UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXIV° Cycle

SSD: ING-INF/05

Methods and technologies for the secure collection, sanitization, processing and release of data

Advisor

Prof. Stefano Paraboschi

Doctoral Thesis

Dario FACCHINETTI

Student ID 1029668

Academic year 2020/2021

Abstract

The last decade has seen a significant increase in usage of cloud services. This trend is not only related to the low cost and high availability of cloud providers, but also to the ease of use of the service and the reliability over time. Digital devices become rapidly obsolete and are subject to failures, hence outsourcing data permits to reduce the risks linked to data loss.

Albeit there are advantages in uploading data to the cloud, there are also several security and privacy challenges. The experience gained by the Research and Industry communities attest that it is not enough to just change the visibility of data by applying a cryptographic transformation, to ensure an adequate level of protection. A cloud-oriented architecture has a wide attack surface, hence it is necessary to pay attention to the whole data lifecycle, from data collection and sanitization, to storage and processing, and finally the release. This doctoral thesis analyzes each of these stages, proposing solutions that push forward the current state of the art.

The first part of the thesis deals with the collection of data, in particular in the mobile scenario. The mobile environment is especially relevant as smartphones are devices with limited storage, that are connected to the network, and with the ability to sense and log confidential data and Personal Identifiable Information. To access this information, an application must be granted the proper permission. Yet, all the components running inside the application (either trusted or included from third-parties) share the same execution environment, thus have the same visibility and access constraints. This is a limitation of the current mobile Operating Systems. Focusing on the Android, which is open source and available to researchers, we propose a set of modifications to achieve internal application compartmentalization leveraging the Mandatory Access Control (MAC) layer. With this approach, the developer can add a policy module to the application to confine each component, effectively restricting access to the application internal storage, to services, and to isolate vulnerability prone components.

After the data are collected, a user or a company may apply to it sanitization before being uploaded to the cloud or being released to a consumer. Data sanitization (or anonymization) is a process by which data are irreversibly altered so that a subject (referenced within the data) cannot be identified, given a certain security parameter, while the data remain practically useful. The

second part of the thesis presents an approach based on k-anonymity and ℓ-diversity to apply data sanitization over large collections of sensors data. The approach described can be applied in parallel in a distributed environment and is characterized by a limited information loss.

The third part of the thesis investigates the storage and processing stages. In this scenario, the cloud provider is typically considered honest-but-curious, which assumes that it will always comply with the requests issued by the user, but may abuse the access to the information provided. Hence, the goal is to support the execution of queries over outsourced data with a guarantee that the cloud provider does not have access to the data content. Unfortunately, the simple use of deterministic encryption does not offer a real protection against a curious provider, as the encrypted data maintain the same distribution of the original data. The approach presented in this thesis is applicable to relational data, and enables the execution of queries involving evaluation of equality and range conditions over attributes. The data is saved encrypted to the server into equally large blocks containing a fixed number of tuples. The blocks are managed by the server as single atomic units, and accessed through an encrypted multidimensional index also stored by the server. By doing this, the cloud provider is unable to identify the single items stored within each block. Local maps are saved by the client to search the index efficiently. The approach proposed provides perfect indistinguishability to an attacker with access to the stored data. This is achieved applying probabilistic encryption to the blocks storing the data, and by destroying (i.e., flattening) the frequencies of the encrypted index. The index is built as an evolution of the partitioning technique presented in the second part of the thesis to sanitize the dataset.

The last part of the thesis addresses the data release stage. The goal is to provide a solution that can be used to schedule the release of chunks or partitions of data at a future point in time. Due to the confidential nature of data, we cannot rely on any honesty assumption. Hence, we move to a decentralized environment in which the parties (i.e., the network nodes) are mutually distrusting. In this setting, we model the parties as rational (or rather driven by pure economic interest), and propose a solution that is only based on economic incentives and penalties.

All the technologies detailed in this thesis have been released under open source licenses and can be readily integrated with real systems.

Contents

Abstract **1**

1 Introduction **15**
1.1 Document structure 19
1.2 Publications 23

2 Data collection **25**
2.1 Introduction 26
2.2 Android security for apps 27
2.3 Motivation 30
2.4 Policy language 34
2.5 Policy configuration 39
2.6 Implementation 44
2.7 Experimental results 51
2.8 Related work 56
2.9 Conclusions 58

3 Data sanitization **59**
3.1 Introduction 60
3.2 Distributed anonymization 62
3.3 Experimental results 64
3.4 Conclusions 66

4 Data storage & processing **69**
4.1 Introduction 70
4.2 Basic concepts and problem statement 71
4.3 Partitioning 73

4.4 Indexing . . . 82
4.5 Query translation . . . 84
4.6 Implementation . . . 88
4.7 Experimental results . . . 92
4.8 Related work . . . 102
4.9 Conclusions . . . 105

5 Data release 107
5.1 Introduction . . . 108
5.2 Background . . . 110
5.3 The ITYT protocol . . . 111
5.4 Economic model . . . 118
5.5 Implementation . . . 122
5.6 Discussion . . . 126
5.7 Experimental results . . . 128
5.8 Related work . . . 130
5.9 Conclusions . . . 132

6 Conclusions 133
6.1 Future Work . . . 134

Acknowledgments 137

Bibliography 139

A Seapp documentation 157
A.1 Use case 1 . . . 158
A.2 Use case 2 . . . 159
A.3 Use case 3 . . . 161
A.4 Policy module . . . 163

B Sds documentation 167
B.1 Requirements . . . 167
B.2 Deployment . . . 168
B.3 Usage . . . 168
B.4 Results . . . 170

C Secidx documentation **173**

C.1 Artifact 173

C.2 Proofs 179

D Ityt documentation **181**

D.1 Model 181

D.2 Contract 182

D.3 sMPC 183

List of Figures

2.1 Evolution of the MAC policy in Android. Before 4.3, MAC was not used. Starting with 4.3, MAC protects system components. Since 9, categories offer rigid MAC protection for apps. Our proposal offers flexible MAC protection to apps. 28
2.2 Security Enhanced App 31
2.3 SEApp policy structure 42
2.4 Installation process 46
2.5 Application launch 48
2.6 File relabeling 49
2.7 Installation time overhead for apps with different complexity 51
2.8 Cumulative install time overhead when installing the top 100 free apps on Google Play Store with our policies 53
2.9 Install time overhead for the three policy sizes 54

3.1 An example of a dataset (a), its spatial representation and partitioning (b), and a 3-anonymous and 2-diverse version (c), considering quasi-identifier `Age` and `Country` and sensitive attribute `TopSpeed` 60
3.2 Architecture and working of the distributed anonymization system . . . 63
3.3 Execution time of the centralized version and distributed version varying the number of workers 65

4.1 Plaintext relation (a), and its encrypted and indexed version (b) 72
4.2 Graphical representation of the cuts performed by procedure Cut over the relation in Figure 4.1(a) 77
4.3 Algorithm for computing a k-flat partition 78
4.4 Algorithm for cutting a k-flat partition 79
4.5 Algorithm for checking the validity of a cut 80

4.6 Examples of partitions for different cases of the **Check** procedure, k=10 81
4.7 Spatial representation of the running example and corresponding MAP . 82
4.8 Physical representation of the relation in Figure 4.1(a) 84
4.9 Information stored at the client-side for representing functions $map_{\texttt{Age}}/id.map_{\texttt{Age}}$ (a)-(c) and $map_{\texttt{State}}/id.map_{\texttt{State}}$ (b)-(d) 86
4.10 Server-side representations: (a) Relational DBMS with GIDs, (b) Relational DBMS with Tokens, (c) Key-value store with GIDs, and (d) Key-value store with Tokens. 91
4.11 Map size for each sample of `usa2019` 93
4.12 Average execution time of queries on WAGP column with a 10 ms latency. 95
4.13 Average execution time of queries on both OCCP and WAGP columns with a 10 ms latency. 97
4.14 Execution time and performance ratio of queries on WAGP column with the addition of latency. 99
4.15 Average global execution time and performance ratios of queries on both OCCP and WAGP columns with the addition of latency. 100
4.16 Overhead in the number of tuples downloaded with queries on the WAGP column. 101
4.17 Overhead in the number of tuples downloaded with queries on both OCCP and WAGP columns. 101
4.18 Execution time and performance ratio of queries on WAGP column with a latency of 10 ms and, varying K and bandwidth. 103
4.19 Execution time and performance ratio of queries on both OCCP and WAGP columns with a latency of 10 ms and, varying K and bandwidth. 104

5.1 Protocol initialization algorithm (executed by the owner). 115
5.2 Shareholder commitment algorithm 116
5.3 Share whistleblow algorithm 116
5.4 Secret whistleblow algorithm 117
5.5 Share disclose algorithm 117
5.6 Reward withdraw algorithm 118
5.7 Ityt constraints representation 121

