UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Codes und Kryptographie

Privacy-Preserving Collection and Evaluation of Log Files

Bachelor's Thesis

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science

by
Angelina Koch

submitted to:
Prof. Dr. Johannes Blömer
and
Prof. Dr.-Ing. Juraj Somorovsky

Paderborn, May 25, 2023

Contents

1 Introduction

New technology evolves fast and so do new threats. The number of found vulnerabilities per year increases[1] and with them the possibilities for new attacks. Since attacks can have huge impacts on companies, it is important to avoid or at least detect them. Otherwise, the companies could face a decreasing reputation due to downtimes from denial-of-service attacks or be threatened by blackmail. To avoid these situations, the companies can use log files. Each entry of such a log file contains data about a request that was submitted by an employee to a specific website. By using log files, it is possible to find out where malicious software came from or who and which device accessed a malicious website and downloaded malware. This can be very useful, but not with regard to preserving privacy. In our scenario, the data in these log files can be sensitive. In particular, they contain information about when an employee of a company visited which websites. Every time an employee sends a request to the internet, a server stores the necessary data as a new log file entry in the company's log file database. Then this data can be analyzed to e.g., find out which employee infected the company's device with the detected malware. Since not all employees can be blamed for a malicious download, it is not necessary to see all employees' log entries in the clear. Otherwise, it would violate their privacy without good reason.

To underline the importance of privacy, we first try to explain what it means. Privacy can be described as information about us that we do not want to be known publicly, not being revealed to the public nor a single unauthorized person. But as described in [Par83], privacy is more complex since it cannot be preserved for anything already public and only as long as the persons themselves want their data to be private. Furthermore, privacy has many principles that are defined in [PH10]. The most interesting ones for us are anonymity and unlinkability. In our context, anonymity means that a log file entry cannot be linked to the employee who caused it. Hence, each issuer of a request stays anonymous. Moreover, unlinkability refers to the fact that it cannot be told whether two different log entries are related. Additionally, we are talking about pseudonymity if we store pseudonyms instead of the names of the employees. Privacy has many more principles, but we do not focus on those. Plus, in the cryptographic sense, privacy can also be defined by an adversary, that interacts with a function over database entries without learning more about the entries than the function's results [GLP11]. This should give a good idea of what privacy is to understand its importance. And this is why we investigate a way to collect and evaluate log files for more security without violating the privacy of innocent employees.

[1] `https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time`

1.1 Related Work

Some previous work on privacy-preserving log files anonymize the sensitive parts of data to improve privacy [Rat16, OHS13]. The paper of Oliver et al. [OHS13] proposes their framework that helps to anonymize the sensitive data inside the log files. For this, the users must define some *clean-up* functions that remove unnecessary data. Those functions could for example, generalize a stored location to only the country or hash the user's ID. Whether this completely preserves privacy, depends on how well these clean-up functions are made. And this requires deciding which data is sensitive and which is not. If the log file is unstructured, in contrast to our log files which contain predefined values, finding the sensitive values can be challenging. For example, if we find a person's name in a search query, this can be sensitive data, if it is the issuer's name. But it would not be sensitive if it is some celebrity the issuer searches for. A solution to clearly divide sensitive data from non-sensitive is presented in [Rat16]. They designed a system with two streams of log files. One stream contains the original, raw data including all sensitive data. This one needs to be encrypted to ensure privacy and security. But this first stream is not used for analysis. Instead, Rath [Rat16] recommends creating a second stream which only contains sensitive data that is anonymized. For this stream, the logging function takes templates (without any sensitive data) and the sensitive data as two distinct inputs. Due to this distinction, the users know the sensitive data and can replace sensitive strings by strings of * or only their first letter. These anonymized log file entries are stored unencrypted and are used for analysis. But there is a problem with those self-created anonymizing functions: Imagine a company has an employee named Yorgens, such that the anonymizing function would change Yorgens to Y and this log file entry would then be stored unencrypted. Since everyone has access to the unencrypted log file and Yorgens is the only employee in this company that starts with Y, everyone can derive that the entries with Y are from Yorgens. Hence, for him, we cannot preserve privacy anymore. Therefore, we want a solution, that does not only shorten the values in the log files but makes them completely anonymized.

The second idea we discuss has been designed for the log files of search engines, such as Google and Yahoo [BCV16]. A goal of search engine owners is to sell the logged data to make a profit with it. But if the data would be anonymized, this would not be possible anymore because they then cannot extract and infer the information from the logged data as before [BCV16]. Due to this, Bondia-Barceló et al. created a logging system that is privacy-preserving while it allows to sell the log entries' content for profit. To do so, they first remove sensitive data, that can reveal the sender's identity, from each search query. Second, they analyze the words inside the remaining content of each query to find its overall purpose. These purposes could for example, be health, science, or shopping. Log file entries of the same overall purpose are stored in the same collection. If this collection is filled with a specified number of log entries, they are *anonymized*. For this, one of the log entries and one of the metadata of the entries is each chosen at random[2]. Then, Bondia-Barceló et al. replace the metadata of the chosen entry by the

[2]Per definition, the chosen log file entry and metadata should not originate from the same search query.

randomly chosen metadata and achieve a log file entry with metadata from a different person. This is considered to be privacy-preserving because no information about the issuer, but of an issuer with the same overall interest, is leaked. This approach is also not optimal for our purposes because the change of each request's metadata makes it impossible to find the person that downloaded the malicious software.

Next, we take a look at Differential Privacy. This was first mentioned by Dinur, Nissim, and Dwork in [DN03, DMNS06]. Afterward, many papers were published that either make use of Differential Privacy for their specific purposes or that try to further improve it [DR14, MKB$^+$19, Vin22]. In the use case of Differential Privacy, a function is evaluated on sensitive data. To protect the users' privacy, the function's output is afterward added with random noise[3]. This leads to changed output such that the given output does not reveal too much information about the sensitive data that was used. Nevertheless, the altered output is still close enough to the correct output to be useful. We discuss Differential Privacy and whether it can be applied to our scenario in more detail in Section 5.2.

Moreover, there also exist cryptographic approaches to preserve the users' privacy. The first one we mention here, is [UKH$^+$21, UKK$^+$22], which is another work in the context of logging users' queries to web search engines. To avoid the users from being linkable to their queries, Ullah et al. group the users [UKH$^+$21]. After that, each query of a user is forwarded by another member of his group such that the true identity of each query's issuer is unknown to the web search engine. Later on, Ullah et al. extended their work such that group members are as different as possible to improve unlinkability [UKK$^+$22]. A disadvantage of this general approach is that it heavily depends on symmetric and asymmetric encryption, which is used to avoid the members of a group reading each other's queries and responses. Furthermore, the search queries themselves can contain sensitive data that reveal the issuer's identity. Therefore, this information must additionally be removed to hide the identity of each search query's sender. Thus, we are still looking for a better solution to preserve privacy in our scenario.

The second cryptographic approach we mention here is called Searchable Encryption [ABO07, BBO07]. This fits our scenario very well since its goal is to encrypt the cells of database tables while enabling search queries on the data. For this, one computes tags for each value in the database's tables. Those tags can be computed with the value itself as input, as well as from its ciphertext. Therefore, these tags can be used to form queries and respond to them without accessing the real data. But the problem with this is, that it focuses more on searchability than on privacy and thus data is revealed if this is necessary to respond to a query. Moreover, one of the schemes that belongs to this concept, needs the stored values to have a "high min-entropy" [BBO07]. But this requirement cannot be met in our scenario, since we most likely have small spaces of e.g., IP addresses of the employees since those are limited to a fixed and not huge

[3]It is also possible to add random noise to the function's inputs and then apply the function on the altered input. But this version of Differential Privacy is not discussed in this work, since the distribution of the noise can be learned then [AS00][DMNS06, page 268].

number.[4] Additionally, we want to regard encryption as one of our basic approaches to give a first impression of what can be achieved by using less complex mechanisms. For those approaches, the work about searchable encryption is already too complex and hence not suitable as basic approach. Though searchable encryption fits our scenario and the goals of this work well, we do not discuss it as advanced approach since our focus is already on Prio [CB17].

There exist many more cryptographic approaches for privacy. One of them is the commercial solution XOR, which is introduced by Inpher and makes use of multi-party computation [Inp22]. Another paper from Liu et al. describes a privacy-preserving way to track Covid-19 infections with a warning system [LZCW23]. This approach uses blockchains, group signatures, zero-knowledge proofs, and more to ensure privacy in each context of monitoring the spread of the virus.

More importantly, we discuss other related work such as Prio [CB17] with and without log files, newer variants of Prio, Poplar [BBC$^+$21], Private Set Intersection [DMRY09], and Differential Privacy in *Advanced Approaches* (Chapter 5) in detail.

1.2 Contribution

In this work, we design a system that allows private collection and evaluation of data stored in log files. Our contribution is to define *evaluation functions* (Section 3.2) that can be computed using those data. Furthermore, we define *privacy goals* (Section 3.4) which describe the privacy properties we want to ensure in the scenario we assume in this work. Given this basic construction, our work consists of analyzing the different cryptographic approaches, that we find most interesting for our purposes. Those make use of hash functions, encryption, Prio [CB17], Poplar [BBC$^+$21], Differential Privacy [DN03, DMNS06], and Private Set Intersection [DMRY09]. We use these thoroughly selected cryptographic approaches to design our privacy-preserving log file collection and evaluation system. For this, we introduce a grading system (Section 3.5) that is used to grade and compare all those approaches. Regarding our encryption approaches, we prove that we cannot recognize whether two ciphertexts originate from the same plaintext unless our used probabilistic encryption scheme is insecure against chosen plaintext attacks (Section 4.2 and Appendix B.1). For Prio and Poplar, we describe in all details how those approaches are applied to our evaluation functions (Sections 5.1 and 5.3). This is more complicated than for other approaches since each evaluation function must be realized differently. Moreover, for one of those aggregation functions from Prio (maximum), we also figure out how it works in particular for our setting and implement it (Section 5.1.4). Additionally, for the less complex approaches, we design a construction which allows us to check whether malicious employees tampered their request data before the evaluation (Section 5.3). In the end, we use our findings to create our privacy-preserving log file collection and evaluation system (Chapter 6).

[4]This problem can be reduced with bucketization, where we output shorter digests to achieve more collisions for the tags. See [BBO07] for more details.

1.3 Overview

In the following, we describe and analyze the cryptographic approaches that can be useful for our privacy-preserving log file collection and evaluation system. To do so, we first list some preliminaries in Chapter 2 to get started with the formal definitions and notations we use in this thesis. After that, we describe the scenario in Chapter 3 for which our privacy-preserving log file system is designed. In that chapter, we also discuss which values in our scenario's log entries are sensitive and which are necessary for our evaluations. Next, we analyze the different approaches. The first of them are basic approaches (Chapter 4). There, we analyze hashing and encryption and grade how well they work for our scenario. After that, we consider more advanced approaches in Chapter 5. Thereby, our main interest is in Prio [CB17]. But we also name advantages of newer variants of Prio, such as Prio3 [BBC$^+$19], Prio+ [AGJ$^+$22], Poplar [BBC$^+$21], and our Prio extension using log files in Section 5.3. Moreover, we investigate as advanced approaches the concepts of Differential Privacy [DN03, DMNS06] and Private Set Intersection [DMRY09]. After that, in Chapter 6, we compare the results of the functionality and privacy grading of all analyzed approaches. This helps us find a solution for our scenario that is as well privacy-preserving as functional regarding the defined functions. In the last chapter (Chapter 7), we conclude the thesis by summarizing the important results and findings.

2 Preliminaries

In this chapter, we list definitions that are fundamental to the cryptography we use in this work. At first, we mention negligibility and poly-bounded adversaries, which are central to all concepts that are defined in this chapter. Next, we explain the concepts of hashing and encryption. Additionally, we require some mathematical constructions that are used by our advanced approach Prio in Section 5.1. The following definitions are inspired and partly adopted from *A Graduate Course in Applied Cryptography* written by Boneh and Shoup [BS22]. But first, we name the notations that are used in this work.

2.1 Notations

The first notation we use is $\mathbb{N}$ which contains 0, hence we denote $\mathbb{N} = \{0, 1, 2, \dots\}$. We also use the notation $a := b$ to denote that the variable a is set to value b. Moreover, a **field** $\mathbb{F}_p$ contains the set $\{0, \dots, p-1\}$ and all mathematical operations are executed in mod p. We use $\mathbb{P}[A]$ to denote the **probability** that some event A occurs. The notation $\vec{v}$ is used for **vectors**. Additionally, by $x \leftarrow X$, we denote that the value x is **chosen uniformly at random** from the space X. Furthermore, we define $s \in \{0,1\}^n, n \in \mathbb{N}$ to be a string and $|s| = n$ its length. With $s[0..z]$ we denote a substring of s with length $z \leq |s|$. Hence, it holds that $s[0..|s|] = s$ and $s[0..0]$ is the empty word.

2.2 Terms of our Scenario

Next, we define the notations that belong to our **log file database** scenario: By `Log_Files` we denote our database's table which contains all important data regarding issued requests.[1] `Log_Files` contains $R \in \mathbb{N}$ **log file entries** entry_{row} with $row \in \{1, 2, \dots, R\}$. A log file entry entry_{row} is represented as a row in the `Log_Files` table. Moreover, this table has $|\mathcal{CO}|$ **columns** where each $col \in \mathcal{CO}$ is also represented as column[2]. If we only talk about a specific column but the log entry is arbitrary, we write $\mathsf{entry}[col]$ to address a column of a log file. Further, we write $\mathsf{entry}_{row}[col]$ when we want to address a single value stored in a **cell** of entry entry_{row} in column col. For example, if we want to investigate the *timestamp* of the 5th log entry, we address this value with $\mathsf{entry}_5[timestamp]$. For clarity, the terms *log file entry*, *column*, and *cell* are also displayed in Figure A.1. The value set $\mathcal{V}_{col}$ defines which values are

[1]There are two other tables in our database, but they are only for administrative purposes and are hence not defined in more detail than in Section 3.1.

[2]The columns that we use in our specific scenario are defined in Section 3.1.

valid for which of the columns, such that an entry $\mathsf{entry}_{row}[col]$ is valid if and only if $\mathsf{entry}_{row}[col] \in \mathcal{V}_{col}, row \in \{1, 2, \ldots, R\}$.

2.3 Negligibility, Super-poly, Poly-bounded, Adversaries, and Security Games

In this section, we explain important terms in the sense of cryptography. We start with negligibility, which can roughly be understood as *so tiny that it is nearly 0.*

Definition 2.1 (Negligibility)**.** *We define a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ to be **negligible**, if and only if it holds that $\forall p \in \mathbb{R}_{>0} : \exists x_0 \in \mathbb{N}\backslash\{0\} : \forall x \geq x_0$ with $x \in \mathbb{Z} : |f(x)| < x^{-p}$. Or in other words: For any x that is larger than some boundary x_0, $f(x)$ becomes very small.*

The term negligibility is often used in cryptography to describe the advantage that an adversary[3] should have in the best case. This would mean, that an adversary that plays a security game, has a very low probability of succeeding or guessing the correct value.

The next important term is super-poly. A function can never be super-poly and negligible simultaneously.

Definition 2.2 (Super-poly)**.** *We define a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ to be **super-poly** if its inverse $g := 1/f$ is negligible as defined above.*

The term super-poly is often used to describe a number of queries that can be sent by e.g., an adversary.

The last essential term is poly-bounded. With this term, an adversary is described that can at most submit a polynomial number of queries. For understandability, one can think of an adversary that only has limited computational power available. The term *adversary* is defined afterward.

Definition 2.3 (Poly-bounded)**.** *We define a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ to be **poly-bounded**, if it holds that $\exists p, q \in \mathbb{R}_{>0} : \forall x \in \mathbb{N} : |f(x)| \leq x^p + q$.*

Additionally, we understand by the term adversary a person or software that tries to harm the company's system or its employees. In our case, this adversary tries to violate the employees' privacy.

Definition 2.4 (Adversary)**.** *We define **adversary** $\mathcal{A}$ as an algorithm that uses internal randomness to output y on input x. If $\mathcal{A}$ is poly-bounded as defined above, we call $\mathcal{A}$ an **efficient adversary**.*

Additionally, we define the term *winning* for the following security games.

Definition 2.5 (Winning a Security Game)**.** *We define that an adversary $\mathcal{A}$ **wins** a security game G against a challenger if and only if the challenger outputs 1 at the end of G. Otherwise, we say that $\mathcal{A}$ loses G.*

Next, we list important definitions of the approaches that we regard during this thesis.

[3]The formal definition of adversary follows in Definition 2.4.

2.4 Hashing Definitions

One of our basic approaches is hashing. For this, we consider three different types of hash functions: Keyless hash functions e.g., SHA-256 and SHA-3, hash functions with salt where a random value is additionally used as input, and keyed hash functions.

The first version of hashing that we want to regard is keyless hashing. For this, the hash function is a function that maps from a very large *message space* to a much smaller *digest space* without any additional input. We refer to the elements of a message space as *plaintexts*. Additionally, we call the elements from the digest space *digests*.

Definition 2.6 (Keyless Hash Functions)**.** *We define a* ***keyless hash function*** *H to be a deterministic function that maps from an infinite message space $\mathcal{M} = \{0,1\}^*$ to a much smaller digest space $\mathcal{D} = \{0,1\}^l$ with $l \in \mathbb{N}$. Hence, our keyless hash function is defined as $H : \mathcal{M} \to \mathcal{D}$ and takes no additional input.*

Often the spaces $\mathcal{M}$ and $\mathcal{D}$ are used as binary ones, e.g., $\mathcal{M} = \{0,1\}^*$ and $\mathcal{D} = \{0,1\}^l$. Then our hash function can take inputs of arbitrary length and always outputs a digest with the fixed length l. But in practice, the hash function H can take arbitrary ASCII symbols as input and outputs a hexadecimal value of fixed length $l/4$, which is not human-readable anymore. In our scenario, most of the values in the log entries are non-numerical. Hence, it would not suffice to assume binary preimage and image spaces. But, whenever we argue with $\mathcal{M}$ and $\mathcal{D}$ being binary, these considerations can be easily applied to scenarios where input and output are not binary. For this, we only have to translate the ASCII symbols and hexadecimals into bits or vice versa.

But there is another problem with keyless hashing that cannot be solved as easily. Since hashes can be precomputed using rainbow tables[4], we need another version of hashing to protect our log entries. Thus, we next regard hashing with salt (or pepper). For this, a randomly chosen value, the salt is also given as input to the hash function. If this value differs for e.g., each user[5], adversaries must compute one rainbow table per user to *reverse* all hashes of the database. Hence, the salt complicates the precomputation of digests that can be done with rainbow tables. Therefore, we understand salted hashing as:

Definition 2.7 (Salted Hash Functions)**.** *We define a* ***salted hash function*** *H_{salt} to be a deterministic function that maps from an infinite message space $\mathcal{M} = \{0,1\}^*$ and a finite salt space $\mathcal{S} = \{0,1\}^\mu, \mu \in \mathbb{N}$ to a finite digest space $\mathcal{D} = \{0,1\}^l, l \in \mathbb{N}$. Hence, our hash function is defined as $H_{\text{salt}} : \mathcal{M} \times \mathcal{S} \to \mathcal{D}$ and takes the μ-bits salt $\in \mathcal{S}$ as additional input. Thus, our salted hash function is defined over the triple $(\mathcal{M}, \mathcal{S}, \mathcal{D})$.*

[4]Rainbow tables use a reduction function $\mathsf{R} : \mathcal{D} \to \mathcal{M}$. This function R cannot compute the inverse of the hash function, but it maps from the digest space to the message space and is (alternating with the hash function) applied to a digest. This approach allows computing hashes of a huge part of the message space if R is well-designed. This way, huge tables called rainbow tables are created. With these, adversaries can look up the preimages of the precomputed digests. Thus, rainbow tables can get large and need a lot of storage. Hence, if we force adversaries to compute many different rainbow tables, to precompute a single database, we slow them down and increase their need for storage. See [KKJ+13] for more.

[5]In our scenario they are named *employees*.

Unlike pepper, which is used as salt, but stored in a second database, the salt of salted hashing is stored along with the data it is used for. Therefore, it is all the more important to generate a new random salt for each user. Otherwise, if adversaries get access to the database, they additionally get to know the salt that is stored there. Then, they could use this salt as input to the hash function to compute a single rainbow table, that would suffice to reverse all digests from the database.

The third version of hash functions also complicates the precomputation of hashes that uses rainbow tables. Those hash functions are keyed hash functions, where instead of the salt, a key is used as a second input to the hash function.

Definition 2.8 (Keyed Hash Functions)**.** *We define a* ***keyed hash function*** H_{key} *to be a deterministic function that maps from an infinite message space* $\mathcal{M} = \{0,1\}^*$ *and a finite key space* $\mathcal{K} = \{0,1\}^\lambda, \lambda \in \mathbb{N}$ *to a finite digest space* $\mathcal{D} = \{0,1\}^l, l \in \mathbb{N}$. *Hence, our keyed hash function is defined as* $H_{\text{key}} : \mathcal{M} \times \mathcal{K} \to \mathcal{D}$ *and takes the* λ*-bit key* $k \in \mathcal{K}$ *as additional input. Moreover, we identify the hash function over the triple* $(\mathcal{M}, \mathcal{K}, \mathcal{D})$.

The key k of a keyed hash function H_{key} must not be secret. It is used to pick a hash function from a large set of hash functions at random, and thus complicates the precomputation of rainbow tables. Moreover, this key cannot be compared to keys that are used in the context of encryption. Those keys can encrypt and decrypt, while this *hash key* can only be used to hash messages, but the reverse is impossible.

Security Definitions of Hashing

In practice, it is important to choose whether the applied hash function uses salts or keys and which specific function should be used. Hence, we discuss some security definitions regarding hash functions that we consider important for our analysis of hashing in Section 4.1. Since we only regard security definitions for keyed hash functions, we denote them by H instead of H_{key}. And as the key k for a keyed hash function H is not meant to be secret, k is always given to the adversary in the following security games.

We use two of the existing security definitions: One-wayness and collision resistance. The first one, we want to focus on, is one-wayness. If a hash function is one-way, then it is hard for any efficient adversary to find any preimage of the hash function's outputs.

Definition 2.9 (One-wayness of Keyed Hash Functions)**.** *We define* ***one-wayness*** *for a keyed hash function* H *that is defined over the spaces* $(\mathcal{M}, \mathcal{K}, \mathcal{D})$ *by introducing the following security game between a challenger and an arbitrary adversary* $\mathcal{A}$*:*

Security Game: One-wayness of Keyed Hash Functions

Challenger		$\mathcal{A}$
$m \leftarrow \mathcal{M}$		
$k \leftarrow \mathcal{K}$		
	$\xrightarrow{(k, d := H(m,k))}$	
		$\ldots$
	$\xleftarrow{m'}$	
if		
$m' \in \mathcal{M}$ *and*		
$H(m', k) = d$		
output 1		
else *output 0*		

*Hence, we call H **one-way** if and only if every efficient adversary $\mathcal{A}$ wins the previously defined security game with negligible probability.*

Hence, one-wayness as a security requirement for our hash function, ensures that it is hard to find any preimage of a hashed log file entry. For us, that means, that no one can easily regain the original values of the log file entries after we hashed them.

The second and last security definition we want to name is collision resistance. To do so, we first explain the term collision, which describes the problem of different messages being hashed to the same digest.

Definition 2.10 (Collision). *We define a **collision** as two different messages $m, m' \in \mathcal{M}, m \neq m'$ being mapped to the same digest $H(m,k) = H(m',k)$ with $k \leftarrow \mathcal{K}$.*

This property can become very important for our analysis. For the hashing approach, we hash the values in our log files. If different values would be hashed to equal digests, we would recognize them as identical values by mistake. Next, we present the security definition for collision resistance:

Definition 2.11 (Collision Resistance of Keyed Hash Functions). *We define **collision resistance** for a keyed hash function H that is defined over the spaces $(\mathcal{M}, \mathcal{K}, \mathcal{D})$ by introducing the following security game between a challenger and an arbitrary adversary $\mathcal{A}$:*

Security Game: Collision Resistance of Keyed Hash Functions

Challenger		$\mathcal{A}$
$k \leftarrow \mathcal{K}$		
	$\xrightarrow{\quad k \quad}$	
		$\dots$
	$\xleftarrow{\quad (m, m') \quad}$	
if		
$m, m' \in \mathcal{M}$ and		
$m \neq m'$ and		
$H(m, k) = H(m', k)$		
output 1		
***else** output 0*		

*Hence, we call H **collision resistant** if and only if every efficient adversary $\mathcal{A}$ wins the previously defined security game with negligible probability.*

In other words, if it is hard for every efficient adversary to find two different values with the same digests, we denote the hash function H as collision resistant. Important is that it is only *hard* to find such two colliding messages. But since the digest space $\mathcal{D}$ can be much smaller than the message space $\mathcal{M}$, we cannot completely prevent collisions. It thus can always happen that some of our log file entries are hashed to the same digest, though they are completely different. We deal with this problem in Section 4.1, where we discuss the approach of hashing in detail. Nevertheless, we prefer using hash functions that are collision resistant because their probability for collision and hence their rate of incorrectly identified log file entries is lower.

2.5 Encryption Definitions

In the following section, we list the definitions that are useful for our second basic approach, which is encryption. For this, we first identify deterministic encryption and the necessary security definition. Next, we regard probabilistic encryption and its security definition as well. But we discuss in Section 4.2 why probabilistic encryption cannot serve our purposes well enough.

At first, we focus on the definition of deterministic encryption schemes. The most secure ones are Shannon ciphers, which are perfectly secure but only for keys with the same length as the plaintext. Since some values in our log entries, such as the URL, can become very long, we consider computational ciphers instead of Shannon ciphers.

Definition 2.12 (Computational Ciphers)**.** *We define a **computational cipher** E as a tuple of two deterministic algorithms* $(\mathsf{Enc}, \mathsf{Dec})$ *for message space $\mathcal{M}$, key space $\mathcal{K}$, and ciphertext space $\mathcal{C}$. Those algorithms are defined as follows:*

- $\mathsf{Enc}(m, k)$ *takes as input a message* $m \in \mathcal{M}$ *and a key* $k \in \mathcal{K}$ *and outputs a ciphertext* $c \in \mathcal{C}$*.*

- $\mathsf{Dec}(m, k)$ *takes as input a ciphertext* $c \in \mathcal{C}$ *and a key* $k \in \mathcal{K}$ *and outputs a message* $m \in \mathcal{M}$*.*

*For the **correctness** of a computational cipher E, we require that:*

$$\forall m \in \mathcal{M} : \forall k \in \mathcal{K} : \mathsf{Dec}(\mathsf{Enc}(m, k), k) = m.$$

One may recognize that we defined Enc to be deterministic, while it is usually defined as a probabilistic algorithm. This restriction is necessary because the ciphers would otherwise not be useful for our scenario. When we analyze the encryption approach in Section 4.2 we encrypt each log file entry and we evaluate our evaluation functions which are defined in Section 3.2 with the encrypted log entries as input. Hence, we need that two identical messages are encrypted to the same ciphertext. Or more formally said:

For each pair of messages $m_1, m_2 \in \mathcal{M}$ with $m_1 = m_2$ and each key $k \in \mathcal{K}$, we require that $\mathsf{Enc}(m_1, k) = \mathsf{Enc}(m_2, k)$ with probability 1.

Hence, we can only regard Enc algorithms that are deterministic such that their outputs are still comparable. Some ciphers that can be used for our analysis in Section 4.2 are e.g., stream ciphers which use pseudorandom generators or block ciphers in ECB mode. To specify the security of the deterministic computational ciphers from above, we use the security game for semantic security:

Definition 2.13 (Semantic Security)**.** *We define **semantic security** for a deterministic cipher E that is defined over the spaces* $(\mathcal{M}, \mathcal{K}, \mathcal{C})$ *by introducing the following security game between a challenger and an arbitrary adversary* $\mathcal{A}$*:*

Security Game: Semantic Security

Challenger		$\mathcal{A}$
$b \leftarrow \{0,1\}$		
		$\dots$
	$\xleftarrow{(m_0, m_1)}$	
$k \leftarrow \mathcal{K}$		
$c := \mathsf{Enc}(m_b, k)$		
	$\xrightarrow{c}$	
		$\dots$
	$\xleftarrow{\hat{b} \in \{0,1\}}$	
if		
$m_0, m_1 \in \mathcal{M}$ *and*		
$\lvert m_0 \rvert = \lvert m_1 \rvert$ *and*		
$b = \hat{b}$		
output 1		
else *output 0*		

Hence, we call a cipher E semantically secure if and only if every efficient adversary $\mathcal{A}$ outputs the correct $b = \hat{b}$ with probability at most $1/2 + \varepsilon$, with ε being negligible. This means, that each adversary $\mathcal{A}$ must be slightly better than simply guessing the correct b.

Nevertheless, if we only use deterministic ciphers, the security and therefore the privacy of the log file entries suffers. Since we can recognize ciphertexts of equal log entries, any employee and adversary can do so as well. For example, they can send a request and compare the ciphertext of the URL with all other ciphertexts of URLs in the log file. Thus, they can get information about which websites have been visited. Moreover, they can do this for any value in the log entries since we use deterministic encryption. Therefore, it would be better to use probabilistic encryption. This can be achieved if the algorithm Enc of our computational cipher E is probabilistic. The probabilistic cipher E is correct if it holds that:

$$\forall m \in \mathcal{M} : \forall k \in \mathcal{K} : \mathbb{P}[\mathsf{Dec}(\mathsf{Enc}(m, k), k) = m] = 1.$$

Then, for the definition of E's security, we consider the following security game:

Definition 2.14 (Security against Chosen Plaintext Attacks)**.** *We define **security against chosen plaintext attacks** for a cipher E that is defined over the spaces $(\mathcal{M}, \mathcal{K}, \mathcal{C})$ by introducing the following security game between a challenger and an arbitrary adversary $\mathcal{A}$:*

Security Game: Chosen Plaintext Attack

Challenger		$\mathcal{A}$
$b \leftarrow \{0,1\}$		
$k \leftarrow \mathcal{K}$		
		for $i \in \{1,2,\dots\}$ *send queries:*
		$\dots$
	$\xleftarrow{(m_{i,0}, m_{i,1})}$	
$c_i := \mathsf{Enc}(m_{i,b}, k)$		
	$\xrightarrow{c_i}$	
		$\dots$
	$\xleftarrow{\hat{b} \in \{0,1\}}$	

```
if
$\forall i : m_{i,0}, m_{i,1} \in \mathcal{M}$ and
$\forall i : |m_{i,0}| = |m_{i,1}|$ and
$b = \hat{b}$
output 1
else output 0
```

Hence, we call a cipher E secure against chosen plaintext attacks (or shortly: CPA secure) if and only if every efficient adversary $\mathcal{A}$ outputs the correct $b = \hat{b}$ with probability at most $1/2 + \varepsilon$, with ε being negligible. This means, that each adversary $\mathcal{A}$ must be slightly better than simply guessing the correct b.

To get a cipher that is CPA secure, Enc must be probabilistic.

All previously defined encryption schemes are symmetric, but we could use asymmetric encryption as well. Hence, in the following we describe asymmetric encryption which uses different keys for encryption and decryption. Moreover, the key for encryption is public such that there is no need to share different keys with different communication partners.

Definition 2.15 (Asymmetric Encryption Schemes)**.** *We define an* ***asymmetric encryption scheme*** *E as a triple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *for message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$.*

- $\mathsf{Gen}()$ *outputs a secret key sk and a public key pk.*
- $\mathsf{Enc}(m, pk)$ *takes as input a plaintext $m \in \mathcal{M}$ and a public key pk and outputs a ciphertext $c \in \mathcal{C}$.*
- $\mathsf{Dec}(c, sk)$ *takes as input a ciphertext $c \in \mathcal{C}$ and a secret key sk and outputs a plaintext $m \in \mathcal{M}$.*

*For the **correctness** of an asymmetric encryption scheme, we require that*

$$\mathbb{P}[\mathsf{Dec}(\mathsf{Enc}(m, pk), sk) = m] = 1.$$

This implies that Dec *must always be deterministic because otherwise it could not be ensured that any ciphertext is correctly decrypted to the original plaintext.*

In Section 4.2 we explain that we can only use deterministic encryption for the purpose of comparing ciphertexts instead of their plaintexts. Hence, we can only use deterministic encryption schemes. Moreover, this leads to Enc being deterministic and thus it already suffices for the correctness of a deterministic asymmetric encryption scheme to fulfill:

$$\mathsf{Dec}(\mathsf{Enc}(m, pk), sk) = m.$$

But it can be shown that asymmetric encryption schemes that are also deterministic are not CPA secure. The definition of CPA security for asymmetric encryption schemes is similar to Definition 2.14 unless for asymmetric encryption schemes, the keys are chosen differently. Hence, the challenger picks $(pk, sk) \leftarrow \mathsf{Gen}()$ and also sends pk to the adversary. Additionally, the encryption of the adversary's queries is done by the asymmetric encryption algorithm $\mathsf{Enc}(m_{i,b}, pk)$. To win this security game, an adversary would only have to pick the same message for m_0 twice and two different messages for m_1 and can already determine whether the chosen bit b is 0 or 1. Moreover, deterministic asymmetric encryption schemes are not semantically secure as well. For this, the adversary $\mathcal{A}$ only has to compute the ciphertexts of the messages m_0, m_1 he chose using the public key pk. Then, he compares them to the provided c to decide which of the messages was encrypted by the challenger. Therefore, we use another security game called message recovery attack, which can be ensured for deterministic asymmetric encryption schemes.

Definition 2.16 (Security against Message Recovery Attacks)**.** *We define **security against message recovery attacks** for an asymmetric encryption scheme E that is defined over spaces $(\mathcal{M}, \mathcal{C})$ and uses algorithms* (Gen, Enc, Dec) *by introducing the following security game between a challenger and an arbitrary adversary $\mathcal{A}$:*

Security Game: Message Recovery for Asymmetric Encryption

Challenger		$\mathcal{A}$
$m \leftarrow \mathcal{M}$		
$(pk, sk) \leftarrow \mathsf{Gen}()$		
$c \leftarrow \mathsf{Enc}(m, pk)$		
	$\xrightarrow{(pk, c)}$	
		$\dots$
	$\xleftarrow{m'}$	
if		
$m' \in \mathcal{M}$ *and*		
$m' = m$		
output 1		
else *output 0*		

Hence, we call a cipher E ***secure against message recovery attacks*** *if and only if every efficient adversary $\mathcal{A}$ wins the previously defined security game with probability at most* $1/|\mathcal{M}| + \varepsilon$*, with ε being negligible.*

Since in the previously defined security game, the adversary must output the whole message m correctly, this security definition is weaker than CPA security and semantic security. But nevertheless, we can ensure this security definition with deterministic asymmetric encryption schemes such that it is useful though.

