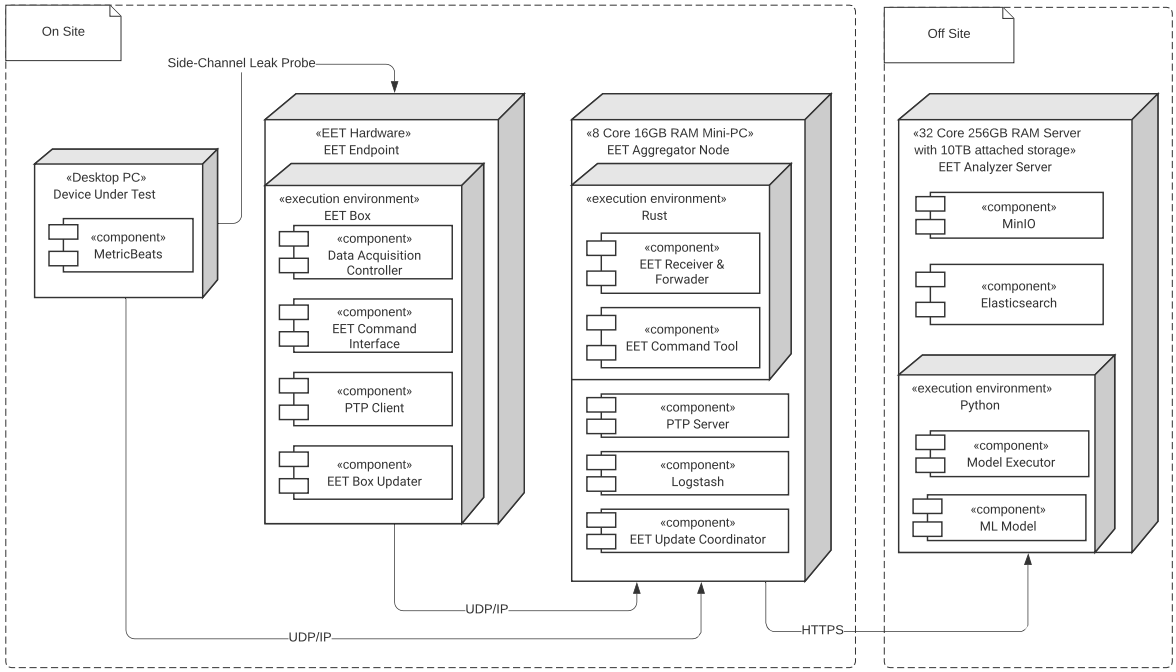


5.3 Deployment View

Deployment of EET generally follows the deployment shown in Figure 3.2. The deployment diagram in this section provides more details about the hardware, resources and the environment while describing changes related to the deployment as seen in Figure 5.1.

EET is capable of detecting threats such as ransomware, malware and system modifications similar to an IDS. Figure 5.2 shows the framework to enable side-channel information collection. EET makes use of three components out of four components: these are the EET Box, EET Aggregator Node and EET Analyzer. The EET Box interacts with

Figure 5.2: Deployment Diagram of EET



the leak probe, which measures the side-channel information through the Data Collector sub-component of the EET Box. The stream of information passes to the EET Aggregator Node, where meta-information and logs attach to the side-channel information stream. Inside the Analyzer component, the storage system stores the side-channel information. The Analyzer initiates the analysis process using callback methods and uses side-channel information, meta-information, and logs to detect attacks.

The architecture is capable of adapting to different leak probes as described in Chapter 3. In EET's case, the side-channel source is power consumption data. To capture power consumption data, EET Box uses the sub-component, Data Collector, to interact with the leak probe. For this purpose, the Data Collector contains several sub-components, including an Analog-to-digital converter (ADC) and a Field-programmable gate array (FPGA). Using these sub-components, Data Collector collects data with the help of the leak probe.

EET supports data collection at different sampling rates up to 1MS/S. Analysis of side-channel information collected during the case study indicates that 1MS/S preserves

data resolution where machine learning models can extract patterns. Due to different requirements, sampling rates can increase. By updating the leak probe and the Data Collector sub-component achieving sampling rates higher than 1MS/S is possible.