5.8 State machine representing the valid state transitions of the ITYT protocol. Each transition name maps to an action (an Ethereum smart contract function) that can be invoked by participants to update the state. Square brackets state additional conditions that must be met to consider the transition valid . . . 123
5.9 Avoid the exposition of $\mathcal{S}$ before sc activation . . . 125
5.10 sMPC protocol time and memory consumption: (a) Single-phase vs two-phase sMPC execution, (b) Two-phase time consumption, (c) Two-phase memory consumption . . . 130

A.1 MainActivity . . . 157
A.2 Use case 1 views . . . 159
A.3 Use case 1 logcat . . . 159
A.4 Use case 2 views . . . 160
A.5 UC2 exploit . . . 160
A.6 Use case 2 logcat - SELinux denial . . . 161
A.7 Use case 2 logcat - Activity termination . . . 161
A.8 Use case 3 views . . . 162
A.9 Use case 3 logcat - SELinux denial . . . 162

C.1 Differences between an initial dataset and its 25-flat representation . . . 176
C.2 Example of construction of the encrypted index, the client-side maps, and the encrypted k-flat relation . . . 177
C.3 Dataset upload to the storage provider . . . 178
C.4 Translation and resolution of a runtime query . . . 178
C.5 Theoretical versus experimental token collision probability varying the number of tokens and their size . . . 179

D.1 Solving the model from command line using the Z3 solver . . . 182
D.2 Simulation of the decentralized protocol using Brownie . . . 183
D.3 *Single-phase* protocol execution for N = 2 . . . 185

List of Tables

2.1 Application policy module CIL syntax 35
2.2 SEApp macros to grant permissions to local types 39
2.3 Policy size 54
2.4 Cold and warm start performance for activities and services 55
2.5 File creation performance 56

3.1 DP and GCP information loss with 100% and 0.01% sampling 66

4.1 Relative size of the client-side maps given the size of the initial dataset . 93

5.1 Sample configurations (economic amounts are expressed as ratio of V) . 122
5.2 Gas cost for each smart contract function with $k = 2$ 128

B.1 Docker containers and URLs associated 168

D.1 Parameters and economic amounts constrained by the solver 182
D.2 List of targets available 184
D.3 Variables to customize the sMPC experiment 185

To my family

Chapter 1

Introduction

The outsourcing of data to cloud providers is a growing trend. There are many facilitating factors for that, including the low cost and high reliability over time. Moreover, after the data have been uploaded to the cloud, they are immediately available to a number of different devices, with almost no need for setup.

Despite there are clear advantages in uploading data to the cloud, there are also several security and privacy challenges. Compared to a traditional *client-server* architecture, in which all resources are internal to an organization and the security perimeter well defined, a *cloud-oriented* architecture has a wider attack surface, hence a greater risk of data breach or exposure. Unfortunately, the simple addition of a protection layer to data, for example with the application of a generic cryptographic transformation, is not enough to ensure an adequate level of protection. Attention must be payed to the collection of data and to its sanitization, storage and processing must be carried out by trusted parties, and data release planned in advance. Thereby, a cloud-oriented architecture relies on more complex policies and on an extensive collection of tools to support the whole data lifecycle. This thesis aims to provide techniques to support each of these stages.

The first part of the thesis deals with the collection of data, in particular in the mobile scenario. The mobile environment is especially relevant, as smartphones are devices constantly connected to the network, with the ability to log confidential data, and limited processing and storage capabilities. Advancements in this area have an impact on a broad spectrum of users, with a potential that goes even beyond the data collection process.

Currently, the access to cloud services is mostly achieved via applications (or simply apps). Modern mobile Operating Systems (e.g., Android) isolate apps from each other and from the system, implementing a dedicated sandbox for each app [137]. To be able to access confidential information such as the location, or to execute privileged operations as connecting to the network, the app must be granted the proper permissions [91]. However, all the components and processes running inside the application sandbox share the same execution environment, to which the permissions are assigned. This is a limitation that could compromise the confidentiality of data even before they are shared or outsourced. Two examples of threat are the presence of internal vulnerabilities (e.g., unprotected broadcast receivers), and the use of (untrusted) third-party libraries. The goal is therefore to reduce the exposure of confidential data to internal app components, and introduce further containment measures targeting third-party libraries.

Focusing on Android, which is open source and available to researchers, in Chapter 2 we propose a set of modifications to achieve internal app compartmentalization leveraging the Mandatory Access Control (MAC) layer. To benefit from the additional security functions, a developer can simply load inside the application archive a dedicated policy module. The policy module is written in a Common Intermediate Language (CIL) [131] dialect, and allows the developer to regulate the permissions associated with the components in a declarative way. Each component can be confined to a dedicated SELinux domain [168], effectively restricting its access to the internal storage, restricting the list of system services it can interact with, and finally denying to the component the access to the network.

After the data are collected in a safe environment, a user or a company may apply to it sanitization before being uploaded to the cloud or being released to a consumer. Data sanitization (or anonymization) is a process by which data are irreversibly altered so that a subject (referenced within the data) cannot be identified, given a certain security parameter, while the data remain practically useful. There are many approaches to apply sanitization to data, in this thesis we focus on the techniques based on generalization and suppression. Generalization involves replacing a value with a less specific but semantically consistent one [163], while suppression involves removing a datum from a data collection. These operations are usually applied in sequence until a dataset satisfies a minimum privacy requirement. One of the most famous privacy requirements is the concept of k-anonymity [65], that was

introduced to address the risk of re-identification of sanitized data through publicly available linkage datasets. To satisfy k-anonymity, a dataset is transformed by the application of multidimensional partitioning algorithms as Mondrian [121]. Mondrian identifies the partitions with a top-down recursive approach. Once the partitions are identified, generalization is applied to the them. The process is inevitably associated with a loss of information, which can be quantified with metrics like the Normalized and Global Certainty penalties, and the Discernability Penalty [188].

Data sanitization is a time and resource consuming task. In Chapter 3 of this thesis we propose an approach based on Mondrian to sanitize large collections of sensors data. The approach also ensures that the sanitized dataset satisfies the ℓ-diversity requirement [130], an extension of k-anonymity which reduces the granularity of data representation. The proposed approach is scalable, and it permits to apply sanitization in parallel, using an arbitrary number of nodes, without compromising the quality (i.e., the opposite to information loss) of the anonymized dataset.

The third part of the thesis investigates the storage and processing stages. Modern cloud providers offer state of the art security when it comes to data protection. This gives guarantees against external attackers that want to gain access to the data outsourced by a customer. The cloud provider is instead considered honest-but-curious, which assumes that it will always comply with the requests issued by the data owner, but may abuse the access to the information provided. Just to give an example, the provider may be interested in understanding which are the users the outsourced data refers to. Hence, the goal of the client is to support the execution of queries over outsourced data with a guarantee that the cloud provider does not have access to the data content.

The solution typically used to try to solve this problem is to apply an additional protection layer to the data. When choosing the encryption scheme to be used to transform the data, three key aspects must be considered: i) the encrypted data must not leak information due to its distribution, ii) the provider must be able to carry out computations over the encrypted data without requiring access to a secret decryption key, and iii) the client should be able to access the data obfuscating the access pattern. These are non-trivial requirements to be satisfied. Traditional cryptosystems do not handle requirements *i* and *ii*, only Semi or Fully Homomorphic Encryption schemes (FHE) [99] accommodates for that. Despite recent advancements in the field, for example with the availability of libraries such as Microsoft SEAL [139],

the cost of FHE, in terms of loss of performance and memory consumption, may not suit all applications. An alternative way to prevent the cloud storage provider to access the data content is the use of Oblivious Random Access Memory (ORAM) protocols [75, 171]. ORAM protocols accommodate for requirements *i* and *iii*, but assume the computation to be executed in a *safe environment* (either at the client or inside a Trusted Execution Environment) after the data of interest are pulled from the memory. Yet, even in this case, the associated overhead may be too high [40].

The approach presented in Chapter 4 is applicable to relational data, and enables the execution of queries involving evaluation of equality and range conditions over attributes. The approach addresses requirements *i* and *ii*, and provides partial protection on requirement *iii*. In particular, the data is saved encrypted to the server into equally large blocks containing a fixed set of tuples. The blocks are managed by the server as single atomic units, and accessed through an encrypted multidimensional index also stored by the server. By doing this, the cloud provider is unable to identify the single items stored within each block (more details in the chapter). Local maps are saved by the client to search the index efficiently. The approach proposed provides perfect indistinguishability to an attacker with access to the stored data (requirement *i*). This is achieved applying probabilistic encryption to the blocks storing the data, and by destroying (i.e., flattening) the frequencies of the encrypted index. The approach relies on both the client and (partially) the server to resolve the query (requirement *ii*). The index was built as an evolution of the partitioning technique used to sanitize the dataset presented in part two of the thesis.