We use the previously defined ciphers and encryption schemes to analyze their privacy-preservation for our scenario in Section 4.2.

2.6 Prio Definitions

For our use of Prio as advanced approach in Section 5.1, we use additional definitions that are necessary for the understanding of Prio from Corrigan-Gibbs and Boneh [CB17].

Since our scenario for Prio is different from the one described in Chapter 3, we also introduce other terms that substitute those from Section 2.2. For Prio, each client (in our scenario *employee*) $t \in \{1, 2, \dots, T\}$ owns a secret sec_t that should be sent to the servers and analyzed. All those secrets sec_t can either be binary ($\in \{0, 1\}$) or numerical in $\mathbb{N}$. A secret is *valid* if it lies in our value set $\mathcal{V}$. The servers compute an aggregate function $f(sec_1, sec_2, \dots, sec_T)$ over all (valid) secrets sec_t.

The following definitions are adopted from the paper [CB17] of Corrigan-Gibbs and Boneh where we use our notations for vectors, secret data, and string truncation and mention the inputs and outputs of the AFE's algorithms. For Prio we need more advanced math, thus we explain briefly what arithmetic circuits are:

Definition 2.17 (Arithmetic Circuits)**.** *We define an **arithmetic circuit** AC to be a mapping $\mathbb{F}^n \to \mathbb{F}$. Hence, AC takes as input a vector $\vec{v} = (v_0, v_1, \dots, v_{n-1}) \in \mathbb{F}^n$ of size $n \in \mathbb{N}$ and outputs a value in $\mathbb{F}$. Additionally, AC can be represented as directed acyclic graph with the following vertices:*

- *Each* Input *vertex represents a variable v_i from the input vector $\vec{v} \in \mathbb{F}^n$ or a single value $u \in \mathbb{F}$. Moreover, each input vertex has no incoming input edges.*
- Gate *vertices represent a mathematical symbol that describes the operation that is applied on the values from both incoming edges. Those symbols can be '$+$' for addition and '$\times$' for multiplication. One single edge forwards the result to the next vertex.*
- *The last vertex contains AC's result and is called* output *vertex. There is exactly one output vertex per arithmetic circuit and it has no outgoing edges.*

Following the directed edges constructs a term that only contains variables from $\vec{v} = (v_0, v_1, \dots, v_{n-1}) \in \mathbb{F}^n$ and $u \in \mathbb{F}$ that are summed or multiplied. The result of this term is the result of the circuit AC.

In Prio, arithmetic circuits are used to ensure the validity of the clients' inputs. How this works in particular, is explained in Section 5.1.1.

Next, we explain affine-aggregatable encodings. Those are used to enable Prio to perform different aggregation functions f. Therefore, AFEs increase the set of possible functions for Prio.

Definition 2.18 (Affine-Aggregatable Encodings)**.** *An **affine-aggregatable encoding (AFE)** is defined as a triple of efficient algorithms* $(\mathsf{Encode}, \mathsf{Valid}, \mathsf{Decode})$*. It computes the result of an aggregation function f on input secrets $(sec_1, sec_2, \dots, sec_T)$ from T clients. Moreover, each AFE has the parameters $\kappa, \kappa' \in \mathbb{Z}$ with $\kappa' \leq \kappa$. The three algorithms of an AFE make use of a field $\mathbb{F}$ and are defined as follows:*

- Encode *takes as input a secret sec_t and outputs an encoding $e_t \in \mathbb{F}^\kappa$ from client $t \in \{1, 2, \dots, T\}$ of this secret.*
- Valid *takes as input an encoding $e_t \in \mathbb{F}^\kappa$ and outputs 0 or 1 depending on e_t.*
- Decode *takes as input all encodings e_t and outputs the result of the aggregation function f.*

*For the **correctness** of an AFE that computes an aggregation function f, we require that for all combinations of secrets $(sec_1, sec_2, \dots, sec_T)$:*

$$\mathsf{Decode}(\sum_{t=1}^{T} \mathsf{Encode}(sec_t)[0..\kappa']) = f(sec_1, sec_2, \dots, sec_T).$$

*Moreover, an AFE is **sound**, if it holds that*

$$\forall e \in \mathbb{F}^\kappa : \mathsf{Valid}(e) = 1 \textit{ if and only if } \exists sec : \mathsf{Encode}(sec) = e.$$

And an AFE is ***private*** *regarding a function* $\hat{f}$ *if there exists an efficient simulator* Sim *that for all possible combinations of secrets* $sec_1, sec_2, \ldots, sec_T$ *simulates such that the distributions of* $\mathsf{Sim}(\hat{f}(sec_1, sec_2, \ldots, sec_T))$ *and* $\sum_{t=1}^{T} \mathsf{Encode}(sec_t)[0..\kappa']$ *are indistinguishable.*

Intuitively, this means that an efficient simulator Sim takes as input $\hat{f}(sec_1, sec_2, \ldots, sec_T)$ and simulates the inputs of Encode in a way that the input of Decode is indistinguishable from $\sum_{t=1}^{T} \mathsf{Encode}(sec_t)[0..\kappa']$. In particular, it picks random $sec_t, \forall t \in \{1, 2, \ldots, T\}$, computes their encoding by simulating Encode and outputs a sum of truncated encodings. This sum is indistinguishable from the input of Decode that is produced by an arbitrary adversary that runs the AFE. According to [CB17], the function $\hat{f}$ reveals few more information than the aggregation function f. This additional leakage is quantified with $L(sec_1, sec_2, \ldots, sec_T)$ and depends on the executed AFE.

Given these security and privacy properties, an AFE does improve the functionality and privacy of Prio. We discuss this in more detail in Section 5.1. In the following chapter, we define our scenario for which we want to analyze the approaches for our privacy-preserving log file collection and evaluation system.

3 Scenario

For this thesis, we imagine the following scenario. In a company, there are many employees that use their devices e.g., computers and tablets (also called clients in the following) to communicate with websites and services over the internet. Each of the employees' requests is sent to a proxy server that forwards the requests to the internet and caches the responses in a web cache for later use[1]. Additionally, the proxy server filters known malicious websites to prevent employees from accidentally downloading malware from there. For this purpose, it can be helpful to learn about new malicious websites to add them to the proxy server's blacklist.

In the following, we consider four types of actors. First, the efficient *adversaries* that were defined in Definition 2.4. Those actors try to get access to the stored log files and try to learn the sensitive data. Then, there are the *employees* that, in the worst case, have legal access to the log files. Hence, if we do not protect or hide the sensitive data in the log files, the adversaries and employees are able to read them all. The other two actors are *admins*[2] (special privileged employees) and *data analysts* from third parties, which analyze the data that is produced by all employees[3]. Thus, both admins and data analysts have special knowledge and privileges i.e., reading permission on the log files and secret keys which they could misuse to injure the employees' privacy.

3.1 Storage of Log Files and their Content

Additionally, each of the employees' requests is forwarded to the log file server. This server takes the requests and parts of the corresponding responses. Then it stores the necessary data in the log files, which are realized with our *log file database*. In the following, we assume that the log file server is run by the company and thus trustworthy. Otherwise, since it receives the requests and thus the employees (sensitive) data, it would be a great risk for privacy. Each of the produced log file entries is stored in the `Log_Files` table. And each log entry contains the column `user_id`, which identifies the submitting employee, and a column with `device_id`, which specifies the used device. The `device_id` depends on the device that was used to send a request, while the `user_id` depends on which employee is currently logged in on this device. This distinction seems useful because it is important to find an employee that accessed a malicious website. For this, we need the `user_id` because devices are not necessarily fixed to a single employee.

[1] `https://www.avg.com/en/signal/proxy-server-definition`

[2] We use the term admin to identify the employees that use their privileges and knowledge to analyze the stored data. Those actors are not related to the typical understanding of admins.

[3] As admins work for the company as well, they also produce (sensitive) data that must be protected.

If we only used `device_id`, a worst-case scenario would be that a malicious employee downloaded malware before being fired. The malware is first discovered weeks later when a new employee is already using the infected device. The log entries would allow the conclusion that this new employee is guilty because he uses the infected device, but that would be wrong. Nevertheless, the `device_id` is also necessary because when malicious software is found on a specific device, the task is to find its source. Due to the stored `device_id` we must only analyze a subset of all log entries that have to do with this device. As both IDs are not used otherwise than in the log files, those are only known to admins and data analysts since those can access `Users` and `Devices`. Additionally, the log files we analyze contain the `client_mac_address` and the `client_ip_address`, which store the MAC and IP address of the sending device. Those are useful to filter all log entries for the infected device, too. Furthermore, for any company that does not store a `user_id` nor a `device_id`, they can use `client_mac_address` and `client_ip_address` instead to find connections between devices and websites. We also collect data about the accessed website's URL and IP address in `target_url` and `target_ip`. For better readability, we prefer `target_url`, but since a website can have different URLs, we also store the corresponding IP address in `target_ip` for clarity. For downloads, we also store the number of downloaded bytes in `session_bytes`. If there was no download, the value of this column is zero. Furthermore, we use a column to store the `status_code` of the response. This is important because malware could be downloaded after the client is redirected to another, malicious website. Otherwise, we would identify a non-malicious website as malicious and thus accidentally blacklist it. Last, each entry contains a `datestamp` and `timestamp`. This allows us to reduce the set of possibly interesting log file entries to a subset of entries from the affected period. As usual, each entry has a `log_id` that makes it unique inside the `Log_Files` table and corresponds to the *row* of the entry. It holds that $\texttt{entry}_{row}[log\_id] = row$. In summary, an entry of the `Log_Files` table consists of columns

$$\mathcal{CO} := \{\texttt{log\_id}, \texttt{user\_id}, \texttt{device\_id}, \texttt{client\_ip\_address}, \texttt{client\_mac\_address}, \texttt{target\_url}, \texttt{target\_ip}, \texttt{session\_bytes}, \texttt{status\_code}, \texttt{datestamp}, \texttt{timestamp}\}^{4}.$$

The database also stores two other tables. The first one is `Users`, which maps `user_id` to the corresponding employee's full name. The other one is `Devices`, which is a mapping from `device_id` to e.g., the number of the workplace of a computer or a tablet's name. Since, these two tables store very sensitive data, we assume that both are encrypted and can only be accessed and decrypted by admins. Hence, we assume that it is not possible to get access to these tables. Though this data must be stored such that there is the possibility to learn e.g., the names of malicious employees. Although all previously introduced columns seem very important, some of them could be redundant in practice. Thus, we briefly recommend in Chapter 6 how the amount of stored data can be kept to a minimum.

For a better understanding of our assumed scenario, Figure 3.1 shows all servers, actors, and their relationships. We decided that the proxy server and the log file server

[4] Thus, $|\mathcal{CO}| = 11$.

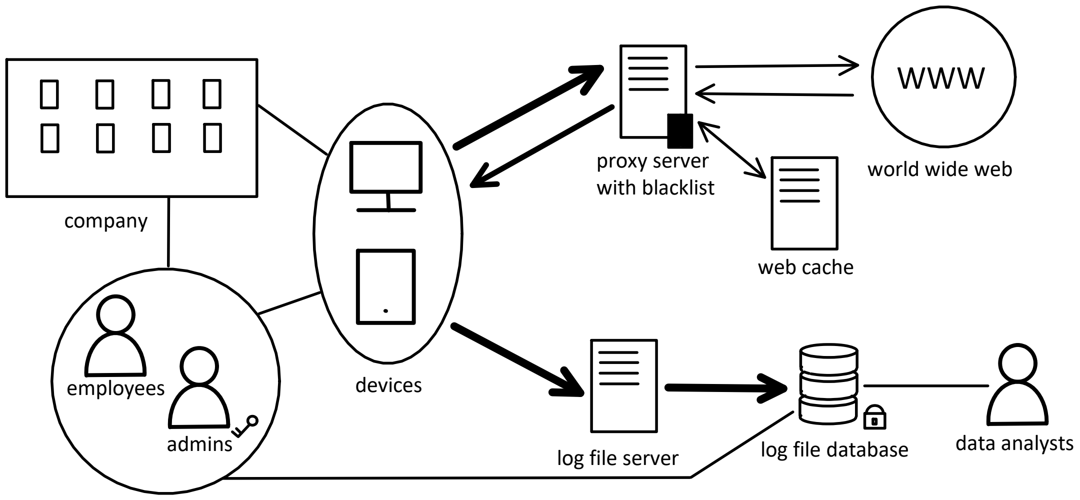


Figure 3.1: Diagram of our assumed scenario

should be displayed as two different entities since they represent different functionalities. Thus, we can easily substitute the log file server depending on what is necessary for our analyzed approaches while keeping the functionality of the proxy server. However, in practice, the tasks of the log file server and the proxy server could also be done by a single server to save resources.

3.2 Evaluation Functions

All these collected data can then be analyzed for the following purposes. These are mainly discovering new malicious websites by finding the sources of detected malware and if possible learning who downloaded it. After the malicious website is found, it could be interesting to reveal how many devices are also affected by the malware. The data can be analyzed by the company's admins or this task can be given to the data analysts from a third party.

For our analysis, we define the following evaluation functions that can be executed to get useful information from the log files. Those can then be used for the previously described purposes. In the following, we use three different notations for the columns: `user_id` refers to the column of the database, *user_id* denotes a variable containing a value from the column and user_id is used in texts.

1. Max_Download_Size(), Avg_Download_Size(): At first, we would like to know the maximum and the average number of downloaded session_bytes. A very large download or download sizes that strongly derivate from the average can be indicators for downloaded malware.
   Input: `session_bytes`, Output: *result* $\in \mathbb{R}$ (float)

2. Number_Requests(): Second, we want to evaluate the number of requests per device_id (in a given period e.g., day or month). A device with significantly many requests could be infected with malware and trying to perform a denial-of-service attack. So, with this function, we have an indicator for malware on a device.
Input: *device_id*, Output: $\mathit{result} \in \mathbb{N}$ (integer)

3. Most_Visited_Websites(): We also have an evaluation function that outputs the website that has been visited most often and the number of accesses to this page. These results can be a good indicator to find a website which is attacked with a denial-of-service attack by one of the devices. Hence, we can use it to find an infected device, too[5].
Input: -, Output: *target_ip*, *target_url*, $\mathit{result} \in \mathbb{N}$ (integer)

4. Malicious_Source(): Next, we want to find a target_ip or target_url that corresponds to a specific number of downloaded session_bytes. This can be used to find the website that is the source of detected malware of size *session_bytes*. Each log entry that is found by this function is marked as *suspicious* by adding its log_id to a set of IDs of suspicious log entries. We can add the website to the proxy server's blacklist, after we ensured that it is indeed a source of malware.
Input: *session_bytes*, Output: *target_ip*, *target_url*

5. Infected_Devices(): Further, we are interested in the number of devices (device_ids) that accessed a given website specified by target_ip or target_url. Thus, we can detect which of all devices can also be infected with that malware because they accessed the same website.
Input: *target_ip*, *target_url*, Output: $\mathit{result} \in \mathbb{N}$ (integer)

6. On_Purpose(): This evaluation function should detect whether an employee accessed a malicious website on purpose or not. For this, it takes the log_id of a given log entry and checks whether the status_code is a redirection or not. If it is a redirection, the output should be '0' which stands for *not on purpose* and '1' otherwise. If the access was a redirection, the log entry is no longer marked as suspicious and we protect that entry in the following investigations.
Input: *log_id*, Output: $\mathit{result} \in \{0, 1\}$

7. Malicious_Issuer(): Additionally, we want an evaluation function that outputs the user_ids of employees that accessed a given target_ip or target_url with a thus infected device. Thus, we can find the employee that downloaded the malware and start further investigations on this.[6] Since we output a user_id, which is only a

[5]Furthermore, this evaluation function could be used to detect websites that are essential for the daily work in the company. Thus, the websites we find with it should never be blacklisted to avoid disrupting the processes of the company. But this functionality must be kept secret, because otherwise it could be misused to avoid malicious websites from being blacklisted by querying them very often.
[6]This function can be used for arbitrary websites. Hence, it depends on its users e.g., admins and data analysts to only search for malicious websites and to not use this function for tracking the employees' behavior.

pseudonym and therefore allows conclusion on who really owns the data in this log entry, we need to lock the usage of this evaluation function. For this, a password is required as additional input and it is only known to the admins. Only if this password is correct, the computation starts. Otherwise, the function should wait a random time until it outputs null[7]. Hence, only admins and third party data analysts are able to execute this evaluation function.
Input: *password*, *device_id*, *target_ip*, *target_url*, Output: *user_id*

8. Period_Preselection(): At last, we also introduce a function that should improve the performance of all other functions. For this, it takes datestamp and timestamp and only outputs log entries of the therefore specified period.
Input: *datestamp*, *timestamp*, Output: list of log entries entry_{row}

All evaluation functions should work together. The functions Avg_Download_Size(), Max_Download_Size(), Number_Requests(), and Most_Visited_Websites() should be at help to find infected devices inside the company's system. Then, Malicious_Source() can be used to find the source of detected malware on a device. All log entries that were found by it are suspicious since they could be the reason for the infected device. Moreover, the function Infected_Devices() must be started to find all devices that could be infected with the found malware, too. Next, we discover whether the malware was downloaded on purpose with On_Purpose(). And last but not least, the function Malicious_Issuer() is useful to find the employee that accessed the website with the malware and hence infected his device. In fact, On_Purpose() and Malicious_Issuer() are only used on suspicious log entries. The evaluation function Period_Preselection() can be used to preselect the log entries that are given to any of the other evaluation functions. For this, each of the other evaluation functions takes *datestamp* and *timestamp* as optional input[8]. If they are provided, each function first executes Period_Preselection() on these inputs and then computes on the remaining list of log entries. Hence, we can improve the performance of each evaluation function, by providing a period that should be investigated. With all these functions, we are able to construct a log file collection and evaluation system that can be used for the previously named purposes.

3.3 Privacy Risk Assessment

Since preserving privacy is one of our main goals, we need to think about which columns must not be revealed to the public to ensure the employees' privacy. Then, for all sensitive data, it holds that: If the log entry was completely analyzed in its hidden form i.e., using all evaluation functions except Malicious_Issuer() and is still classified as suspicious, we allow revealing sensitive data. This is necessary because if we completely forbid revealing the identity of an issuer, it would for example, be impossible to find a malicious employee that downloaded malware on purpose or to find an adversary in the system. Whether this *opening* is necessary and how this should be applied then, depends

[7]The random waiting should complicate password guessing attacks.
[8]This is not denoted in the previous listing of inputs to avoid redundancy.

on the approach we use and is discussed in the corresponding section. As a side note, the value log_id is only stored for administrative purposes and should thus not contain any sensitive information. This is why we ignore it when we evaluate the privacy of our approaches.

Firstly, the most important value to hide is user_id because this has a direct mapping to the name of the employee that caused this log entry. So, if we cannot hide the user_id from unauthorized reading access[9], we can never ensure anonymity. One could claim, that the values in `user_id` are indeed a pseudonym and thus not a threat to the anonymity since it can be chosen independently from the employees' names. But pseudonymity only fulfills its purpose as long as the connection between true name and fixed pseudonym is not revealed. Hence, we do not want to rely on pseudonymity and hide the values in `user_id` whenever possible. Moreover, we want to hide the values of the `device_id` column from being read. Those values directly map to the number of a working place or a specific device. We assume, that these numbers can be seen by every employee because they are e.g., the devices' names. Thus, if the company has a fixed seating arrangement or if the employees use the same device every day, this would enable finding the connection between a device_id and the name of the employee that uses it given access to the `Devices` table. The same as already discussed for `user_id` and `device_id` also holds for the values of `client_ip_address` and `client_mac_address`. Hence, we also hide them from any unauthorized access. The values target_ip and target_url are independent from the employees, since every employee is allowed to open every URL. Thus, `target_ip` and `target_url` cannot reveal sensitive data of the employees and it is thus possible to not hide those columns. This also comes in handy when we want to find out where malicious software of a known byte size has been downloaded. Moreover, for the values of the `session_bytes` column is also no need to hide them. As before, the data is independent from the employees, since every employee can download every file of a specific byte size. Further, this decision is very useful since we can then compute maximums and averages over a true value, which would not be possible if every byte size was mapped to a random value. Next, there is the risk that is caused by the value in `status_code`. Again, it holds, that this is independent from its issuer and hence we do not have to hide it. Finally, the datestamp and timestamp values of a log file can reveal information about the employees that caused them if they are out of the norm. For example, if there is only one employee that stays in the office until 10 pm. Therefore, it is clear that all log entries with this timestamp belong to this specific employee. But we assume for this work, that the company has fixed days per week and fixed time slots at which the employees are working there. Thus, the company is closed during out-of-norm periods such that no log entries can be caused. Moreover, it is important to not hide the values in `datestamp` and `timestamp` because if we mapped them to randomly chosen values, we could not find the entries of a specified period. Hence, we do not try to hide the values in `datestamp` and `timestamp`.

[9]Unauthorized in this sense is everyone that (1) has no permission to analyze the log entries, meaning someone not working as a data analyst or admin for the company or (2) someone that tries to read log entries that were not classified as suspicious.

Thus, the columns `user_id`, `device_id`, `client_ip_address`, and `client_mac_address` are considered to be sensitive. With this knowledge of sensitive and less sensitive values in our log entries, we can better judge whether privacy is preserved or not. Further, we can decide, which values in the entries must be hidden to preserve privacy and which must stay unchanged to ensure the functionality of the evaluation functions.

3.4 Privacy Goals

For the discussion of whether our approaches in Chapter 4 and Chapter 5 preserve privacy, we define the following privacy goals that should be ensured by each approach.

1. *Unreadability*: Sensitive data as defined in Section 3.3 is hidden, such that it is hard to reveal the data by e.g., decrypting it.

2. *Anonymity*: For everyone with access to the log files, it should be hard to find the issuer of any log entry.

3. *Unlinkability*: It is hard to decide whether two or more log entries are related. For example, it is hard to decide whether two different log entries were caused by the same employee.

4. *Presumption of Innocence*: Our evaluation functions should not output sensitive data of innocent employees. Thus, if we need to reveal sensitive data, then only from suspicious log entries.

5. *Admin-Privacy*: Our approaches target to hide the sensitive data from all employees. But the admins, as special employees, also have special privileges. Therefore, they can leak sensitive data to enable the evaluation of our functions. Hence, we use this privacy goal to denote whether we can protect the employees' privacy from their admins and all actors that gain access to similar privileges.

6. *Ephemerality*: The data of the request (i.e., in the log entries), but especially the sensitive data, is not permanently stored by the admins. Hence, they have no permanent access to those (sensitive) data.

7. *Validity*: Given that each column $col \in \mathcal{CO}$ has a value set $\mathcal{V}_{col}$ that contains all valid values for column *col*. We want to ensure that no adversary or employee is able to send invalid requests to influence the results of the evaluation functions. Otherwise, those invalid requests could harm other (innocent) employees.

The goal of *Pseudonymity* is already fulfilled, since we never store the employees' names in our database `Log_Files` and only pseudonyms (user_ids) instead. Moreover, the privacy goals *Unreadability*, *Anonymity*, and *Unlinkability* should hold for all non-privileged actors, namely for all normal employees and all adversaries that gained access to the system. For the privileged admins and data analysts, we use *Admin-Privacy* to denote whether their special knowledge and permissions allow injuring one of our goals *Unreadability*, *Anonymity*, or *Unlinkability*.

Table 3.1: Example of Functionality Grading

Evaluation Functions	Approach
Avg_Download_Size()	×
Max_Download_Size()	×
Number_Requests()	✓
Most_Visited_Websites()	×
Malicious_Source()	∼
Infected_Devices()	×
On_Purpose()	×
Malicious_Issuer()	✓
Period_Preselection()	×
Result	2.5 P.

Table 3.2: Example of Privacy Grading

Privacy Goals	Approach
Unreadability	×
Anonymity	×
Unlinkability	×
Presumption of Innocence	✓
Admin-Privacy	×
Ephemerality	×
Validity	×
Result	1 P.

3.5 Grading

For the comparison of the different approaches, we introduce a grading system. Each approach is represented as a column in a table that displays the overall grading with regard to all evaluation functions (see Section 3.2). An example of such a table is Table 3.1. For each evaluation function that can be realized using the analyzed approach, this is denoted with a '✓'. If an evaluation function cannot be realized, this is denoted with '×'. For approaches that only produce an approximately correct solution, we denote this with '∼'.

To also introduce some kind of ranking between the approaches, each '✓' is worth 1 P., each '∼' is worth 0.5 P. and the '×' is worthless. Then, we can compare the results of our analysis based on larger or smaller results. But thanks to the symbols in the table, we still have the information, which approach fulfills which requirements and we get a good overview of the results. In Section 3.2 we described that most of the evaluation functions depend on each other. This means, that if we cannot evaluate Malicious_Source() we have no input for the functions Infected_Devices() and Malicious_Issuer(). Nevertheless, we analyze each evaluation function individually to decide whether it could be computed if we had the necessary input.

Since we want to discuss the trade-off of privacy and functionality, we also introduce a similar grading system for privacy using the previously defined privacy goals (Section 3.4). As we did for the Functional Grading, we use the symbols '✓' for ensured privacy goals and '×' for injured ones. If we can ensure a privacy goal only for under some conditions, we denote this with '∼'. Further, we also introduce a mapping from the symbols to numerical values for better comparability: The '✓' is worth 1 P. and '∼' is worth 0.5 P., while '×' is again worthless.

The two grading tables that sum those grading results are designed as Tables 3.1 and 3.2.

4 Basic Approaches

For the previously described scenario, we could provide that all evaluation functions from Section 3.2 can be computed, if we just stored all data in the clear. But this would mean that the collection of data we achieve is not privacy-preserving at all. In particular, we would not fulfill a single privacy goal (Section 3.4) since all data, including the sensitive data, could be read by every employee of the company.

Thus, in this chapter, we discuss the basic approaches with hashing and encryption that we use for our privacy-preserving log file collection and evaluation system. For each of the basic approaches, we discuss and grade whether our defined evaluation functions from Section 3.2 can be realized. Further, we grade each approach's preservation of the employees' privacy regarding our privacy goals from Section 3.4.

4.1 Hashing

At first, we want to analyze the basic approaches only depending on hash functions as defined in Section 2.4. For this, we first discuss general aspects that are important for the choice of the used hash function.

Design Decisions

In Section 2.4 we mention the security definitions that we find important for hashing log file entries. The first of them is one-wayness. If our used hash function H is one-way, it holds that no employee, admin, or adversary can compute any preimage of a given digest. This also holds for the original preimage which was the original input to the hash function H. Therefore, we use a hash function H that is one-way as this ensures that no actor can compute the preimages of the hashed log entries.

Moreover, when we use hash functions, we have to deal with the risks of hash collisions. Those can influence the outcome of our evaluation functions and lead to false results. For example, if two different values from `target_ip` were hashed to the same digest, the output of Most_Visited_Websites() could be influenced. If both websites together are visited more often than any other website, the function outputs those websites as most visited, though only their summed accesses is the maximum. Hence, we use a collision resistant function H to decrease the possibility of accidental hash collisions in the following basic approaches (Chapter 4). Additionally, it becomes hard for adversaries to create colliding log entries to affect our evaluation functions on purpose.

Before we can start applying a hash function to our log entries, we must first decide which hash functions can be used. To not lose generality, we do not choose a specific hash function, but which kind of hash function is useful in our scenario. For this, we previously defined keyless, salted, and keyed hash functions in Chapter 2. If we used a keyless hash function, then adversaries could use rainbow tables to compute the hashes' preimages e.g., for MD5 [KKJ+13]. Hence, we could not preserve the privacy, especially the *Unreadability*, of the sensitive data in our log files. To avoid those *rainbow table attacks*, the hashes can be salted. Then, in addition to the data in the log entry, a random $salt \leftarrow \mathcal{S}$ is input to hash function H. The difference between salting a hash function and its related approach called peppering is that the salt is stored along with the hashed data. Hence, every person that has access to the log files also has access to the salt, which is thus not secret. As a result, the employees and adversaries that gained access to the database can therefore look up the salt and precompute a rainbow table using this. Then, we have the same problem as with keyless hash functions. Therefore, for hashing passwords, it is recommended to use different salts for each user. If we apply this recommendation to our scenario, we must use a new salt for every user_id since this value differs for each of the employees. But if we do so, we cannot recognize equal entries from different employees anymore. For example, a website that has been visited by two different employees would be hashed to different digests due to the different salts. Then we lose a lot of information and it would be hard to evaluate our functions from Section 3.2. Therefore, we need to use keyed hash functions, where the key k is picked uniformly at random from a huge set of keys $\mathcal{K}$. Then, it is very unlikely that any adversary already precomputed a rainbow table for the specific key we choose[1]. But, we cannot use different keys for different employees because this would lead to the same problem as if we used one salt per employee. Then, there would be different hash functions for each employee and thus identical values from different employees would be hashed to different digests with high probability. Hence, we could not compare digests from different employees in `Log_Files`.

The hash function H we picked in the previous considerations is applied to each cell of our database `Log_Files`[2] separately. This means, that we compute $H(\mathsf{entry}_{row}[col], k)$ for all rows $row \in \{1, 2, \ldots, R\}$, a subset of all columns $col \in \mathcal{CO}^* \subseteq \mathcal{CO}$[3], and our key $k \leftarrow \mathcal{K}$. Otherwise, if we hashed each column $col \in \mathcal{CO}$ of the table completely, we could not argue on any data that was presented in a specific log entry. And if we used a complete log entry entry_{row} (namely a row of the table) as input to the hash function, we could not argue about values of different columns $\mathsf{entry}[col]$. For example, we could not tell whether a specific log entry is from the same day as any other log entry. Therefore, we lose the least amount of information by hashing the database table cell by cell. In the following, we distinguish between two different kinds of hashing approaches. The

[1]The choice of keyed hash functions is useful for theoretic considerations, such as proving the security of a hash function. Nevertheless, in practice, keyless hash functions with salt are often used instead.

[2]In this work, we only focus on how to protect the `Log_Files` database. The other databases `Users` and `Devices` were only introduced for administrative purposes, since those must exist somewhere in practice. Though the following findings also apply to those databases, they are not part of our analysis.

[3]In fact, we should not hash the column `log_id` at all or use a different hash function than for all other columns. This column would contain digests of all numerical values from 1 to R sorted in ascending order. Thus, for a large amount of numerical inputs, the digests are already precomputed, which simplifies learning the mapping of numerical values and their digests.

first one named Complete Hashing hashes the cells from all columns, specifically from columns in $\mathcal{CO}^* = \mathcal{CO}$. Whereas the second approach called Partial Hashing only hashes cells from some of our columns, namely columns from $\mathcal{CO}^{*}$[4] $\subset \mathcal{CO}$. Cells in the columns $\mathcal{CO} \backslash \mathcal{CO}^*$ thus stay preimages, i.e., are not hashed.

Since for keyed hashing the key k is considered to be public as it defines the used hash function, we cannot hide it from any of the actors. Hence, if we want to avoid the employees from starting the evaluation functions on their own, the functions must take a password as additional input. This password should be known only to admins or third party data analysts. In the next sections, our two hashing approaches are described in detail, analyzed, and graded.

4.1.1 Complete Hashing

For our first approach with hash functions, we hash each cell of each column in $\mathcal{CO}$ in our log file database. This provides a lot of privacy since it is hard for every actor to make sense of the digests stored in our log file database. Thus, we fulfill a lot of our privacy goals with our approach of Complete Hashing. On the other hand, it is hard to do any computations on this hashed data. Therefore, we only have a very small subset of evaluation functions (Section 3.2) that can be computed. To show these statements, we explain how Complete Hashing is applied to our scenario in the next section. After that, we analyze and grade the functionality and privacy of this approach according to the grading system that was introduced in Section 3.5 and we discuss the results afterward.

Application

Since the application of our hash function is already discussed in Section 4.1 we only summarize the results briefly. We use a keyed hash function that is collision resistant and one-way. Moreover, we hash each cell $\text{entry}_{row}[col], \forall row \in \{1, 2, \ldots, R\}$, $col \in \mathcal{CO}^* = \mathcal{CO}$ separately. Hence, our Complete Hashing is applied as follows: Given a keyed hash function H and a key $k \leftarrow \mathcal{K}$ that is chosen uniformly at random: For all $row \in \{1, 2, \ldots, R\}$ with $R \in \mathbb{N}$ being the number of log file entries and for all columns $col \in \mathcal{CO}$ we compute $H(\text{entry}_{row}[col], k)$ to individually hash each cell of our log files.

Functionality Grading

Since the application of Complete Hashing is defined in the previous section, we can grade which of our evaluation functions can be evaluated using this application. In particular, we show in this section that we can only evaluate Number_Requests() and Most_Visited_Websites() only partly. These results can be seen in Table 4.1.

Avg_Download_Size() For the evaluation function Avg_Download_Size() we need the values from the `session_bytes` column to compute the average of all downloaded

[4]The complete definition of $\mathcal{CO}^*$ can be found in the *Application* section of Partial Hashing in Section 4.1.2

bytes (from a specific period). But all values are hashed in the `session_bytes` column and this can hardly be reversed by any actor since the hash function H is one-way. Therefore, admins and data analysts cannot do any useful computation with this data as input.

Max_Download_Size() The functions Avg_Download_Size() and Max_Download_Size() take the same input. Thus, we can conclude that there exists the same problem for Max_Download_Size() as for Avg_Download_Size(): Due to the hashed and not reversible values in the `session_bytes` column, admins and data analysts cannot compute the maximum of all downloaded bytes (from a specific period).

Number_Requests() This evaluation function takes as input a *device_id* and outputs the number of submitted requests per device (in a given period). To do so, the evaluation function first hashes its input, specifically it computes $H(device\_id, k)$. Then, this function compares the digests in the `device_id` column with its hashed input. We can ensure that each pair of identical digests corresponds to a pair of identical preimages, since H is deterministic and also collision resistant. For each such match, this function increases its *result* by one and outputs it at the end. Therefore, it exists a possibility to compute Number_Requests() with our Complete Hashing approach[5].