The cost of EET hardware can change due to changing requirements. For example, sampling rate-related changes to the EET Box can cause an increase in cost per unit. Thus, when making changes to the architecture, non-functional requirements should be considered with cost in mind.

Figure 5.2 describes the deployment of the EET architecture without instantiating. For example, the number of EET Boxes connected to the EET Aggregator Node is unknown. In this case, to determine the number of units connected to the Aggregator node instance, we use quantitative measures while adding an estimated overhead. Thus, to provide a decision concerning the instances in an EET deployment, we use the quantitative measures listed in Section 5.4. The measures indicate that the Aggregator must support 2 Gbps of outgoing traffic.

5.4 Quantitative Measurements

In the case study, we deploy EET with 20 Interceptors, 1 Aggregator, and 1 Analyzer deployed as shown in Section 5.3. We know that deploying an Interceptor with different configurations is possible, including differing sampling rates. The side-channel data collection in the study case occurs at the sampling rate of 1MS/S. The sampling rate determines the maximum amount of information transmitted from a single EET Box to the rest of the pipeline. Thus, we can assess whether or not architecture scales based on sampling rates involved. To evaluate the scaling of the EET architecture, we require quantitative measures such as the throughput of the data pipeline.

A data pipeline consists of layers that transfer data to each other and store it. Thus, we also need to assess our storage needs based on the targeted sampling rate. Throughout this thesis, we have focused on providing a scalable system to collect side-channel information by a decentralized pipeline, and SAAM suggests there are possible improvements to our architecture concerning quality attributes we have determined in Chapter 4. By providing quantitative measures, we compare whether SAAM offers valuable information concerning the decentralized side-channel data pipeline.

5.4.1 Throughput

We provide the theoretical and real throughput from a deployment of EET with 50 EET Boxes, 1 EET Aggregator Node and 1 EET Analyzer. With the sampling rate of 1MS/S for multiple systems. Due to the size of the information being transferred, scalability is affected; hence, we use the throughput of the data pipeline to show that the architecture scales to meet the required throughput of EET. By showing how the proposed architecture manages scalability, we show a possible solution to a side-channel data pipeline that can collect side-channel data from several systems.

We analyze the theoretical throughput of the pipeline given 50 EET Boxes connected to an Aggregator instance to determine the scalability and hardware requirements of the proposed system. Calculating the throughput on each component level allows users to leverage the throughput information to determine hardware requirements with increased precision.

Theoretical throughput calculation considers the sampling rate, size of each sample, and network overhead. The sampling size, for instance, depends on the ADC inside the Data Collector; as the representation supported by it is 2 bytes, the Data Collector stores each sample as a 2-byte value. Network overhead, in our case, is the overhead in bytes required to form UDP packets. We add 7% overhead to the total size of a single UDP packet as overhead. Based on these assumptions, we calculate the theoretical throughput as follows.

Sampling Rate(MS/S)	**Rate Per Box (Mbps/s)**	**Rate Per Aggregator (50 Boxes) (Mbps/s)**
1 MS/S	17,12	856,00
0.5 MS/S	8,56	429,00
0.1 MS/S	1,71	85,60
0.01 MS/S	0,17	8,56

Table 5.1: Theoretical throughput of the pipeline

As shown in Table 5.1, the Aggregator node requires a minimum of two 1 Gbps network cards working in a dual setup for 50 EET Boxes running the maximum sampling rate. Based on the specification, the number of connected EET Boxes can reduce the cost of the hardware requirements of the EET Aggregator. Based on the theoretical output, EET

can use up to 50 EET Boxes with a single Aggregator instance. The Analyzer is the other component that is affected by the throughput. The Analyzer may require a load-balancing network with large bandwidth to handle the throughput when using the maximum sampling rate. SAAM did not find a similar deficiency with regards to the components. Therefore, by making changes to the architecture's networking, the architecture's scalability can further be improved.