The last part of the thesis explores the data release stage. In this scenario, we want to provide a solution to schedule the release of a secret data to a future point in time. Conversely to the idea envisioned by May in 1993 [136], we want to avoid the use of a cryptographic puzzle, as it requires a receiving party to run a decryption procedure for a long time. Furthermore, we want to ensure there is no need for the data owner for the disclosure to happen, and there is no dependency on a trusted party (or service).

The idea presented in Chapter 5 is to use the Blockchain to witness the elapse of time, and to split the secret data among a group of users who have to cooperate to recover the secret information using a pre-defined protocol. We program the protocol with a Smart Contract [172], and to ensure the parties cooperate as intended we develop a set of economic incentives and penalty. As we move to a decentralized

environment in which the parties (i.e., the network nodes) are mutually distrusting, the honest-but-curious assumption is refuted. The parties are instead modeled as rational, or rather driven by pure economic interest. This permits to analyze the protocol as an extended form game whose outcome can be determined based on participants' expected utility.

1.1 Document structure

This thesis is organized in six chapters.

Chapter 1 illustrates the structure of the document and the publications that set the basis for this thesis.

Chapter 2 describes SEApp [160], a set of modifications to the AOSP to extend the Mandatory Access Control layer to Android apps. SEApp leverages SELinux to restrict access to the internal storage, restrict access to services, and isolate vulnerability prone components. This is achieved executing components on dedicated processes. A dedicated app policy module (written in CIL) regulates the permissions associated with each process. The security measures introduced by SEApp facilitate the isolation of third-party components running inside an application. This is of particular importance in the development of secure frontends collecting confidential user information.
The chapter is organized as follows.

- Section 2.1 presents the scenario and discusses the background.
- Section 2.2 introduces the techniques currently enforcing access control in Android.
- Section 2.3 presents the motivation for the introduction in Android of dedicated components. A set of use cases is used to showcase the security measures introduced by SEApp.
- Section 2.4 details the SEApp policy module, the policy module language and the policy constraints.
- Section 2.5 illustrates the SELinux and SEAndroid policy configuration files.

- Section 2.6 discusses the changes introduced by SEApp and our implementation.
- Section 2.7 presents the experimental evaluation, in which we measure both the installation time and runtime overhead introduced by SEApp.
- Section 2.8 discusses the major differences between SEApp and other literature proposals.
- Section 2.9 concludes the chapter.

Chapter 3 describes a scalable and distributed approach to sanitize large collections of sensors data [83]. This chapter also illustrates how sanitization can be applied in parallel with limited information loss. The availability of such a tool is crucial for the outsourcing and the release of data. An open source artifact based on the content of this chapter was implemented [80]. The artifact was awarded as the *Best Artifact* at *IEEE PerCom 2021*.
The chapter is organized as follows.

- Section 3.1 presents the scenario and discusses the background.
- Section 3.2 discusses the architecture of the sanitization tool and the sanitization algorithm.
- Section 3.3 illustrates the experimental evaluation. To demonstrate the scalability of our approach a centralized version (i.e., a portable, single core version) of the sanitization tool was developed. Information loss in instead measured by the Discernability Penalty and the Global Certainty Penalty.
- Section 2.9 concludes the chapter.

Chapter 4 describes SecIdx [81, 82], an approach to support point and range queries on outsourced encrypted data. Data are stored by the server into equally large blocks, which are indexed by a multidimensional encrypted stucture also stored by the server. Local maps are saved by the client to search the index efficiently. Perfect indistinguishability of the information outsourced to the server is

guaranteed by the use of non-deterministic encryption for the blocks, and by flattening the distribution of the values in the encrypted index. Runtime query resolution involves a translation strategy based on the local maps, that permits to pull the blocks of interest from the server to the client efficiently. An *in-memory* DB is used by the client to filter spurious tuples.
The chapter is organized as follows.

- Section 4.1 introduces the rationale of the approach.
- Section 4.2 presents the basic concepts and formulates the problem statement.
- Section 4.3 details the partitioning algorithm used by the multidimensional index to organize the data into fixed sized partitions.
- Section 4.4 illustrates the construction of the encrypted index and the client-side maps.
- Section 4.5 explains runtime query translation.
- Section 4.6 details our implementation, separating operations into preprocessing and runtime.
- Section 4.7 presents an extensive experimental evaluation in which we compare the performance of relational and non-relational backends, under different configurations. The experimental evaluation demonstrates the limited performance overhead associated with our approach.
- Section 4.8 discusses the major differences between SecIdx and other literature proposals.
- Section 4.9 concludes the chapter.

Chapter 5 presents ITYT [43], a novel approach of implementing time-locked secrets. ITYT is a primitive that can be used to schedule the release of chunks or partitions of data. The approach proposed in the chapter permits to avoid the use of trusted third-parties and cryptographic puzzles. Instead, it uses threshold cryptography to split the data among a group of peers, which have to cooperate to recover the secret data using a pre-defined protocol programmed as a smart contract. The

blockchain is used to witness the elapse of time, and secure Multi-Party Computation to avoid any single point of trust.
The chapter is organized as follows.

- Section 5.1 illustrates the scenario.
- Section 5.2 describes the background.
- Section 5.3 presents an overview of the ITYT protocol introducing the preliminary definitions, the roles of the participants, and the main functions.
- Section 5.4 provides an economic model to push rational participants to cooperate as intended, strictly adhering the protocol.
- Section 5.5 illustrates how to implement ITYT leveraging existing frameworks.
- Section 5.6 discusses how ITYT ensures the methods to report misbehavior are not bypassable, how denial of service (DOS) can be mitigated, and how deadlocks can be prevented.
- Section 5.7 presents our experimental evaluation.
- Section 5.8 discusses other proposals to enable time-locked secrets from literature.
- Section 5.9 concludes the chapter.

Chapter 6 draws the conclusions of the thesis and discusses future work.

1.2 Publications

This section lists some of the publications produced during the Ph.D. course that set the basis for this thesis.

Papers in proceedings of international conferences

- Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. "**SEApp: Bringing Mandatory Access Control to Android Apps.**". *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3613-3630. 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, and Pierangela Samarati. "**Scalable distributed data anonymization**". *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 401-403. IEEE, 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, and Pierangela Samarati. "**Artifact: Scalable distributed data anonymization**". *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 450-451. IEEE, 2021.

- Enrico Bacis, Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Giovanni Livraga, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. "**Multi-provider secure processing of sensors data**". *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 349-351. IEEE, 2019.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, and Pierangela Samarati. "**Multi-dimensional indexes for point and range queries on outsourced encrypted data**". To appear in *In 2021 IEEE global communications conference (GLOBECOM)*. IEEE, 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, and Pierangela Samarati. "***k*-flat Secure Indexing for Encrypted Databases**". Under submission.
- Enrico Bacis, Dario Facchinetti, Marco Guarnieri, Marco Rosa, Matthew Rossi, Stefano Paraboschi. "**I *Told* You Tomorrow: Practical Time-Locked Secrets using Smart Contracts**". *In Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES)*, pp. 1-10. 2021.

Chapter 2

Data collection

SEApp: restricting access to in-app confidential data

The first part of this thesis investigates the collection of data, in particular in the mobile environment. The mobile environment is especially relevant, as smartphones are devices with access to a huge amount of personal and confidential information.

Operating Systems (OS) play a strategic role in protecting access to confidential information. Whenever an application (or simply *app*) tries to access a resource deemed sensitive, the OS intervenes making sure the app is acting on behalf of the user. Should the user grant the proper permission, then the request is authorized.

The OS has great success in restricting access to confidential information. However, access granularity is only at app level. This is a limitation of the current mobile operating systems. Focusing on Android which is open source, we identified three major weaknesses: *i)* all app components have full access to the internal storage, *ii)* third-party libraries may abuse the privileges granted by the user to the whole app, and *iii)* vulnerability prone components are not easy to isolate.

This chapter presents an approach to overcome these limitations, giving developers the power to define ad-hoc Mandatory Access Control policies for their apps, supporting the internal compartmentalization of app components. This guarantees stronger protection of the data collected by the app as, for example, third-party components included in the app can be prevented to access to them. Also, it permits to implement a finer control on the information accessible to the app from the

system services, selectively restricting inter-process communication on an internal compartment basis.

The approach is a natural evolution of the security mechanisms already available in Android, and the results of this part of the thesis have a potential that is even more general than the data collection phase. However, its realization requires to consider that the security of system components must be maintained, the solution must be usable by developers, and the performance impact should be limited. This proposal meets these three requirements. The proposal is supported by an open-source implementation, which has also passed the *30th Usenix Security Symposium* artifact evaluation process.

2.1 Introduction

Security in operating systems has greatly evolved and has been able to address many of the threats originating by an extensive and varied collection of adversaries.

The mitigation of security threats is particularly important for *mobile operating systems*, due to their wide deployment and the confidential information they hold.