Most_Visited_Websites() This evaluation function takes no input and should output the website with the most accesses (in a given period). For this, the function first groups the log entries by their values in `target_ip` (or `target_url`) and computes the size of each group. Since H is collision resistant, there are as many groups as distinct values in `target_ip`. Hence, we can compute the number of accesses to each website only using the digests instead of their preimages. Then, the evaluation function outputs the size of the largest group. But it cannot output useful information about the most visited website's URL or IP address since those are hashed and due to H being one-way their preimages cannot easily be reconstructed. Therefore, Most_Visited_Websites() can only output a part of the necessary data and hence only be realized partly.

Malicious_Source() For computing, Malicious_Source() takes *device_id* and *session_bytes* as input. Given those, it should output websites that could be the source of a given malicious download with size *session_bytes*. This function could compute digests of its inputs to find the relevant log entries and output their *target_ip*s or *target_url*s. But since our output values in `target_ip` and `target_url` are hashed, the resulting values are only digests, which are unreadable. Thus, and because we cannot compute any preimages as H is one-way, this data is not useful. As a result, the Malicious_Source() evaluation function cannot be realized with Complete Hashing.

[5]This construction is linear in the size of rows R of the log file.

Table 4.1: Functionality Grading for Complete Hashing

Evaluation Functions	Complete Hashing	Comments
Avg_Download_Size()	×	unreadable input data
Max_Download_Size()	×	unreadable input data
Number_Requests()	✓	-
Most_Visited_Websites()	∼	partly unreadable output data
Malicious_Source()	×	unreadable output data
Infected_Devices()	×	unreadable output data
On_Purpose()	×	unreadable input data
Malicious_Issuer()	×	unreadable output data
Period_Preselection()	×	unreadable input data
Result	1.5 P.	

Infected_Devices() Also for Infected_Devices() we cannot do any computations due to the previously described problem of Malicious_Source(). This means, that the input, which is a target_ip or target_url could be hashed to find the relevant log entries. But the values we are looking for (in this case values from `device_id`) are hashed. Thus, they are unreadable strings and since H is one-way their preimages can also not be regained. Therefore, Infected_Devices() cannot be computed when we use Complete Hashing as our approach.

On_Purpose() To compute this function, we need to regard the values in `status_code` but those are also hashed. Since H is one-way it is hard to find the preimages of the cells' content. Therefore, we cannot make any sense from the data in `status_code` and thus not compute this evaluation function.

Malicious_Issuer() For this function, we have a similar problem as for Malicious_Source(). It takes as input a *target_ip* or *target_url* and *device_id* and should output values from the `user_id` column. To find all necessary log entries, it could hash its inputs and compare them to the digests in the `device_id` and `target_ip` respectively `target_url` columns to find the interesting log entries. But given the resulting log entries, their values in `user_id` are useless since those are hashed. This is a problem since we cannot compute their preimages because H is one-way and the digests themselves reveal no useful information about their preimages. Thus, we cannot compute Malicious_Issuer() with Complete Hashing.

Period_Preselection() We cannot compute our Period_Preselection() evaluation function since the input, namely the values from `datestamp` and `timestamp`, is hashed. Hence, admins and data analysts cannot figure out, which log entries belong to a specified period and which do not. Therefore, this evaluation function cannot be realized using Complete Hashing.

Privacy Grading

Next, we grade which of our privacy goals from Section 3.4 can be ensured with our approach Complete Hashing. For this, we use the grading system that was introduced in Section 3.5. Further, we regard efficient adversaries $\mathcal{A}$ that can gain access to the company's system and therefore access the `Log_Files` database. But those adversaries should not be able to read the requests' data which is sent between the internet, employees, log file server and database. As we also assume that every employee can access this database (but not `Users` and `Devices`) both actors are considered in privacy goals *Unreadability*, *Anonymity*, and *Unlinkability.* Additionally, the admins and data analysts have the same privileges as employees since they are also allowed to access the `Log_Files` database. But they do not own special knowledge because the key k of the hash function H is public. We show that this approach of Complete Hashing allows us to ensure three of our privacy goals from Section 3.3. Those are *Unreadability*, *Anonymity*, and *Presumption of Innocence*, which is denoted in Table 4.2. The privacy goals *Admin-Privacy* and *Validity* can only be ensured under conditions we further explain in the following section.

Unreadability We could not ensure this privacy goal, if sensitive data could be read by any employee or adversary with access to the `Log_Files` database. Since, we use a hash function to hide the table's content, every of these actors has to compute the digests' preimages to make sense of it. But because our hash function H is one-way, computing these preimages is hard for every actor. Thus, `Log_Files`' content cannot easily be retrieved and read. Therefore, we can ensure *Unreadability* with high probability[6].

Anonymity We could not ensure this privacy goal, if employees or adversaries could find the issuer of a log entry given the data in `Log_Files`. Regarding the considerations from Section 3.3 the columns `user_id`, `device_id`, `client_ip_address`, and `client_mac_address` reveal information about which employee caused a given log entry. But, as already described, Complete Hashing ensures *Unreadability* for all columns. This means, that none of the regarded actors can learn the sensitive data which corresponds to exactly those four named columns. Since, employees and the regarded adversaries cannot learn the values in those sensitive columns, they can also not link the digests to any employee. But it would be feasible for e.g., a small set of user_ids to compute the corresponding digests to learn the mapping of preimages and digests. This could then be used to identify each log entry's value in `user_id`. Nevertheless, all *user_id*s are kept secret, `Users` stores the mapping to the names, and neither employees nor adversaries can access those database tables. Hence, the identity of the issuer could though not be revealed in this scenario. Thus, we can ensure *Anonymity* for Complete Hashing.

[6]The probability that this privacy goal can be injured depends on the used hash function H. It furthermore is the probability that an efficient adversary wins the security game for one-wayness (Definition 2.9) against H. As H is one-way, this probability is negligible.

Unlinkability The hash function H is collision resistant and collisions are thus unlikely. Hence, we can compare digests to detect whether two different log files are e.g., submitted by the same employee. This property is used to evaluate the log files. But this also means, that every employee or adversary is able to detect whether two distinct log entries contain equal digests. Hence, they learn whether two entries are linked to each other. Therefore, we cannot ensure *Unlinkability* when using Complete Hashing.

Presumption of Innocence We could not ensure *Presumption of Innocence* if we revealed sensitive data to evaluate the functions from Section 3.2. But since H is one-way the actors can hardly reveal the data since computing the digests' preimages is hard. Therefore, we can ensure *Presumption of Innocence* with Complete Hashing.

Admin-Privacy We could not ensure *Admin-Privacy* if the admins or data analysts could use their special privileges to injure *Unreadability*, *Anonymity*, or *Unlinkability*. H is one-way and thus it is hard for every actor given H and k to compute preimages of any digests. Therefore, this is also hard for the admins and data analysts since their privileged knowledge e.g., k is public in the security game of one-wayness (Definition 2.9) and they thus have no advantage compared to all other actors. Thus, we can ensure nearly the same privacy goals for admins and data analysts as for all non-privileged actors. Nevertheless, admins and data analysts know the existing *user_id*s and can compute the digests of all those. Due to this and their access to `Users` they can learn the identity of the employees that caused given log entries. Hence, if $\mathcal{V}_{user\_id}$ is small enough to allow this attack, admins and data analysts can injure *Anonymity*. As this also holds for all value sets from sensitive columns, those privileged actors can additionally injure *Unreadability*. Therefore, we only ensure *Admin-Privacy* as long as all $\mathcal{V}_{col}$ are huge.

Ephemerality We cannot ensure *Ephemerality* since all data is permanently stored in the log file database and admins and data analysts can access the database.

Validity And for the privacy goal of *Validity*, we need to ensure that invalid log entries do not influence the evaluations' results and thus harm innocent employees. Since for this approach we use the scenario as it is defined in Chapter 3, the log file server would need to check whether the values in incoming requests are valid. This means, that it must check whether session_bytes are in a range of meaningful values ($\mathcal{V}_{session\_bytes}$). But it would be hard to define such a value, since conspicuously huge values could be invalid values or originate from attacks that try to overflow the memory. Moreover, the log file server must check whether information is missing that would be needed to create the log entries. It also has to check whether the status_codes are in the range from 100 to 599[7]. But the biggest problem is to check all IP addresses and URLs to be part of the existing websites. Since there exist more than 1.88 billion websites worldwide[8] it is infeasible to check for each

[7] `https://httpwg.org/specs/rfc9110.html`

[8] `https://www.statista.com/chart/19058/number-of-websites-online/`

Table 4.2: Privacy Grading for Complete Hashing

Privacy Goals	Complete Hashing	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	$\times$	digests are comparable
Presumption of Innocence	✓	-
Admin-Privacy	$\sim$	only for huge value sets $\mathcal{V}_{col}$
Ephemerality	$\times$	permanent storage
Validity	$\sim$	only for small value sets $\mathcal{V}_{col}$
Result	4 P.	

incoming request whether it uses existing URLs and IP addresses. Hence, only most of the values could be checked by the log file server before it creates the log entries. Therefore, Complete Hashing can partly ensure *Validity.*

Result

Regarding our results from the previous analysis, Complete Hashing is not very useful for our scenario. As we can see in Table 4.1, we are only able to partly compute two of the evaluation functions. This is far from what we want to achieve. Nevertheless, we can already assure 3 of 7 privacy goals completely. Therefore, hashing the values in our log entries seems to be a good first approach, but we need more functionality in the next one.

An interesting fact it that we require large value sets $\mathcal{V}_{col}$ for *Admin-Privacy* and small sets $\mathcal{V}_{col}$ for *Validity.* Since those two requirements contradict each other, we are not able to ensure *Anonymity* and *Validity* at once.

Moreover, when we use hashing, we always have the risk of collisions appearing. For this, we use a hash function that is collision resistant. Yet, it can only ensure that collisions are unlikely, but they are possible though. For both realizable evaluation functions Number_Requests() and Most_Visited_Websites() those collisions would lead to incorrect results. Nevertheless, as H is collision resistant and collisions are very unlikely in practice, we grade the resulting risk as low and thus acceptable.

To improve this construction, we could use different hash functions (i.e., different keys k) per column. Then, it would be harder to link the stored digest to known preimages, since this must be done for each column individually. But we only profit from this, if equal data is stored in different columns. Otherwise, finding a digest of a preimage in column a would not help to learn preimages from digests in column b. Additionally, we could use ephemeral keys k. Then, for each period, a new hash function would be used. The advantage of this is that we can ensure forward-secrecy and backward-secrecy. Hence, if any adversary can reverse a hash function, he can still not learn the preimages from older or newer periods.

As Complete Hashing only enables the evaluation of two functions, we further improve our hashing approach. Thus, in the next section, we introduce our second hashing approach, which enables more functions while preserving our privacy goals from Complete Hashing.

4.1.2 Partial Hashing

Since our previous attempt to apply hashing to the stored log entries led to a huge number of not realizable evaluation functions (Section 3.2), we introduce Partial Hashing. This time, we only hash the sensitive data defined in Section 3.3. All other columns of the log file entries should be stored as preimages. We expect to not lose any privacy with this approach since we still hide the sensitive data. But due to some unhashed log entries, we can perform more computations on the stored data.

Application

As already defined in Section 4.1, we hash each cell $\mathsf{entry}_{row}[col]$ in our log files separately. Hence, we can preserve most information from the storing hashed data. Further, we also use the one-way and collision resistant keyed hash function from Section 4.1.

Then, we apply Partial Hashing as follows: Given a keyed hash function H that is one-way and collision resistant and a key $k \leftarrow \mathcal{K}$ that is chosen uniformly at random: For all $row \in \{1, 2, \ldots, R\}$ with $R \in \mathbb{N}$ being the number of log file entries and for all columns $col \in \mathcal{CO}^* = \{\texttt{user\_id}, \texttt{device\_id}, \texttt{client\_ip\_address}, \texttt{client\_mac\_address}\} \subset \mathcal{CO}$ we compute $H(\mathsf{entry}_{row}[col], k)$ to hash each cell in our log files individually.

Functionality Grading

At first, we grade which of our evaluation functions (Section 3.2) are realizable with Partial Hashing. Indeed, we can compute all our evaluation functions except the function Malicious_Issuer(). This result is also denoted in the corresponding Table 4.3. How the evaluation functions must be realized to achieve these results, is described in the following.

Avg_Download_Size(), Max_Download_Size() Since the input of these functions is not hashed, admins or data analysts can use all preimages (of a given period) from the `session_bytes` column to compute the maximum and the average of all (non-zero) values in `session_bytes`. Therefore, we can realize both functions with Partial Hashing.

Number_Requests() For the Number_Requests() evaluation function, the same implementation is possible as for the Complete Hashing approach in Section 4.1.1. To do so, the function first hashes its input *device_id*. Then, the function compares its hashed input to every digest in the `device_id` column and counts the matches. The used hash function H is collision resistant and deterministic and thus, we can assume that each digest originates from exactly one preimage. Therefore, it is allowed and useful to compare the digests instead of their preimages. Thus, this evaluation function is also realizable.

Most_Visited_Websites() This function first groups the log entries by their values in `target_ip` (or `target_url`) and computes the size of each group. After that, Most_Visited_Websites() outputs the size of the largest group and the *target_ip*s or *target_url*s from the log entries in that group. Those values are also stored as preimages such that they can be output as readable data. Hence, this function can additionally provide information about which websites were visited most often. Thus, with Partial Hashing, admins and data analysts can completely compute Most_Visited_Websites().

Malicious_Source() This evaluation function takes *session_bytes* as input and compares it to all preimages in the `session_bytes` column (in the specified period). This comparison is no problem, since the cells in `session_bytes` are not hashed. Then, it outputs every preimage *target_ip* and *target_url* that is in the matching log entries. For this, the admins and data analyst do not have to compute the preimages, which would be hard given that H is collision resistant, since they are stored as preimages. Additionally, they only have to consider a subset of log entries with the given *device_id* in the corresponding column. To find those, the function must compute $H(device\_id, k)$ and selects all log entries with the resulting digest in their `device_id` column. This only works if H is collision resistant. Otherwise it would also collect log entries from different devices, if their preimages would map to the same digest as *device_id*. Hence, the Malicious_Source() is feasible and can be evaluated using Partial Hashing.

Infected_Devices() This function compares its input, which is a *target_ip* or *target_url* to the preimages in `target_ip` and `target_url` (of the specified period). Since the used hash function H is collision resistant and deterministic, it is unlikely that different preimages map to the same digest. Moreover, for each preimage exists only one digest. Given these facts , it is useful to compare the digests instead of their preimages. In the next step, Infected_Devices() groups the matching log entries by their values in `device_id` and outputs the number of groups. Hence, admins and data analysts can learn the number of devices that are also infected with the detected malware. Thus, we conclude that Infected_Devices() is realizable using Partial Hashing.

On_Purpose() Since all values in the `status_code` column are stored as preimages, this function only needs to find the log entry with the given *log_id*, which is also not hashed. Then, it outputs whether the access to the (malicious) URL was on purpose by outputting 0 if the value in `status_code` lies between 300 and 399 and 1 else. Hence, we can compute this evaluation function with our approach Partial Hashing.

Malicious_Issuer() This evaluation function gets values from `target_url` or `target_ip` as input. At first, it decreases the amount of interesting log entries by only picking the

Table 4.3: Functionality Grading for Partial Hashing

Evaluation Functions	Partial Hashing	Comments
Avg_Download_Size()	✓	-
Max_Download_Size()	✓	-
Number_Requests()	✓	-
Most_Visited_Websites()	✓	-
Malicious_Source()	✓	-
Infected_Devices()	✓	-
On_Purpose()	✓	-
Malicious_Issuer()	×	unreadable output data
Period_Preselection()	✓	-
Result	8 P.	

entries from a device with *device_id*. As H is collision resistant and deterministic, we can assume that the comparison of digests instead of preimages is successful. Since all target_urls and target_ips are stored as preimages in the entries, Malicious_Issuer() can find all log entries that contain the access to the specified website. But as the data it should output i.e., *user_id* is hashed, admins and data analysts can still not learn the malicious issuer. This also holds since H is one-way and those actors can thus not compute the digests' preimages. Therefore, we cannot compute the Malicious_Issuer() function.

Period_Preselection() This evaluation function can be computed since the values from `datestamp` and `timestamp` are not hashed. Thus, it only needs to pick the log entries that were created in the specified period. Therefore, Period_Preselection() is realizable and can be used by admins and data analysts to preselect log entries from a given period.

Privacy Grading

In this section, we analyze the preservation of privacy of Partial Hashing with regard to our privacy goals from Section 3.4. The adversaries we consider in the following section have access to the company's system. Thus, they can read all data that is stored inside the `Log_Files` database. Further, we assume that the requests, responses and communication with the log file server and database are encrypted. Hence, those adversaries cannot read the network traffic. We learn in this section that Partial Hashing ensures the same privacy goals as Complete Hashing. Those are *Unreadability*, *Anonymity*, *Presumption of Innocence*, and partly also *Admin-Privacy* and *Validity*. Again, the results are summed in Table 4.4.

Unreadability We could not ensure *Unreadability* if the sensitive data was stored as preimages. Since we applied H on all sensitive columns $col \in \mathcal{CO}^*$ defined in Section 3.3 this is not the case. Moreover, *Unreadability* could be injured if any of the employees or adversaries with access to the log file database was able to compute the preimages of the sensitive data given their digests. Then, that data would become readable to them and this would injure the employees' privacy. Nevertheless, computing the preimages is hard since the digests were produced with a one-way hash function H. Thus, it is very unlikely that any of those actors is able to compute preimages of the stored digests. Hence, we can ensure *Unreadability* for Partial Hashing with high probability.

Anonymity The privacy goal *Anonymity* would be injured if any actor was able to link a specific log entry to the employee that caused this entry. Since the values in the columns $col \in \mathcal{CO} \setminus \mathcal{CO}^*$ are independent from their issuers (Section 3.3), those cannot be useful to link log entries to employees. All other columns $col \in \mathcal{CO}^*$ are hashed using H on input k. Thus, it is necessary to compute the preimages from those columns' data to link the entries to the corresponding employees. But again, computing those preimages is hard for all actors since H is one-way. Thus, we can prevent the employees and adversaries from injuring *Anonymity*.

Unlinkability For this privacy goal, we have nearly the same issue as when we used Complete Hashing. We enable the comparison of digests to evaluate our log files in a hidden way. This only works since the hash function H is collision resistant and deterministic. But this also allows all employees and adversaries with access to the database to compare digests and preimages to learn whether some log entries are related. Therefore, we cannot ensure *Unlinkability* with Complete Hashing.

Presumption of Innocence We could not ensure this privacy goal if we revealed sensitive data during our evaluations. Due to this, we avoid revealing sensitive data for the evaluation of the functions (Section 3.2) and analyze sensitive data only as digests. Furthermore, as H is one-way it would also be hard for admins and third party data analysts to reveal those hashed data if it should be revealed. Therefore, we can ensure this privacy goal for Partial Hashing.

Admin-Privacy If admins or data analysts had a higher probability to win the security game for one-wayness (Definition 2.9) against H, they could use their privileges to injure *Unreadability* or *Anonymity*. But the only privileged knowledge (i.e., the key k) of admins and data analysts is already considered public. Therefore, they have the same probability of winning the game as all other actors and this is negligible since H is one-way. But, if for example, the number of employees is so small, that any privileged actor could efficiently compute $H(user\_id, k)$ for all $user\_ids$, they can injure both *Unreadability* and *Anonymity*. For this, admins or data analysts compare the computed digests to the ones in `Log_Files` to learn the stored data. Since they have access to the `Users` and `Devices` tables, they can also learn the issuers of arbitrary log entries. As a result, we can only ensure *Admin-Privacy* if the value set $\mathcal{V}_{col}$ of all sensitive columns is huge.

Table 4.4: Privacy Grading for Partial Hashing

Privacy Goals	Partial Hashing	Comments
Unreadability	✓	only for sensitive data
Anonymity	✓	-
Unlinkability	×	digests are comparable
Presumption of Innocence	✓	-
Admin-Privacy	∼	only for large value sets $\mathcal{V}_{col}$
Ephemerality	×	permanent storage
Validity	∼	only for small value sets $\mathcal{V}_{col}$
Result	4 P.	

Ephemerality This specific privacy goal cannot be ensured since we permanently store the log entries in our log file database. Moreover, the admins and data analysts can access the database whenever they want.

Validity Since we use the same scenario as for Complete Hashing, and thus the same log file server, it holds the same argument as in Section 4.1.2. The log file server can check most of the values that are provided by each request before creating the corresponding log entries. But that server is not able to check IP addresses and URLs. Therefore, Partial Hashing can ensure the privacy goal *Validity* only partially.

Result

The second hashing approach Partial Hashing is more successful than Complete Hashing. We can still ensure all privacy goals, we also ensure with Complete Hashing. Additionally, we can compute nearly every of our evaluation functions if we use Partial Hashing.

As a side note: We considered some columns to be not sensitive for our analysis, such as `timestamp` and `target_url`. But if an employee works until late one day or if there is an employee that is the only one that works with a specific website, privacy can be threatened. Then, their colleagues can link their log entries to them if those entries are not hidden in any way. Hence, every preimage from a log entry could be linkable to its issuer, if the value derivates from the norm. Therefore, in practice, Partial Hashing is a risky approach. But this problem does not only exist for our specific scenario but for all situations in practice. Therefore, Partial Hashing is good given all our assumptions but bad in practice. Due to this, we next investigate an approach for which we can hide all data and none of them is stored as preimages. In particular, our next section discusses our basic approach Encryption.

4.2 Encryption

Our two previous approaches use hashing and we can already realize many useful evaluation functions at least with Partial Hashing. Furthermore, we can ensure many of our privacy goals with those hashing approaches. But one thing that cannot be circumvented is the small risk of hash collisions that could influence the results of our functions. This is a risk we are willing to take, but we though discuss another approach where those collisions are impossible. Thus, the next basic approach we analyze for our privacy-preserving log file collection and evaluation system is Encryption (Definitions 2.12 and 2.15). A correct[9] encryption scheme would never provide a pair of identical ciphertexts that would be decrypted to different plaintexts. Otherwise, it would not be clear to which plaintext a ciphertext should be decrypted and the correctness would be injured. Moreover, with Encryption, we can permanently hide the data inside the `Log_Files` database table and reveal it with decryption to allow the evaluation of our functions from Section 3.2. As long as the used cipher or encryption scheme is secure, it is hard for all efficient adversaries to learn the stored data given its ciphertexts. Thus, in contrast to Partial Hashing we can hide the data and it is though useful for our evaluations. All in all, Encryption enables a large set of realizable evaluation functions (Section 3.2) while preserving privacy as well. In the following, we describe how in particular the Encryption approach is applied to our scenario and we grade its functionality and privacy. The corresponding grading tables are Tables 4.5 to 4.7.

Application

At first, we need to discuss which encryption schemes are feasible for our application. For all possible encryption schemes, we only allow admins access to the secret/private key to decrypt the suspicious log entries for further investigations. Thus, in contrast to the previous hashing approaches, the key must be given to each evaluation function as additional input. This ensures that only admins can start the evaluation functions and none of the other employees of the company. To also allow the analysis by third party data analysts but though preserve the employees' privacy, it would be preferable to only give encrypted data to them. Then, we could ensure that admins are the most powerful actors and expect them to be trustworthy. But in practice, it is more useful if the data analysts have access to the same data as the admins, as they sometimes substitute admins or cannot analyze ciphertexts. Thus, data analysts have as well access to the decryption keys and the database, but those are given to them by the admins. Moreover, we do not choose a specific encryption scheme or cipher such that we avoid losing generality. Instead, we discuss whether symmetric or asymmetric encryption is more useful and why probabilistic encryption is not at all[10]. Then, we describe how these encryption schemes can be applied to the data in our log files.

[9]The correctness of the used encryption schemes is defined in Definitions 2.12 and 2.15.

[10]With searchable encryption, we could indeed use probabilistic encryption to hide the data because deterministic tags are computed as well. Those tags are then used to find the interesting log entries. For more information on searchable encryption, see [BBO07, ABO07]

Different Kinds of Encryption First, we need to discuss the influence of different kinds of encryption. Our log file database could use encryption at rest, such that the hard drives themselves are encrypted. But since the data is still unencrypted for every legal user of the database, this does not serve our purposes and is thus not further considered. Additionally, we do not deal with the encryption that is done for the communication from the devices to our database with e.g., TLS[11]. TLS can serve our purposes well since adversaries cannot read the data that is sent to the log file server. Nevertheless, this kind of encryption is out of scope of this thesis. We assume the worst case that not only data analysts but also employees have legal access to the database. Hence, to avoid all of them reading the sensitive data, we encrypt the log files such that only ciphertexts are stored in the database.

Symmetric or Asymmetric Encryption The first discussion is whether to prefer a symmetric or an asymmetric encryption scheme. In our scenario, the log file server is the only entity that processes and thus encrypts the data from the requests. Hence, the advantage of asymmetric encryption that allows everyone to encrypt data is not well-used here. Due to this, symmetric encryption would be more useful for our considerations. But to give more control to the employees, we could slightly change our scenario for the application of asymmetric encryption. Instead of sending their requests to the log file server, every employee's device already constructs a log file entry on its own. Then, it encrypts this log file with the public key and forwards it to the log file server. In this adapted scenario, the server has the simple task to store these received and encrypted log entries into the log file database. Thus, the employees must no longer rely on the log file server to correctly encrypt their data because they are responsible themselves. Plus, the trust in the log file server can be low, since it only stores the encrypted log entries inside the database, but never sees sensitive data. Additionally, the data is already encrypted before transmitting such that using transit encryption as included in TLS is less important. On the other hand, we cannot ensure that the employees correctly encrypt their log entries. If a malicious employee wants to avoid his log entry being decrypted and read, he only needs to use a wrong public key for the encryption. This would cause that the admins or data analysts cannot correctly decrypt and therefore not read the corresponding entry. Furthermore, our log file server can no longer check the data it stores as this is decrypted. Thus, the employees could send arbitrary log entries with data that does not originate from their requests. Therefore, we cannot ensure *Validity* in the asymmetric case, which is also mentioned in Section 4.2.

But whether we use symmetric encryption schemes and the scenario defined in Chapter 3 or asymmetric encryption schemes and the adapted scenario instead does not influence the following functionality grading. This is because we only regard the content of the log file database, which is in both cases encrypted data that can be decrypted

[11] We are aware that there exist older versions of TLS that still have weaknesses regarding security. Thus, we assume for this thesis that the newest version (currently TLS 1.3) is used instead. The advantages of TLS 1.3 are described at `https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/` and in [Res18].

using the corresponding key.

Deterministic or Probabilistic Encryption Next, we discuss whether we need a deterministic or a probabilistic encryption scheme. If we use a probabilistic encryption scheme, then for two equal plaintexts their ciphertexts differ with high probability. This would mean that we cannot compare the encrypted cells in our log files since there are different ciphertexts for each plaintext. For example, if we want to find the most often visited website, we would only count the appearances of each ciphertext. But different ciphertexts can originate from the same URL. Thus, we would distribute the number of accesses of a website to all its ciphertexts and can hence not hope for correct results. Hence, it would be preferable if we could recognize that two different ciphertexts originate from the same plaintext such that we can count them as one URL.

But in Appendix B.1 we prove the following claim:

Claim 4.2.1. *Given a (symmetric) probabilistic cipher E defined over the set of spaces $(\mathcal{M}, \mathcal{K}, \mathcal{C})$ that is also secure against Chosen Plaintext Attacks as we defined it in Definition 2.14, we cannot distinguish whether two different ciphertexts would be decrypted to the same plaintext. Or more formally said: For two distinct ciphertexts $c_0, c_1 \in \mathcal{C}$ we cannot decide whether $\mathsf{Dec}(c_0, k) = \mathsf{Dec}(c_1, k)$ holds.*

This claim also implies that if we could decide whether different ciphertexts belong to the same plaintext, then the used encryption scheme E would not be CPA secure. Hence, we cannot use any probabilistic encryption schemes for our purposes and only discuss deterministic encryption in the following. We should be aware that if our encryption scheme is an asymmetric one, it cannot be CPA secure and deterministic. This is because, in the CPA game, an adversary could encrypt both chosen messages himself and if the scheme is deterministic he can recognize and decide by comparing the ciphertexts whether b is 0 or 1. Nevertheless, deterministic asymmetric encryption schemes are secure against message recovery attacks (Definition 2.16). Thus, we can at least ensure that it is hard for all efficient adversaries to regain the plaintexts from the database's cells given their stored ciphertext. But given a small set of possible plaintexts, we cannot ensure that those adversaries are able to determine which of those plaintext's ciphertext is stored in a given cell. In comparison, the symmetric encryption scheme E is considered to be semantically secure (Definition 2.13), which is more secure than the asymmetric scheme.

Inputs Last but not least, we learned in the previous section that we preserve the largest amount of information if we only hash our data cell by cell. This idea can also be used in this Encryption approach. Therefore, we encrypt each cell $\mathsf{entry}_{row}[col]$ with $row \in \{1, 2, \dots, R\}$ and $col \in \mathcal{CO}$ individually.

Moreover, the encryption scheme we use should be a correct one as defined in Definitions 2.12 and 2.15 such that we can decrypt the ciphertext and use the resulting plaintexts for our evaluations. These considerations lead to the following application of our Encryption approach: In the symmetric case, we use a correct, semantically secure computational cipher E_{sym} with two deterministic algorithms $(\mathsf{Enc}, \mathsf{Dec})$. And we pick $k \leftarrow \mathcal{K}$ as the symmetric key. Then, we compute $\mathsf{Enc}(\mathsf{entry}_{row}[col], k)$ for all $row \in \{1, 2, \ldots, R\}$ and for all $col \in \mathcal{CO}$ using E_{sym}. Alternatively, for the asymmetric scenario, we use the correct encryption scheme E_{asym} with algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Where E_{asym} is secure against message recovery attacks. We pick $(pk, sk) \leftarrow \mathsf{Gen}()$, with public key pk and secret key sk. Then we apply our approach to the log file by computing: $\mathsf{Enc}(\mathsf{entry}_{row}[col], pk)$ for all rows $row \in \{1, 2, \ldots, R\}$ and all columns $col \in \mathcal{CO}$.

Functionality Grading

In the following, we analyze which of our evaluation functions can be evaluated using this Encryption approach. We have the possibility to decrypt our stored data and therefore get useful information, though everything is stored as ciphertexts. Due to this, we can compute all of our evaluation functions. How the functions must be applied, is described in the following listing and the results can also be seen in Table 4.5. In the following, we use the fact that our encryption scheme is deterministic and correct to analyze ciphertexts as well as their plaintexts depending on the function. Since the scheme is deterministic, there exists exactly one ciphertext per plaintext. And since it is correct, there must be one plaintext per ciphertext, because otherwise Dec could not output the correct plaintext. All evaluation functions take the symmetric key k in the symmetric scenario and the secret key sk in the asymmetric scenario as additional input. Therefore, only admins and data analysts can start the following functions:

Avg_Download_Size() This function should compute the average of all values that are stored in `session_bytes`. Since, we cannot compute the average of the stored ciphertexts[12], the function first decrypts the necessary data in `session_bytes`[13]. As we encrypted each cell individually, we can decrypt those cells without decrypting any other ciphertext. Then, Avg_Download_Size() computes the average of all resulting, non-zero plaintexts and outputs the result. Therefore, this evaluation function is realizable by using the Dec algorithm.

Max_Download_Size() The next function, Max_Download_Size() should output the maximum of all values in `session_bytes` (of a given period). To find the maximum, we need to compare the size of the plaintexts, which is not possible given their ciphertexts[14]. Therefore, this function decrypts the ciphertexts first. Then, Malicious_Issuer() only searches for the largest plaintext (of the given period) and outputs it. Hence, admins can compute the Max_Download_Size() function with our Encryption approach.

[12]Alternative encryption schemes where decryption is not necessary to compute an average of ciphertexts are presented in the section about *Useful Extensions*

[13]We assume that the data is only decrypted inside the memory of the executing device instead of decrypting the whole database.

[14]In the section *Useful Extensions*, we discuss an alternative encryption scheme that would allow even this comparison of ciphertexts.

Number_Requests() This function takes a *device_id* as input and outputs the number of requests that were sent by the specified device. It can be computed because we use a deterministic encryption scheme and ciphertexts from identical plaintexts in `device_id` are also identical. Thus, this function encrypts its input *device_id* with the given key and compares the result to all ciphertexts in the `device_id` column. For each match, it increases the *result* by 1 and outputs *result* at the end. Since we do not want employees to start the functions, all keys must be given as input to this function, though we do not use the private key here. Hence, admins can compute this evaluation function even without decryption.

Most_Visited_Websites() This function regards the `target_ip` (or if given instead the `target_url`) column and should output the website with the most visits (in a given period). To do so, it groups all entries by their ciphertexts in `target_ip` and outputs the number of log entries in the largest group. Since admins might also be interested in the corresponding IP address and URL, Most_Visited_Websites() randomly picks one of the log entries in the largest group, decrypts its value in `target_ip` and outputs it. Then, the same is done to receive a URL from (most probably) another log entry. Hence, admins can use this evaluation function to successfully determine the number of requests per device with Encryption.

Malicious_Source() This evaluation functions gets a *device_id* as input and uses it to select all log entries that were issued by this device. For this, it computes the ciphertext of its input and selects all log entries with that ciphertext in their `device_id` column. Additionally, it gets as input a value *session_bytes* and sldo it encrypts this input with the used cipher or encryption scheme. Then, it compares the encrypted input to all values in the `session_bytes` column. For each matching log entry, Malicious_Source() outputs the found ciphertexts in `target_ip` and `target_url`. But those values are encrypted and thus not readable to humans. Hence, to output useful data, the function also decrypts the resulting ciphertexts. Since this information is not sensitive and only belongs to suspicious log entries, their decryption is allowed. Therefore, admins can compute this evaluation function with the Encryption approach.

Infected_Devices() We can realize this evaluation function such that it successfully outputs all devices that accessed a given (malicious) website. For this, it encrypts its input *target_ip*[15] and compares it to the ciphertexts in the `target_ip` column. After that, Infected_Devices() counts the number of different values in `device_id` in all matching log entries and outputs the result. Thus, admins can compute the Infected_Devices() evaluation function using our approach Encryption.