Sampling Rate(MS/S)	**Rate Per Box (Mbps/s)**	**Rate Per Aggregator (50 Boxes) (Mbps/s)**
1 MS/S	16.73	836.50
0.5 MS/S	8.34	417.00
0.1 MS/S	1.71	85.50
0.01 MS/S	0.17	8.50

Table 5.2: Throughput of the pipeline based on data arriving at MinIO

With the hardware requirements based on our theoretical throughput measurement, we can now compare our findings to a real example with the EET deployment given in the Table 5.2. Throughput, in this case, is measured with MinIO internal monitor. The theoretical and actual throughput is nearly identical, meaning scaling the architecture based on the theoretical throughput is also possible. Scaling-wise, the architecture does not fell short in any area and delivers the expected results with the specified hardware of dual 1Gbps network cards.

5.5 SAAM vs. Reality

In the SAAM, we noted that the architecture is open to improvements in terms of reliability and maintainability. To increase maintainability SAAM suggests adding a controller to the Aggregator and adding an update system to the architecture. SAAM also provides suggestions about making the architecture more reliable by introducing fail-safe methods and redundancy. EET adopts changes for maintainability while felling short on the changes with regards to reliability.

During the case study, problems arose regarding both maintainability during the development of the EET. Thus, EET introduces design changes and remedies to the architecture

with regards to maintainability. For instance, EET introduces an update system to update the Interceptors. However, changes concerning reliability are lacking as some of the issues were identified later when the pipeline was operating. For instance, the case study shows that the data pipeline failed to operate due to the lack of safeguards within the architecture.

EET and the architecture deploys and targets multiple systems. Thus having a method to upgrade the software components helps increase maintainability. EET improves the proposed architecture with sub-components under the EET Box and the EET Aggregator Node, which are the Interceptor and the Aggregator in the proposed architecture.

SAAM findings suggest that introducing remote configuration abilities to the Aggregator will increase the maintainability. EET makes changes with regards to this suggestion by introducing an SSH server to the architecture under the Aggregator. The SSH Server allows establishing remote access to Aggregators, increasing the maintainability of the Aggregator.

Reliability-wise EET does not make any changes to the structure of the architecture. However, it uses the configuration options of the MinIO, such as hash comparing on upload. Nonetheless, the analysis provided in Chapter 4 captures the shortcomings of the reliability of the architecture. During the deployment of EET the pipeline failed due to the following reasons, long recovery times due to lack of alerts based on the telemetry information available, twice due to misconfiguration of the Aggregator software, three times due to MinIO instance failing to access the storage attached to the Analyzer, and twice due to Interceptor software failing without recovery caused by bugs. Thus, we can claim that SAAM shows the shortcomings of the architecture correctly by pointing the problems around reliability, such as the lack of fail-safe mechanisms and redundancy.

Real-world findings indicate that architecture has room to grow in terms of maintainability and reliability. This finding is parallel to SAAM and the suggestions made within Chapter 4 can successfully increase the respective quality attributes as seen with the example of maintainability and the changes within EET concerning the maintainability.

Given the context of relatively new items in the literature such as data pipelines and side-channel information collection, SAAM drawing a picture very similar to the real world indicate that assessing data pipelines with SAAM yields an accurate depiction of the capabilities of a data pipeline and a side-channel information acquisition system.

Chapter 6

Conclusion

The thesis provides and discusses a data pipeline architecture to collect side-channel information from a number of devices in a decentralized approach. The data pipeline consists of four different steps. These are: (1) collecting, (2) transmitting, (3) enriching, and (4) storing side-channel data. The architecture design consists of four major components including the target device and referred to as Interceptor, Aggregator, Analyzer and Device Under Test. Each major component implement parts of the four-step pipeline. We provide a short summary of each step below.

Collecting data occurs via a side-channel leak probe, and the pipeline interacts with the probe through the help of its components. For example, in the case study, EET collects power consumption data.

The transmission of data happens between the components until the data gets stored in the storage system. Interactions taking place until data gets stored is considered part of the transmission steps. Interactions such as sending commands, control events, logs and side-channel information fall under the transmission step.

The enrichment step provides contextual information to data. By adding meta-information bits to the time-series data being collected, enrichment enables different techniques for machine learning and analysis, such as using meta-information data as labels to train supervised machine learning models. The architecture supports querying by the attached meta-information to fetch data.