Both Android and iOS have seen significant investments toward the realization of advanced security techniques, which have led to a great increase in the level of protection offered to users [137]. The strength of security and the value of protected resources is testified, for instance, by the payouts associated with working exploits in markets like Zerodium [190], where the payouts for mobile operating systems are the highest[1].

A peculiar threat that characterizes mobile operating systems is the need to balance on one side the high sensitivity of the information, and on the other hand the need for users to install into the system a large number of applications (called simply *apps* in this domain) often produced by unknown developers, which may hide malicious functions. A first level of protection is offered, both in iOS and Android, by a preliminary screening of apps before they are made available on the platform market [9] or installed to a device, but this approach cannot provide a strong guarantee. Security mechanisms internal to the operating system are needed in order

[1]At the time of writing, US$2.5M and US$2M are paid for a zero click solution able to subvert the security of Androd and iOS, respectively.

to constrain the apps to only operate within the boundaries specified by the device owner at installation time.

The approach used in the design of mobile operating systems considers as the first requirement the protection of system resources. Focusing on Android, which is open source and more accessible to researchers, we notice a significant evolution in its internal security architecture. This architecture is quite rich and consists of many security measures [91, 137]. In this environment, we specifically look at the role of SELinux. SELinux implements the *Mandatory Access Control* (MAC) mechanism, which relies on a system-level policy to declare the operations that a process can execute over a resource based on the security labels associated with them. Compared to classical *Discretionary Access Control* (DAC), still used in Android in an extensive way, MAC is more rigid and provides stronger guarantees against unwanted behaviors. When SELinux was introduced into Android 4.3 in 2013 (see Figure 2.1), it used a limited set of system domains and it was mainly aimed at separating system resources from user apps. In the next releases, the configuration of SELinux has progressively become more complex, with a growing set of domains isolating different services and resources, so that a bug or vulnerability in some system component does not lead to a direct compromise of the whole system.

The introduction of SELinux into Android has been a clear success. Unfortunately, the stronger protection benefits do not extend to regular apps which are assigned with a single domain named `untrusted_app`. Since Android 9, isolation of apps has increased with the use of categories, which guarantees that distinct apps operate on separate security contexts. Our proposal, SEApp, builds upon the observation that giving app developers the ability to apply MAC to the internal structure of the app would provide more robust protection against other apps and internal vulnerabilities.

2.2 Android security for apps

One of the major requirements considered in the design of mobile operating systems is the need to constrain the ability of apps to manipulate the execution environment. Apps may hide functions that are meant to gain system privileges or capture valuable information from other apps. Compared to classical desktop operating systems, there is greater reliance on the use of apps to access resources or get services,

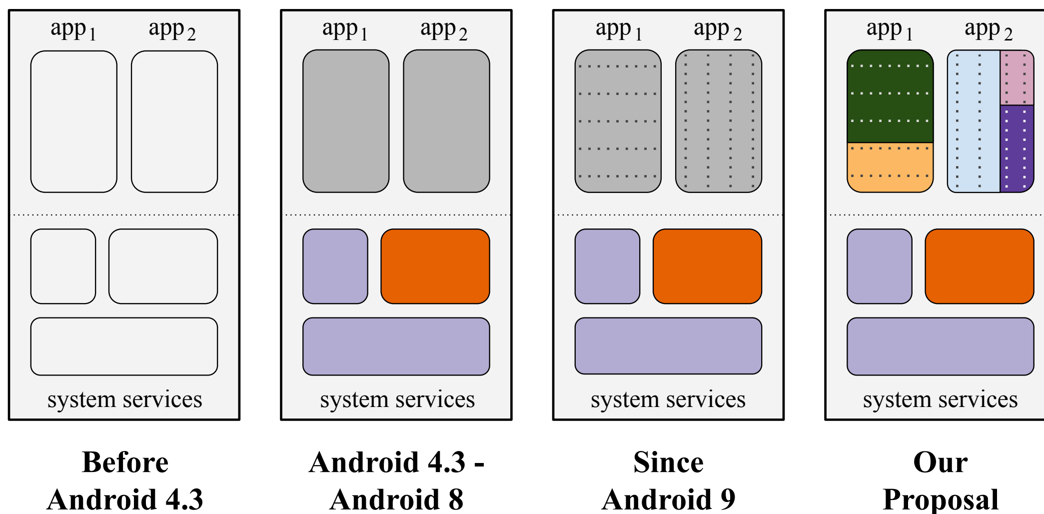


Figure 2.1: Evolution of the MAC policy in Android. Before 4.3, MAC was not used. Starting with 4.3, MAC protects system components. Since 9, categories offer rigid MAC protection for apps. Our proposal offers flexible MAC protection to apps.

with more attention paid to limit the ability of apps to operate in the system. Advancements in this context can have an impact on how security for applications is managed in other domains [5].

The basic principle adopted to manage the threat introduced by apps is the design of a *sandbox*, a restricted environment for app execution, where anomalous actions by the app are not able to access resources beyond what has been authorized at app installation time. The sandbox can be considered a realization of the "least privilege" security principle.

The construction of the app sandbox is based on three access control mechanisms: Android permissions [21,91,92], Discretionary Access Control (DAC) [62], and Mandatory Access Control (MAC) [165]; each of them roughly aligning with how users, developers, and the platform grant consent, respectively.

Android permissions restrict access to sensitive data and services. In file `AndroidManifest.xml` [23], each app statically lists the Android permissions needed to fully operate. Not all of them may be granted; depending on the threat they pose from a security and privacy standpoint, they may be granted as part of the installation procedure, or prompted to the user when the app needs them.

DAC restricts access to resources based on user and group identity. By assigning each application a unique UNIX user ID (UID) and a dedicated directory, Android isolates apps from each other and from the system. However, UID sandboxing has a number of shortcomings. As an example, processes running as root are not subject to these restrictions. For this reason, when such a process is misbehaving, for instance due to a bug, it can access private app data files. DAC discretionality itself is a problem. Indeed, as apps and system processes could override safe defaults, they are more susceptible to dangerous behavior, such as leaking files or data across security boundaries via IPC or fork/exec. Despite its deficiencies, UID sandboxing is still the primary enforcement mechanism that separates apps from each other, establishing the foundation upon which further sandbox restrictions have been built.

MAC dictates which actions are allowed based on the security policy defined by the system. Specifically, only actions explicitly granted by the policy are permitted. To decide whether to permit or deny an action, a set of policy rules concerning the *security contexts* (i.e., collections of security labels that classify resources) of the involved parties is evaluated.

In Android, MAC is implemented using SEAndroid, a set of kernel modifications part of the Linux Security Module (LSM) framework [186]. Since its first introduction with the Security Enhanced Android (SEAndroid) project [168], SELinux has been extensively applied to protect system components. Initially, it was used to assert the security model requirements during compatibility testing, then its usage grew further at each release. In the current version Android 11, SELinux is also used to isolate the rendering of untrusted web content (by the `isolated_app` domain), to restrict *ioctl* system calls [116], thus limiting the reachability of potential kernel vulnerabilities, and to support multi-user separation and app sandboxing with SELinux categories. This last aspect permits to enforce app separation both at DAC and MAC. Android dynamically assigns categories to apps during app installation, so that: (i) an app running on behalf of a user cannot read or write files created by the same app on behalf of another user (since Android 6 [16]); and, (ii) an app cannot read or write files created by another app (since Android 9 [18]). Before Android 9, this separation was only enforced at DAC level. This overlap of security measures is of extreme relevance to the enforcement of the Android Security Model and our proposal moves in the same direction. To bypass these protections, a process should be granted root permissions, `DAC_OVERRIDE`

or DAC_READ_SEARCH, and run as SELinux mlstrustedsubject; only a few critical system services run in this configuration.

Android restricts the SELinux implementation to the policy enforcement, ignoring most policy management functions. The motivation is that the system policy only changes between releases, therefore support to runtime changes is not needed.

2.3 Motivation

As discussed above, SELinux and the MAC support have been a crucial factor in the realization of a secure design and the construction of a robust app sandbox. A limitation of the current design is that this is the only way that apps can benefit from MAC support. There is currently no option to let the app developer control the use of the MAC level, as only platform, vendor, ODM and OEM developers are allowed to introduce new policy segments [31]. Our solution overcomes this limitation, giving the application developer the power to specify new SELinux types and associated permissions.

Use cases

We envision several scenarios that justify the use of SEApp. Many of them have been previously considered by researchers as motivations for the introduction in Android of dedicated components [46, 84, 113].

In this section, we give a tour of SEApp capabilities using a showcase app[2]. The architecture of the showcase app is shown in Figure 2.2. Our description is based on three use cases: fine-granularity in access to files, fine-granularity in access to services, and isolation of vulnerability prone components. Each of the use cases emphasizes the intra-app security features introduced by SEApp. A dedicated description, along with policy files that show concretely how to enforce these use cases, appears in the Appendix A; we provide there a technical demonstration of how SEApp can provide protection against a number of common security problems in Android apps [103] that were implemented in the showcase app.