On_Purpose() This function outputs whether the response to an employee's request that is stored in a log entry was a redirection or not. To do so, it first applies Enc on its

[15]We could alternatively use *target_url* as input, if we have no IP address given. But the IP address is more useful since it covers all different URLs that could have been used to access the website or download.

input *log_id*. Then, it picks the log entry with the corresponding ID and analyzes its ciphertext in the `status_code` column. Since there exist at most 100 status codes for a redirection, the evaluation function can encrypt all those possible codes, store their ciphertexts for later use. If $\mathsf{entry}_{log\_id}[status\_code]$ is equal to one of the stored redirection ciphertexts, this function outputs 0 and otherwise 1. Thus, admins can compute On_Purpose() without decryption of the employees' data.

Malicious_Issuer() The next function is defined to output the user_ids of employees that accessed a given malicious website from an infected device with *device_id*. For this, the function first detects the relevant log entries by choosing the ones with $\mathsf{Enc}(device\_id, k)$ (or $\mathsf{Enc}(device\_id, pk)$ in the asymmetric case) in the `device_id` column. Additionally, Malicious_Issuer() gets *target_ip* or *target_url* as input. It computes $\mathsf{Enc}(target\_ip, k)$ (or $\mathsf{Enc}(target\_ip, pk)$ in the asymmetric case) to compare the resulting ciphertext to all ciphertexts in the `target_ip` column. For all matching log entries, it outputs the corresponding value in `user_id`, which reveals the identity of its issuer. But this data is also encrypted and decrypting it means revealing the most sensitive data. On the other hand, if the value from `user_id` is output of this function, then because it is suspicious and we do not necessarily preserve privacy for suspicious log entries. In this situation, the log entry is suspicious since it accessed a website we think malware was downloaded from. Therefore, we allow the revealing of the `user_id` because otherwise admins could never find a malicious employee. Thus, they can compute the Malicious_Issuer() function as well.

Period_Preselection() This function selects all log entries of a given period. All values in the `datestamp` and `timestamp` columns are encrypted but the ciphertexts reveal no information about the order of the log entries and when which were created. Hence, the function decrypts log entries to find out when they were created. To avoid encrypting all of them, it decrypts a randomly chosen log entry $\mathsf{entry}_{row}[datestamp]$ with log_id $row \in \{1, 2, \ldots, R\}$ and depending on whether it is too new or too old, it picks another log entry $row' < row$ or $row' > row$. This is done until the two log entries are discovered where the one is still outside the period and the next is already inside. Hence, admins can compute Period_Preselection() and only need to encrypt some log entries for this.

Useful Extensions As we can see, Encryption relies on the decryption algorithm Dec for 6 out of 9 evaluation functions, which is much. Using decryption means revealing data, which can also reveal sensitive information. Moreover, since our encryption scheme E must be deterministic to allow more computations, every revealing of the data allows admins to memorize the connection between some plaintext and its ciphertext. And this weakens the privacy of this approach since ciphertexts become readable once someone has memorized the corresponding plaintext. Therefore, it would be preferable to use other encryption schemes that allow the evaluation of some functions without using the

Table 4.5: Functionality Grading for Encryption

Evaluation Functions	Encryption	Comments
Avg_Download_Size()	✓	with decryption
Max_Download_Size()	✓	with decryption
Number_Requests()	✓	-
Most_Visited_Websites()	✓	with partial decryption
Malicious_Source()	✓	with decryption
Infected_Devices()	✓	-
On_Purpose()	✓	-
Malicious_Issuer()	✓	with decryption of sensitive data
Period_Preselection()	✓	with decryption
Result	9 P.	

decryption. For this, we could use homomorphic, order-preserving, and prefix-preserving encryption schemes.

With homomorphic encryption, we could sum two ciphertexts from `session_bytes` without decrypting them [Gen09, OTD13, EDG14]. For this, we only need to sum the ciphertexts and decrypt the result to receive the sum of their plaintexts. Hence, if we used homomorphic encryption to encrypt the `session_bytes` column, we could compute Avg_Download_Size() without decrypting all ciphertexts in it.

With order-preserving encryption schemes [BCLO09] the ciphertexts would have the same order as their plaintexts. Hence, the largest plaintext also results in the largest ciphertext. Thus, we could detect which of the ciphertexts from `session_bytes` is the largest to find their maximum. This way, we can compute Max_Download_Size() by only decrypting the largest ciphertext in `session_bytes`. But to achieve that both evaluation functions Avg_Download_Size() and Max_Download_Size() can be evaluated without using Dec, we would need the `session_bytes` column twice. One of them would be encrypted using homomorphic encryption and the other one would be encrypted with an order-preserving encryption scheme.

Moreover, we could use prefix-preserving encryption [XFAM02, XY12] which leads to plaintexts with the same prefix being encrypted to ciphertexts with equal prefixes as well. Then, the admins or data analysts would be able to detect whether all the client_ip_addresses originate from the network of the company or if there are requests with completely different IP addresses[16]. This would work since the requests of the company all originate from the same subnet, such that at least the first two parts of the IP addresses would be identical. Second, we could also compute our Period_Preselection() function without using the decryption on every cell. Given the date format YYYY-MM-DD HH:mm:ss, we can see which ciphertexts are from the same year, month, day, and so

[16]For this, the storage of the data must be outside the LAN to prevent that Network Address Translation (NAT) translates the global IP addresses to local ones.

forth. The ciphertexts from datestamps from e.g., the same year would have the same prefixes and we can use that to pick all log entries from e.g., the past year. For this, we only have to find all log entries where the first four characters of the ciphertext in `datestamp` are equal.

Hence, if we used all these different encryption schemes, we could enable to compute 6 of 9 evaluation functions without the usage of Dec. This would mean that we could ensure more privacy since the risk of memorizing the connection between plaintexts and their ciphertexts decreases. But on the other hand, it means high effort to use at least three different encryption schemes to only protect a single log file. Moreover, it is inefficient to store redundant columns only to encrypt them in two different ways. And since we want to discuss Encryption as a basic approach, using all these useful but expensive encryption schemes is out of scope of this work. Hence, we leave this idea for further research.

Privacy Grading

In this section, we regard the privacy preservation of Encryption. For this, we use our grading system from Section 3.5 to decide which of our privacy goals can be ensured and which not. As before, we regard efficient adversaries that gained access to the systems of the company. Thus, those adversaries and the employees have access to `Log_Files` database. But we assume the communication inside the system to be encrypted such that those actors cannot read the sent requests. We analyze the privacy goals that can be injured by those actors in *Unreadability*, *Anonymity*, and *Unlinkability.* Moreover, there are the admins and data analysts that additionally own the keys (k or (pk, sk)) to decrypt the log files. Their privacy-injury is regarded in *Admin-Privacy.* The result of this analysis is that only *Unreadability*, *Anonymity*, and *Presumption of Innocence* can be completely ensured by our Encryption approach. We can also ensure *Validity*, but not for all columns in the database and only for symmetric encryption schemes. As before, the analysis is further described in the following and the results for symmetric and asymmetric encryption schemes are summed in Tables 4.6 and 4.7.

Unreadability To injure the privacy goal *Unreadability* the employees and adversaries with access to the database must be able to read the content of the log files. In the symmetric case, our computational cipher E is semantically secure. That means that for each efficient adversary (which could be any of the actors) it is hard to detect which one of two different plaintext's ciphertext they see. Thus, they are unlikely to decrypt a given ciphertext without knowing the symmetric key k. And in the asymmetric case, the encryption scheme E is secure against message recovery attacks. This implies that it is hard for them to learn the plaintext of a given ciphertext. But given that asymmetric schemes are not CPA secure, the adversaries could at least learn some information about the plaintexts that allow them to distinguish their ciphertexts from other ciphertexts. Nevertheless, in both cases, the actors cannot easily decrypt the ciphertexts that are stored inside the database. Therefore, they only see the ciphertexts which reveal no helpful information about the plaintext. Hence, we can ensure *Unreadability* with Encryption.

Anonymity To ensure *Anonymity*, we must show that none of the values stored in the database reveals information about the employee that caused the specific log entry. Given the considerations from Section 3.3, we can limit the columns containing such information to the sensitive columns. Those are `user_id`, `device_id`, `client_ip_address`, and `client_mac_address`. For all those columns, we only store ciphertexts in the database's table. If they could be easily read and understand, the actors with access to the database could link those sensitive information to the employee that issued the corresponding requests. Nevertheless, since E is semantically secure in the symmetric and secure against message recovery in the asymmetric scenario, there is no easy connection between ciphertext and plaintext that reveals information about the stored plaintext. Otherwise, the adversaries in the games (Definitions 2.13 and 2.16) would be able to learn the plaintext m and would win their games with more than negligible probabilities. But since E is secure in those senses, the adversaries cannot learn enough from the ciphertexts to learn the plaintext. Therefore, they cannot use the content of the ciphertexts to link a log entry to its issuer. One might also think, that due to allowing the decryption of the `user_id` column to enable Malicious_Issuer() we cannot ensure *Anonymity*. But we defined in Section 3.2 that for this evaluation function a password must be given as additional input, which is only known to the admins. Thus, the actors we consider in this section (employees and adversaries with access to the database), cannot start that evaluation function or see its results. Hence, we can ensure *Anonymity*.

Unlinkability Since we use deterministic encryption, ciphertexts from identical plaintexts are also identical. This is necessary to compare ciphertexts, but it can also be used maliciously. Due to this, adversaries can detect whether two different log entries are similar by comparing the ciphertexts in each column. They can for example, detect whether two entries accessed the same IP address or were submitted by the same employee. As E is also correct (Definitions 2.12 and 2.15), two equal ciphertexts must always belong to the same plaintext. Thus, we cannot ensure *Unlinkability* for the Encryption approach.

Presumption of Innocence This privacy goal means, that the evaluation functions are only allowed to output sensitive data if the analyzed log entry is suspicious. For most of the evaluation functions, we only need to decrypt ciphertexts of non-sensitive columns, unless for the function Malicious_Issuer(). That evaluation function is the last one in the order of all evaluation functions. This means that if we execute Malicious_Issuer(), we find malware on a company's device, detect its source and ensure that the website was not accessed due to a redirection. Hence, the only missing piece of information is the employee that accessed a malicious website on purpose. Thus, we decrypt the user_id of a suspicious employee. Therefore, we can protect the privacy of innocent employees and can ensure this privacy goal using the Encryption approach.

Admin-Privacy We cannot ensure these privacy goals if the admins (or data analysts) own any special privileges or knowledge that allow them to injure *Unreadability*, *Anonymity*, or *Unlinkability*. In fact, they own the symmetric key k respectively the key pair (pk, sk). Those allow the admins (or data analysts) to decrypt the ciphertexts inside the database and injure the *Unreadability* though they are not allowed to. Moreover, if they can read this data in plaintexts, they can use this to find the employee that issued a given request. For example, they could decrypt the value from the `user_id` column and learn the user_id of the employee. Then, they could use the `Users` table to find the employee's name and hence *Anonymity* is injured by the admins (or data analysts). Hence, we cannot ensure *Admin-Privacy* for Encryption.

Ephemerality As we permanently store all data of the log files in our database and admins can access that data, we cannot ensure this privacy goal.

Validity The privacy goal *Validity* depends on what our log file server can do. For Encryption, we use again our scenario from Chapter 3 where the log file server is described as the server that creates log entries from the requests that are sent to the internet. As already mentioned above, we cannot ensure *Validity* at all, if E is asymmetric. Since the log file server only sees ciphertexts and owns no secret key, it is not able to check any of the values inside the log entries. In the symmetric case, the log file server must check whether correct user_ids and device_ids are given. Additionally, it needs to check whether status_code and session_bytes are values in meaningful ranges. Those tasks can be done, though they slow down the log file server. The main problem is checking whether a URL or IP address is valid. Since there exist many websites world-wide, it is hard to compare all of them to a single value in a request. And this is even harder, thinking of the many requests that must be processed per second. Hence, the log file server can check values only for some columns $col \in \mathcal{CO}$. Thus, *Validity* can only be partly ensured by a symmetric E.

Result

For this Encryption approach, we can compute all of our evaluation functions. This is possible, as the option to decrypt and therefore reveal hidden data allows us to evaluate all functions while still storing the data encrypted. Furthermore, compared to Partial Hashing none of the columns must be stored unhidden to enable the evaluation functions. On the other hand, Encryption only ensures 3 of 7 privacy goals. In comparison to the previous hashing approaches, we lose *Admin-Privacy* privacy. To avoid that malicious admins can decrypt the stored data though they are not meant to, we could use secret-sharing. Then, the key k or sk would be split with addition and each

Table 4.6: Privacy Grading for Symmetric Encryption

Privacy Goals	$\mathsf{Encryption}_{sym}$	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	$\times$	E is deterministic and allows comparison
Presumption of Innocence	✓	-
Admin-Privacy	$\times$	admins own the key (pair)
Ephemerality	$\times$	permanent storage
Validity	$\sim$	only for small value sets $\mathcal{V}_{col}$
Result	3.5 P.	

Table 4.7: Privacy Grading for Asymmetric Encryption

Privacy Goals	$\mathsf{Encryption}_{asym}$	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	$\times$	E is deterministic and allows comparison
Presumption of Innocence	✓	-
Admin-Privacy	$\times$	admins own the key (pair)
Ephemerality	$\times$	permanent storage
Validity	$\times$	log file server cannot check ciphertexts
Result	3 P.	

admin (or data analyst) gets a part of the key. As a result, the admins can only decrypt the data by cooperating and a subset of malicious admins cannot decrypt any secret data. The more admins we share the key to, the more admins must be malicious to be still successful. But this also complicates the computations if all admins are needed to compute a single function.

Nevertheless, if we use deterministic encryption, there is always the risk that the actors that can see the ciphertexts as well as the plaintexts can memorize their connections. Hence, any actor could recognize a ciphertext and already knows which plaintext it belongs to since he has already seen their connection. But this problem cannot be avoided in our case because we need to use deterministic encryption to compute our evaluation functions with at least decryption as possible.

To further improve this scheme, we could use ephemeral keys. Hence, the symmetric key or in the asymmetric case the public key pair is only used for a specified period. And e.g., monthly or weekly, a new key (pair) must be generated and used to encrypt all log entries that are created in the following. This would mean, that if we decrypt ciphertexts from a given column, then only those from the last e.g., month or week. Thus, we can ensure better privacy for older entries that are not part of our interest and would be decrypted and thus revealed unintentionally. But on the other hand, we would have to store all deprecated keys for a much longer time than a month or week. Since infected devices can sometimes be detected months after they got infected, we need those keys to also analyze the older log entries. Hence, storing all those keys takes more storage than only one key (pair).

Additionally, we could use one key (pair) per column $col \in \mathcal{CO}$. Then, only one key (pair) could be given to each admin, such that the keys for all cells are distributed among all admins of the company. This would ensure that a single admin can at least see the sensitive data of one column if he maliciously decrypts them. Moreover, he could not link any information from different columns since all other columns are encrypted by the other admins and are therefore not human-readable. Hence, we can ensure *Unlinkability* and *Admin-Privacy*, unless for the cell for which he owns the key (pair). On the other hand, this would mean that for each evaluation function, many admins must work together. We defined that the keys must be given as additional input to the evaluation functions. Hence, all admins must input the keys they hold. This is very impractical since no computations can be done if one of the admins is not at work on a given day. But even, if only the keys of the necessary columns must be given as additional input, there is still need for teamwork to start a function. Moreover, we need to store eleven different key (pairs) which again needs more storage than simple storing one *global* key (pair) for all columns $col \in \mathcal{CO}$.

Another idea is to use a a key (pair) per employee. But then, the same problem occurs as for salted and keyed hashing. Equal plaintexts from different employees would be encrypted to different ciphertext. Then all evaluation functions that rely on comparing the ciphertexts to each other would no longer be realizable.

Since we cannot ensure enough privacy goals with Encryption, we analyze one last basic approach that combines the two best of our previously described approaches.

4.3 Hashing and Encryption

For this last basic approach named Hashing & Encryption we combine Partial Hashing with Encryption. In Section 4.2 we decrypted almost none of the sensitive data. Hence, in this approach, we hash all sensitive data such that no one, not even the admins, can reveal it. Furthermore, we encrypt all data that has been stored as preimages in Partial Hashing, namely the set of non-sensitive columns. Thus, we can still compute our evaluation functions using decryption, but the data is only stored as ciphertexts.

In the following sections, we describe how Hashing & Encryption is applied to our scenario and grade its privacy as well as its functionality.

Application

As for the other approaches before, we use a collision resistant and one-way keyed hash function H. In addition to transit encryption and encryption at Rest, we use a correct[17] deterministic encryption scheme to encrypt our data such that each actor only sees the ciphertexts when accessing the database. For this approach, we can use a symmetric encryption scheme as well as an asymmetric encryption scheme. As described in Section 4.2 of our previous approach, the deterministic asymmetric encryption scheme only provides security against message recovery attacks. Thus, if an adversary could reconstruct ciphertexts, those only contain non-sensitive data such that no privacy goals are injured. In both symmetric and asymmetric cases, the data is stored encrypted and can be decrypted with a key that is only owned by the admins or data analysts. Hence, whether we use symmetric or asymmetric encryption does not influence the analysis of the functionality. Instead, they ensure different privacy goals, which is regarded in Section 4.3. Additionally, in the asymmetric case we would have a slightly different scenario[18] which is described in Section 4.2, Symmetric or Asymmetric Encryption. As for Encryption in Section 4.2 the decryption key must be given to each evaluation function as additional input to ensure that only admins are able to start these functions (and to allow the functions to decrypt data). As before, many companies do not have special employees that could work as admins and thus hire third party data analysts. In those cases, these analysts need the same privileges and knowledge as we allow to admins. Therefore, those have access to the keys of the cipher or encryption scheme E. Additionally, those analysts are considered for the privacy grading of *Admin-Privacy*. Moreover, we hash and encrypt each cell individually. All in all, Hashing & Encryption is applied as follows:

We define the subset $\mathcal{CO}^*$ to be the set of all sensitive columns meaning $\mathcal{CO}^* = \{\texttt{user\_id}, \texttt{device\_id}, \texttt{client\_ip\_address}, \texttt{client\_mac\_address}\} \subset \mathcal{CO}$. Given a keyed hash function H, we pick a key $k_H \leftarrow \mathcal{K}_H$ uniformly at random. In the symmetric case, we use a semantically secure computational cipher E with deterministic algorithms $(\mathsf{Enc}, \mathsf{Dec})$ and a symmetric key $k_E \leftarrow \mathcal{K}_E$. Otherwise, in the asymmetric case, we use

[17]Hence, it should hold the definitions for correctness in Definition 2.12 for our symmetric case and Definition 2.15 for the asymmetric case.

[18]See Chapter 3 for the original scenario.

an asymmetric encryption scheme E with algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where Enc must be deterministic for our purposes. In this case, the keys are chosen by computing $(pk, sk) \leftarrow \mathsf{Gen}()$. Additionally, E should be secure against message recovery attacks (Definition 2.16).

Then, for each log entry $row \in \{1, 2, \ldots, R\}$ and for each column $col \in \mathcal{CO}^*$, we compute $H(\mathsf{entry}_{row}[col], k_H)$. For all remaining columns $col \in \mathcal{CO} \backslash \mathcal{CO}^*$ (namely all non-sensitive columns) and for all log entries $row \in \{1, 2, \ldots, R\}$, we compute the ciphertexts with $\mathsf{Enc}(\mathsf{entry}_{row}[col], k_E)$ if E is symmetric or $\mathsf{Enc}(\mathsf{entry}_{row}[col], pk)$ if E is asymmetric.

Functionality Grading

In this section, we regard each of our evaluation functions from Section 3.2 under the application of Hashing & Encryption. For this, we first describe which of the functions can be realized and how this is done. The results are as usual denoted in the corresponding Table 4.8. We can see, that for this approach, most of the evaluation functions can be realized. Only Malicious_Issuer() is not realizable, since we hash the sensitive data inside the column `user_id`. Each of the evaluation functions additionally takes the secret key k or key pair (pk, sk) as input. More details of this analysis are given in the following listing.

Avg_Download_Size() The function's inputs are the ciphertexts from the `session_bytes` column. Thus, it should compute the average of all session_bytes. For this, it must decrypt the stored ciphertext in the `session_bytes` column and computes the average of all non-zero plaintexts. Alternatively, we could use homomorphic encryption as already discussed in Section 4.2, Useful Extensions. Hence, the admins can use this evaluation function to compute the average of all downloaded bytes.

Max_Download_Size() To find the largest plaintext in the `session_bytes` column, the stored ciphertexts must be decrypted first. Hence, Max_Download_Size() decrypts all those ciphertexts with the key that was given as input and compares the resulting plaintexts to find their maximum. Therefore, admins or data analysts can run Max_Download_Size() to obtain the largest download (in a given period).

Number_Requests() This functions gets as input a value *device_id*. Since the values from column `device_id` are sensitive, they are stored as digests. Therefore, the function Number_Requests() computes $H(device\_id, k_H)$ with our hash function H and the previously chosen key k_H to compare its input to all digests in the `device_id` column. For each matching log entry, it increases *result* by one and outputs it after it compared all log entries (from a given period). Hence, the admins or data analysts can run Number_Requests() to learn the number of sent requests per device and thus detect running denial-of-service attacks.

Most_Visited_Websites() This evaluation function should output the most often visited websites. For this, it groups all log entries (of a given period) by their ciphertexts in the `target_ip` (or `target_url`) column. Then, it outputs the size of the largest group. Additionally, it decrypts and outputs an IP address and a URL from that group by picking a random log entry for each IP and URL. Hence, admins or data analysts get all necessary data as output, since decryption can be used. Thus, they can compute this evaluation function with the Hashing & Encryption approach.

Malicious_Source() This function takes a value *session_bytes* as input. Since, the corresponding values in column `session_bytes` are encrypted, Malicious_Source() encrypts its input as well. This means, that in the symmetric case, it computes $\mathsf{Enc}(\mathit{session\_bytes}, k_E)$ where k_E is the key we picked at the beginning. Otherwise, the encryption of its input must be computed by $\mathsf{Enc}(\mathit{session\_bytes}, pk)$. It then compares the encrypted input to all ciphertexts in the corresponding column. For every matching log entry, it decrypts the ciphertexts in the `target_ip` and `target_url` columns and outputs them in the end. Therefore, admins and data analysts can execute the evaluation function Malicious_Source() with our Hashing & Encryption approach.

Infected_Devices() For this, the evaluation function gets *target_ip* (or *target_url*) as input. As all values in `target_ip` are encrypted, it then encrypts its input by computing $\mathsf{Enc}(\mathit{target\_ip}, k_E)$ with k_E being the key we picked in Section 4.3 if E is symmetric. In the asymmetric case, it instead computes $\mathsf{Enc}(\mathit{target\_ip}, pk)$. Next, the function compares its encrypted input to all ciphertexts in the `target_ip` columns and groups all matching log entries by their digests in `device_id`. After that, it outputs the number of different groups. Hence, admins and data analysts can compute this evaluation function with Hashing & Encryption.

On_Purpose() The output of this function is stored in the `status_code` column, which is encrypted. If a symmetric encryption scheme is used, On_Purpose() firstly encrypts the given *log_id* by computing $\mathsf{Enc}(\mathit{log\_id}, k_E)$ with k_E being the encryption key that was picked in Section 4.3. If the encryption scheme is asymmetric instead, On_Purpose() encrypts its input with $\mathsf{Enc}(\mathit{log\_id}, pk)$. After that, it picks the log entry with the corresponding ciphertext in `log_id`. Then, On_Purpose() compares the corresponding ciphertext in `status_code` to all ciphertexts belonging to 300 - 399. If the ciphertext in `status_code` belongs to a redirection code, On_Purpose() outputs 0 and otherwise 1. Thus, we can realize this evaluation function with Hashing & Encryption.

Malicious_Issuer() Since the values in the `user_id` column are hashed with our one-way hash function H, it is hard to compute their preimages. Moreover, we do not want to try this since we use hash functions to avoid that the sensitive data can be revealed. Therefore, we cannot output the (malicious) issuer of any suspicious request and thus admins and data analysts cannot compute this evaluation function with Hashing & Encryption.

Period_Preselection() For this evaluation function, we can do the same computations as for the previous approach with Encryption. The function needs to decrypt a

Table 4.8: Functionality Grading for Hashing with Encryption

Evaluation Functions	Hashing & Encryption	Comments
Avg_Download_Size()	✓	with decryption
Max_Download_Size()	✓	with decryption
Number_Requests()	✓	-
Most_Visited_Websites()	✓	with decryption
Malicious_Source()	✓	with decryption
Infected_Devices()	✓	-
On_Purpose()	✓	with decryption
Malicious_Issuer()	×	unreadable output data
Period_Preselection()	✓	with decryption
Result	8 P.	

randomly picked value in the `datestamp` column and if it is out of range for the given period, it picks another log entry that is newer respectively older than the previous one. This way, it only decrypts a few log entries until it finds the last log entry that still belongs to the given period and the first entry that is already in that period. Then, it outputs all log entries with log_ids in between.[19] Thus, admins can also compute Period_Preselection() with the Hashing & Encryption approach to preselect interesting log entries.

Privacy Grading

This section again contains the analysis of our privacy goals (Section 3.4). This time, we consider them under the application of Hashing & Encryption. In addition, we assume that the regarded adversary has access to the `Log_Files` database but cannot read the traffic in the company's system since it is encrypted with e.g., TLS. Again, the privacy goals *Unreadability*, *Anonymity*, and *Unlinkability* are only regarded for employees and the above described adversaries. While admins and data analysts are considered in *Admin-Privacy*. Our resulting grading table (Table 4.9) shows that we can ensure 3 of 7 privacy goals. While *Validity* can only be ensured for most columns and *Admin-Privacy* only if the value sets $\mathcal{V}_{col}$ are large. Why we can achieve these goals is further described in the following listing.

Unreadability We could not ensure this privacy goals if any employee or adversary with access to the database was able to read the data stored in our log file database. For the sensitive columns $col \in \mathcal{CO}^*$, those actors must make sense of the given digests

[19]Since $\texttt{entry}_{row}[\texttt{log\_id}] = row$, all log entries are ordered by their *log_id*. Otherwise, it would be hard to find all log entries from a given period. Moreover, we would need to decrypt all log entries, since the ones we are looking for could be widely distributed among the other entries outside our period.

or compute the digests' preimages. Since H is one-way, every efficient adversary can learn the preimage of a given digest with negligible probability. Thus, for all regarded actors, it is hard to learn the data in the sensitive columns $\mathcal{CO}^*$. All other columns $col \in \mathcal{CO} \backslash \mathcal{CO}^*$ are encrypted using E_{sym} or E_{asym}. In both cases, the ciphertexts must reveal information about their plaintext such that these employees or adversaries can reveal the data in the database. But E_{sym} is semantically secure (Definition 2.13), which implies that efficient adversaries are unlikely to recognize the ciphertext of a known plaintext. And E_{asym} is secure against message recovery attacks (Definition 2.16) such that it is hard for efficient adversaries to decrypt a given ciphertext. Hence, in both cases, the ciphertexts do not reveal enough information to learn their plaintexts. Therefore, the regarded actors cannot learn any information from the stored data with high probability. Thus, for all columns $col \in \mathcal{CO}$ we can ensure *Unreadability.*

Anonymity Since the hash function H is used to hide the sensitive columns $col \in \mathcal{CO}^*$, we can conclude the same as for the hashing approaches (Sections 4.1.1 and 4.1.2). The only columns that reveal information about a log entry's issuer are the sensitive ones. In the usual case, their data can help find such an issuer if they are readable to the actors with access to the database. But as already described previously, we ensure *Unreadability.* This means that the sensitive data cannot be learned and thus also not be linked to any employee. Hence, we can ensure *Anonymity.*

Unlinkability Since our encryption scheme E and our hash function H are deterministic, we can compare ciphertexts with ciphertexts and digests with digests. Moreover, the correctness of E and the collision-resistance of H imply that equal ciphertexts (or digests) belong to equal values. Hence, comparing ciphertexts or digests is equivalent to comparing their plaintexts or preimages. This can also be used to e.g., identify whether two different log entries were issued by the same employee. Thus, we cannot preserve *Unlinkability* for Hashing & Encryption.

Presumption of Innocence The privacy goal *Presumption of Innocence* would only be injured if we revealed sensitive data to enable our evaluation functions (Section 3.2). As H is one-way, computing a preimage given a digest is hard. Moreover, none of the previously defined evaluation function includes the revealing of a sensitive column. Thus, we do not reveal any sensitive column's content (for any employee) and therefore *Presumption of Innocence* can be ensured using Hashing & Encryption.

Admin-Privacy To injure *Unreadability*, *Anonymity*, or *Unlinkability*, admins or data analysts need specific tokens or knowledge to reveal the hidden data. As the sensitive columns $col \in \mathcal{CO}^*$ are hashed with the one-way hash function H, the privileged actors only know the key k_H. But this is already considered to be public in the security game (Definition 2.9) and thus gives no advantage to them compared to all non-privileged actors. On the other hand, the admins and data analysts own the secret key k_E in the symmetric case and the key pair (pk, sk)

Table 4.9: Privacy Grading for Hashing and Encryption

Privacy Goals	Hashing & Encryption	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	$\times$	H, E are deterministic and allow comparison
Presumption of Innocence	✓	-
Admin-Privacy	$\sim$	only for large value sets $\mathcal{V}_{col}$
Ephemerality	$\times$	permanent storage
Validity	$\sim$	only for small value sets $\mathcal{V}_{col}$
Result	4 P.	

in the asymmetric one. With those keys, the privileged actors can easily regain the plaintexts from the stored ciphertexts. In particular, they can decrypt all ciphertexts in non-sensitive columns $col \in \mathcal{CO} \backslash \mathcal{CO}^*$. But for *Unreadability* we are only interested in hiding the sensitive columns. And for *Anonymity*, only sensitive columns could reveal information about the identity of an employee. But again, if any of the sensitive value sets $\mathcal{V}_{col}$, $col \in \mathcal{CO}^*$ is small, malicious privileged actors can efficiently compute all corresponding digests. Thus, they can learn the data stored in the sensitive columns and injure *Unreadability*. Additionally, given access to the `Users` database, they can learn the identity of the log files' issuers and also injure *Anonymity*. Hence, the privileges of admins and data analysts can enable them to injure already ensured privacy goals. Therefore, Hashing & Encryption ensures *Admin-Privacy* as long as $\mathcal{V}_{col}$, $col \in \mathcal{CO}^*$ is huge.

Ephemerality As all the data, sensitive and non-sensitive, is permanently stored in our log file database and can be accessed by all admins and data analysts, we cannot ensure *Ephemerality*.

Validity If we want to provide this security goal, we need to ensure that our log file server is able to check all values from all columns $col \in \mathcal{CO}$ since we use the scenario from Chapter 3. Our log file server needs to verify the values for `session_bytes` and status_code to be in valid ranges. Moreover, it must check the values for client_ip_address and client_mac_address to be valid addresses of the company's network. Otherwise, such a request would be suspicious. The data from columns `target_ip` and `target_url` is hard to check since for those the value set is extremely huge and the log file server has not much time to process a single request. Hence, we conclude that the log file server can only ensure *Validity* for most of the database's columns. Therefore, we denote *Validity* as only partly ensured.

Result

After we have analyzed all interesting aspects of our Hashing & Encryption approach, we can compare this approach to the previously described ones. With regard to functionality, this approach is much better than Complete Hashing because the possibility of decryption enables the computation of nearly all evaluation functions. But on the other hand, we cannot realize as many functions as with Encryption because this time Malicious_Issuer() cannot be computed. If we only regard their privacy preservation we can ensure the same privacy goals as we can ensure with Complete Hashing and Partial Hashing. Moreover, we can ensure more privacy goals as with Encryption. Indeed, Hashing & Encryption ensures the same privacy goals and enables the same evaluation functions as Partial Hashing. Nevertheless, Hashing & Encryption is better regarding privacy since none of the data is stored unhidden. We can summarize that for Hashing & Encryption, we can compute the most evaluation functions while still ensuring the most privacy goals. Hence, Hashing & Encryption combines the best of all previously described approaches. One thing that stands out in all analyzes is that we can never ensure *Unlinkability* or *Ephemerality*. Moreover, we cannot completely ensure *Validity* in all our basic approaches. Another huge disadvantage of the basic approaches is that we require deterministic algorithms. Thus, given small value sets, there always exists the risk of malicious actors computing all digests or ciphertexts from known plaintexts. Thus, the (sensitive) data would no longer be hidden if those mappings are learned. Due to this, we next analyze our advanced approaches e.g., Prio. Those use randomness to avoid this simple connection between secret and hidden data. Moreover, they can ensure *Unlinkability* and *Ephemerality* as well as *Validity*.

5 Advanced Approaches

In the following chapters, we concentrate on the advanced approaches. Those use several cryptographic techniques to allow privacy-preserving evaluation. In particular, we analyze Prio [CB17], Differential Privacy [DMNS06], and Private Set Intersection [DMRY09]. Since they are private per design, we expect them to ensure more privacy goals than our basic approaches. Moreover, these are the promising candidates for our privacy-preserving log file collection and evaluation system.

5.1 Prio

The previously regarded basic approaches are already good as we can realize all evaluation functions with Encryption and half of our privacy goals can be ensured by each approach. But we are still hoping for a better approach that also guarantees *Unlinkability* and *Ephemerality*. Therefore, the next and most promising approach we investigate is Prio, based on the paper of Corrigan-Gibbs and Boneh [CB17]. Hence, the following description of Prio and its privacy and security properties in Sections 5.1.1 and 5.1.2 are mainly based on that paper from Corrigan-Gibbs and Boneh [CB17]. We add information on how this approach can be adapted to our scenario (Chapter 3) in Section 5.1.3. Further, we use their protocol to realize our evaluation functions (Section 3.2) in Section 5.1.4 and grade the privacy that is preserved in our scenario by using our privacy goals (Section 3.4) in Section 5.1.4.

Prio allows the aggregation of secret data that belongs to many different clients. This is done in a privacy-preserving way by using at least two servers and secret-sharing. The Prio protocol from Corrigan-Gibbs and Boneh [CB17] can recognize invalid data that is provided from a client, such that we can ensure *Validity*. Furthermore, Prio can avoid the computing servers from learning the clients' secret data and thus preserves the privacy of the clients i.e., the employees. Additionally, Prio ensures anonymity. Those security and privacy properties of Prio are discussed in more detail in Section 5.1.2. Moreover, Prio is already used by Mozilla[1] and to trace Covid-19 infections during the pandemic[2].