The storing step is responsible for saving the data for analysis, which can be triggered based on a callback or after storing the data. In this step, the architecture offers two types of storage: hot storage and cold storage. Hot storage refers to the storage partition with a

demand to write, read, and modify data, often requiring high I/O. The architecture stores incoming data for 24 hours in the hot storage and then moves the data to the cold storage. The cold storage is built with relatively cost-effective hardware that does not support high bandwidth I/O. Using cold storage helps preserve the dataset for potential use for 30 days based on the requirements of the architecture.

We designed the architecture based on the four steps metioned above. Design and implementation of an architecture is a subjective task, and the design can include opinionated details. To determine whether the design and the structure captures the required functionality we conducted an assessment. The analysis of an architecture involves mapping the structure and evaluating the quality attributes of the architecture.

Assessing architectures involves structured methods that evaluate the design, implementation and structure of architecture. In this thesis, we used SAAM to assess the architecture. The application of SAAM requires a set of quality attributes; in Chapter 4 we introduced maintainability, reliability, reusability and scalability as quality attributes which the architecture gets evaluated against.

The results of applying SAAM showed that the architecture has shortcomings concerning maintainability and reliability. Chapter 4 suggested changes to the architecture to overcome the shortcomings of the architecture including but not limited to, implementing an update system to the architecture.

Reliability-wise improvements to the proposed architecture include but are not limited to redundancy and fail-over methods within the architecture. Both methods improve reliability by ensuring that the risk of losing data is less compared to the architecture without suggested changes. At the moment, EET does not implement any of the suggested approaches concerning reliability. Findings from the real-world case study with EET indicate that SAAM provided an accurate analysis on the reliability topic by pointing to the right shortcomings of the architecture.

Maintainability-wise SAAM describes a set of suggested improvements such as implementing a controller on the Aggregator. The case study of EET follows the suggestions and implements them in a similar way to the suggested method. For instance, instead of adding a controller to the Aggregator, EET adds an SSH Server to the Aggregator to connect to it remotely. Based on the issues faced during designing and implementing EET, SAAM provides analysis parallel to the real world.

During the implementation and design phase of the case study of EET the conclusions drawn were parallel to the results from applying SAAM. For instance lack of redundancy and fail-safe mechanisms impact the reliability of the architecture. While implementing EET same issues arose due to lack of these features or countermeasure, indicating that

SAAM provided us with the same and correct points. As such our findings indicate that SAAM is applicable to assess data pipeline architectures.

The proposed architecture has shown that it reduces the overhead of redesigning analysis and machine learning components for new data by providing a reusable interface. After analyzing SAAM and the case study, we also see that our proposed architecture applies to other domains to enable machine learning and analysis.

The architecture provided in this thesis enables the collection of large side-channel information datasets with a scalable and decentralized approach. Additionally adapting the framework for another type of data source is a possibility based on the SAAM process in Chapter 4 and a possible future work which can contribute to the literature of different domains.

When applying SAAM, we identified the shortcomings and the advantages of the proposed architecture. Scalability of the architecture and features such as the querying mechanisms are advantages, whereas the lack of redundancy and control systems over the pipeline affected maintainability and reliability of the architecture negatively. Reducing or removing these shortcomings require improvements in the architecture and Chapter 4 provides improvement strategies and details regarding these shortcomings.

Reducing the shortcomings by implementing alternative strategies provided in Chapter 4 also lays the ground work for another round of SAAM. Reapplying SAAM would yield a comparison point between the improved version of the architecture and the proposed architecture.

Data pipelines are gaining traction due to the increased connectivity of systems, assessing the quality of data pipeline architectures has also become a necessity. With applications growing in the big-data, machine-learning, and deep-learning domain, the software architectures of data pipelines, and data processing pipelines may involve more than the quality attributes established in this thesis, expanding the analysis with quality attributes such as deployability can provide other useful insight about the architecture. In the future, a method can be developed to determine which quality attributes should be assessed to analyze a data pipeline architecture. Furthermore future research can be done towards tailoring a methodology just like SAAM but only focusing on assessing data pipeline architectures.