[2]The showcase app is available in the SEApp repository along with the set of modifications to the AOSP.

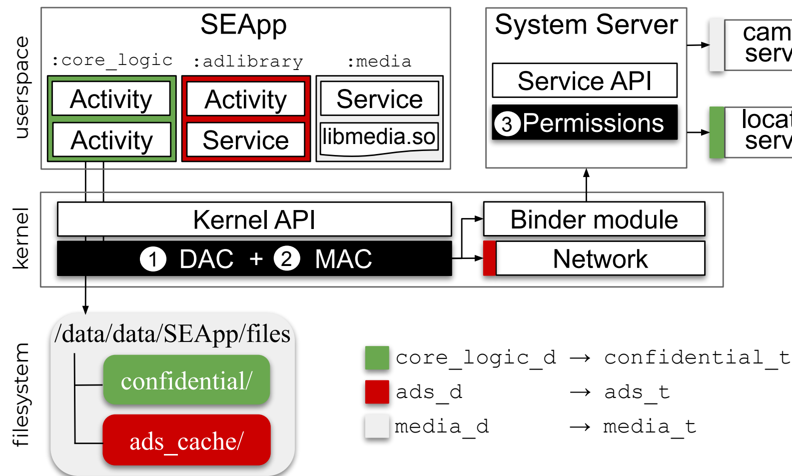


Figure 2.2: Security Enhanced App

Fine-granularity in access to files

Android apps can collect data from multiple sources, and the system provides many options to store it. The default one is *Internal Storage*: a filesystem region, located at `/data/data/packageName`, reserved to each package. Its content is available to all app's internal components and inaccessible to any other app. Since data can be extremely sensitive, the developer may be interested in restricting its visibility to only some internal components, labeling sensitive and non-sensitive data with distinct SELinux types (use case 1). Yet, in the current Android security model, apps do not have the option to assign distinct MAC labels to different resources, as all internal files are labeled `app_data_file`. SEApp allows the developer to introduce dedicated types, and to organize the app's structure with a separation between components managing non-sensitive data and those requiring access to sensitive data. The sensitive components will be associated with a more stringent MAC domain. Figure 2.2 shows an example in which the *confidential* files are made accessible to *:core_logic* processes and inaccessible to any other process.

In Appendix A.1 we give a demonstration of how *confidential* files are made inaccessible to non-confidential components in the presence of a path traversal vulnerability.

Fine-granularity in access to services

Often developers introduce into their applications code coming from external sources, which they do not fully trust [79, 93, 146]. For instance, a common need of app developers is to get revenue from their apps and a simple approach is to include an Ad delivery library within the app. The library is a relatively complex piece of code, with local computation necessities and the need to manage a dialogue with remote servers. The app developer is clearly interested in supporting the execution of the library, but may want to have guarantees that the library cannot abuse the access privileges granted by the user to the whole application sandbox (use case 2). A common concern is preventing access to system services such as *location*. These requirements can be managed by SEApp with the definition of a separate MAC domain for the library. The process managing the delivery of Ads will be associated with this domain, which will provide only the necessary privileges to access the dedicated resources needed for the library execution. SELinux will then guarantee the confinement of the library, preventing access to the location service even if the `ACCESS_FINE_LOCATION` permission is granted to the app. Figure 2.2 shows an example in which the *:adlibrary* process is granted access to the network but is prevented from accessing `location service`.

In Appendix A.2 we give a demonstration of how the showcase app can support the execution of the Unity Ads [179] framework with a dedicated SELinux domain. We also describe in detail how SEApp prevents a malicious component, which was deliberately injected by us into the library process, to capture the device location.

Isolation of vulnerability prone components

App developers often have to consider that the input provided to the app can come from untrusted sources. A typical example is the rendering of complex Javascript code performed by WebView. The solution currently offered by Android is to execute these potentially dangerous actions within a sandbox using *isolatedprocess*, i.e., a special process that is isolated from the rest of the system and has no permissions of its own [13]. It runs under a dedicated UID and SELinux domain, and it can only interact with a restricted number of services [15].

A common need of app developers is to take advantage of complex media or processing libraries, components that are not considered malicious, but due to their

size and complexity are more likely to have security bugs. The developer is then interested in isolating these potentially vulnerable components (use case 3). *Isolatedprocess* offers a high protection level in Android, however, its use imposes several restrictions on the developers. For instance, *isolatedprocess* cannot perform many of the core Android IPC functions, and the only way to interact with it is through the bound service API [14]. Also, *isolatedprocess* can only access already open app files received over Binder. Another shortcoming is that each invocation of an *isolatedprocess* requires the creation of a new process. If a series of requests are made by the app, the performance impact can be significant. SEApp offers an easier way to do this compared to *isolatedprocess*, as it permits to assign a domain to the process in which the component is executed, and then configure the required permissions at MAC level. In terms of performance, the management of multiple requests does not require the system to activate a new process with a new UID and a dedicated SELinux category. Figure 2.2 shows how to confine the *:media* component.

In Appendix A.3 we give a demonstration of how the showcase app can support the execution of media components relying on a native library in a dedicated process. We also describe how the developer can leverage SEApp to prevent the code of the library from the execution of unwanted or unintended operations, like opening a network connection.

Modular app compartmentalization

The motivations presented above become more frequent as apps increase their size and complexity, and several important apps see a continuous increase in these parameters. For instance, Facebook Messenger version 285 contains more than 500 components and WhatsApp Messenger version 2.20 more than 300. This increase in size and the need to manage it is testified by the development of App Bundles [11], Android's new, official publishing format that offers a more efficient way to build and release modular applications.

In these large and modular apps, developers find it difficult to fully control which components of an app are using sensitive data[3]. The availability of a solution such as SEApp can greatly reduce such risk. A better compartmentalization can reduce the impact of internal vulnerabilities in modular apps, since each module can be asso-

[3]The topic was explicitly considered in [38], an interview with Android's VP of Engineering.

ciated with a dedicated policy fragment. From a security and software engineering standpoint, SEApp permits to separate the activities of security policy maintenance and development of new features.

Compatibility with Android design

Looking at the evolution of Android, it is clear that our proposal is consistent with the evolution of the operating system and the desire of its designers to let app developers have access to an extensive and flexible collection of security tools. The major obstacles, as perceived by OS developers, on offering to app developers the use of MAC services are: weakening of the protection of system components; performance impact; usability by app developers. The work we did solves these concerns: our approach guarantees that app policies do not have an impact on the system policy (Section 2.4); the app policy can be specified declaratively and attention has been paid to let developers adopt the approach in a convenient way (Section 2.5); and, experiments demonstrate the acceptable performance impact, with a quite limited overhead at app installation time, and a negligible runtime impact (Section 2.7).

Compatibility with other proposals

As presented in Section 2.3, SEApp by itself provides protection against a broad spectrum of attacks (see Appendix A), but its merit does not end there. As multiple literature proposals (e.g., [48, 113, 187]) build upon process isolation and use it to accomplish separation of privileges at the application layer, SEApp could be used as building block to enforce such restrictions at the MAC layer too, enabling defense in depth. Moreover, SEApp could also work in conjunction with other solutions that work at MAC level such as *FlaskDroid* [56], to benefit of its Userspace Object Managers (USOMs) coverage of the Android system services and provide finer granularity in access to services.

2.4 Policy language

To support the use cases presented in Section 2.3, we want the developer to have control of the SELinux security context of subjects and objects related to her security enhanced app. To each of them is assigned a *type* (also called *domain* when it

labels processes). As types directly relate to groups of permissions, the evaluation of security contexts is the foundation of each security decision. Since apps may offer many complex functions, the policy language has to provide the flexibility of defining multiple domains with distinct privileges so that the app, according to the task it has to do, may switch to the least privileged domain needed to accomplish the job.

Policy module syntax	
blockStmt	→ (`block` *blockId cilStmt*∗)
cilStmt	→ *typeStmt* | *typeAttrStmt* | *typeAttrSetStmt* | *typeBoundsStmt* | *typeTransStmt* | *macroStmt* | *allowStmt*
typeStmt	→ (`type` *typeId*)
typeAttrStmt	→ (`typeattribute` *typeAttrId*)
typeAttrSetStmt	→ (`typeattributeset` *typeAttrId* (⟨*typeId* | *typaAttrId*⟩+))
typeBoundsStmt	→ (`typebounds` *parentTypeId childTypeId*)
typeTransStmt	→ (`typetransition` *sourceTypeId targetTypeId classId* [*objectName*] *defaultTypeId*)
macroStmt	→ (`call` *macroId* (*typeId*))
allowStmt	→ (`allow` ⟨*sourceTypeId* | *sourceTypeAttrId*⟩ ⟨*targetTypeId* | *targetTypeAttrId* | `self`⟩ *classPermissionId*+)

Table 2.1: Application policy module CIL syntax

The app policy is specified in a module, provided by the app to describe its own types. The policy module is processed at app installation time by a component of the system, called *SEApp Policy Parser*, responsible to verify that the policy is correct and does not introduce vulnerabilities into the system. The addition of a policy module is managed by combining the new module with the platform policy and the previous installed ones, producing after policy compilation a single binary representation of the global policy.