[1]Mozilla's blog: `https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/`,
Mozilla's presentation of Prio application: `https://www.facebook.com/atscaleevents/videos/scale-2019-firefox-origin-telemetry-with-prio/468109414055086/`,
Gibbs' presentation slides: `https://rwc.iacr.org/2020/slides/Gibbs.pdf`,
GitHub: `https://github.com/mozilla/prio-processor`

[2]`https://www.abetterinternet.org/post/prio-services-for-covid-en/` and [AG21] for more details.

5.1.1 How Prio works

The main goal of Prio is to compute an aggregation function over the secret data of different clients without revealing the secret data to the servers computing the aggregates [CB17]. Hence, it fits our scenario where many clients (used by our employees[3]) send their requests containing data we would like to analyze. In order to work correctly and in a privacy-preserving way, Prio needs at least two servers (i.e., Prio servers in the following) for the evaluation. Only then, secret-sharing can be used such that each server only computes a part of the clients' inputs and can thus not learn their secret data. As Prio can only compute sums of client data, Affine-Aggregatable Encodings (AFEs) (Definition 2.18) are used. Those run the Encode algorithm on the clients' secret data before it is sent to the servers and the Decode algorithm after the aggregate is computed. These encodings allow Prio to evaluate several different aggregation functions i.e., average, variance, boolean AND and OR, frequency count, union of sets, see [CB17] for more details. Before starting the evaluation function, public parameters e.g., the necessary AFE with its parameters κ, κ' (and more, depending on the AFE) are given to all clients and servers. Moreover, Prio can ensure the *Validity* of the secrets with secret-shared non-interactive proofs (SNIPs). Due to the use of SNIPs and the AFEs' Valid algorithm, the servers can check which client's secret shares are valid without access to the whole secret. Hence, with Prio, it is ensured that only valid inputs influence the output of the evaluation functions while preserving privacy. Next, we describe the execution of Prio in detail.

1. Each client t owns secret data sec_t (e.g., a visited website's URL) which should be used as input to the evaluation functions without revealing sec_t to the computing Prio servers. Before sending sec_t to the servers, each client must compute the encoding $e_t = \mathsf{Encode}(\mathrm{sec}_t)$ using the Encode algorithm of the used AFE.

2. Next, the clients need to create the secret shares of their encoded secret e_t for all S servers. Each such share $[e_t]_s$ is created for and sent to server s. It holds that $e_t := \sum_{s=1}^{S} [e_t]_s$. Moreover, $S - 1$ shares are chosen at random (i.e., $[e_t]_s \leftarrow \mathbb{N}$ for numerical secrets)[4]. Thus, the shares are independent from sec_t. This construction is called *additive secret-sharing* [CB17].

3. Then, to ensure the *Validity* of the sent shares, each client must prepare a SNIP proof[5].

   a) For this, the clients compute $\mathsf{Valid}(e_t)$, where the algorithm Valid depends on the chosen AFE. Valid uses an arithmetic circuit AC (Definition 2.17) to

[3]Each of the employees uses a device which is the client in this setting. Thus, for each employee exists exactly one client that executes the Prio protocol with the secret data that is produced by this employee.

[4]According to Corrigan-Gibbs and Boneh [CB17] this can also be done more efficiently for shared vectors by using a pseudo-random generator (PRG). Then, instead of shares, PRG keys are chosen at random that represent the much longer shares. See [CB17, Appendix H] for details.

[5]For a detailed description of SNIP proofs see [CB17].

compute whether the given encoding $e_t = \mathsf{Encode}(sec_t)$ is valid ($\mathsf{Valid}(e_t) = 1$).

b) In the next step, the clients compute two functions $f(x)$ and $g(x)$ that each represent one of the inputs of multiplication gate x of the circuit AC.

c) From those functions, the clients compute a third function $h(x) = f(x) \cdot g(x)$ which represents the outputs of each multiplication gate x.

4. After that, the clients send their tuple of shares $([e_t]_s, [h]_s, [\alpha]_s, [\beta]_s, [\gamma]_s)$[6] to server s using a secure and authenticated channel as provided by TLS. As the clients only send a single share of the proof to the servers and there is no additional client-server communication, Prio's zero-knowledge proof system is *non-interactive*[7]. The shares $[\alpha]_s, [\beta]_s, [\gamma]_s$ are created from $\alpha, \beta, \gamma \in \mathbb{F}$ with regard to $\gamma = \alpha \cdot \beta \in \mathbb{F}$. The servers need them for the multiplication of shares which is necessary to check the SNIP. The share $[h]_s$ is a share of h's coefficients, which is more efficient than transmitting shares of h's whole truth table[8].

5. After receiving these shares, each server s verifies the SNIP proof by checking whether h delivers the necessary result that Valid outputs 1 and thus the received secret sec_t is valid. For this, they first reconstruct the shares $[f]_s$ and $[g]_s$ given the shares from sec_t and h. Second, the servers ensure that $f \cdot g = h$ holds. In particular, those three functions are only evaluated on a randomly chosen input $r \leftarrow \mathbb{R}$ since this is more efficient than checking each value in $\mathbb{R}$. If one of those two checks fails, the share is invalid and thus ignored in the following[9].

6. If e_t is a validly encoded secret share from client t, a server s adds $[e_t]_s[0..\kappa']$ to its internal sum_s, which is initialized with 0. Whereas, κ' is a parameter of the chosen AFE and describes the length of the share that should be regarded.

7. In the end, all internal server-sums sum_s are published. Hence, any server can compute the Decode algorithm of the chosen AFE with $\mathsf{Decode}(\sum_{s=1}^{S} sum_s)$ to get the final result of the aggregation function. In fact, this task can be done by a *leader server*, which coordinates the whole protocol, computes the result from all *sum*s and stores the resulting aggregate along with a datestamp and timestamp. But this server should be the most trusted one. The stored result is the chosen aggregate on input all secrets $sec_t, t \in \{1, 2, \ldots, T\}$ (see Appendix B.2 for a formal proof

[6] In particular, each client must send as many shares of (a, b, c) triples as AC has multiplication gates since each triple is only used once.

[7] `https://www.youtube.com/watch?v=dE0KJxxe3xI`

[8] Another even more efficient solution would be to already evaluate h on each point on client-side and to send shares of that evaluation of h to the Prio servers [CB17].

[9] The Prio servers use multi-party computation (MPC) to verify the correctness of the SNIP proofs. This is necessary since the servers must combine their intermediate results of the arithmetic circuit AC to reconstruct the final result of Valid. In particular, MPC is required to compute the product of two shares and for adding the servers' final resulting shares of Valid's output. Thus, the servers learn whether Valid outputs 1 without learning the values of the internal nodes in the circuit AC. For more details see [CB17].

of the fundamental sum aggregation). To compute different aggregation functions (i.e., evaluation functions), different AFEs must be chosen. For example, an AFE for computing an average is necessary to compute Avg_Download_Size().

The whole protocol of Prio is also shown in the following figures. For better readability, it is split into client-side (Figure 5.1) and server-side executions (Figure 5.2).

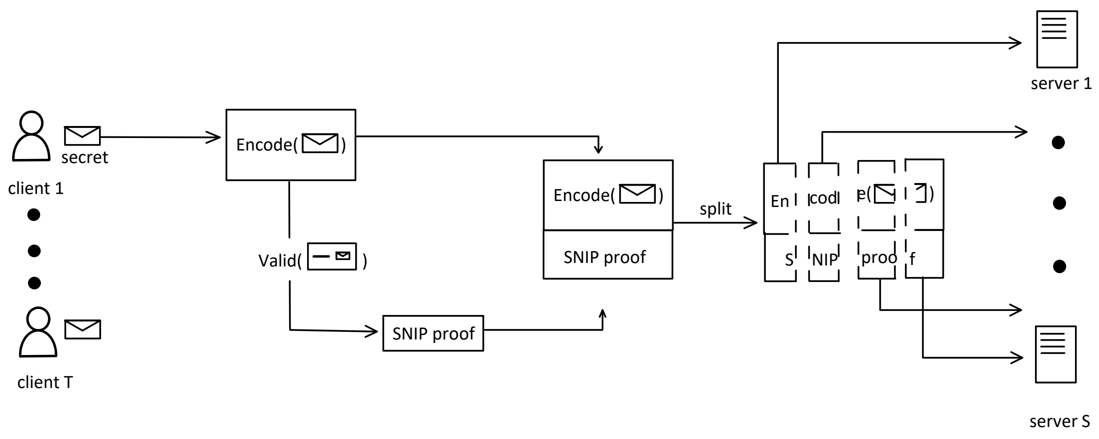


Figure 5.1: Execution of Prio protocol, client-side

5.1.2 Security and Privacy of Prio

For the security and privacy of Prio there are several types of adversaries (e.g., malicious clients and servers) that have to be considered. First of all, adversaries that take control of one or more servers can impersonate them to influence the computations of the aggregates. Adversaries that are able to interrupt client-server-communication can cause unavailability of the whole system or exclude specific clients from participation. Moreover, malicious clients (or clients controlled by adversaries) could try to send wrong secrets to influence the aggregates. According to Corrigan-Gibbs and Boneh [CB17], Prio is secure against adversaries that can control clients and servers as long as one server is still trustworthy[10]. They additionally consider adversaries with access to the company's network that can arbitrarily manipulate the sent packets. These adversaries can thus cause attacks we describe in Section 5.1.4. Nevertheless, protection against adversaries reading the transmitted traffic is provided by Prio by assuming secure and authenticated channels, as provided by TLS. In comparison, the adversaries we considered in the basic approaches (Chapter 4), only passively observed the log file's content. Due to this, the adversaries we regard for Prio have many more abilities and are thus stronger.

[10]A Prio server is trustworthy if it executes the Prio protocol correctly i.e., as described in Section 5.1.1 or [CB17].

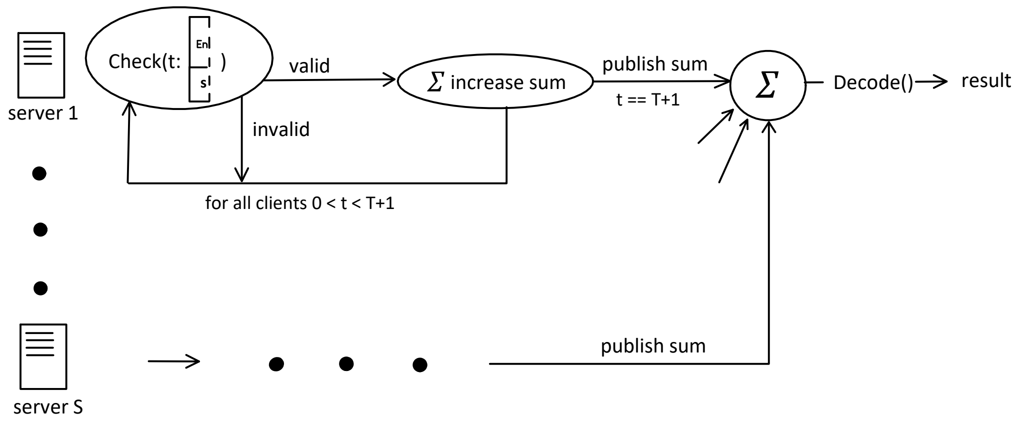


Figure 5.2: Execution of Prio protocol, server-side

As Prio makes use of the AFEs and SNIP proofs, both must ensure security and privacy properties such that Prio can be secure and private as well. For this, an AFE should be private, correct, and sound as described in Definition 2.18. Moreover, a SNIP must also be correct and sound and should further provide zero-knowledge. In detail, for correctness, a SNIP must ensure that each client's secret w that is inside the defined language $\mathcal{L}$ of valid inputs and correctly encoded, is accepted by the Prio servers. While soundness implies that all others secrets where $w \notin \mathcal{L}$ or $\mathsf{Valid}(\mathsf{Encode}(w)) \neq 1$ are rejected by the servers with high probability[11]. Furthermore, SNIPs must provide zero-knowledge such that none of the single secrets w can be learned by the servers and only whether $\mathsf{Valid}(\mathsf{Encode}(w)) = 1$ holds. Then, given AFEs and SNIPs with these security and privacy properties, Prio can ensure f-privacy, anonymity, validity, and robustness. These properties are informally described in the following and based on [CB17].

- **f-privacy** implies that the Prio servers do not learn the clients' inputs w given the final aggregate of function f. In particular, there exists a simulator Sim that simulates the view of all malicious clients and servers of the computation without any secret w as input. For the formal definition of this simulator, see [CB17, Appendix A,G]. In other words, the view is a tuple of all transmitted data in the Prio protocol. It contains the data that is generated by the malicious and trustworthy clients i.e., secret shares and SNIP proofs. Additionally, the internal *sums* of all malicious and trustworthy servers are simulated by Sim and thus also part of the view. For this, the simulator takes as input the final aggregate $f(sec_1, sec_2, \ldots, sec_t)$ and the data that is generated by malicious clients and servers. This shows us, that Sim

[11]This also holds for very efficient malicious client i.e., super-polynomial time ($\Omega(n^p), p \in \mathbb{N}$, Definition 2.2) adversaries .

executes the protocol's steps in a wrong order i.e., the output is taken as input to simulate the protocol without the clients' secrets as input. The distribution of those simulated data is then computationally indistinguishable from the distribution of all transmitted data in the maliciously controlled execution of the protocol. Even during the collaborative computation of the multiplication gates of the arithmetic circuit AC, the Prio servers learn no information that aids reconstructing the secrets as long as the SNIP provides zero-knowledge. Thus, Prio can protect the privacy of the employees against malicious clients and servers. Nevertheless, if all servers were malicious, they could exchange their secret shares and reconstruct the secrets. Thus, at least one of the servers must be trustworthy to ensure f-privacy[12]. Overall, the AFE must be private (see Definition 2.18) and the SNIP must provide zero-knowledge. The detailed proof of this statement can be found in [CB17, Appendix G]. For us, this means, that an adversary learns at most $S-1$ of S shares and the final aggregate, but not the secret w of any client.

- **Anonymity** can be provided by Prio for all symmetric and f-private aggregation functions f. A function f is called symmetric if it holds that f outputs the same result regardless in which order it receives its inputs. Or more formally, it must hold that $f(w_1, w_2, \dots, w_T) = f(w_{\pi(1)}, w_{\pi(2)}, \dots, w_{\pi(T)})$ for all client inputs $w_i, i \in \{1, 2, \dots, T\}$ and permutation π. The proof of this is located in [CB17, Appendix A]. For our scenario, this means that the aggregate of the evaluation function does not reveal any information about the owner of a secret besides what is revealed by the aggregate itself. Thus, the protocol protects the clients i.e., the employees from malicious servers.
- **Validity**[13] implies that (with high probability) only valid client-inputs w are used to update the servers' aggregate *sum*. An input is valid if and only if it holds that $w \in \mathcal{L}$ and $\mathsf{Valid}(\mathsf{Encode}(w)) = 1$. For our scenario, validity ensures that all secrets *sec* that are not inside the value set $\mathcal{V}$[14] or wrongly encoded are rejected by the Prio servers with high probability. Otherwise, if $sec \in \mathcal{V}$ and $\mathsf{Valid}(\mathsf{Encode}(sec)) = 1$ the secret is aggregated to the servers' *sum* and thus influences the final aggregate. Due to this, Prio servers are protected from malicious clients.
- **Robustness** in the sense of failing servers is also ensured by Prio. But it requires all servers to be trustworthy. Corrigan-Gibbs and Boneh [CB17] mention that one could also construct Prio to ensure robustness given a subset of trustworthy

[12] In the protocol, the Prio servers communicate with each other in order to verify the secrets by checking the SNIP proofs. For this, they compute the result of the arithmetic circuit AC without learning the values of the internal nodes. If all clients were malicious, they could sum their values from the internal nodes and also reconstruct the values from the input node. Thus, they learn the clients' secrets. Due to this, at least one server must be trustworthy to ensure privacy.

[13] In the original paper called *robustness*, but this collides with the robustness against failing Prio servers, such that we use validity instead.

[14] We define a $\mathcal{V}_{col}$ for each column *col*. This is necessary since `session_bytes` stores different data than `target_ip` and thus needs another range of valid values. But we omit the index whenever possible to simplify the expressions.

servers, but then, the privacy of the whole protocol suffers. In the case of some defect servers, the remaining servers must compute the aggregate without them, to ensure robustness. But this means that also a subset of servers suffices to reconstruct the secrets. Thus, if exactly this number of servers were malicious, they could regain the clients' secrets and hence injure the privacy of the employees.

5.1.3 Adapted Scenario

In this section, we describe how we adapt our scenario to allow evaluations with Prio, as it is very different to how we analyze data in the previous chapters. In contrast to all previously shown approaches, we do not store log files anymore. Instead, we directly evaluate all incoming data and only store the results of the data's aggregate. This leads to more privacy since the data of the clients' requests is not completely stored anymore. On the other hand, it is hard to do any evaluation once the data is gone. Thus, it is important to start as much evaluation functions as possible to store enough results for further investigations. Furthermore, we remove the log file server, which was only meant to create the log file entries for the database. We replace it with at least two Prio servers which get their inputs directly from the clients (i.e., devices) and store the results in a storage we name *result storage*. As we divided proxy and log file server into different entities, the proxy server is not affected by the adaptions we need to do for the usage of Prio. Moreover, for Prio, each client t needs to prepare the computation of an evaluation function itself. Thus, it needs to pick the necessary values called its *secret data* sec_t from the previously submitted request. Additionally, all admins and data analysts only have access to the result storage using their personal logins and can then use those aggregated data for their investigations. Hence, the secret data of the clients remains secret as long as it cannot be concluded from the results of the evaluation functions.

As described in [CB17], the servers aggregate shares until they received a share from each client. Thus, in the original Prio scenario, each client is only allowed to *vote* once. This suffices, if their admins for example, want to learn the most used browsers. But in our scenario, we want to find the device with most requests and if we only allowed each device to vote once, before computing the result of the evaluation function, we would never get a useful result. Moreover, if we tried to ensure that each client sends exactly one share per evaluation, the clients (and thus the employees) must wait with further requests until everyone has sent its share. Or alternatively, we would have to ignore all meanwhile incoming requests, which is also not preferable. Hence, in our scenario, we need to allow that every device can send as many votes as it submits requests in a given period. Thus, the given inputs must be evaluated after a predefined period is over. Alternatively, the servers accept a predefined number of shares $K > 1$ before publishing their *sum*s. This second option is more useful, since we always know the number of received input shares per execution of an evaluation function.

As the clients now participate in the evaluation by sharing their secrets, they need to know which data must be encoded with which AFE. It can be seen that we need to execute some evaluation functions periodically with the same kind of input data. Thus, we do not need to notify each client separately to encode its secret for a specific evaluation

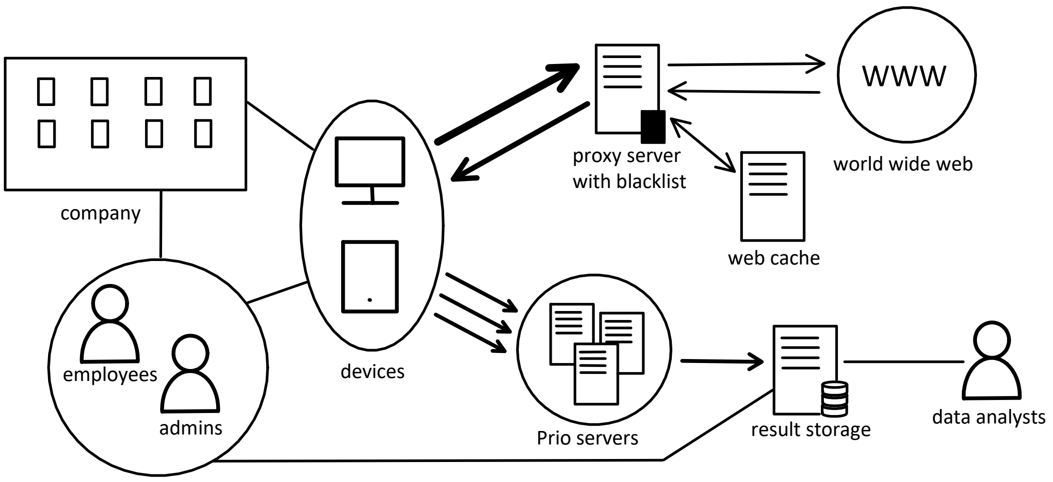


Figure 5.3: Diagram of the adapted scenario for our approach using Prio

function. Instead, we implement all clients to automatically construct the shares from their requests. Moreover, they take the necessary *user_id* from the currently logged in employee and the *device_id* from themselves. Then, for each evaluation function that is specified in Section 5.1.4, each client picks the demanded values from the requests, produces the shares and proofs and sends everything to the Prio servers which compute that particular evaluation function[15].

The whole adapted scenario for Prio can be seen in Figure 5.3.

5.1.4 Evaluation

Application

For each of our evaluation functions from Section 3.2 we need different AFEs to realize them. The overall application from creating shares of the secret data over sending the shares to the servers to aggregating the server's internal sums is equal for all evaluation functions and can be seen in Section 5.1.1. The details of each application are described in the following section.

Functionality Grading

In this section, we analyze which evaluation functions from Section 3.2 can be realized if our used approach is Prio. As we can see, there are only four of them realizable. This is caused by Prio only being able to compute aggregate statistics and using live data.

[15]As the Prio servers cannot see the secrets, they cannot determine to which evaluation function a share should be aggregated. In practice, one could send (not shared) identifiers with each share that tell the Prio servers (i.e., their leader) for which evaluation function the share should be used.

Which functions in particular can be realized and how this is done in detail is described in the following. The results of this functionality grading are summed in Table 5.1.

Avg_Download_Size() This evaluation function gets as input the values that would belong to the column `session_bytes`[16] and computes the average of all those values. Given an upper border B[17] such that for each secret sec it holds that $sec \leq 2^B - 1$ (each secret can be represented as B-bit binary). We use the AFE of Prio that can compute the average of the (non-zero) secret inputs [CB17]. First, the Encode algorithm computes the binary representation of a client's secret meaning bits $b_i \in \{0,1\}$ with regard to $sec = \sum_{i=0}^{B-1} 2^i \cdot b_i$ and outputs the tuple $e_t = (sec, b_0, b_1, \ldots, b_{B-1})$. Then, the clients use additive secret-sharing to produce the shares as described in Section 5.1.1. Additionally, they create the SNIP proofs from the AFE's Valid algorithm and produce their shares as well. After that, all those shares are sent to the servers. Those use the SNIP proof to check whether the given tuple contains the secret sec and its bit representation. Hence, all client-provided secrets are rejected that were e.g., no integers in $[0, 2^B - 1]$. If a share is valid, each server s only adds the first part ($\kappa' = 1$) of the encoding e_t, which is the share of sec, to its sum_s. After the servers treated all K received shares this way, they publish their sum_s and one of them (or their leader) executes Decode on input $\sum_{s=1}^{S} sum_s$. Since the result is already the sum of all secrets, Decode only has to divide its input by the number of shares which is K. Again, the proof of the correctness of the used sum-AFE (which is equal to the avg-AFE except that Decode does nothing) is given in Appendix B.2. Hence, this evaluation function can be correctly computed with Prio and thus admins can use it to compute the average of downloaded bytes.

Max_Download_Size() This function takes all values from `session_bytes` (from a given period) as input and outputs their maximum. For this, each client must send the size of its downloaded *session_bytes* after issuing a single request. Hence, the secret of each client is $sec_t = session\_bytes_t$. In order to find the maximum $\max_{1 \leq t \leq T}(session\_bytes_t)$ of all secrets, Prio's max-AFE must be used to encode each sec_t [CB17]. We assume that we already defined an upper border B that defines the largest size of a download. The Encode algorithm of the max-AFE creates encodings of the secrets by representing each sec_t in unary. Thus, Encode(sec_t) $= (b_0, b_1, \ldots, b_{B-1})$ such that the first b_0 to $b_{sec_t - 1}$ are 0 and all b_{sec_t} to b_{B-1} are 1. The servers then use the bitwise and-AFE to aggregate the shares. Hence, each 1 in the encoding is replaced by a string 0^n and each 0 is replaced by a random string in $\{0,1\}^n$ with $n \in \mathbb{N}$ being one of the public parameters. The Valid algorithm does always accept because Encode outputs randomly

[16]Since we do not need databases containing log entries for Prio there are also no columns with data in it. Nevertheless, we use the names of the columns in $\mathcal{CO}$ to identify which kind of data is analyzed.

[17]We can find the upper border B of all those session_bytes with the AFE for finding the maximum. But that AFE also needs an upper border B as input. Hence, it can be complicated to find B. This can e.g., be done by regarding previous maximums and adapting B to them.

chosen strings. Thus, every binary share is accepted. Then, bitwise xor is used instead of sum to produce the shares, update the servers' *sum*s, and compute $aggregate = \bigoplus_{s=1}^{S} sum_s$. The servers run the Decode algorithm with input *aggregate* and the algorithm translates each string only containing 0s to 1 and else to 0. The resulting string $(b'_0, b'_1, \ldots, b'_{B-1})$ is the AND of all encodings and its smallest j with $b'_j = 1$ is the maximum $\max_{1 \leq t \leq T}(session\_bytes_t) = j$ [CB17]. Therefore, with this evaluation function, admins and data analysts can find the largest downloads.

Number_Requests() This function takes as input a *device_id* and outputs the number of requests that were issued by the specified device (in a given period). Using the previous approaches, we had to run this function for each single device_id, but with Prio we can evaluate it for a small value set $V \subseteq \mathcal{V}_{device\_id}$ at once. For significant large sets of e.g., device_ids, this application is inefficient, thus an approximate variant is explained in [CB17]. For this, Corrigan-Gibbs and Boneh adapt the data structure *count-min sketches*[18] [MDC16] to the existing Prio protocol. Nevertheless, for our purposes, the simple version efficiently covers all sizes of sets that are reasonable in the context of device_ids, thus $V = \mathcal{V}_{device\_id}$. To realize this function with Prio, we use the AFE for frequency counting that is described in [CB17]. After sending a request, a client has to encode its secret $sec = device\_id$, which represents the currently used device, as follows: Given a vector $\vec{v} = (v_0, v_1, \ldots, v_{|V|-1})$ that represents all valid device_ids, a client sets $\vec{v}$ to be zero everywhere except a single v_i. That value $v_i, i \in \{0, 1, \ldots, |V| - 1\}$ represents its secret *device_id* and is set to 1. Then, for each request, a client creates the S shares from its encoded secret Encode(sec) $= \vec{v}$ using additive secret-sharing. Moreover, each client needs to prepare the SNIP proof using Valid from the used AFE and sends a share $([\vec{v}]_s, [h]_s, [\alpha]_s, [\beta]_s, [\gamma]_s)$ to server s with $s \in \{1, 2, \ldots, S\}$. At the servers, the Valid algorithm of the AFE accepts a share if it only contains a single 1 and the rest is all 0. Then, the servers add each accepted share to their $\overrightarrow{sum}$ and output the resulting vector after receiving K shares. After that, all servers' $\overrightarrow{sum}$s are added again by one of the servers and the result is a vector that identifies which device_id issued how many requests (in the regarded period). Thus, the Decode algorithm does nothing since the output already contains the number of requests per device. The admins only need to store the mapping between device_id and its number of requests. Thus, Max_Download_Size() can be realized as well if we use Prio.

Most_Visited_Websites() To apply this evaluation function, we use the frequency counting AFE of Prio [CB17]. For this, each client prepares a vector $\vec{v} = (v_0, v_1, \ldots, v_{|W|-1})$ that represents with $W \subset \mathcal{V}_{target\_ip}$ all interesting websites from e.g., *Alexa top*

[18]Count-min sketches are two-dimensional arrays A that use one hash function H_j per row j to efficiently store and update the frequency of each possible value i.e., secret *sec*. To encode their *sec*, the clients increase the value $A_j[H_j(sec)]$ in each row j and column $H_j(sec)$ of A by 1. After the servers added all received shares of A, the minimum of all values $A_j[H_j(sec)]$ of all rows j is an upper border of the frequency count of *sec*. See [CB17, MDC16] for more details.

1000[19]. Except the value $v_i = 1, i \in \{0, 1, \ldots, |W| - 1\}$ that represents the website that has been accessed by the client, $\vec{v}$ must be all 0. Hence, the constructed $\vec{v}$ is the encoding of the client's secret using Encode. Then, each client creates the shares for $\vec{v}$ using additive secret-sharing on each v_i and sends a share $[\vec{v}]_s$ to server s. The Valid algorithm rejects all $\vec{v}$ that contain more or less than a single 1. Next, each server adds the shares it receives to its *sum* and publishes this after all K shares are processed. In the end, one of the servers sums the published *sum*s and achieves a vector that describes the amount of accesses to each website in $\vec{v}$ (in the regarded period). Since the result is already the result we expect when using Prio's frequency count, the Decode algorithm does nothing in this case. To find the most visited website, the admins must only find the largest value in this vector and output the e.g., URL it belongs to. Hence, using this implementation, the evaluation function outputs much more information than needed. In particular, it also outputs which website is visited how often, while admins might only be interested in the most visited websites. To solve this problem, the Decode function could be defined to output the i for the largest v_i in $\vec{v}$. Then, the admins must only look up the corresponding URL (or IP address). But still the number of visits of each website can be read by the executing Prio server, though we do not store it in the result storage.

Alternatively, to cover a larger set of investigated websites than W, we can use the extension of Prio called *Poplar*. This approach is introduced in [BBC$^+$21] and solves the heavy-hitters problem. If we use Poplar, we can find all URLs that are visited at least o times. It does investigate all possible URL strings and not only a predefined subset of most popular websites as before. This is more useful in our scenario, since the website that an infected device starts a denial-of-service attack against must not be one of those popular websites. To use Poplar, each client must first translate the target_url (or target_ip as long as it is consistent) into bits $b_i \in \{0, 1\}$. Then, the client produces a binary tree by following the given rules. The tree is zero everywhere except on a given path, which depends on the string's binary. If b_0 is 0 then the left child of the tree's node (starting from the root) is 1. Otherwise, if $b_0 = 1$, the right child is set to 1. This is done for each level of the tree until the last bit b_n of the secret and therefore the tree's last level is reached. Thus, such a tree represents an arbitrary string with n bit representation. For this tree, the shares are computed and sent to the Prio servers. To allow efficient transmission of huge trees, Distributed Point Functions [GI14] are used to transmit the non-zero path in the binary tree. For more details on this, see [BBC$^+$21]. The servers then sum the shared trees and their result is thus a tree that in its leaves denotes the numbers of accesses for each possible URL string. Since this tree is too large to completely traverse, Boneh et al. [BBC$^+$21] decided to truncate the branches of the tree that are smaller than o. Hence, they dramatically decrease the size of the resulted tree by each level's investigation. In the end, the servers' result is a set of websites that have been visited at least o times. Hence, the admins could

[19] `https://www.htmlstrip.com/alexa-top-1000-most-visited-websites`

use this extension to investigate a much larger set of visited websites. Moreover, depending on o is chosen, the admins receive several websites that have been visited suspiciously often and not only a single one. And again this function leaks more information than originally needed. Moreover, it outputs the frequency of each substring. Nevertheless, we can realize Most_Visited_Websites() using Prio.

Malicious_Source() This function takes as input a value *session_bytes* and should output a website where malware of *session_bytes* bytes has been downloaded. If we knew the download size *session_bytes* before the malware has been downloaded, we could ask all clients that downloaded files of that size to send the corresponding *target_ip* (or alternatively *target_url*) as their secret data. Then, we could compute the union of those secret with the corresponding AFE described in Prio ([CB17]) to output all suspicious websites. But in the original setting, this evaluation function should find the results for a single infected device with *device_id*. This is also more useful as the websites with the corresponding size must not be malicious, especially if the devices that visited those were not infected. Therefore, we are only interested in a single client's input and we also want to output the union of all secret data that is sent by this client. Specifically, we could demand the client that uses the device with *device_id* to send all requests' e.g., target_ips that downloaded files of size *session_bytes*, to the Prio servers. Those servers could then perform the union-AFE of Prio and output a set of all target_ips that were requested by the device and caused a download of size *session_bytes*. But nevertheless, this construction does not work since we do not know the *session_bytes* before the malware is detected on a device. Hence, when we find the malware and want to do the evaluation with Prio, the necessary data is already lost since it is not stored. Moreover, this construction can neither ensure *Anonymity* since the issuer is always the same, nor *Unlinkability* because all secret data is from the same issuer, nor *Unreadability* as the secret data of the client is shown in the result as set of all secret data. Hence, it would destroy the whole idea of Prio if we realized this evaluation function as described above. Thus, Malicious_Source() is not realized using Prio.

Infected_Devices() This evaluation function takes as input a *target_ip* or *target_url* and should output the number of devices that accessed the therefore specified websites in a predefined period. Given a malicious website, we could let the clients vote whether they accessed this website or not. Hence, each one would only need to send a bit 1 for *accessed* and 0 otherwise. Then, the servers could use the sum-AFE to compute the sum of all 1s, and hence output how many devices could be infected due to visiting this website. But for this, we need the knowledge of the malicious website. If we use votes from current requests after we have found the malicious website, we could as well add the website to our blacklist to block all further accesses. Hence, if we want to learn which devices visited this website a long time ago, we need to store the requests' target_ip*s* or target_urls. Then, the clients would have to send votes for all their stored requests in hindsight. Thus, we would need a storage for that data to compute this evaluation function correctly and usefully. Since a storage or log files are not part of the Prio approach, we cannot realize Infected_Devices().

On_Purpose() This evaluation function takes as input a *log_id* and should output the corresponding value in `status_code`. As log files could be recognized by their unique log_id this was useful to find the necessary log entry in the database. Otherwise, in Prio there exist no log files anymore and thus the *log_id* cannot be used to find any data. Alternatively, a combination of datestamp, timestamp (and maybe one more) could be unique for all issued requests and be used instead to find the needed request. Then, there would remain the problem that Prio only analyzes live data. But, we only know the request that should be investigated after it has been issued. Thus, its status_code is no longer available. Moreover, we would have to output the status_code of a single request, which would be the secret data. Hence, after summing the *sum*s, the executing server learns the secret data of the single participating client, which is against Prio's zero-knowledge requirement. Due to these aspects, we do not realize On_Purpose() with Prio.