References

[1] MITRE History. https://cve.mitre.org/about/history.html.

[2] CVE Program Report for Q2 Calendar Year 2021, 2021. https://cve.mitre.org/blog/July282021_CVE_Program_Report_for_Q2_Calendar_Year_2021.html#metrics.

[3] CVE-2019-9977. National Vulnerability Database, March 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-9977.

[4] CVE-2016-9361. National Vulnerability Database, February 2017. https://nvd.nist.gov/vuln/detail/CVE-2016-9361.

[5] CVE-2019-9019. National Vulnerability Database, February 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-9019.

[6] Gartner forecasts worldwide security and risk management spending to exceed $150 billion in 2021. Gartner.

[7] Gabriel Hospodar, Benedikt Gierlichs, E. D. Mulder, I. Verbauwhede, and J. Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1:293–302, 2011.

[8] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 20–33, Cham, 2015. Springer International Publishing.

[9] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ascad database. *ANSSI, France & CEA, LETI, MINATEC Campus, France. Online*

verfügbar unter https://eprint. iacr. org/2018/053. pdf, zuletzt geprüft am, 22:2018, 2018.

[10] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, Caroline Sporleder, et al. Acoustic side-channel attacks on printers. In *USENIX Security symposium*, volume 9, pages 307–322, 2010.

[11] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[12] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[13] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[14] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys Tutorials*, 20(1):465–488, 2018.

[15] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer, 2016.

[16] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328, 2011.

[17] Haider Adnan Khan, Nader Sehatbakhsh, Luong N Nguyen, Milos Prvulovic, and Alenka Zajić. Malware detection in embedded systems using neural network model for electromagnetic side-channel signals. *Journal of Hardware and Systems Security*, 3(4):305–318, 2019.

[18] Carlos Moreno and Sebastian Fischmeister. On the security of safety-critical embedded systems: Who watches the watchers? who reprograms the watchers?. In *ICISSP*, pages 493–498, 2017.

[19] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[20] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The em side—channel (s). In *International workshop on cryptographic hardware and embedded systems*, pages 29–45. Springer, 2002.

[21] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[22] Nathan Liu, Carlos Moreno, Murray Dunne, and Sebastian Fischmeister. vProfile: Voltage-based anomaly detection in controller area networks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 1142–1147. IEEE, 2021.

[23] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[24] Danilo Bzdok, Naomi Altman, and Martin Krzywinski. Statistics versus machine learning. *Nature Methods*, 15(4):233–234, Apr 2018.

[25] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[26] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Side channel attack: an approach based on machine learning. *Center for Advanced Security Research Darmstadt*, 29, 2011.

[27] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[28] Bertrand. Clarke. *Principles and Theory for Data Mining and Machine Learning*. Springer Series in Statistics. Springer New York, New York, NY, 1st ed. 2009. edition, 2009.

[29] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

[30] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[31] Imola K Fodor. A survey of dimension reduction techniques. Technical report, Lawrence Livermore National Lab., CA (US), 2002.

[32] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[33] Tong Lin and Hongbin Zha. Riemannian manifold learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(5):796–809, 2008.

[34] Alexander N Gorban, Balázs Kégl, Donald C Wunsch, Andrei Y Zinovyev, et al. *Principal manifolds for data visualization and dimension reduction*, volume 58. Springer, 2008.

[35] Eric J Kostelich and Thomas Schreiber. Noise reduction in chaotic time-series data: A survey of common methods. *Physical Review E*, 48(3):1752, 1993.

[36] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[37] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[38] Tom M Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11):30–36, 1999.

[39] Ludwig Schmidt, Shibani Santurkar, Dimitris Tsipras, Kunal Talwar, and Aleksander Madry. Adversarially robust generalization requires more data. *arXiv preprint arXiv:1804.11285*, 2018.

[40] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[41] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[42] Joao Carreira, Eric Noland, Chloe Hillier, and Andrew Zisserman. A short note on the kinetics-700 human action dataset, 2019.

[43] Kyle Wiggers. Openai disbands its robotics research team, Jul 2021.

[44] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[45] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300, 2019.