In this section, we provide a description of the SEApp policy language and the restrictions each module is subject to. Policy configuration is detailed in Section 2.5, while policy compilation and runtime support are discussed in Section 2.6.

Choice of policy language

SEAndroid supports two languages for policies, Type Enforcement (TE) [177] and Common Intermediate Language (CIL) [131]. TE was the language available in the early implementations of SELinux, while CIL was later introduced to offer an easy

to parse syntax that avoids the pervasive use of general purpose macro processors (e.g., M4 [95]). Another aspect that differentiates them is that, in Android, TE representations are internally converted into CIL before being compiled into the SELinux binary policy. To avoid the additional translation step being performed at each policy module installation, we decided to use CIL over TE.

Definition of types and type-attributes

CIL offers a multitude of commands to define a policy, but only a subset has been selected for the definition of an app policy module. This was done to control the impact of the policy module on the system and it may, as a side effect, facilitate the work of the app developer writing the policy.

The syntax is described in Table 2.1. To declare a *type*, the `type` statement can be used. This permits to declare the types involved in an access vector (AV) rule, which grants to a source type a list of permissible actions over a target type. AV rules are defined through the `allow` statement.

When writing a policy, there is frequently the need to assign the same set of authorizations to multiple types. To avoid the repetition of multiple `allow` declarations, it is convenient to refer to multiple types using a single entity, the *type-attribute*. Using the `typeattributeset` statement we associate with a `typeattribute` a set of types and type-attributes. Each type-attribute essentially represents the set of types that is produced by the (possibly multi-step) expansion of its definition. The semantics is that each of the types that directly or indirectly (using type-attributes) appears as the source of an allow rule will be authorized to operate with the specified permission on each of the types directly or indirectly appearing as the target. This improves the conciseness and readability of the policy.

After defining the domains with the least group of permissions necessary to fulfill the task, the developer can also configure the domain transitions using the `typetransition` statement. By doing so, it is possible to ensure that important native processes run in dedicated domains with limited privileges, leading to intra-app compartmentalization.

Policy constraints

The introduction of dedicated modules for apps raises the need to carefully consider the integration of apps and system policies. The first requirement is that an app policy must not change the system policy and can only have an impact on processes and resources associated with the app itself. To preserve the overall consistency of the SELinux policy, each policy module must respect some constraints. Since Android supports the side-loading of apps [10], we cannot rely on app markets to verify app policies. Therefore, the enforcement of constraints is done on the device, by both the SEApp Policy Parser and the SELinux environment. If any of these components raises an exception, during the verification or compilation of the policy, app installation is stopped.

To ensure that policy modules do not interfere with the system policy and among each other, a first necessity is that policy modules are wrapped in a unique namespace obtained from the package name. This is done through the `block` CIL statement, which prevents the definition of the same SELinux type twice, as the resulting global identifier is formed by the concatenation of the namespace and the local type identifier. Also, the use of a namespace specific for the policy module permits to discriminate between local types or type-attributes T_A (namespace equal to the current app package name), types or type-attributes of other modules $T_{A' \neq A}$ (namespace equal to some other app package), and system types or type-attributes T_S (system namespace). At installation time, the SEApp Policy Parser determines the origin of each type, with an explicit prohibition for policies to refer to types or type-attributes defined by other policy modules, while use of system types or type-attributes is subject to restrictions.

With regard to the `allow` statement, a dedicated analysis is performed by the SEApp Policy Parser. For each rule, the global origin of source and target types is determined. We refer to system origin *S*, when the type is directly or indirectly associated with a system type in the expansion of its definition, while to local origin *A* otherwise. Based on the origin of source and target of each rule, there are four cases. The case *AllowSS*, i.e., a permission with system origin both as source and target, is prohibited, as it represents a direct platform policy modification. The case *AllowAA* is always permitted, as it only defines access privileges internal to the app module. The cases *AllowAS* and *AllowSA* are more delicate.

An *AllowAS* originates when a local type needs to be granted a permission on a system type. A concrete example is shown in Section 2.3, where the *:media* process needs access to the `camera_service`. The case cannot be decided locally by the SEApp Policy Parser, therefore it is delegated to the SELinux decision engine during policy enforcement. This crucial postponed restriction depends on the constraint that all app types have to appear in a `typebounds` statement [45], which limits the bounded type to have at most the access privileges of the bounding type. As Android 11 assigns to generic third-party apps the `untrusted_app` domain, this is the candidate we use to bound the app types. If the *AllowAS* rule gives to the local type more privileges than those associated with `untrusted_app`, and at runtime these privileges are used, the SELinux decision engine identifies the policy violation and prohibits the action.

AllowSA rules are the key to regulate how system components access internal types. To be compliant with Android, the local types introduced by the app policy module must ensure interoperability with system services crucial to the app lifecycle. As an example *Zygote* [36], the native service which spawns and configures new app processes, can only execute processes labeled with the type-attribute `domain`, which is assigned by default to `untrusted_app`. However, giving app developers the freedom to directly define *AllowSA* rules would lead to two major issues: (i) the rules would depend on system policy internals, leading to a solution with limited abstraction and modularity; (ii) explicit *AllowSA* rules could lead to violations of the security assumptions of a system service, with the risk of introducing vulnerabilities (e.g., leading to a *confused deputy attack* [55]). For these reasons we prohibit their explicit use. To limit system types to only those already dealing with untrusted content and simplifying the policy, we rely on CIL macros, a set of function-like statements that, when invoked by the SEApp policy module, produce a predefined list of policy statements. This approach permits to retain control on the rules produced, ensuring no violation of the default system policy. Also, it makes the work of the developer easier, by abstracting away system policy internal details. To preserve the interoperability with system services, third-party app functionality has been broken down into the CIL macros listed in Table 2.2. This list has been identified looking at the internal structure of the `untrusted_app` domain. With this design philosophy, the developer can grant a basic set of permissions to a type

Macro	**Usage**
md_appdomain	to label app domains
md_netdomain	to access network
md_bluetoothdomain	to access bluetooth
md_untrusteddomain	to get full untrusted app permissions
mt_appdatafile	to label app files

Table 2.2: SEApp macros to grant permissions to local types

(by calling one or more macros), and then add to it fine-grained authorizations with *AllowAS* rules.

With regard to the `typeattributeset` statement, the SEApp Policy Parser uses a verification strategy similar to the one used for allow rules. First, the global origin of the type-attribute and of the set expression of types and type-attributes is determined. All statements that directly or indirectly relate to system types are blocked. This avoids implicit permission propagation from system and local types.

Similarly, for the `typetransition` statement, the SEApp Policy Parser verifies the origin of the types involved, with a prohibition for all the statements that relate to system types, as they may lead to an escalation of privileges.

2.5 Policy configuration

In this section, we explore the structure of application policy modules. Before describing the content of SEApp configuration files, we give a short description of how SEAndroid defines the security contexts of processes, files and system services. There are strong similarities between the structure of system and app policies. Indeed, we designed our solution as a natural extension of the approach used to protect the system. Also, our design maintains full backward compatibility. Developers who are not interested in taking advantage of MAC capabilities do not have to change their apps.

SEAndroid policy structure

Compared to a traditional Linux implementation, Android expands the set of configuration files where SELinux [25] security contexts are described, because a wider set of entities is supported. SEAndroid complements the common

SELinux files (i.e., `file_contexts` and `genfs_contexts`) with 4 additional ones: `property_contexts`, `service_contexts`, `seapp_contexts` and `mac_permissions.xml`. Also, the implementation of the SELinux library (*libselinux*) [178] has been modified introducing new functions (to assign domains to app processes and types to their dedicated directory). We concisely describe the role of SEAndroid context files.

Processes

With reference to app processes, Android assigns the security context based on the class the app falls in. The specification of the classes and their security labels are defined in the `seapp_contexts` policy file. Most classes state two security contexts: one for the process (`domain` property) and the other one for the app dedicated directory (`type` property). A number of *input selectors* determine the association of an app with a class. Among these, `seinfo` filters on the tag associated with the X.509 certificate used by the developer to sign the app. The mapping between the certificate and the `seinfo` tag is achieved by the `mac_permissions.xml` configuration file. Since the enumeration of all third-party app certificates is not possible a priori, all third-party apps are labeled with the `untrusted_app` domain by default.

Files

SELinux splits the configuration of security contexts of files between `file_contexts` and `genfs_contexts`, with the former used with filesystems that support extended file attributes (e.g., `/data`), while the latter with the ones that do not (e.g., `/proc`). To apply `file_contexts` updates, two approaches are available: either rebuild the filesystem image, or run *restorecon* operation on the file or directory to be relabeled (this is the default method used by permissioned system processes). Conversely, to apply `genfs_contexts` changes, a reboot of the device or a sequence of filesystem *un-mount* and *mount* operations has to be performed.