Malicious_Issuer() In our scenario (Chapter 3) this function should take a *device_id* and a *target_ip* (or *target_url* if given instead, as well as the password) as input. Then, it should output the *user_id* of the client i.e., employee that issued the specified (malicious) website. If we knew the malicious website's *target_ip* when the website is issued and the malware downloaded, we could execute the following: We could ask the client to encode its $sec_t = user\_id_t$ with a vector $\vec{v}$ that represents all user_ids. For this, it must only set the corresponding v_i to 1 and the rest of $\vec{v} = (v_0, v_1 \dots, v_{|V|-1})$ to 0 (for all device_ids $V = \mathcal{V}_{device\_id}$). Then, using the sum-AFE, the Prio servers could compute the aggregate of all those client-provided vectors. The result would be a vector that represents with $v_j > 0, j \in \{0, 1, \dots, |V|-1\}$ the user_ids of clients that might have downloaded the malware. But as already said, we need the *target_ip* before the malware is downloaded to realize this idea. Since that is not possible, those values must be stored, which is not contained in Prio. And to observe whether the guilty employee accesses the malicious website with the same device again, is not useful. Since the malware is already downloaded, that employee would beware to repeat that suspicious action. Thus, we can also not realize Malicious_Issuer() with this approach.

Period_Preselection() Prio cannot output whole log entries since it only outputs aggregate data. But we could define a different version that is similar to Period_Preselection(): Additionally to the secret data, each client also gives its datestamp and timestamp as input. Those should be part of the encoding. And the Valid algorithm verifies whether the given secret data is inside the specified period. Then, the servers would only aggregate shares from the correct period. Nevertheless, the clients can only send live data, as they do not store any additional data. Thus, if the Prio servers asked for another period than the current one, the clients would send and produce the shares and proofs, but all of them would be rejected by Valid. Hence, to allow the clients to send secrets from previous periods, we would have

Table 5.1: Functionality Grading for Prio

Evaluation Functions	Prio	Comments
Avg_Download_Size()	✓	with avg-AFE
Max_Download_Size()	✓	with max-AFE
Number_Requests()	✓	with frequency-count-AFE
Most_Visited_Websites()	✓	with frequency-count-AFE or Poplar
Malicious_Source()	×	needs storage
Infected_Devices()	×	needs storage
On_Purpose()	×	would injure zero-knowledge
Malicious_Issuer()	×	needs storage
Period_Preselection()	×	not useful here
Result	4 P.	

to store the data, which is not part of Prio. Moreover, it would be useless to start Period_Preselection() for the current period since per construction the clients are only able to send current requests' data to the servers. Therefore, there is no need to ensure that secrets originate from the current period. Thus, Period_Preselection() is unnecessary and not realizable as long as there is no older data stored.

A note on Prio's max-AFE In the original paper [CB17] the description of max-AFE is missing detailed information on how the shares are constructed. Those are important since that AFE depends on binary shares such that additive secret-sharing would not work. Moreover, the paper demands to use or-AFE for realizing the max-AFE, though it only works if and-AFE is used instead. Therefore, we had to figure out which construction is needed to create a working max-AFE. Thus, we realize max-AFE using and-AFE. Moreover, we use the xor function to create the shares from binary secrets[20]. Thus, for all secrets sec, it holds that $sec = \bigoplus_{s=1}^{S}[sec]_s$. Since only or-AFE is explained in detail in [CB17], we also figured out how to change Encode to realize the and-AFE. For this, we adapt Encode to map each 1 to a string of 0s and each 0 to a binary string that is chosen uniformly at random. The Valid algorithm still does nothing in this case, as all random binary strings are possible and valid. For the Decode algorithm, we map each string that only contains 0s to 1 and everything else to 0. That the previously described construction works correctly, is shown by our implementation of the max-AFE[21].

[20] Later on, we discovered that the xor function is also used to create shares for binary secrets in Prio+ [AGJ+22].

[21] `https://github.com/anliko/prio_max_afe`

Privacy Grading

In this section, we analyze the privacy we can preserve with Prio. Hence, we discuss which of the defined privacy goals from Section 3.4 can be ensured. For this, we regard the adversaries described in Section 5.1.2. We again consider adversaries that can read the traffic of the company in *Unreadability*, *Anonymity*, and *Unlinkability*. Those adversaries that gain the privileges of admins and data analysts or learn the aggregates by controlling the (leader) server are regarded in *Admin-Privacy*. In fact, Prio is able to preserve all our privacy goals. The whole discussion can be read in the following paragraphs and Prio's privacy grading table is Table 5.2. Additionally, do a brief, separate privacy grading for the second variant of Most_Visited_Websites() using Poplar. This is useful since Poplar is based on the same idea but still different to Prio and also ensures other privacy and security properties. That grading can be found in Section 5.1.4

Unreadability We could not ensure *Unreadability* for Prio if a secret sec could be read (i.e., learned) by any client or adversary except its owner. Thus, the first opportunity to see such a secret is while it is send to the Prio servers. Since any actor that eavesdrops all send shares can easily reconstruct the secret, this would injure the privacy of the transmitting client. But shares are only sent via a secure channel that is constructed by e.g., TLS. Intuitively, it would be hard for each eavesdropping actor to reconstruct the secrets from the encrypted shares. Thus, the phase of transmission is considered to be secure and private. The next privacy issue would occur if the secret could be reconstructed and read by the Prio servers. As already mentioned in Section 5.1.2, Prio can ensure f-privacy respectively zero-knowledge. This means that it is hard for every adversarial actor to reconstruct the secrets of the clients given the result that is output of the Prio servers. Or put more formally, given the result of aggregation function f on input secrets $sec_t, t \in \{1, 2, \ldots, T\}$, a simulator Sim can execute the servers' steps of the Prio protocol without knowledge of the secrets sec_t. Additionally, the simulated credentials have the same distribution as in the execution that is done by the malicious servers. Thus, as long as at least one server executes the protocol correctly, the malicious servers cannot reconstruct the secrets given their shares. Therefore, each server only learns the shares which are chosen uniformly at random ($[e_t]_s \leftarrow \mathbb{N}$ or $[e_t]_s \leftarrow \{0,1\}^n, n \in \mathbb{N}$) and thus reveal no information about a secret. Additionally, the servers are not able to learn the secrets from the result as long as Prio is f-private for the aggregation functions we used. Those are avg, max, and frequency count and according to [CB17] all of them are f-private. Nevertheless, if we use the aggregation functions sum, average, or maximum, we need at least two shares as input. Otherwise, the result would be the secret itself[22]. Then, any malicious leader server, admin, or data analyst, knowing that there was only one secret, can learn this by reading the result. But as the number of evaluated shares is always $K > 1$, we avoid those

[22] An interested reader might recognize the relation to zero-knowledge proof systems. In particular, the protocol is witness indistinguishable in that case, but not zero-knowledge. Those who are not yet familiar with zero-knowledge proof systems might be interested in [FS90].

issues. On the other hand, some aggregates as set union reveal all secrets because it is their purpose to output them[23]. Since we do not need that aggregation function, this does not influence our privacy grading[24]. As a result, we can ensure *Unreadability* if $K > 1$ and there is at least a single trustworthy Prio server. And both conditions are already part of how Prio should be constructed.

Anonymity As is already proved by Corrigan-Gibbs and Boneh in [CB17], Prio ensures *Anonymity* if the used function is f-private and symmetric (Section 5.1.2). If we regard our implementations of the evaluation functions, we make use of Prio's avg-AFE, max-AFE, and frequency-count-AFE. According to their paper, all those AFEs are f-private. Moreover, we realize the functions Avg_Download_Size(), Max_Download_Size(), Number_Requests(), and Most_Visited_Websites() using Prio. For the both first of them, it does not matter in which order they receive their inputs since they only use mathematical operations sum, division, and max that are commutative. The evaluation function Most_Visited_Websites() uses the sum-AFE and it does not matter in which order we add the vectors that represent the device_ids. This holds for the vector-based realization of Most_Visited_Websites(), as well. Furthermore, in our scenario, the admins must not notify the clients to send their secrets since it is automatically done for each request. Otherwise, if the admins asked a small set of clients explicitly to send secrets, they would be known by the admins and *Anonymity* could not be ensured. Thus, if there is at least one trustworthy server and no sensitive data is output, we can ensure *Anonymity*. The condition of one trustworthy server is fulfilled, as it is a requirement for our approach and the zero-knowledge property of Prio ([CB17]). In addition, in the previous functionality grading (Section 5.1.4), no sensitive data is part of the Prio server's result. Thus, we can ensure *Anonymity* with Prio.

Unlinkability We could not preserve this privacy goal if it were possible to find connections between different secrets. This means recognizing equal secrets as well as secrets that originate from the same client. But those secrets are only seen by their owners. In particular, their shares are transmitted via the secure and authenticated channel that is produced by e.g., TLS. Intuitively, this implies that the transmitted shares are hard to decrypt and thus learn for every efficient adversary that observes the traffic. Since each share $[e_t]_s$ is chosen uniformly at random from $\mathbb{N}$ or $\{0,1\}^n$, comparing their ciphertexts even if deterministic encryption is used, reveals nothing about the secret or the share's issuer. A subset of malicious servers could reconstruct the secrets to find connections to other secrets or issuers. But since Prio ensures f-privacy, we can provide that the secrets of the clients cannot be reconstructed from the results. Hence, adversarial servers cannot learn

[23] In [Cel22] it is also discussed that a subset of the aggregation functions outputs more data than needed. One of them is variance, which is not used in this thesis and thus no risk for our scenario.

[24] In scenarios where such functions are needed, k-anonymity could be used to improve the privacy. Then, only a subset of all secrets that were sent at least $k \in \mathbb{N}$ times should be inside the result. This could be realized with frequency counting or Poplar.

the secrets and thus not compare them. And since the shares that are seen by the servers are chosen uniformly at random, connections to other shares are only coincidence. Thus, they also cannot reveal information about the secrets except an upper border if they know the length of the shares. Hence, with Prio, we can ensure *Unlinkability* which was not possible with any of the previous approaches.

Presumption of Innocence We have already shown that Prio is f-private (Section 5.1.2) for all evaluation functions we realize in Section 5.1.4. Due to that, the servers are not able to learn the secrets given the shares and aggregates as long as at least a single server is trustworthy. Thus, the admins are neither able to learn any of the provided inputs (i.e., secrets) nor their senders (i.e., the employees). Moreover, we can ensure *Unreadability* and *Anonymity*. As we never output sensitive data such as user_ids, we further avoid that the owner of any secret can be found. Thus, we cannot reveal a secret or the identity of a suspicious employee. But that is less a problem regarding that Prio ensures *Validity* and thus avoids malicious employees and adversaries from significantly influencing our resulting aggregate. As neither the secret data of innocent nor malicious employees can be revealed, we ensure *Presumption of Innocence*.

Admin-Privacy As already mentioned for the previous approaches, data analysts need the same privileges and knowledge as admins to correctly perform their work. Thus, both admins and data analysts are able to see the results that are stored within the result storage. Hence, they learn everything that can be concluded from those aggregates. Given that Prio is f-private (Section 5.1.2) they cannot learn the single secrets of the clients. Hence, depending on the used aggregation function of Prio, admins learn at most the aggregates of the secrets. But in the worst case, if the function is for example set union, all secrets are a visible part of the result and thus leaked. Moreover, if the other aggregation functions are only computed using a single input, then most results would as well reveal that secret. But we pick $K > 1$ and therefore we can ensure that the aggregation functions we use (i.e., average, maximum, and frequency count) do not output a secret as aggregate. One main advantage is that analysis with Prio is done by the Prio servers and not by the admins. Hence, once the evaluation functions that should be executed periodically are defined, the admins cannot misuse their privileges to injure the employees' privacy. Furthermore, in this case, admins do not own any additional knowledge compared to all non-privileged actors. Their sole advantage is the access to the result storage, which again does not help learning the secrets as Prio is f-private, union-AFE is not used, and $K > 1$. Thus, we can also ensure *Admin-Privacy* using the Prio approach.

Ephemerality Since Prio is realized without log files, none of the requests are stored. Instead, we store the results of the evaluation functions in the result storage. This includes the *session_bytes* and accessed *target_ip*s or *target_url*s but only as aggregated data. The admins and data analysts can permanently access the aggregates in the result storage, but regarding our functionality grading from Section 5.1.4

no sensitive data (defined in Section 3.3) is directly stored there. As already discussed for the previous privacy goals, the result storage stores only aggregates that contain no secrets since $K > 1$. Moreover, no secrets from sensitive data can be concluded from the stored data, as Prio is f-private. Therefore, Prio is private in the sense of our privacy goal *Ephemerality*.

Validity One might first think that we cannot ensure *Validity* since the servers never see the true secrets from the clients and can thus not check them. But thanks to Prio's usage of SNIP proofs and the Valid algorithm of the AFEs, we can indeed ensure *Validity*. For this, the SNIPs ensures soundness. Thus, all clients' secrets sec_t that are encoded incorrectly or lie outside a predefined range lead to $\mathsf{Valid}(\mathsf{Encode}(sec_t)) = 0$ with high probability. Then, those secrets are ignored by the servers and not aggregated to the *sum*s. Since SNIPs are also correct, all valid secrets sec_t that are encoded using the correct Encode algorithm and for which secret $sec_t \in \mathcal{V}$ holds, lead to $\mathsf{Valid}(\mathsf{Encode}(sec_t)) = 1$ for sure. Thus, with high probability the result of the aggregation function that is computed is only created using valid secret $sec_t \in \mathcal{V}$. Then, the result cannot be negatively influenced by out of range secrets e.g., unrealistically huge *session_bytes* that should cover the true maximal downloaded session_bytes. Hence, we can ensure *Validity* with Prio.

Privacy Grading of Poplar Poplar ([BBC+21]) is used in this thesis to realize the Most_Visited_Websites() evaluation function for a significantly larger set of possible secrets than with Prio. Thus, we need to discuss the privacy that is ensured by Poplar. First, it ensures robustness (respectively *Validity*) against malicious clients since it uses validity proofs as well [BBC+21]. Hence, if clients send secrets that are invalid, those are ignored with high probability. But they are still able to send incorrect secrets without being spotted. Similar to Prio, Poplar ensures privacy if at least one of the computing servers executes the protocol correctly. Hence, the servers only learn the aggregate $f(sec_1, sec_2, \ldots, sec_K)$ and none of the K secrets. Moreover, Poplar ensures completeness if all servers and clients are trustworthy [BBC+21]. This property means that the aggregate on input the clients' secrets can be computed correctly. As in Prio, we do not need to store data to realize Most_Visited_Websites() with Poplar. Thus, Poplar ensures *Ephemerality* in our scenario. On the other hand, Poplar does reveal more information than only the aggregated result. As the setting is the same as for Prio, the admins and data analysts have access to the result storage. Therefore, they learn the aggregated result, the secrets that are part of this result and the additionally leaked data. Thus, admins or data analysts also learn the prefixes of the most visited website(s) and how often those were accessed. The concrete leakage is computed in [BBC+21]. But according to Boneh et al. *Anonymity* is not injured by this additional output of information [BBC+21]. Therefore, the owner of any secret URL cannot be spotted and the data of a specific client cannot be revealed. Hence, *Presumption of Innocence* is ensured as well. Since each server only sees a binary tree with nodes containing random values, actors that see those shares i.e., (malicious) server and adversaries analyzing the network traffic learn nothing useful to compare those shares to each other. Only the final

Table 5.2: Privacy Grading for Prio

Privacy Goals	Prio	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	✓	-
Presumption of Innocence	✓	-
Admin-Privacy	✓	-
Ephemerality	✓	-
Validity	✓	-
Result	7 P.	

result can be used to e.g., learn which website is visited more often than other websites or which websites are nearly as often visited as each other. As this data can only be seen by the admins (or data analysts), only those actors can injure *Unlinkability.* Thus, *Unlinkability* can be ensured and *Admin-Privacy* not. Since Poplar outputs all secrets of URLs that are visited at least o times (o-anonymity), we can only partly ensure *Unreadability.* Thus, we can conclude that Poplar at ensures *Anonymity*, *Unlinkability*, *Presumption of Innocence*, *Ephemerality*, and *Validity.*

Result

Prio ensures all privacy goals, which is the best result we could hope for. But on the other hand, we can only realize 4 of 9 evaluation functions from Section 3.2. This is caused by Prio only using the live data that is provided by current requests. As Prio is specialized to compute statistical aggregates, all of our statistical evaluation can be implemented. In particular, those are the functions we need to find malware on a device, which is an interesting circumstance. All other evaluation functions are not realizable, as they need data from the moment the malware was downloaded and hence need storage to access those requests' data at a later time. Moreover, Prio only accepts integers (decimal or binary) as input, which complicates evaluations of strings. For those, Poplar can be used instead, though it suffers from huge proof sizes due to the binary-tree-shares.

Though Prio is the most privacy-preserving approach we investigated thus far, it also has some disadvantages and issues that should not be omitted. Firstly, Prio can ensure *Validity* (i.e., robustness in [CB17]) against malicious clients using the AFE's Valid algorithm and SNIP proofs. But they cannot protect from malicious servers. Thus, each malicious server can tamper the received shares arbitrarily without any party recognizing the changes. The reason we renamed that requirement from robustness to *Validity* is that Prio can only ensure that values from outside a predefined range are rejected. For this, it relies on the fact that each client sends exactly one share. Thus, if each client can only send one vote for a given *target_ip*, we can ensure that this vote is not influenced by malicious clients. But in our scenario we have to allow arbitrary many requests by arbitrary clients as we cannot ensure that each client sends exactly one request in the regarded period. Thus, to influence the result, a malicious client only has to send many shares with the same *target_ip* to e.g., push the number of visits to the maximum. Moreover, Prio is not able to check whether malicious clients use exactly the data from their requests because the sent shares do not reveal any information about the secrets. Additionally, Prio also requires that at least one server is trustworthy to ensure *Unreadability* i.e., zero-knowledge. Moreover, as already described in Section 5.1.2, Prio can only ensure robustness i.e., successful computation despite failing servers, if all servers are trustworthy. Otherwise, the construction would decrease the privacy Prio can preserve.

Furthermore, Prio is vulnerable against selective denial-of-service and intersection attacks [CB17]. The first ones can be executed by an adversary that blocks $T-1$ from T many clients. Then, this adversary pretends to be all blocked clients by sending $T-1$ secrets to the Prio servers. This kind of attacks where a single adversary pretends to be many clients (or other entities) is also called Sybil Attack [Dou02]. As the aggregate is computed with only one unknown input, the adversary can easily compute the secret of the remaining client given his own secrets and the aggregate. To avoid this attack, Corrigan-Gibbs and Boneh [CB17] recommend to e.g., let the servers check the number of participating distinct clients by instructing each client to sign its secret with a secret key sk. Hence, they can count the number of different participating clients given the different signatures. Then, the servers must only wait for some other clients to send their secrets. For our scenario, this means that the evaluation is slowed down to protect from those attacks and we need to ignore incoming secrets from already participating clients, which influences our results. Alternatively, the ignored secrets could be stored and send to a later time, which is descried for Prio & Log Files in Section 5.3. The second attack is the insertion attack which allows the adversary e.g., a malicious client to learn the secret of one of the participating clients. For this, the adversary first learns the aggregate $result_1$ computed from T secrets. After that, he interrupts the connection of client t and waits for the Prio servers repeating the evaluation with the same clients sending the same secrets. Given the result of the second evaluation ($result_2$ computed from $T-1$ secrets), the adversary can then compute the secret of the interrupted client. For example, if f is the average, the adversary computes $sec_t = T \cdot result_1 - (T-1) \cdot result_2$. The fact that our Prio approach only analyzes the live data is an advantage here, as the clients' secrets change for each evaluation. Thus, this attack does not work in our scenario.

Another disadvantage of Prio is that the SNIP proofs can be very large due to the usage of arithmetic circuits [Cel22]. Hence, the variants of Prio described in the following section target reducing the size of those proofs. In addition, in their paper [CB17] Corrigan-Gibbs and Boneh already suggest a different way to evaluate the arithmetic circuit which is more efficient, especially if many arithmetic circuits $AC_1, AC_2, \ldots, AC_N$ are used. For this, Valid outputs 0 for accept, which is in contrast to the previous design. But due to this, each server verifies $\sum_{i=1}^{N} r_i \cdot v_i = 0$ with $N \in \mathbb{N}$ many arithmetic circuits, $r_i \leftarrow \mathbb{N}, 1 \leq i \leq N$ and v_i being the output of AC_i. Thus, the sum of those server-side computations should be 0 if the client's input is accepted by all arithmetic circuits.

The issues of this Prio version are solved with its newer variants Prio2, Prio3 and Prio+. Those are shortly regarded in the following section. Another advantage of Prio is that it does not require public-key cryptography (except the cryptography used in e.g., TLS). Thus, it is easier to implement and faster than comparable approaches. Since Differential Privacy is important for Prio2 and Prio3, we discuss this in detail in Section 5.2. And to allow the realization of more than only four evaluation functions, we introduce Prio & Log Files in Section 5.3. For this, storing the missing data on the client-side is the solution to all problems that appear in this functionality grading (Section 5.1.4).

5.1.5 Extensions and related work: Prio2, Prio3, Prio+, VDAF, PPM

Due to the previously mentioned problems of the first version of Prio from [CB17], namely huge SNIP proofs or more data leakage than desirable, many variants of Prio were designed. Those are described briefly in this section.

Prio2 is an extension of the original Prio that was created by a cooperation of Apple and Google. Their version of Prio served tracking infections during the COVID-19 pandemic. In addition to the original construction, Prio2 uses Differential Privacy to randomize the secrets before they are shared with the Prio servers [AG21]. This further decreases the amount of data that is leaked in addition to each function's resulting aggregate. Moreover, they adapt the SNIP proofs to their dedicated purposes. As well as Prio, Prio2 only allows secrets that are numerical [Cel22]. Hence, Prio2 improves the privacy of Prio, but it does not allow any additional aggregation functions [AG21].

Prio3 uses a fully linear proof system to reduce the size of the SNIP proofs that are necessary for Prio [BBC+19]. Since the arithmetic circuits that are used for Prio's SNIP proofs can become huge, smaller proofs can decrease the size of the shares and reduce the computation time for the clients. Then, also the servers profit as they do not need to verify the SNIPs. Moreover, this variant of Prio still allows only numerical secrets and thus can execute the same aggregation functions as the original Prio and Prio2. As this is the newest variant of Prio, the IETF currently discusses this construction as a candidate for the PPM standardization process[25].

Prio+ is another version of Prio that also tries to reduce the size of the SNIP proofs. For this, the authors use boolean secret-sharing[26] instead of the SNIP proofs [AGJ+22] for checking the shares' validity [27]. This results into easier computation tasks on the client-side. Nevertheless, for the aggregation functions except MIN, MAX,

[25]`https://www.ietf.org/archive/id/draft-gpew-priv-ppm-01.html`

[26]With boolean secret-sharing, the numerical shares are represented as binary and xor is used to create the shares. Prio's or and and-AFE are used similarly.

[27]But for some aggregation functions such as variance and linear regression, SNIP proofs are still necessary [AGJ+22].

AND and OR, the binary shares must be converted to additive shares, which is expensive. Regardless of the efficiency, Prio+ allows the same aggregation functions as Prio does.

VDAF This idea is based on an internet-draft published by the IETF[28]. Verifiable Distributed Aggregation Functions (VDAFs) is an interface that allows the computation of aggregates over distributed (and sensitive) user data in a way that none of the computing devices learns parts of or complete users' inputs. Hence, it also takes several clients' secrets as inputs and uses them to evaluate an arbitrary aggregation function. For this, the whole procedure is split into four different steps: *Sharding*, *Preparation*, *Aggregation* and *Unsharding*[29]. In the case of Prio this means creating the shares in *Sharding*, checking the SNIP proofs in *Preparation*, summing the shares to the servers' *sums* in *Aggregation*, and adding all *sums* in *Unsharding*. Additionally, the *Verifiable* in VDAF stands for the validity check that we already know from Prio. Thus, it can be seen that Prio and its variants are all elements of the large group of VDAF protocols.

PPM Additionally, the IETF is currently standardizing privacy preserving measurement (PPM)[30], which implements the VDAF interface. PPM targets to combine already existing approaches as Prio3 and Poplar to a general class of privacy preserving measurements. It has the same goal as the previously described extensions, to compute aggregates of (sensitive) user data without the computing servers learning those data. For this, secret sharing is used as well and the servers have dedicated roles. One role is the *collector*, which collects the encrypted measurements (in our scenario the metadata from the requests). All other servers are *aggregators* and in specific each of them is either *helper* or *leader*. In this setting, the leaders coordinate the execution of the protocol by instructing the helpers. Those helpers perform the actual computation of the aggregated shares.

5.2 Differential Privacy

Our next approach Differential Privacy is already briefly introduced in Section 1.1. We rely on [DN03, DMNS06] for the information we require in the following. Given these information, we discuss whether Differential Privacy can be useful for our scenario (Chapter 3). Differential Privacy describes the technique of obfuscating sensitive data to increase privacy. Put more formally, each secret data *sec* is added[31] with random noise $\varepsilon \leftarrow D$ that is chosen uniformly at random from a distribution D e.g., binomial, gaussian, or Laplace. Hence, the resulting data $sec^* = sec + \varepsilon$ is only an approximate value for the true data *sec*. This complicates the computation of the evaluation functions, since the larger

[28] `https://www.ietf.org/archive/id/draft-irtf-cfrg-vdaf-03.html`
[29] See Footnote 28
[30] `https://www.ietf.org/archive/id/draft-gpew-priv-ppm-01.html`
[31] This is possible for each type of data, if we represent it as e.g., binary and then add the also binary noise to it.

we pick ε, the more incorrect becomes the result. But since the data is obfuscated, this also ensures some privacy for the employees. We can apply Differential Privacy on each input of an evaluation function $f(sec_1^*, sec_2^*, \dots, sec_J^*), sec_j^* = sec_j + \varepsilon_j, \forall j \in \{1, 2, \dots, J\}$ or on its output $f(sec_1, sec_2, \dots, sec_J) + \varepsilon$.

If we used Differential Privacy as a stand-alone approach, we would store all data in the `Log_Files` database as described in Chapter 3. But this time, each cell stores the obfuscated data $\mathsf{entry}_{row}[col]^* = \mathsf{entry}_{row}[col] + \varepsilon_{row,col}, \varepsilon_{row,col} \leftarrow D$ to improve the privacy of the stored data. Given these data, we could compute our evaluation functions as already described in the basic approaches in Chapter 4. Thus, we could compute the maximum and average of all obfuscated values in `session_bytes`. But as each cell is added with its own noise (i.e., ε depends on *row* and *col*), this construction is similar to randomized encryption. Thus, we cannot compare different *device_id*s to compute the number of requests for a device, since equal *device_id*s are mapped to different *device_id**s. Moreover, 6 of 9 evaluation functions cannot be realized if we cannot compare the obfuscated values to each other. Therefore, we must pick a fix ε for each new value (or each column *col*) such that Differential Privacy becomes deterministic. Only then, we can compare the obfuscated data to realize the evaluation functions as already shown in the basic approaches (Chapter 4). This construction does not ensure *Ephemerality* as the data is still stored with access for all employees, admins, data analysts, and adversaries with access to the company's system. Furthermore, *Unreadability* cannot be ensured as values in the columns `session_bytes` and `status_code` are too close to the original data if ε is small. Especially, the values in `status_code` are easy to reconstruct since only few values (e.g., 100-103, 300-308[32]) are indeed used. As Differential Privacy must be deterministic to enable the evaluation functions, *Unlinkability* is again injured as well. In some cases, the noise that was added to the inputs can be removed from the result to achieve the true results [DN03]. This would improve the correctness of the evaluation functions, but then, we could not ensure *Admin-Privacy* as all privileged actors can remove the noise and learn the secret data. Thus, Differential Privacy as a stand-alone approach does either allow few evaluation functions or injures half of the privacy goals. Additionally, the more we try to improve the privacy, the more we increase ε and the less accurate is the result of the evaluation function. Hence, by now, Differential Privacy is no improvement compared to our previous approaches.

Due to these findings, Differential Privacy can only improve the privacy for statistical evaluations and not for *look-up-tasks*. That is exactly the kind of evaluation functions, we realized with Prio. Hence, Differential Privacy can be used to improve the privacy of Prio by obfuscating the aggregated results. This is already recommended in Prio [CB17] and realized by Prio2 and Prio3. For some aggregation functions, Prio leaks more data than necessary. One of them is the variance aggregate, which also reveals the expectation [CB17]. This additional leakage can be reduced by obfuscating the result of Prio with random noise $\varepsilon \leftarrow D$. Furthermore, in Section 5.3 we discuss the problem of the union-AFE revealing all secrets, as they are the desired output. If this result was obfuscated with ε, we could increase the privacy, since only approximate secrets

[32] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`

would be leaked. But this only works if each element in the resulting set is obfuscated separately. Otherwise, if the set was perturbed as a whole, the result could become useless. In Section 5.1, we discussed selective denial-of-service attacks where all but one client are blocked such that the adversary can learn the remaining client's secret from the aggregate. If the aggregate is additionally obfuscated, the adversary could not compute the client's secret correctly. Moreover, if the clients added noise to their secrets before executing the Prio protocol, they could protect from malicious servers. Thus, if all servers were malicious and reconstructed the secrets, they would only learn the obfuscated secrets due to the Differential Privacy. Since Prio already ensures some privacy, the amount of noise ε that must be added to the result can be small enough to still allow correct evaluations. As a result, Differential Privacy is not recommended as a stand-alone approach in our scenario but works very well in addition to Prio. Combined, they improve Prio's privacy and complicate the selective denial-of-service attack.

5.3 Prio with Log Files

The approach Prio was already very successful with regard to privacy. Therefore, with Prio & Log Files we want to realize the evaluation functions (Section 3.2) that could not be realized with Prio. Those successful functions are all executed periodically to detect malware or running attacks. But the remaining functions must be executed on demand, when malware has been found on a device. Due to this, we use Prio as before and use the information of Prio's construction from the paper of Corrigan-Gibbs and Boneh from 2017 [CB17]. But in the following, we extend this protocol to enable the remaining evaluation functions and verify the received secrets. For this, we slightly adapt the scenario from Section 5.1.3 and analyze the functionality and privacy of the resulting approach in Section 5.3. The main difference to Prio is that the required data is now stored by the clients. Thus, Prio & Log Files allows the evaluation of the remaining functions while preserving nearly as much privacy goals as Prio.

Application

For this approach, each client locally stores a reduced log entry with columns $\mathcal{CO}' = \{$`target_url`, `target_ip`, `session_bytes`, `status_code`, `datestamp`, `timestamp`$\} \subset \mathcal{CO}$. Additionally, each device knows its ID (`device_id`) and the currently logged in employee (`user_id`). Hence, the clients must not store sensitive data in their log files and are also the only entity that stores this data permanently. In addition, their log files can be encrypted such that the clients' data is collected in a privacy-preserving way. When malware is detected, the admins (or data analysts) start the evaluation functions. For this, the public parameters of the required AFE, its parameters and the kind of needed input data are distributed to all clients. Thus, only if asked, each client accesses its stored data, picks the necessary values from it and sends them as secrets to the Prio servers. For each of the evaluation functions we want to realize, we output a set of secrets as a result. Unless Period_Preselection(), which is only an extension of the Valid algorithm that detects secrets from invalid periods.

Correctness of the Clients' Log Files Prio already ensures validity checks, but those cannot verify whether a client really e.g., visited the website it pretends to. Thus, for malicious clients, it would be easy to store and send wrong data to the Prio server to avoid being caught. Hence, we introduce the following construction to avoid that malicious clients can send tampered, out-of-order secrets. For this, we re-introduce the log file server. This receives all clients' requests (via a secure, authenticated channel e.g., TLS) and constructs the reduced log entries. Additionally, each client has a public key pair (pk, sk) and the log file server encrypts each log entry[33] with the corresponding client's public key pk. Moreover, we want to prevent malicious clients from decrypting, tampering, and encrypting ciphertexts in their log files. To do so, the log file server stores all accessed *target_ip*s, *target_url*s, and all existing *user_id*s as digests in the *digest storage*. For this, the hash function H from our hashing approaches (Section 4.1) can be used[34]. Then, given a result of Prio, the admins (or data analysts) hash all elements from the provided set and remove all elements whose digests are not stored in the digest storage. This provides that malicious clients could not change their secrets to arbitrary data. Nevertheless, we do not store which digests belong to which client to not injure *Anonymity*. Thus, malicious clients are still able to change their secrets to not out-of-order data. For example, those clients can replace their *target_ip*s by other IP addresses they (or other clients) accessed. On the other hand, the set of all existing user_ids is only stored in the `Users` table and thus only known to admins and data analysts. Due to this, we can ensure that a malicious client cannot replace its *user_id* to hide that it accessed the malware's website. Hence, invalid and false out-of-order secrets cannot influence our final results. Additionally, we could secret share the secret key sk of the client such that client and log file server each own a share. Then, the clients could only decrypt their data when they are allowed to i.e., the log file server uses its second share. This further complicates tampering secrets for malicious clients. Without this construction, we must trust that each client correctly creates its log file entries from the requests. Thus, as it becomes feasible if we store the data and provides more trust in the correctness of the client-provided secret shares, our construction is useful for Prio & Log Files.

Functionality Grading

In this section, we describe how the remaining evaluation functions can be realized with Prio & Log Files. Hence, we describe which kind of data and which AFE are needed to achieve the results we defined in Section 3.2. Thereby, some evaluation functions are realized slightly differently than defined. This is necessary as the original functions are

[33]In particular, each cell is again encrypted separately, such that the clients can pick any ciphertext of their log file and construct shares of it without need to decrypt ciphertexts from other columns.

[34]Using a different hash function H_t (or different key k_t) for each client t could increase the privacy of the employees. But this would lead to the problem that we must know which client contributed which secret to correctly perform the checks. And since Prio ensures *Anonymity*, this cannot be guaranteed.

designed for centralized log files only. Then, we can realize all the remaining functions with Prio & Log Files, which is also shown in Table 5.3.