[46] Axel Van Lamsweerde. From system goals to software architecture. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 25–43. Springer, 2003.

[47] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[48] Martin Glinz. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*, pages 21–26. IEEE, 2007.

[49] Timothy Christian Lethbridge and Robert Laganiere. *Object-oriented software engineering*, volume 11. McGraw-Hill New York, 2005.

[50] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987.

[51] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2020.

[52] Maarten Van Steen and A Tanenbaum. Distributed systems principles and paradigms. *Network*, 2:28, 2002.

[53] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.

[54] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005.

[55] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[56] Apache Parquet. https://parquet.apache.org. Accessed: 2021-05-30.

[57] Apache ORC. https://orc.apache.org/docs/. Accessed: 2021-05-30.

[58] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, page 356–361, New York, NY, USA, 2014. Association for Computing Machinery.

[59] Zheng Qin, Xiang Zheng, and Jiankuan Xing. *Introduction to software architecture*. Springer, 2008.

[60] R. Kazman, L. Bass, G. Abowd, and M. Webb. Saam: A method for analyzing the properties of software architectures. In *Proceedings of 16th International Conference on Software Engineering*, pages 81–90, 1994.

[61] Len Bass, Paul Clements, Rick Kazman, and Mark Klein. Models for evaluating and improving architecture competence. Technical Report CMU/SEI-2008-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2008.

[62] Kurt R Linberg. Software developer perceptions about software project failure: a case study. *Journal of Systems and Software*, 49(2):177–192, 1999.

[63] Bill C. Hardgrave, Fred D. Davis, and Cynthia K. Riemenschneider. Investigating determinants of software developers' intentions to follow methodologies. *Journal of Management Information Systems*, 20(1):123–151, 2003.

[64] Micheal A. Chilton, Bill C. Hardgrave, and Deborah J. Armstrong. Person-job cognitive style fit for software developers: The effect on strain and performance. *Journal of Management Information Systems*, 22(2):193–226, 2005.

[65] Robert D. Austin. The effects of time pressure on quality in software development: An agency model. *Information Systems Research*, 12(2):195–207, 2001.

[66] Harsh Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the pentium processor. 1994.

[67] BBC News. UK Computer problems hit CSA payouts. *BBC News*, 2003.

[68] Joseph Herkert, Jason Borenstein, and Keith Miller. The boeing 737 max: Lessons for engineering ethics. *Science and engineering ethics*, 26(6):2957–2974, 2020.

[69] Hosagrahar V Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[70] Deliverable Leader SUH and Ekaterini Ioannou. A configurable real-time data processing infrastructure mastering autonomous quality adaptation. 2016.

[71] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, et al. Software quality attributes and trade-offs. *Blekinge Institute of Technology*, 97(98):19, 2005.

[72] Anne Immonen, Pekka Pääkkönen, and Eila Ovaska. Evaluating the quality of social media data in big data architecture. *Ieee Access*, 3:2028–2043, 2015.

[73] Pekka Pääkkönen and Daniel Pakkala. Reference architecture and classification of technologies, products and services for big data systems. *Big Data Research*, 2(4):166–186, 2015.

[74] ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Technical report, International Organization for Standardization, Geneva, CH, March 2011.

[75] David J Smith. *Reliability, maintainability and risk: practical methods for engineers*. Butterworth-Heinemann, 2017.

[76] R Prieto-Diaz and P Freeman. Classifying software for reusability. *IEEE software*, 4(1):6–16, 1987.

[77] Michael R. Lyu. *Handbook of software reliability engineering*. IEEE Computer Society Press; McGraw Hill, 1996.

[78] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: A big data - ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1328–1347, 2021.

[79] Naresh Jotwani Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. Mcgraw-Hill Education, 2 edition, 2008.

[80] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[81] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *Software, IEEE*, 13:47 – 55, 12 1996.

[82] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Citeseer, 2000.

[83] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys (CSUR)*, 47(4):1–33, 2015.

[84] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[85] Sandeep Bhatt, Pratyusa K. Manadhata, and Loai Zomlot. The operational role of security information and event management systems. *IEEE Security Privacy*, 12(5):35–41, 2014.