Services

Unlike what happens for system processes, a system service requires the assignment of a security context to both its processes and its *Binder* [24], to be fully compliant

with SEAndroid. The *Binder* is the lightweight inter-process communication primitive bridging access to a service. Its retrieval is enabled by the *servicemanager*, a process started during device boot-up to keep track of all the services available on the device. Based on the labels specified in the `service_contexts` file, it is then possible to control which processes can register (*add*) and lookup (*find*) a *Binder* reference for the service, and therefore connect to it. However, since *Binder* handles resemble tokens with almost unconstrained delegation, denying a process to get the *Binder* through the *servicemanager* does not prevent the process from obtaining it by other means (e.g., by abusing other processes that already hold it). Furthermore, preventing a process from obtaining a *Binder* reference prevents the process from using any functionality exposed by the service.

SEApp policy structure

Developers interested in taking advantage of our approach to improve the security of their apps are required to load the policy into their Android Package (APK). A predefined directory, `policy`, at the root of the archive, is where the SEApp-aware package installer will be looking for the policy module (see Figure 2.3). Inside this directory, the installer looks for four files (which we refer to as *local*), that outline a policy structure similar to the one of the system. Specifically, the developer is able to operate at two different levels: (i) the actual definition of the app policy logic using the policy language described in Section 2.4 (in the local file `sepolicy.cil`), and (ii) the configuration of the *security context* for each process (in the local files `seapp_contexts` and `mac_permissions.xml`) and for each file directory (in the local file `file_contexts`).

Processes

SEApp permits to assign a SELinux domain to each process of the security enhanced app. To do this, the developer lists in the local `seapp_contexts` a set of entries that determine the security context to use for its processes. For each entry, we restrict the list of valid input selectors to `user`, `seinfo` and `name`: `user` is a selector based upon the type of UID; `seinfo` matches the app seinfo tag contained in the local `mac_permissions.xml` configuration file; `name` matches either a prefix or the whole process name. The conjunction of these selectors determines

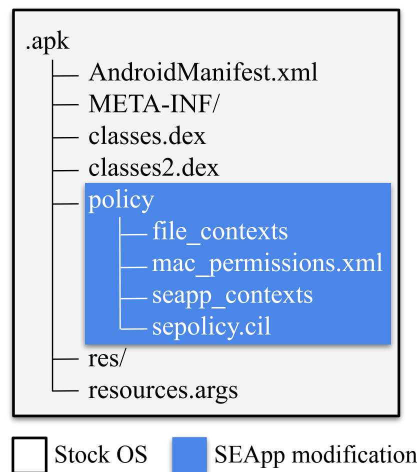


Figure 2.3: SEApp policy structure

a class of processes, to which the context specified by `domain` is assigned. To avoid privilege escalation, the only permitted domains are the ones the app defines within its policy module and `untrusted_app`. As a process may fall into multiple classes, the most selective one, with respect to the input selector, is chosen. An example of valid local `seapp_contexts` entries is shown in Listing 2.1, which shows the assignment of the *unclassified* and *secret* domains to the *:unclassified* and *:secret* processes, respectively.

In Android, developers have to focus on components rather than processes. Normally, all components of an application run in a single process. However, it is possible to change this default behavior setting the `android:process` attribute of the respective component inside the `AndroidManifest.xml`, thus declaring what is usually called a *remote* component. Furthermore, with the specification of an `android:process` consistent with the local `seapp_contexts` configuration, we support the assignment of distinct domains to app components. To execute the component, the developer is only required to create the proper *Intent* object [28], as she would have already done on stock Android for remote components. The assignment to the process of the correct domain is handled by the system. This design choice allows us to support Android activities, services, broadcast receivers and content providers, while avoiding changes to the *PackageParser* [152], as there are no modifications to the manifest schema.

Files

The developer states the SELinux security contexts of internal files in the local `file_contexts`. Each of its entries presents three syntactic elements, `pathname_regexp`, `file_type` and `security_context`: `pathname_regexp` defines the directory the entry is referred to (it can be a specific path or a regular expression); `file_type` describes the class of filesystem resource (i.e., directory, file, etc.); `security_context` is the security context used to label the resource. The admissible entries are those confined to the app dedicated directory and using types defined by the app policy module, with the exception of `app_data_file`. Due to the regexp support, a path may suit more entries, in which case the most specific one is used. Examples of valid local `file_contexts` entries are shown in Listing 2.2: the first line describes the default label for app files, second and third line respectively specify the label for files in directories `dir/unclassified` and `dir/secret`.

In SELinux, the security context of a file is inherited from the parent folder, even though `file_contexts` might state otherwise. Since, for our approach, it is essential that files are labeled as expected by the developer, we decided to enforce file relabeling at creation. Therefore, a new native service has been added to the system (see Section 2.6). We then offer to the developer an alternative implementation of class `java.io.File`, named `android.os.File`, which sets file and directory context upon its creation, transparently handling the call to our service.

System services

To support any third-party app, the `untrusted_app` domain grants to a process the permissions to access all system services an app could require in the `AndroidManifest.xml`. As an example, in Android 11, the `untrusted_app_all.te` platform policy file [35] permits to a process labeled with `untrusted_app` to access `audioserver`, `camera`, `location`, `mediaserver`, `nfc` services and many more.

To prevent certain components of the app from holding the privilege to bind to unnecessary system services, the developer defines a domain with a subset of the `untrusted_app` privileges (in the local `sepolicy.cil` file), and then she ensures the components are executed in the process labeled with it. Listing 2.3 shows

an example in which the `cameraserver` service is made accessible to the *secret* process.

```
1 user=_app seinfo=cert_id domain=package_name.unclassified name=
   package.name:unclassified
2 user=_app seinfo=cert_id domain=package_name.secret name=package
   .name:secret
```

Listing 2.1: `seapp_contexts` example

```
1 .*                 u:object_r:app_data_file:s0
2 dir/unclassified  u:object_r:package_name.unclassified_file:s0
3 dir/secret        u:object_r:package_name.secret_file:s0
```

Listing 2.2: `file_contexts` example

```
1 (block package_name
2   (type secret)
3   (call md_appdomain (secret))
4   (typebounds untrusted_app secret)
5   (allow secret cameraserver_service (service_manager (find)))
    ...)
```

Listing 2.3: Granting `cameraserver` access to secret domain

2.6 Implementation

In this section, we describe the main changes introduced in Android by SEApp. We first analyze the modifications required to manage policy modules, both during device boot and at app installation. We then describe how the runtime support was realized.

Policy compilation

Boot procedure

Since the introduction of Project Treble [17], policy files are split among multiple partitions, one for each device maintainer (i.e., platform, SoC vendor, ODM, and OEM). This feature facilitates updates to new versions of Android, separating the Android OS Framework from the device-specific low-level software written by the

chip manufacturers. Yet, each time a partition policy (i.e., a segment) changes, an on-device compilation is required.

The *init* process divides its operations in three stages [26]: (i) *first stage* (early mount), (ii) SELinux setup, and (iii) *second stage* (init.rc). The first stage mounts the essential partitions (i.e., `/dev`, `/proc`, `/sys` and `/sys/fs/selinux`), alongside some other partitions specified as early mounted (since Android 10 using an `fstab` file in the first stage ramdisk, in Android 9 and lower adding fstab entries using device tree overlays). Once the required partitions are mounted, *init* enters the SELinux setup. As the name suggests, this is the stage where *init* loads the SELinux policy. As the `/data` partition, where policy modules are stored, is not yet mounted, it is not yet possible to integrate them with the policy of the system. Then, as last operation of the SELinux setup stage, *init* re-executes itself to transition from the initial `kernel` domain to the `init` domain, entering the second stage. As the second stage starts, *init* parses the `init.rc` files and performs the builtin functions listed there, among them mounting the `/data` partition. Now, the policy modules are available, and we can produce with *secilc* [33] (the SELinux CIL compiler) the binary policy consisting of the integration among the system policy, the SEApp macros and the app policy modules. To trigger the build and reload of the policy, we implemented a new builtin function, and modified the `init.rc` to call this function right after `/data` is mounted. The policy is considered immediately after the `/data` partition is available and this ensures that the policy modules are loaded far before an application starts, making the policy not bypassable.

Even though most Android devices supporting Android 10 were released with Treble support and, therefore, execute their SELinux setup stage on the `sepolicy.cil` fragments scattered among multiple partitions, *init* still supports the use of a legacy monolithic binary policy. For compatibility towards devices using a monolithic binary policy, additional changes are required, as SEApp needs the system policy written in CIL to be compiled alongside with app modules. To this end, we modified the Android build process to push the `sepolicy.cil` files onto the device even for non-Treble devices. New entries in the device tree were added to make the policy segments available during *init* SELinux setup stage [29].