Malicious_Source() This evaluation function allows us to find all *target_ip*s or *target_url*s where downloads of size *session_bytes* have been downloaded. We expect that one of them is the source of the *session_bytes*-sized malware. Thus, we distribute the *session_bytes* as public parameter to start this function. Then, each client t should send the *target_ip*s (or *target_url*s) of requests that caused downloads of that specific size as its secret sec_t. Using the union-AFE the servers could compute the union of all client-provided websites. But the union-AFE is not efficient for the amount of possible secrets we expect here [CB17]. And the announced approximation of the union-AFE cannot be found in the Prio paper[35]. Thus, we can only enable the analysis of a small subset of all possible websites $W \subset \mathcal{V}_{target\_ip}$. We cannot ensure that the malicious source is inside this subset. But given W, each client prepares the encoding $\mathsf{Encode}(sec) = \vec{v} = (v_0, v_1, \ldots, v_{|W|-1})$ and for each website it accessed with download size *session_bytes* it adds a 1 to the v_i that represents the *target_ip*. Thus, in this case, several v_i can be 1 and all other $v_j, j \neq i$ are set to 0. Since union-AFE is realized using the or-AFE, each client additionally encodes $\vec{v}$ a second time. For this, each 0 in $\vec{v}$ is mapped to 0^n and each 1 to a randomly chosen string in $\{0,1\}^n$, where $n \in \mathbb{N}$ is a public parameter. Valid does nothing, since all arbitrary strings in $\{0,1\}^n$ are allowed. The servers aggregate the shares of $\vec{v}$ using the xor-function. Then, with Decode each string 0^n is mapped to 0 and all others are mapped to 1. Thus, every 1 in the resulting vector represents a website's *target_ip* that caused a download of size *session_bytes*. As a result, the admins could find websites that could be the cause for detected malware with this realization of Malicious_Source(). But in practice there could occur the following problems. First, it would be more successful to only take secrets from infected devices, since those accessed the malicious website for sure. But we cannot know whether other, not recently investigated, devices are infected as well. Additionally, only communicating with a small subset of devices could injure *Anonymity*. The other problem is that an infected device could as well deny participating in the evaluations. Since we ensure *Anonymity* with Prio, we can neither detect them nor find the source of this malware without the device's help.

Infected_Devices() To realize this evaluation function with Prio & Log Files, the malicious website's *target_ip* or *target_url* is part of the public parameter and distributed to all clients. Further, we have two options to implement it. Firstly, each client checks whether this *target_ip* or *target_url* is inside its locally stored log file and sends 1 if it is inside and 0 otherwise. Then, using the sum-AFE of Prio, the servers can compute how many clients accessed the specified malicious website. The second option is to also include the *device_id*s of the clients to additionally

[35]We could instead use the approximate frequency count AFE with only regarding whether the frequency is 0 or 1. But as the used data structure only outputs an upper border, it could be hard to tell whether the true count is 0 or 1.

learn which devices are infected. For this, the or-AFE is used. At first, a vector $\vec{v} = (v_0, v_1, \ldots, v_{|V|-1})$ encodes all possible *device_id*s in $V = \mathcal{V}_{device\_id}$. If a client has stored a request that issued the specific website (in the given period), it sets $v_i, i \in \{0, 1, \ldots, |V|-1\}$ to 1, where i represents its *device_id*. Every other $v_j, j \neq i$ is set to 0. Otherwise, $\vec{v}$ should be all zero, because not sending a share could also reveal information about an employee. Next, each client additionally encodes $\vec{v}$ with the Encode algorithm of the or-AFE. Thus, each 0 is replaced by 0^n and each 1 by a randomly chosen string in $\{0,1\}^n$ with public parameter $n \in \mathbb{N}$. After that, the client creates the S shares by using the xor function and also creates the SNIP proofs for the servers. The Valid algorithm accepts every binary, since arbitrary strings are produced by Encode. After the aggregation with xor, a server runs the Decode algorithm on the aggregated secrets, which maps each string 0^n to 0 and the others to 1. Thus, the result is a vector that contains a 1 in each v_i if the client with the ith device_id accessed a malicious website and could be infected. This second option does not only reveal the number of probably infected devices, but also which clients visited a given website and which not. As a result, we can realize Infected_Devices() with Prio & Log Files.

On_Purpose(), Malicious_Issuer() For these functions, the clients need access to the stored mapping of their *target_ip* (or *target_url*) and the corresponding *status_code*. As in the previous case, the malicious website's *target_ip* (or *target_url*) is part of the public parameters that are given to all clients. With usage of the or-AFE, the Prio servers can compute whether any of the clients ever accessed the specified malicious website (on purpose). For this, a vector $\vec{v} = (v_0, v_1, \ldots, v_{|V|-1})$ is used to encode all existing *user_id*s ($V = \mathcal{V}_{user\_id}$) of the company. Every client has to set the v_i that belongs to its *user_id* to 0 if the website that is asked for, has been accessed with status_code 3xx and to 1 otherwise. The rest of $\vec{v}$ is set to be all 0. If the client did not access the specified website at all, it should create $\vec{v}$ only containing 0s. Again, the clients use the Encode algorithm of the or-AFE. Given the public parameter $n \in \mathbb{N}$, Encode maps every 0 to 0^n and 1 to $\{0,1\}^n$ chosen uniformly at random. The shares are created using xor and the Valid algorithm does accept all binary encodings. When the servers finished the aggregation of all K shares with xor, one of them (or their leader) computes $\mathsf{Decode}(\bigoplus_{s=1}^{S} sum_s)$. Then, the result is a vector of all *user_id*s in $\mathcal{V}_{user\_id}$ where each v_i contains a 1 if the corresponding client i.e., employee accessed the specified malicious website on purpose. In the original scenario, On_Purpose() should reveal whether an issuer is malicious or innocent before revealing his identity, but this cannot be done with this realization. Hence, it would be better to only execute On_Purpose() to avoid revealing the identity of innocent employees. But, we cannot ensure the correctness of the *status_code* as it is no part of the secret. Nevertheless, we can realize both evaluation functions.

Period_Preselection() This function should ensure that the secrets that are sent by the clients really belong to the period that is asked for. Hence, to realize it, each

Table 5.3: Functionality Grading for Prio with Log Files

Evaluation Functions	Prio & Log Files	Comments
Malicious_Source()	✓	with union-AFE
Infected_Devices()	✓	with sum-AFE or OR-AFE
On_Purpose()	✓	with OR-AFE
Malicious_Issuer()	✓	with OR-AFE
Period_Preselection()	✓	as extension of Valid
Result	5 P.	

client must send the *datestamp* and *timestamp* in addition to its secret *sec* to the servers. If we do assume that sending that data to the servers in clear does not injure any privacy goals, we can send them additionally to each share. Thus, the servers can read that data and reject secrets outside the period before starting the validity check. Otherwise, adding the check of *datestamp* and *timestamp* to the Valid algorithm requires adapting Valid for each run of an evaluation function. But then, this function could for example, be executed by creating a vector $\vec{v}$ that contains the secret, the year, the month, and more as single entries v_i. The Valid algorithm must check whether the v_i (most likely from year and month) are inside a defined range. This way, we can define an extension that verifies whether secrets are really from the demanded period and can ensure it using the trustworthy *Validity* check system that is already part of Prio [CB17]. Moreover, since stored data is used as secrets, this evaluation function is useful for Prio & Log Files though it was not for Prio. But we cannot verify whether the sent secrets really originate from the specified period, as we do not store the necessary information. Still, we can realize Period_Preselection() with Prio & Log Files.

Privacy Grading

In the following section, we grade the privacy of Prio & Log Files. For this, we argue for each of the privacy goals that are defined in Section 3.4 whether it can be ensured. We regard the same adversaries as in Prio Section 5.1. Thus, they can control arbitrary many clients and up to $S-1$ from S servers. Moreover, they can read the sent requests and shares unless they are encrypted and can manipulate the transferred packets. In particular, *Unreadability*, *Anonymity*, and *Unlinkability* are investigated for not privileged actors as the employees of the company or passive adversaries that try to learn secrets by observing the traffic. Every malicious client or adversary that gained access to knowledge or privileges of admins injures the privacy goals as defined in *Admin-Privacy*. As a result, we can ensure 6 of 7 privacy goals. This can also be seen in Table 5.4.

Unreadability We denote *Unreadability* to be injured if any of the (sensitive) data defined in Section 3.3 is leaked. Thus, leakage could occur during the transmission of the shares, at the Prio servers which know the shares and the result, at the result storage that contains the results and at the digest server which stores all secrets. Since we assume the log file server to be trustworthy, we do not discuss the leakage of data that could be caused by this entity. Due to the usage of e.g., TLS, the shares are transmitted via a secure and authenticated channel. Hence, we assume that for every efficient (eavesdropping) adversary it is hard to decrypt and reconstruct a secret given the encrypted shares. Moreover, for Prio & Log Files as for Prio, we require that at least one Prio server is trustworthy. Thus, the remaining $S - 1$ servers cannot reconstruct the secrets of the employees given the shares they received and can thus not read a client's secret. Since the non-privileged actors cannot access the result storage, we can conclude that they can also not learn the results of the evaluation functions. Therefore, they do not learn the secret data, though it is a part of the results. And the last problematic entity is the digest storage. If employees or eavesdropping adversaries could learn the data that is stored there, *Unreadability* would be injured. But the log file server computes $H(sec, k)$ before sending the secrets to the digest server. As H should be one-way[36], malicious actors could not learn the secrets even if they had access to the digest server. Therefore, *Unreadability* can be ensured for Prio & Log Files.

Anonymity This goal is injured if any adversary or employee could find the employee that sent a specific secret *sec*. In the usual case, Prio does ensure *Anonymity* if the used aggregation function is f-private and symmetric. Further, this means for each evaluation function f on inputs $sec_1, sec_2, \ldots, sec_K$ and permutation π : $f(sec_1, sec_2, \ldots, sec_K) = f(sec_{\pi(1)}, sec_{\pi(2)}, \ldots, sec_{\pi(K)})$. Moreover, the AFE we use i.e., OR-AFE (and union which is based on OR) are f-private according to Corrigan-Gibbs and Boneh [CB17]. Thus, it exists a simulator Sim that can simulate the view of the Prio servers with a distribution that is computationally indistinguishable from the malicious execution of the protocol. Thus, given the security properties of Prio, we can ensure *Anonymity* as long as there is no other possibility to reveal the identity of the issuer. As this approach also outputs sensitive data, revealing an employee's identity becomes possible. But, this can only be done using the result storage, since all sensitive data is stored there. Since employees and the regarded adversaries have no access to that storage, those actors cannot learn sensitive data from the result storage to reveal identities. Thus, Prio & Log Files also ensures *Anonymity.*

Unlinkability We could not ensure this privacy goal if any client or e.g., eavesdropping adversary was able to find connections between different secrets. First, each client only stores its own secrets, thus finding connections to other clients' secrets is hard. When, the secrets are sent as shares via the secure and authenticated channel, those can only be reconstructed by an adversary that is able to decrypt the shares. But

[36]This could again be injured if the number of preimages was so small that computing all preimages' digests is feasible. Since we store *target_ip*s and *target_url*s which have large value sets $\mathcal{V}_{target\_ip}$ and $\mathcal{V}_{target\_url}$ this is no problem in our scenario.

we assume that this is hard for every efficient adversary as a secure authenticated channel is used for transmission. Alternatively, a malicious actor must control all the Prio servers to reconstruct and learn the secrets to compare them. Thus, at least one of the servers must be trustworthy and not controlled by a malicious actor to ensure *Unlinkability*. Otherwise, comparing the shares that a single server can see is not useful, as those are only picked at random ($[sec]_s \leftarrow \mathbb{N}$ or $[sec]_s \leftarrow \{0,1\}^n, n \in \mathbb{N}$) and can thus reveal nothing about the secrets. Last but not least, malicious employees and the regarded adversaries would need access to the result storage or digest storage to interact with the stored results and secrets. Since those privileges are not given to them and we only regard passive actors for this goal, they have no access to (foreign) secrets. As Prio assumes that at least one server is not under malicious control and the regarded actors cannot gain access to foreign secrets, we can ensure *Unlinkability* for Prio & Log Files.

Presumption of Innocence This privacy goal is important for this approach, since we are able to output results for Malicious_Issuer(). Hence, we are forged to leak sensitive data of employees. In particular, both evaluation functions Malicious_Issuer() and On_Purpose() could injure the privacy of the company's employees as they output *user_id*s. To though ensure *Presumption of Innocence*, we must guarantee that the output of those evaluation functions is correct such that the identity of innocent employees is not revealed. Due to the previously described construction Section 5.3, we can ensure that malicious employees could not provide the *user_id*s of innocent employees instead of their own. This is avoided since all *user_id*s stored in the `Users` table are kept secret and are thus only known to the admins (or data analysts)[37]. Therefore, we can ensure that an employee whose name is output of Malicious_Issuer() or On_Purpose() must be connected to the malicious action. As On_Purpose() is an extension of Malicious_Issuer() that only outputs *user_id*s of suspicious employees, we should only use On_Purpose(). Otherwise, Malicious_Issuer() might output `Users`s of innocent employees as well, which injures *Presumption of Innocence*. Due to this, we can ensure that none of the innocent employees' names are revealed and preserve *Presumption of Innocence*.

Admin-Privacy In this scenario (Sections 5.1.3 and 5.3), the admins (and data analysts) have access to the result storage using their login data. This means, that given any of the results, those actors can learn a huge set of the employees' secrets since each result is a collection of secrets. Moreover, for Infected_Devices(), On_Purpose(), and Malicious_Issuer(), admins or data analysts learn the secrets that were created from sensitive data i.e., user_ids and device_ids. Thus, Prio & Log Files injures *Unreadability* according to the performed evaluation function. We cannot avoid this leakage, since admins and data analysts need a way to learn the identity of malicious clients or employees. Moreover, if we forbid leaking the *user_id*s of em-

[37]To improve the secrecy of the user_ids, arbitrary strings could be used instead of integers. According to the way we realized Malicious_Issuer() and On_Purpose() those IDs must not be numeric but only map to an index of $\vec{v}$ such that IDs as strings become possible.

ployees we could not realize all evaluation functions. To also injure *Anonymity*, admins and data analysts only have to relate the sensitive data from the result storage to the data stored in `Users` and `Devices`. *Unlinkability* could as well be injured with access to the result storage. But admins and data analysts can only learn the secrets but no relationships between them as each secret only appears once. Thus, *Unlinkability* is still ensured against privileged actors. As a result, *Admin-Privacy* cannot be ensured for Prio & Log Files, as admins and other privileged actors can injure two of the already preserved privacy goals.

Ephemerality According to the definition of *Ephemerality*, this privacy goal is ensured as long as admins or data analysts cannot permanently access the employees' request data. Since for this approach we store the data on client-side, admins (or data analysts) should not be able to access it. Moreover, even if they were able to access the log files of the clients, they were still not able to learn their (secret) data within since this is encrypted using the employees' public keys pk. Hence, we can ensure *Ephemerality*, too.

Validity Again, we profit from the advantages of Prio. As described in Section 5.1.2, Prio guarantees soundness and correctness. Thus, each invalid secret $sec \notin \mathcal{V}$ (or wrongly encoded secret $\mathsf{Valid}(\mathsf{Encode}(sec)) \neq 1$) is detected with high probability. And each secret that fulfills $sec \in \mathcal{V}$ and $\mathsf{Valid}(\mathsf{Encode}(sec)) = 1$ is accepted by the Prio servers. Due to the security properties of Corrigan-Gibbs' and Boneh's Prio protocol [CB17], *Validity* is already ensured for Prio & Log Files given the AFE's Valid algorithm and the SNIP proofs. Furthermore, the construction we described in Section 5.3 ensures even more. In the original Prio setting, the employees could send arbitrary data as their secrets as long as those was inside a predefined range $\mathcal{V}$. But our construction limits the set of undetected tampered secrets to the set of secrets that originate from real requests. In particular, the storage of all secrets as digests $H(sec)$ ensures that data cannot be changed from data originating from a request to completely arbitrary data. Furthermore, since H is also collision resistant, it is hard for all malicious actors to send tampered secrets sec' that cause a hash collision in the digest server such that sec' would not be detected. Therefore, each malicious employee could only adapt his client to send secrets that were already sent by him or other employees to avoid being detected. But this, also means that malicious employees must know a $user_id$s to tamper their secrets in Malicious_Issuer(). As those IDs are kept secret, tampering the secret becomes hard. The same holds for the $device\_id$ that must be sent for the Infected_Devices() function. This means, that our application complicates the tampering of secrets, especially for the evaluation functions Infected_Devices(), On_Purpose() and Malicious_Issuer(). Thus, due to Prio, we can ensure *Validity* by Prio & Log Files and further improve it with our additional construction.

Table 5.4: Privacy Grading for Prio with Log Files

Privacy Goals	Prio & Log Files	Comments
Unreadability	✓	-
Anonymity	✓	-
Unlinkability	✓	-
Presumption of Innocence	✓	If Malicious_Issuer() is replaced by On_Purpose()
Admin-Privacy	$\times$	Admins learn secret elements and identities
Ephemerality	✓	-
Validity	✓	-
Result	6 P.	

Result

The advantages of the previously described Prio & Log Files approach is that we can analyze all remaining evaluation functions with it. Additionally, we can preserve 6 out of 7 privacy goals. Thus, storing the clients' secrets and revealing parts of them as result only impacts the privacy that must be preserved against privileged actors. All other privacy goals can be ensured, though.

Moreover, the log files that are stored by the clients only contain non-sensitive columns, namely (`target_ip`, `target_url`, `session_bytes`, `status_code`, `datestamp`, `timestamp`). This is possible since user_id and device_id are both known to the client and must hence not be additionally stored. Furthermore, since the clients store and encrypt their data themselves, their privacy is preserved as long as no adversary gains access to the company's system and the employees' login credentials. On the other hand, this approach depends on much trust to the employees. As those are able to pick and create their shares from data they store themselves, we need to trust that those stored data is indeed sent as secrets. If a malicious employee tampers the stored log entries, he could send manipulated shares to influence the result of the Prio server's analysis. But due to our construction that allows the log file server to see and store the secrets as digests, the impact of those actions can be decreased.

As the main principle is still based on Prio, the already described attacks are still feasible. The insertion attack where an adversary eavesdrops a result, blocks client t and eavesdrops the result again but this time without client t's secret becomes possible for Prio & Log Files. For Prio, it was impossible since Prio is only executed on live data and thus the Prio servers never receive the same K secrets from the same clients. But Prio & Log Files does store the data and thus it could be possible that the same function f is executed on the same inputs as before. But if we apply Differential Privacy to the aggregate results, the adversaries could no longer compute the exact secret of the client given the perturbed results. Thus, we can complicate this attack with Differential Privacy but we cannot completely avoid it. The other attack is the selective denial-of-service attack. For this, an adversary blocks all but one client, sends the $K-1$ shares himself and eavesdrops the resulting aggregate to compute the secret of the not blocked client. This attack can be avoided by using signatures. If each client must sign its shares with a secret key sk, the Prio servers could verify that a huge set of clients participates and therefore prevent that an adversary sends all $K-1$ secrets. Still, this means that the clients cannot send arbitrary many secrets. But as the evaluation is not live anymore, the clients could send a given threshold (e.g., $K \backslash T$ for $K \geq T$ and T clients) of secrets and the remaining secrets are sent during the next execution.

All in all, the approach Prio & Log Files is a successful way to realize the remaining evaluation functions from Section 5.1 with Prio. In fact, the idea of hashing the secrets to compare them to a set of results is similar to the approach we want to analyze next.

5.4 Private Set Intersection

In the following, we alternatively realize the evaluation functions (Section 3.2) that cannot be realized with Prio using our final advanced approach based on Private Set Intersection (PSI)[38]. This provides a way of comparing different sets of two parties without revealing the sets to each other. All information that are given regarding Private Set Intersection are taken from [DMRY09, KRS$^+$19, Wei21]. PSI ensures many of our privacy goals from Section 3.4 because the private data of the employees is not completely revealed to the admins or data analysts.

5.4.1 How Private Set Intersection works

In the setting of PSI participate two parties P and Q where each owns a secret set X respectively Y. The goal is to compute $X \cap Y$ without P learning Q's set Y and vice versa. The intersection can be learned by one party or both, depending on the scenario.

In fact, there exist many different ways to realize PSI. In [DMRY09] they use polynomials P to encode each $x \in X$ such that $P(x) = 0$ and encrypt its coefficients with homomorphic encryption. Other versions make e.g., use of (blind) RSA or El-Gamal encryption schemes. We avoid focusing on a specific variant to remain general in the following. But all those variants rely on the same basic construction that we briefly describe. Each party hides its set X with an algorithm we denote as Hide_k. It is useful if this algorithm differs for each pair (P, Q_ι) which can e.g., be done by picking the key $k \leftarrow \mathcal{K}$ uniformly at random. This key depends on whether PSI uses a hash function, an encryption scheme or another cryptographic approach. Then, these hidden elements $x^* = \mathsf{Hide}_k(x), x \in X$ and $y^* = \mathsf{Hide}_k(y), y \in Y$ can be exchanged and compared to learn the intersection $X \cap Y$.

[38]We only regard not yet realized evaluation functions as Prio already ensures all privacy goals and we can thus not find a better solution for them. Moreover, those realized evaluation functions need statistical analysis, which is better provided by Prio than PSI.

To illustrate this process, we exemplarily describe the variant using oblivious pseudo-random functions (OPRFs)[39]. Each party Q computes $\mathsf{Hide}_k(y_j) := F_k(y_j)$ with key $k \leftarrow \mathcal{K}$, pseudo-random function (PRF) F, and $y_j \in Y$. Then, Q sends all $F_k(y_j)$ to P. After that, both parties use OPRF to compute $F_k(x_i)$ for all $x_i \in X$ of P using the key k from Q. Thus, party P learns its $F_k(x_i)$ and can compare them to Q's $F_k(y_j)$. To learn the intersection $X \cap Y$, P collects all $\{x_i \in X | \exists y_j \in Y : F_k(x_i) = F_k(y_j)\}$[40].

5.4.2 Evaluation

Application

Again, we must slightly adapt our scenario from Chapter 3 to make it fit this approach. For this, the admins (or data analysts) are party P and each employee's client is party $Q_\iota, \iota \in \mathbb{N}$. Further, we want only the admins (or data analysts) P to learn the intersections $X \cap Y_\iota$. Moreover, this approach needs storage to store the necessary data from the requests. To store this data in a more privacy preserving way as considered in the basic approaches (Chapter 4), we again use the application we described for Prio & Log Files in Section 5.3. Hence, the log file server receives the requests and constructs encrypted log entries with columns (`target_ip`, `target_url`, `status_code`, `datestamp`, `timestamp`) that are stored by the clients themselves. Additionally, it stores digests of the sent data (*target_ip*, *target_url*) inside the digest storage. We recommend that for this, a collision resistant and one-way keyed hash function H is used. Then, the elements in $Z = X \cap Y$ can be hashed with H. All resulting digests $H(z), z \in Z$ that are not inside the digest storage are removed from Z. This way, we avoid that data, that was never part of a request and is thus obviously tampered, influences the results of our evaluation functions.

We additionally enable admins (or data analysts) to perform PSI on a huge set of private sets $Y_\iota, \iota \in \{1, 2, \ldots, n\}, n \in \mathbb{N}$, if they apply it round by round. At first, we need to define the set X with which the admins P begin the intersection. If it is impossible or inefficient to define the necessary set, the admins compute it as follows. For this, the admins (or data analysts) initiate PSI between two parties Q_1 and Q_2. The protocol is computed as usual, except that Q_1 and Q_2 sent their hidden elements to P. After that, the admins can perform the intersection to learn $X = Y_1 \cap Y_2$. In particular, they must query the clients on the hidden elements to obtain all $x_i \in X$[41]. It is more useful to query both clients, to avoid that a single client learns the complete intersection. Then, the admins can continue PSI by computing the intersection of their set X and the remaining $Y_\iota, \iota \in \{3, 4, \ldots, n\}$. After each round, P sets X to be the previous intersection. Thus, we use that intersection is associative i.e., $\bigcap_{\iota=1}^{n} Y_\iota = (\ldots((Y_1 \cap Y_2) \cap Y_3) \ldots \cap Y_n)$ to intersect private sets of n clients Q_ι. As the admins execute PSI with a single client per

[39] An OPRF is a pseudo-random function F that is computed by two parties P and Q. Q picks the key $k \leftarrow \mathcal{K}$ and P its input x. Then $F_k(x)$ is computed such that only P learns it. Neither k nor x is revealed to each other [KRS+19].

[40] More details on this kind of PSI are provided in [Wei21].

[41] In fact, P does learn $x_i^* = \mathsf{Hide}_k(x_i)$ for all $x_i \in Y_1 \cap Y_2$. But for the following rounds, it needs all x_i. As those are only known to the clients that computed $x_i^* = \mathsf{Hide}_k(x_i)$, P must query them on all x_i^*. Moreover, as P is learning the intersection of Q_1 and Q_2, x_i must be known to both parties.

row, they can relate the elements in X to the client they communicate with by analyzing the network traffic. Hence, we avoid sending sensitive data such that the admins (or data analysts) cannot learn them, too.

Functionality Grading

As before, we use this section to analyze the functionality of our approach PSI. For this, we describe how each evaluation function from Section 3.2 can be realized. And again, we need to adapt the inputs of some evaluation functions such that they can be realized in this setting. We can realize all five remaining evaluation functions without transmitting any sensitive data. This can also be seen in Table 5.5.

Malicious_Source() This function gets as input a value *session_bytes*. Then, it outputs target_ips or target_urls that caused downloads of that size. Hence, given a set of infected clients, they create their Y_ι with *target_ip*s (or *target_url*s if stored instead) that correspond to downloads of size *session_bytes*. For this, *session_bytes* must be distributed among all clients first. For X, the admins must pick a set of all *target_ip*s (or *target_url*s) which is too large. Hence, $X = Y_1 \cap Y_2$ is computed in the first round as described above. After that, all parties collaboratively compute the intersection $\bigcap_{\iota=1} Y_\iota, \iota \in \{3, 4, \ldots, n\}$ using the round by round construction from Section 5.4.2, Application. The result is a set of websites that could have infected the regarded devices. Therefore, this execution is only useful if all devices Q_ι are infected with the same malware i.e., the malware was downloaded from the same website. Otherwise, the intersection results in an empty set. Nevertheless, the admins or data analysts would still learn that there is more than one malicious website. Our problem is that we do not know all infected devices, as Malicious_Source() prepares the execution of Infected_Devices(). Thus, picking the right clients Q_ι is challenging but not discussed in more detail. As a result, we defined a realization for Malicious_Source() such that admins or data analysts can find sources of malware.

Infected_Devices() This evaluation function outputs the number of devices that accessed a given malicious website. For this, each client Q_ι adds all its *target_ip*s (or the *target_url*s if given instead) from its log file to Y_ι. On the other side, the admins' set X contains all known malicious websites[42]. Then, P and Q_ι execute PSI as described in Section 5.4.1 to compute the intersection of X and a single client's set Y_ι. If it holds that $|X \cap Y_\iota| \geq 1$, the investigated client visited one of the malicious websites and could be infected with malware. An advantage of this idea is that many malicious websites can be checked at once. On the other hand, Infected_Devices() must be performed for each client separately, such that the admins can learn which devices are infected. Since we only want Infected_Devices()

[42]In fact, it contains the malicious websites that should be analyzed. Websites that were already blacklisted a long time ago must not be further regarded here.

to output the number of infected devices, the admins should compute the intersection $X \cap Y_\iota$ without revealing which elements are inside. Told differently, if the admins find a hidden element y^* that equals any of their hidden elements x^*, they know that the investigated device is infected. Thus, they must not find out which x particularly belongs to x^*. This way, they learn whether the device is infected, but not which malicious websites were accessed by that client. Hence, there exists a way to realize Infected_Devices() with PSI.

On_Purpose(), Malicious_Issuer() With these functions, the admins (or data analysts) want to learn whether a given client accessed any of the known malicious websites (on purpose). Thus, each client picks its set Y_ι to contain all *target_ip*s (or alternatively *target_url*s) that were accessed (on purpose). For On_Purpose(), it only picks those that were accessed with $status\_code \notin \{300, 301, \ldots, 399\}$. And the admins' set X again contains all known malicious websites. Then, the parties P and Q_ι execute the PSI protocol such that P learns the intersection $Z = X \cap Y_\iota$. If $|Z| \geq 1$, the regarded client accessed at least one of those malicious websites (on purpose). We can see that On_Purpose() can be used as an extension of Malicious_Issuer(). An advantage of this construction is that several malicious websites can be checked at once. Nevertheless, the goal of our evaluation functions is only achieved if the admins can identify each client after they communicated. Only then, it is possible to learn the identity of a malicious issuer. For this, they could for example, examine the network traffic to find the IP address of the client they executed PSI with. Alternatively, the admins (or data analysts) define their set X to also contain all *user_id*s. Then, given that Y_ι also contains the *user_id* of the client of Q_ι, the intersection Z would reveal the party the admins communicated with. Therefore, this realization delivers the (malicious) issuers of downloaded malware given the admins.

Period_Preselection() To evaluate requests from a given period, the clients must be asked to only use data from that period. To check this, they must send the *datestamp* and *timestamp* along with the hidden set Y_ι. Then, the admins check that data before they execute the PSI protocol with Y_ι. But we cannot verify whether the provided *datestamp* and *timestamp* are correct, as only the clients store them. Nevertheless, we can realize this function with PSI.

Privacy Grading

In the following section, we analyze the privacy of our last advanced approach PSI. For this, we again use our privacy goals from Section 3.4 and discuss which of them can be ensured. Moreover, we consider efficient adversaries with access to the company's system that can thus see all sent data i.e., eavesdropping adversaries. As this also holds for clients, we consider both with the privacy goals *Unreadability*, *Anonymity*, and *Unlinkability*. Further, adversaries that control the party P, can injure privacy by executing PSI and learning the intersections. We analyze the privacy goals that are

Table 5.5: Functionality Grading for Private Set Intersection

Evaluation Functions	PSI	Comments
Malicious_Source()	✓	-
Infected_Devices()	✓	-
On_Purpose()	✓	-
Malicious_Issuer()	✓	-
Period_Preselection()	✓	-
Result	5 P.	

injured by those adversaries with *Admin-Privacy* because they own privileges of admins and data analysts then. In contrast, adversaries that control clients Q_ι can also influence the results but only by sending *target_ip*s (or *target_url*s) that were already accessed once, due to our construction in Section 5.3, Application. Moreover, each variant of PSI ensures different privacy properties. While all of them ensure that party P does not learn Q's private set and vice versa, some also ensure additional protection against malicious clients. But we only assume that the first of those properties is guaranteed because it holds for all variants. In fact, PSI can ensure almost 5 of 7 privacy goals. This result is also displayed in Table 5.6.

Unreadability We could not ensure this privacy goal if any client or adversary with access to the system could learn a secret set or parts of it. At first, each client can only access its own secret set, which does not injure privacy. But beyond this, a client neither learns the sets Y_ι of the clients it performs PSI with nor the admins' set X. This is provided by PSI, which ensures that the clients Q_ι do not learn each others' sets i.e., Y_ι and the set X of malicious websites. Moreover, we regard malicious clients or adversaries with access to the sent data by eavesdropping the secrets $\mathsf{Hide}_k(y)$. In particular, if they knew the key k, they could test for many elements y' whether $\mathsf{Hide}_k(y) = \mathsf{Hide}_k(y')$ holds to learn element y. But this key is only known to the parties that compute the intersection[43]. Furthermore, we can assume that the exchange of this key is done via a secure channel such that stealing it is hard. Additionally, only the admins (or data analysts) P learn the intersection i.e., subsets of the clients' secret sets. But only adversaries that manage to control P can use this to learn the secrets. As those already have the same privileges as admins, we consider this case for *Admin-Privacy*. Hence, there is no way for clients and the regarded adversaries to learn the intersections. Therefore, we can preserve our privacy goal *Unreadability* for PSI.

Anonymity We cannot ensure *Anonymity* if the malicious employees or adversaries we regard here can learn the identities of clients participating in PSI. In fact, we already saw that they cannot learn data from the execution of the protocol. Thus,

[43]In particular, only one of the parties executing the protocol knows the key k.

even if the clients sent sensitive data for PSI, the malicious actors could not learn enough from it to derive the clients' identities. As we only demand that *target_ip* or *target_url*, along with *datestamp* and *timestamp* should be sent and none of them is sensitive, this is no risk to *Anonymity*. Nevertheless, PSI is always executed pairwise. Hence, any eavesdropping adversary can detect which two devices are executing the protocol using a tool as Wireshark[44]. But as this is out-of-scope of this thesis, we leave such possible attacks for future work. As a result, we cannot ensure *Anonymity* since the participating devices can be located in the network[45].

Unlinkability We can ensure this privacy goal as long as the employees and adversaries with access to the company's system cannot find connections between different secrets. To learn whether some secret elements belong to the same client or two clients share a secret element, those malicious actors must learn the intersections. But as already discussed in *Unreadability*, the actors we regard here cannot learn them due to the security properties of PSI. Therefore, we can ensure *Unlinkability* against malicious clients and the regarded e.g., eavesdropping adversaries.

Presumption of Innocence This privacy goal is injured if we leak sensitive data of innocent employees. Firstly, the actors can only learn non-sensitive data from each client, as each one only sends its *target_ip*s or *target_url*s. But to enable Malicious_Issuer(), the admins must reveal the identity of the client they executed PSI with if the intersection is not empty. To do so, they learn the e.g., IP address of the other participant by analyzing the traffic. Due to this, we can only provide *Presumption of Innocence*, if this revealing of the IP address is only done for suspicious clients. As those can be detected with On_Purpose(), we should only execute this function instead of Malicious_Issuer(). Then, we could ensure that the admins or data analysts only reveal identities of suspicious employees. Thus, *Presumption of Innocence* can be ensured as long as Malicious_Issuer() is not executed.

Admin-Privacy This goal is ensured if the privileges of admins and data analysts do not allow them to injure one of the previous privacy goals *Unreadability*, *Anonymity*, or *Unlinkability*. As the admins additionally learn the intersections and should reveal the identity of suspicious clients, they are a greater risk to privacy. First, malicious admins and adversaries that control party P can learn the elements that are inside the intersections from Infected_Devices(), On_Purpose(), and Malicious_Issuer(). For this, they must only find out which of their elements $x_i \in X$ map to the same hidden elements $\mathsf{Hide}_k(x_i) = \mathsf{Hide}_k(z_j)$ as $z_j \in X \cap Y_\iota$. This, also works for Malicious_Source() except that the preimages of the elements must be learned by querying the clients. To increase the amount of learned secret elements, admins and adversaries controlling P can choose huge sets X that contain more than only

[44] `https://www.wireshark.org/`

[45] Prio avoids this problem by communicating with a huge set of clients at once. But that does not work for PSI since either all sets must be sent in clear to the admins and privacy injured or the same key k must be used for all clients though it should only be used once.

malicious websites. But as sensitive data is never part of those sets Y_ι, admins cannot injure our privacy goal *Unreadability*. Further, we demand that the admins are able to learn the clients' identities given e.g., the network traffic. This is necessary because we could otherwise not realize On_Purpose() or Malicious_Issuer(). But due to this, *Anonymity* is injured by admins and the regarded adversaries. To injure *Unlinkability*, the privileged actors only have to compare $P \cap Q_\iota$ and $P \cap Q_{\iota+1}$. If both those intersections contain the same element x, they can conclude, that x is owned by Y_ι and $Y_{\iota+1}$. This holds for the evaluation functions Infected_Devices(), On_Purpose(), and Malicious_Issuer(). And given any intersection $Z = X \cap Y_\iota$ from Malicious_Source(), admins, data analysts or those adversaries also learn that Z is a subset of all previous parties' sets $Y_\psi, \psi < \iota$. Due to this, *Unlinkability* is also injured. Hence, we cannot ensure *Admin-Privacy* for PSI.

Ephemerality We would injure this privacy goal if admins or data analysts could permanently access the clients' request data. But due to our construction, the data is stored on client-side and is thus not accessible for privileged actors. Hence, we can also ensure *Ephemerality* for PSI.

Validity To ensure this privacy goal, we must prevent that invalid sets or sets with invalid elements $y \in Y_\iota$ influence our final intersections. Whereas invalid relates to elements y that are not inside their value sets $\mathcal{V}$. As PSI does not ensure any *Validity* checks besides the ones that are done by the algorithms we described with Hide_k, we need to rely on our constructions. On the one hand, the log file server could check the data from the requests, before it stores those in the digest server. But this does not work as the only data we store are *target_ip*s or *target_url*s and comparing those to a set of valid addresses is infeasible. At least, we can ensure that only truly accessed websites can be part of the resulting intersections. This is possible since our construction allows us to learn whether the elements in intersection $Z = X \cap Y_\iota$ only contain elements from our digest storage. Thus, admins can remove all invalid elements $z \in Z$ such that the resulting intersections are not influenced by the invalid client data. A limitation is that we only recognize whether an element was ever part of a request, but not if it was specifically that client that issued the given website in the regarded period. Moreover, we can not verify the correctness of sent *datestamp* and *timestamp* and thus not ensure *Validity* for Period_Preselection(). Hence, we can ensure *Validity* for most of the evaluation functions with PSI.

Result

Given the previous privacy and functionality considerations, we learn that PSI enables all remaining evaluation functions. Meanwhile, it ensures almost 5 of 7 privacy goals. Additionally, the log files we need for PSI are smaller than those defined in Section 3.1 and also than those for Prio & Log Files. In particular, only a small log file with non-sensitive columns (`target_ip`, `target_url`, `session_bytes`, `status_code`, `datestamp`, `timestamp`)

Table 5.6: Privacy Grading for Private Set Intersection

Privacy Goals	PSI	Comments
Unreadability	✓	-
Anonymity	×	pairwise execution reveals identity in traffic
Unlinkability	✓	-
Presumption of Innocence	✓	if Malicious_Issuer() is replaced by On_Purpose()
Admin-Privacy	×	admins learn intersections
Ephemerality	✓	-
Validity	∼	not possible for Period_Preselection()
Result	4.5 P.	

must be stored. Thus, each reduced log file is stored in a privacy preserving way since only the client can access its own data and PSI is less expansive regarding storage consumption.

Another advantage of PSI is that only one party i.e., admins and data analysts learn the intersections. Moreover, we do not need to publish the malicious websites to the clients to enable the evaluation functions. Thus, malicious clients do not learn malicious websites that could be used to harm the company or whether their own website was already found.

Furthermore, our realization of Malicious_Issuer() does preserve more privacy than the other functions, as all clients can participate and the final result does not only depend on a single client's secret set. But it is not possible to apply this to the other evaluation functions, as we want to know whether a single client is infected or whether particularly this client caused the malicious download. Computing the intersection with all clients means that e.g., client Q_1 is not malicious i.e., accessed none of the malicious websites. Thus, the intersection $X \cap Y_1$ would be empty. This causes that all following intersection for $Q_\iota, \iota > 1$ cause an empty set as well since X is empty. Due to this, we would never find any client that accessed a malicious website if we applied the round by round variant.

In comparison, Prio & Log Files enables the same evaluation functions while preserving more of our privacy goals. On the other hand, as it only realizes a simple set intersection, PSI is easier to understand than approaches using Prio. Moreover, Prio is complex using the encoding and sharing and because each aggregation function requires another AFE. PSI is comparable easy to realize, though this still depends on the specific variant. But if we already realized Prio, Prio & Log Files is the better extension for this.

6 Overall Solution

Finally, we regarded all approaches that are interesting for this work. Hence, we can design our privacy-preserving log file collection and evaluation system using the most promising approaches. For this, we briefly describe the problems that occurred with our basic approaches (Chapter 4) in Section 6.1. After that, we design our log file system with advanced approaches (Chapter 5) in Section 6.2. In the end, we recommend some further improvements to the proposed system.

The reason we only use advanced approaches for the final system design is that basic and advances approaches significantly differ in the way they protect privacy. While the goal of the basic approaches is to privately store data such that it can still be evaluated, the advanced approaches target private evaluation while storing as least data as possible. Due to this, the advanced approaches can ensure more of the privacy goals we defined and are thus better for our privacy-preserving log file system.

6.1 Result of Basic Approaches

As a result, our basic approaches Complete Hashing, Partial Hashing, Encryption, and Hashing & Encryption have two main issues in common. First, all of them depend on the storage of the request data i.e., on centrally stored log files. Thus, their main goal is to secure and hide the (sensitive) data inside to increase the employees' privacy. For this, we need to balance privacy and functionality since too much privacy complicates the evaluations as in Section 4.1. Second, as all basic approaches depend on deterministic hash functions or encryption, it is always possible to memorize the connection of the secret data and its digest or ciphertext. This enables revealing the hidden secret data and thus limits the privacy. While Hashing & Encryption seems to be a good solution to both problems, it still ensures few privacy goals compared to the advanced approaches we analyze in Chapter 5. Therefore, for our privacy-preserving log file system, we need approaches that have none of these problems. We sum our findings of the basic approaches' privacy and functionality grading in Tables A.2 and A.3.

As a result, the basic approaches are good solutions for any company that wants to improve the privacy of the stored employees' data. Since they do not require changing the whole system (as the advanced approaches do) basic approaches are easier and faster to realize. But in practice, log files and databases are often compressed to save storage. This becomes complicated as the encrypted or hashed log entries have a high entropy such that compression algorithms can hardly be applied. Nevertheless, compared to log files that are completely stored in clear, these basic approaches already have a huge positive impact on privacy.

6.2 Result of Advanced Approaches

But for our privacy-preserving log file collection and evaluation system, we desire more private approaches. To ensure more privacy, we must prevent from secret data being stored or sent. And though this work was meant to find a way of privately analyzing log files, we learned that using no (centrally stored) log files is the more private solution. With Prio, there is a way of realizing the evaluation functions (Section 3.2) we thought of without storing secret data or (directly) transmitting it to admins or data analysts. As Prio alone does not enable all evaluation functions, we additionally use Prio & Log Files. If we completely abandoned log files, our system could execute few helpful evaluation functions. But if the requests' data is only stored at the employees' devices and not on a centrally accessible log file, we can keep the stored data private. Thus, our privacy-preserving log file collection and evaluation system is a combination of Prio[1] and Prio & Log Files, as we described those approaches in Sections 5.1 and 5.3. Hence, a part of all requests' data is directly sent to the Prio servers to perform the periodically malware detection. Additionally, the requests are forwarded to the log file server, which creates and encrypts the log file entries. After that, the log file server sends those log entries to the employee that submitted the corresponding request. Moreover, the log file server stores the necessary secrets as digests inside the digest server. This final scenario is displayed in Figure 6.1. Then, all evaluation functions that detect malware are executed with Prio, while the remaining ones can be realized due to Prio & Log Files. Since both approaches ensure nearly the same privacy goals, this combined approach is more private (regarding our privacy goals) than all other analyzed approaches. In particular, our system then ensures all privacy goals except *Admin-Privacy*, which depends on the evaluated function. But since it is necessary that the admins (or data analysts) learn at least few data from the evaluation functions' outputs, we cannot avoid this. Additionally, we can verify some results (e.g., from Max_Download_Size() or Most_Visited_Websites()) computed with Prio by comparing them to the data stored in the digest storage. This becomes possible as all requests' session_bytes and websites are already stored for Prio & Log Files. The grading that can be expected if we combine both approaches can be seen in Tables 6.1 and 6.2[2]. Additionally, a summary of the advanced approaches' gradings can also be found in Tables A.4 and A.5. As a result, our log file system allows private evaluation of secret data due to Prio. Moreover, though we need to store some requests' data, the collection of that data is privacy-preserving as well.

[1]In fact, Prio3 should be preferred since it allows the same aggregation functions but is more efficient due to the new proofs. For more details, see Section 5.1.5.

[2]The $\sim$ in this case means that we can ensure a privacy goal depending on which evaluation function is executed.

Table 6.1: Overall Functionality Grading of Final Construction

Evaluation Functions	Prio	Prio & Log Files	Final Construction
Avg_Download_Size()	✓	–	✓
Max_Download_Size()	✓	–	✓
Number_Requests()	✓	–	✓
Most_Visited_Websites()	✓	–	✓
Malicious_Source()	×	✓	✓
Infected_Devices()	×	✓	✓
On_Purpose()	×	✓	✓
Malicious_Issuer()	×	✓	✓
Period_Preselection()	×	✓	✓
Result	4 P.	5 P.	9 P.

Table 6.2: Overall Privacy Grading of Final Construction

Privacy Goals	Prio	Prio & Log Files	Final Construction
Unreadability	✓	✓	✓
Anonymity	✓	✓	✓
Unlinkability	✓	✓	✓
Presumption of Innocence	✓	✓	✓
Admin-Privacy	✓	×	∼
Ephemerality	✓	✓	✓
Validity	✓	✓	✓
Result	7 P.	6 P.	6,5 P.

6.3 Possible Improvements

As already announced in Section 3.1, we recommend which kind of data can be removed from the log files since it is never used for any of the analyzed approaches. Since in practice each kind of data reveals sensitive or private information, we can improve privacy by storing as least data as possible. At first, it suffices to store either the target_ip or the target_url as both are used for the same. We recommend using the IP addresses, as each website has one IP address but perhaps several registered URLs. Moreover, the data for the status_code must not be stored if a bit identifies whether the request was redirected instead. In addition, if the company analyzes data from a fixed period (e.g., always a complete month), it suffices to store the month and year, instead of the complete datestamps and timestamps. This increases the employees' privacy and avoids that those stamps can be used to identify specific employees given their working hours. Last, we never made use of client_ip_address or client_mac_address. Those must only be stored if a company does not identify each device with a device_id. But then, one must be aware that client_mac_addresses can be tampered by malicious employees and that client_ip_addresses might change over time. If only one of those three options is stored, this suffices for our privacy-preserving log file collection and evaluation system.

To further improve our privacy-preserving log file collection and evaluation system, we could use homomorphic encryption [Gen09, OTD13]. If the clients encrypted their secrets using homomorphic encryption, the shares and servers' *sum* would be encrypted as well. Thus, even if all servers were malicious and cooperated to reconstruct the secrets from their shares, they could only see the encrypted secret. In this case, we would only allow the admins (or data analysts) to decrypt the ciphertexts that are stored in the result storage. But this requires that homomorphic encryption provides all mathematical operations that are used by Prio, which includes xor and displaying integers as binaries or unaries. Since one of Prio's advantages is that it does not require on additional encryption, the performance of this proposed improvement should not dramatically slow down Prio. Thus, further research is necessary to check whether current homomorphic encryption schemes can provide what we ask for. Hence, this improvement could be investigated in future work.

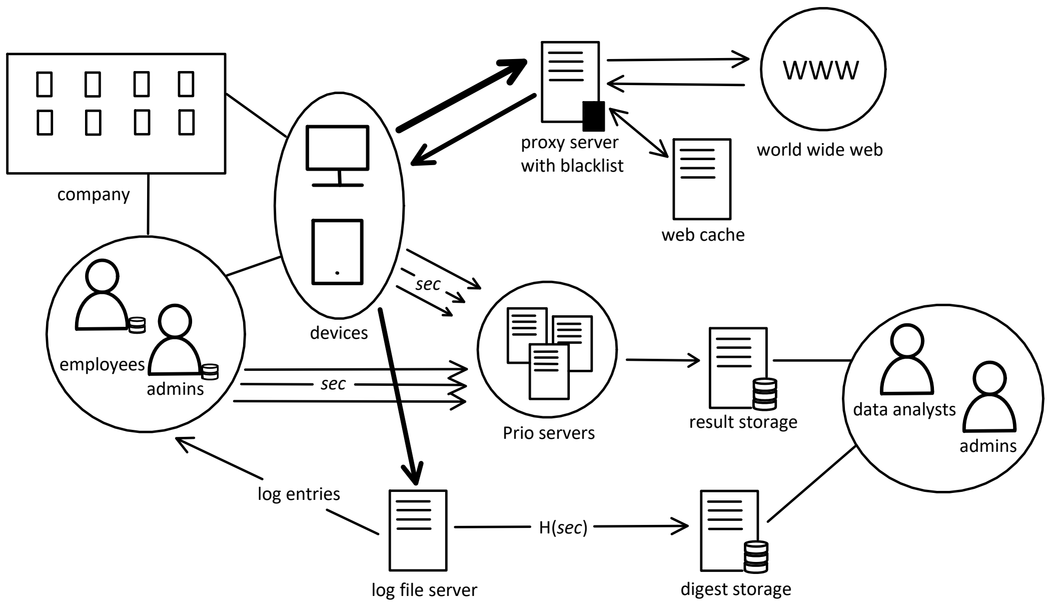


Figure 6.1: Diagram describing the scenario of our privacy-preserving log file collection and evaluation system

7 Conclusion

In this thesis, we regarded several different cryptographic approaches to find a solution for our privacy-preserving log file collection and evaluation system. In the end, our final construction makes use of Prio (Section 5.1) and Prio & Log Files (Section 5.3). Those can evaluate our defined evaluation functions from Section 3.2 without revealing sensitive data of innocent employees. On the other hand, it still allows detecting malicious employees and adversaries that try to harm the company. The important findings of our analysis and our construction of the privacy-preserving log file collection and evaluation system are summarized in Chapter 6.

As we only analyze our specific scenario and a subset of probably useful functions and privacy goals, this work is still not exhaustive. Other researchers might be interested in privacy-preserving storage of databases, where our basic approaches might already be at help by hiding sensitive data. Alternatively, one could use our scenario to analyze different functions that focus on servers' performance or memory consumption of e.g., a proxy server. Nevertheless, the approaches we use in this thesis are also useful for other settings. Prio, for example, can be used for statistical evaluation on distributed secret data. In this thesis, we assumed that some data is sensitive and some not. In practice, each data can reveal information about an employee. Thus, one would have to hide and protect even more data than we did in this work. This leads to e.g., Partial Hashing being less useful in practice than in our specific scenario, since a huge amount of data is stored in clear. In other work, the identification of sensitive data must be adapted to the specific scenario, as this is still hard to decide in general. Depending on the scenario, a name can be sensitive data if it belongs to an employee and might not be sensitive if it is part of the accessed URL. But finding a way to decide this in general could also be interesting future work.

A last limitation of this work is that we assume trust in the log file server. Since this server sees the (sensitive) data from all sent requests, it would be problematic if this entity was untrusted or controlled by any adversary. Then, all (sensitive) data would be leaked and privacy is injured for all employees. But in practice, such a trusted entity is hard to guarantee. Thus, it could be interesting to identify how we can remove this dependency on a trusted log file server without completely changing the system we created here.

7.1 Outlook

We chose a few interesting approaches, but there exist many more cryptographic protocols or schemes that enable private collection and evaluation of log files. One of our planned advanced approaches could not be regarded in detail as it is meant for machine learning and had thus to be used for a completely different scenario than we regarded here. This approach is Federated Learning. Federated Learning [KMY$^+$16, MMR$^+$17] is a technique that allows machine learning models to learn from users' secret data without collecting the data centrally. For this, the current model is sent to the users, updated using their data and sent back to be merged to the previous model version. This could also be used to learn about employees' behavior in the scenario we analyzed. But as for the average of all downloads' sizes, the employees must update a given average and given this update, it would always be possible to learn their provided data. Thus, federated learning must be adapted to avoid this leakage of the users' data. Due to this, using Federated Learning as an approach for private log files is a task that can be regarded in future work. We also briefly mention the concept of Searchable Encryption in Section 1.1. This could be another interesting approach to investigate further.

It might also be interesting to formalize our privacy-preserving log file collection and evaluation system. This would include a formal definition of the necessary evaluation functions and privacy requirements. Given those, it becomes feasible to formally prove which of our approaches ensure which privacy requirements. Moreover, we do not focus on the performance of the proposed system and approaches. Regarding this, Prio is also promising as it provides scalability and does not require (additional) public key cryptography [CB17]. For practical usage of our system, the performance would be interesting and could thus be analyzed in future work. For all this, our thesis can be seen as a groundwork that already defines the scenario and analyzes the functionality and privacy of the presented approaches.

Bibliography

[ABO07] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. New security models and provably-secure schemes for basic query support in outsourced databases. In *21st Annual IFIP WG*, volume 11, pages 8–11, 2007.

[AG21] Apple and Google. Exposure notification privacy-preserving analytics (enpa) white paper. 2021.

[AGJ$^+$22] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 516–539. Springer, 2022.

[AS00] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 439–450. ACM, 2000.

[BBC$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97. Springer, 2019.

[BBC$^+$21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 762–776. IEEE, 2021.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552. Springer, Heidelberg, August 2007.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 224–241. Springer, Heidelberg, April 2009.

[BCV16] Julio Bondia-Barcelo, Jordi Castellà-Roca, and Alexandre Viejo. Building privacy-preserving search engine query logs for data monetization. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016*, pages 390–397. IEEE Computer Society, 2016.

[BS22] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. 2022.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282. USENIX Association, 2017.

[Cel22] Sofia Celi. A note on Privacy-Preserving Measurements Techniques. page 11, 2022. https://www.ietf.org/archive/id/draft-ietf-ppm-dap-02.html.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer, Heidelberg, March 2006.

[DMRY09] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS 09: 7th International Conference on Applied Cryptography and Network Security*, volume 5536 of *Lecture Notes in Computer Science*, pages 125–142. Springer, Heidelberg, June 2009.

[DN03] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In Frank Neven, Catriel Beeri, and Tova Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 202–210. ACM, 2003.

[Dou02] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002,* *Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[EDG14] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1068–1079. ACM, 2014.

[FS90] Uriel Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *22nd Annual ACM Symposium on Theory of Computing*, pages 416–426. ACM Press, May 1990.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178. ACM Press, May / June 2009.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658. Springer, 2014.

[GLP11] Johannes Gehrke, Edward Lui, and Rafael Pass. Towards privacy for social networks: A zero-knowledge based definition of privacy. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 432–449. Springer, 2011.

[Inp22] Inpher. XOR Secret Computing Engine, 2022. https://inpher.io/xor-secret-computing/.

[KKJ+13] Himanshu Kumar, Sudhanshu Kumar, Remya Joseph, Dhananjay Kumar, Sunil Kumar Shrinarayan Singh, Ajay Kumar, and Praveen Kumar. Rainbow table to crack password using md5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*, pages 433–439, 2013.

[KMY+16] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *CoRR*, abs/1610.05492, 2016.

[KRS+19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1447–1464. USENIX Association, 2019.

[LZCW23] Momeng Liu, Zeyu Zhang, Wenqiang Chai, and Baocang Wang. Privacy-preserving COVID-19 contact tracing solution based on blockchain. *Comput. Stand. Interfaces*, 83:103643, 2023.

[MDC16] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-preserving process mining: Differential -privacy for event logs (extended abstract). *Inform. Spektrum*, 42(5):349–351, 2019.

[MMR+17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 2017.

[OHS13] Ian Oliver, John Howse, and Gem Stapleton. Protecting privacy: Towards a visual framework for handling end-user data. In Caitlin Kelleher, Margaret M. Burnett, and Stefan Sauer, editors, *2013 IEEE Symposium on Visual Languages and Human Centric Computing, San Jose, CA, USA, September 15-19, 2013*, pages 67–74. IEEE Computer Society, 2013.

[OTD13] Monique Ogburn, Claude Turner, and Pushkar Dahal. Homomorphic encryption. In Cihan H. Dagli, editor, *Proceedings of the Complex Adaptive Systems 2013 Conference, Baltimore Marriott Inner Harbor at Camden Yards, Baltimore, Maryland, USA, November 13-15, 2013*, volume 20 of *Procedia Computer Science*, pages 502–509. Elsevier, 2013.

[Par83] William A Parent. A new definition of privacy for the law. *Law and Philosophy*, 2(3):305–338, 1983.

[PH10] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf, August 2010. v0.34.

[Rat16] Christof Rath. Usable privacy-aware logging for unstructured log entries. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 272–277. IEEE Computer Society, 2016.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[UKH$^+$21] Mohib Ullah, Rafiullah Khan, Muhammad Inam Ul Haq, Atif Khan, Wael Alosaimi, Muhammad Irfan Uddin, and Abdullah Alharbi. Multi-group obscure logging (mg-oslo) A privacy-preserving protocol for private web search. *IEEE Access*, 9:79005–79020, 2021.

[UKK$^+$22] Mohib Ullah, Rafiullah Khan, Irfan Ullah Khan, Nida Aslam, Sumayh S. Aljameel, Muhammad Inam Ul Haq, and Muhammad Arshad Islam. Profile aware obscure logging (paoslo): A web search privacy-preserving protocol to mitigate digital traces. *Secur. Commun. Networks*, 2022:2109024:1–2109024:13, 2022.

[Vin22] Staal A. Vinterbo. Differential privacy for symmetric log-concave mechanisms. *CoRR*, abs/2202.11393, 2022.

[Wei21] Christian Weinert. *Practical Private Set Intersection Protocols for Privacy-Preserving Applications.* PhD thesis, Technical University of Darmstadt, Germany, 2021.

[XFAM02] Jun (Jim) Xu, Jinliang Fan, Mostafa H. Ammar, and Sue B. Moon. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. In *10th IEEE International Conference on Network Protocols (ICNP 2002), 12-15 November 2002, Paris, France, Proceedings*, pages 280–289. IEEE Computer Society, 2002.

[XY12] Liangliang Xiao and I-Ling Yen. Security analysis and enhancement for prefix-preserving encryption schemes. Cryptology ePrint Archive, Report 2012/191, 2012. `https://eprint.iacr.org/2012/191`.

Appendix A

Exemplary Database Tables and Overall Grading Tables

A.1 Example of the `Log_Files` Database Table

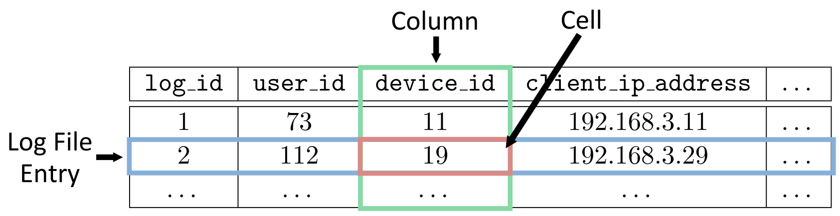
The following Table A.1 is a simplified example of our database table `Log_Files`. It only contains the first columns from $\mathcal{CO}$. In practice, this table would contain all columns in $\mathcal{CO}$ and significantly more log entries. Nevertheless, it gives a first impression of what we are working with.

Table A.1: Example of the database table `Log_Files`

`log_id`	`user_id`	`device_id`	`client_ip_address`	...
1	73	11	192.168.3.11	...
2	112	19	192.168.3.29	...
...	...	...	...	...

A.2 Explanation of Log File Entry, Column, Cell

Additionally, Figure A.1 explains the terms Log File Entry, Column and Cell, which are often used in Chapter 4 about our basic approaches. We hope that it is helpful to see these terms marked in an example of the database table `Log_Files` to understand the explanations and thoughts in this thesis.



`log_id`	`user_id`	`device_id`	`client_ip_address`	...
1	73	11	192.168.3.11	...
2	112	19	192.168.3.29	...
...	...	...	...	...

Figure A.1: Explanation of Log Entry, Column, and Cell

A.3 Overall Grading Tables

Table A.2: Overall Functionality Grading of Basic Approaches

Evaluation Functions	Complete Hashing (Section 4.1.1)	Partial Hashing (Section 4.1.2)	Encryption (Section 4.2)	Hashing & Encryption (Section 4.3)
Avg_Download_Size()	×	✓	✓	✓
Max_Download_Size()	×	✓	✓	✓
Number_Requests()	✓	✓	✓	✓
Most_Visited_Websites()	~	✓	✓	✓
Malicious_Source()	×	✓	✓	✓
Infected_Devices()	×	✓	✓	✓
On_Purpose()	×	✓	✓	✓
Malicious_Issuer()	×	×	✓	×
Period_Preselection()	×	✓	✓	✓
Result	1.5 P.	8 P.	9 P.	8 P.

Table A.3: Overall Privacy Grading of Basic Approaches

Privacy Goals	Complete Hashing (Section 4.1.1)	Partial Hashing (Section 4.1.2)	Encryption$_{sym}$ (Section 4.2)	Encryption$_{asym}$ (Section 4.2)	Hashing & Encryption (Section 4.3)
Unreadability	✓	✓	✓	✓	✓
Anonymity	✓	✓	✓	✓	✓
Unlinkability	×	×	×	×	×
Presumption of Innocence	✓	✓	✓	✓	✓
Admin-Privacy	~	~	×	×	~
Ephemerality	×	×	×	×	×
Validity	~	~	~	×	~
Result	4 P.	4 P.	3.5 P.	3 P.	4 P.

Table A.4: Overall Functionality Grading of Advanced Approaches

Evaluation Functions	Prio (Section 5.1.4)	Prio & Log Files (Section 5.3)	PSI (Section 5.4.2)
Avg_Download_Size()	✓	–	–
Max_Download_Size()	✓	–	–
Number_Requests()	✓	–	–
Most_Visited_Websites()	✓	–	–
Malicious_Source()	×	✓	✓
Infected_Devices()	×	✓	✓
On_Purpose()	×	✓	✓
Malicious_Issuer()	×	✓	✓
Period_Preselection()	×	✓	✓
Result	4 P.	5 P.	5 P.

Table A.5: Overall Privacy Grading of Advanced Approaches

Privacy Goals	Prio (Section 5.1.4)	Prio & Log Files (Section 5.3)	PSI (Section 5.4.2)
Unreadability	✓	✓	✓
Anonymity	✓	✓	×
Unlinkability	✓	✓	✓
Presumption of Innocence	✓	✓	✓
Admin-Privacy	✓	×	×
Ephemerality	✓	✓	✓
Validity	✓	✓	∼
Result	7 P.	6 P.	4.5 P.

Appendix B

Proofs

B.1 Proof of Claim 4.2.1

In this section we prove Claim 4.2.1. It says that CPA security (Definition 2.14) of a (symmetric) probabilistic cipher E implies that we cannot detect whether two different ciphertexts $c_1 \neq c_2$ originate from the same message m. This fact leads to the problem that we cannot use probabilistic ciphers (or encryption schemes) for our basic approach Encryption. The proof is also shortly displayed in Figure B.1.

Proof. Given an adversary $\mathcal{A}$ that takes two different ciphertexts $c_1 \neq c_2$ as input and outputs 1 if and only if both ciphertexts originate from the same message i.e., $\mathsf{Dec}(c_1, k) = \mathsf{Dec}(c_2, k), k \in \mathcal{K}$, we construct an adversary $\mathcal{B}$ from $\mathcal{A}$ that plays the CPA security game (Definition 2.14) against a (symmetric) probabilistic cipher E and a challenger. We construct $\mathcal{B}$ as follows:

1. $\mathcal{B}$ picks three messages $m_1, m_2, m_3 \leftarrow \mathcal{M}$ uniformly at random from the message space. It must hold that $m_2 \neq m_3$ and $|m_1| = |m_2| = |m_3|$[1].

2. $\mathcal{B}$ sends m_1 as $m_{1,0}$ and m_2 as $m_{1,1}$ to the challenger.

3. Since the challenger encrypts one of the received messages $m_{1,b}$ according to $b \leftarrow \{0,1\}$, $\mathcal{B}$ receives the corresponding ciphertext c_1.

4. Then $\mathcal{B}$ sends m_1 as $m_{2,0}$ and m_3 as $m_{2,1}$ to the challenger.

5. After receiving the second ciphertext c_2, $\mathcal{B}$ sends c_1, c_2 to $\mathcal{A}$.

6. If and only if $\mathcal{A}$ outputs 0, $\mathcal{B}$ sends 1 to the challenger, otherwise he sends 0.

Since the challenger encrypts the same message m_1 twice in experiment $b = 0$, $\mathcal{A}$ outputs 1 with high probability. Then, adversary $\mathcal{B}$ would correctly output 0 with the probability that $\mathcal{A}$ succeeds. On the other side, for $b = 1$ the challenger encrypts two different messages $m_2 \neq m_3$ which are encrypted to different ciphertexts due to the definition of correctness of probabilistic ciphers. Hence, $\mathcal{A}$ would output 0 with high probability. Therefore, our adversary $\mathcal{B}$ would output the correct $\hat{b} = 1 = b$ with $\mathcal{A}$'s probability of

[1]If necessary, we pick new values at random until they fulfill these requirements.

succeeding. As a result, adversary $\mathcal{B}$ is efficient and would output the correct $\hat{b}$ with high probability.

Thus, CPA security implies that we **cannot** detect whether two different ciphertexts originate from the same message. $\square$

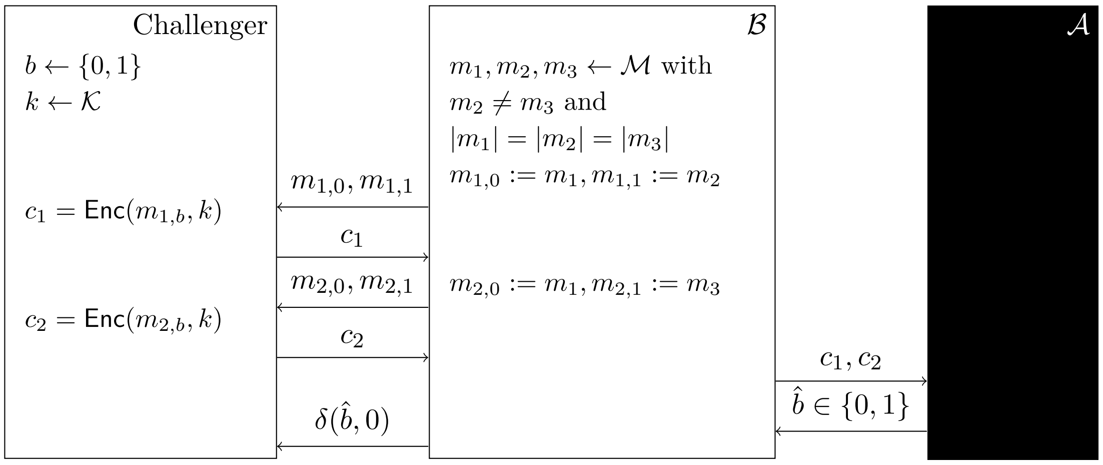

Challenger
$b \leftarrow \{0,1\}$
$k \leftarrow \mathcal{K}$
$c_1 = \mathsf{Enc}(m_{1,b}, k)$
$c_2 = \mathsf{Enc}(m_{2,b}, k)$
$\mathcal{B}$
$m_1, m_2, m_3 \leftarrow \mathcal{M}$ with $m_2 \neq m_3$ and $|m_1| = |m_2| = |m_3|$
$m_{1,0} := m_1, m_{1,1} := m_2$
$m_{2,0} := m_1, m_{2,1} := m_3$
$m_{1,0}, m_{1,1}$
c_1
$m_{2,0}, m_{2,1}$
c_2
$\delta(\hat{b}, 0)$
c_1, c_2
$\hat{b} \in \{0,1\}$
$\mathcal{A}$

Figure B.1: Black Box Diagram displaying the proof of Claim 4.2.1

B.2 Proof of correctness of Prio's sum-AFE

Claim B.2.1. *We claim that the encoding and decoding of the sum-AFE* Prio *uses is correct. Thus, given the correct application of the* Encode *algorithm to each secret sec_t, the output of* Decode *should be the sum of all secrets $\sum_{j=1}^{K} sec_j$ (for K secrets).*

In the following, we prove Claim B.2.1 exemplarily for one of the AFEs that are used by Prio. Let f be the aggregation function that sums the secrets of all K sent secrets: $f(sec_1, sec_2, \ldots, sec_K) = \sum_{j=1}^{K} sec_j$ Then, we pick the AFE that encodes for the sum computation. Hence, Encode computes the tuple $e = (sec, b_0, b_1, \ldots, b_{B-1}) \in \mathbb{F}^{B+1}$ with $sec = \sum_{i=0}^{B-1} 2^i \cdot b_i$ and $\kappa' = 1$ [CB17]. The algorithm Valid then checks whether the set of $b_i, i \in \{0, \ldots, B-1\}$ is a valid bit representation of the first value i.e., sec in the encoding e. Since $\kappa' = 1$, applying $[o..\kappa']$ on the encoded secret already outputs the secret sec itself. Thus, Decode has no impact on its input since it is already decoded. Then, the following holds given all steps executed by the servers $s, s \in \{1, 2, \ldots, S\}$ of the Prio protocol:

Proof.

$$\begin{aligned}
\sum_{s=1}^{S} sum_s &= \sum_{s=1}^{S} \left(\sum_{j=1}^{K} [\mathsf{Encode}(sec_j)]_s[0..\kappa'] \right) \\
&= \sum_{j=1}^{K} \sum_{s=1}^{S} [\mathsf{Encode}(sec_j)]_s[0..1] \\
&= \sum_{j=1}^{K} \sum_{s=1}^{S} [sec_j]_s \\
&= \sum_{j=1}^{K} sec_j \\
\Rightarrow \mathsf{Decode}\left(\sum_{s=1}^{S} sum_s \right) &= \sum_{j=1}^{K} sec_j \\
&= f(sec_1, sec_2, \ldots, sec_K)
\end{aligned}$$

□