As previously mentioned, we decided to store the policy modules in the `/data` partition; even if this choice required us to adapt the boot procedure of the device, it smoothly integrates SEApp with the current Android design. In fact, the `/data`

partition is one of the few writable partitions, it is dedicated to hold the APK the user installs, as well as their dedicated data directories and, therefore, it represents the best option to contain also the app policy modules. Moreover, whenever a user performs a factory reset, Android automatically wipes the `/data` partition, removing the customization the user made to the device configuration, including the apps. By placing the app policy modules and the apps into the same partition, a factory reset removes the policy modules as well.

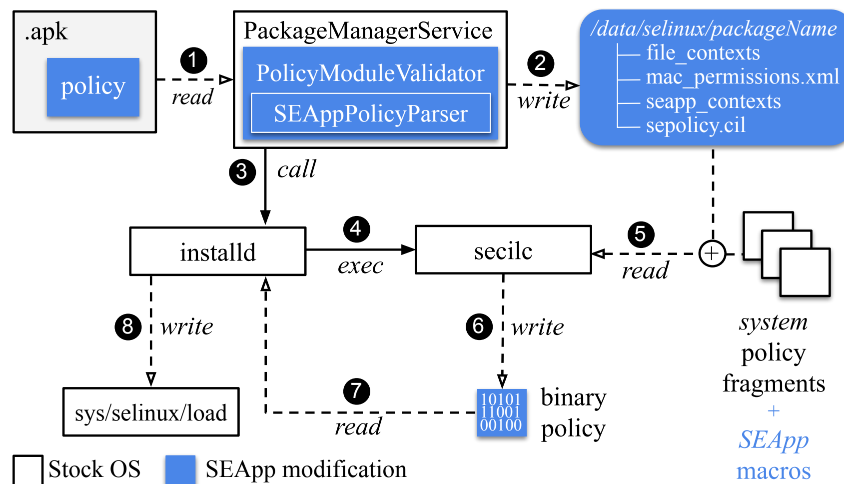


Figure 2.4: Installation process

App installation

As introduced in Section 2.5, the developer willing to define its own policy module is expected to load it in the app package. At app installation, the *PackageManagerService* [30] inspects the APK to identify whether or not the current installation involves a policy module, by looking for the `policy` directory at the root of the archive. When the app has a policy module attached to it (see Figure 2.4), the *PackageManagerService* extracts it (❶) and uses our *PolicyModuleValidator* to verify the respect of all the constraints on `sepolicy.cil` (through the *SEAppPolicyParser*, Section 2.4) and on the configuration files (Section 2.5). In case of a violation of the constraints, the app installation stops. Otherwise, the policy module is stored within `/data/selinux`, in a dedicated directory identified by the package name (❷). Then, the *PackageManagerService* invokes *installd* [27]

through the *Installer* to trigger the policy compilation with an exec call to the *secilc* program (❸, ❹). *Secilc* reads the system `sepolicy.cil` fragments, the SEApp macros and the `sepolicy.cil` fragments of the app policy modules in the `/data/selinux` directory (❺), and builds the binary policy (❻). When the *secilc* execution returns and no compilation errors have been raised, the binary policy is then read by *installd* (❼) and loaded with *selinux_android_load_policy*, which writes the `sys/selinux/load` file (❽).

To load the policy files after *init*, the implementation of SELinux in Android has been slightly modified. In particular, we modified the policy loading function within *libselinux* (function *selinux_android_load_policy*), and changed the system policy to allow *installd* to load the app policy module.

As for the policy configuration files, some changes were introduced to load the application `file_contexts`, `seapp_contexts` and `mac_permissions.xml`. *SELinuxMMAC* [34], i.e., the class responsible for loading the appropriate `mac_permissions.xml` file and assigning `seinfo` values to apks, was modified to load the new `mac_permissions.xml` specified within the app policy module. The loading of `file_contexts` and `seapp_contexts` was configured to treat system and app configuration files apart. So, SEApp-enhanced applications will load exclusively their configuration files, whereas the loading of system's and other apps' configuration files is not needed since their use is prohibited. System services and daemons, instead, load the base system configurations once, and then load the app policy module specific configuration files as they are needed. An example of this are *Zygote* and *restorecon* services, which need to retrieve at runtime `seapp_contexts` and `file_contexts`, respectively (see Section 2.6).

Our implementation also supports the uninstallation of SEApp apps. The regular uninstallation process is extended with a step where the global policy is recompiled, in order to remove the impact of old modules on the overall binary policy. With reference to application updates, the native *installd* runs with the necessary permission to remove and apply new file types based on the content of the `file_contexts`.

Runtime support

In addition to the steps described above, other aspects have to be considered in order to extend SELinux support at the application layer.

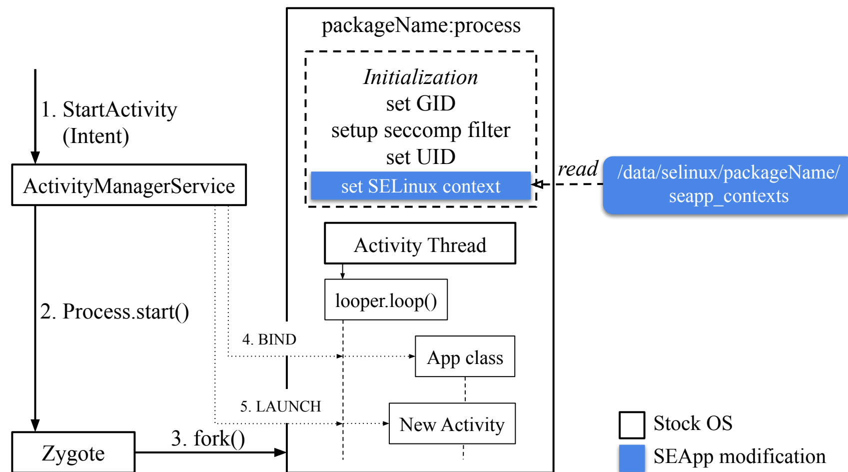


Figure 2.5: Application launch

Processes

Android application design is based on components. Each of them lives inside a process, and can be seen as an entry point through which the system or the user can enter the app.

To activate a component, an asynchronous message called intent, containing both the reference to the target component and parameters needed for its execution, has to be created. The intent is then routed by the system to the *ActivityManagerService* [19] via *Binder IPC*. Before delivering the intent request to the target component, the *ActivityManagerService* checks if the process in which the target component should be executed is already running; if not, the native service called *Zygote* [36] is executed. Its role is to spawn and correctly setup the new application process. To achieve this, it first replicates itself by performing a fork, then, using the input provided by the *ActivityManagerService* (namely, package name, `seinfo`, `android:process`, etc.), it starts configuring the process GID, the *seccomp* filter, the UID and finally the SELinux security context. We adapted the final configuration step, forcing *Zygote* to set the security context based on the `seapp_contexts` located at `/data/selinux/packageName` (i.e., the one provided by the developer for her app). Process name is used to assign the proper context to the process when it starts, before the logic of the process kicks in. In case the developer did not specify a domain, then *Zygote* uses the system `seapp_contexts` as fallback. After the correct labeling, the *ActivityManager-*

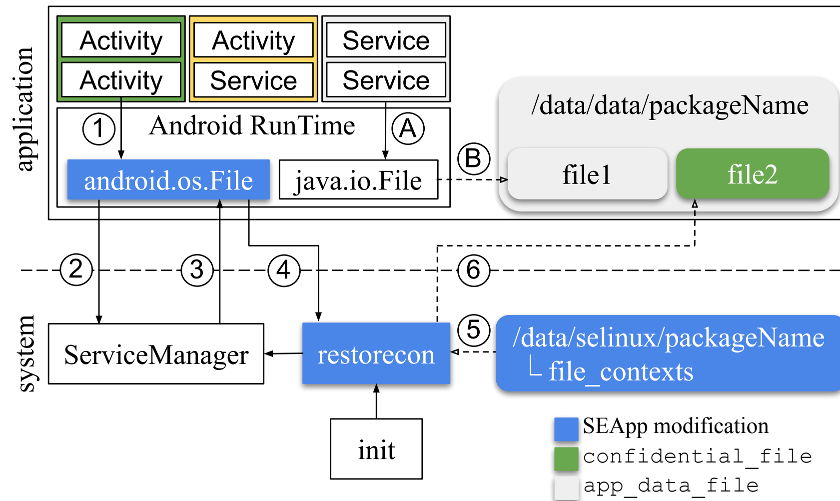


Figure 2.6: File relabeling

Service finishes the configuration by binding the application class, launching the component, and finally delivering the intent message. Figure 2.5 details the process.

This implementation design offers several benefits, including backward compatibility, support for all components, and ease of use. Indeed, a developer who wants to use our solution only has to configure some files; changes in the application code are reduced to a minimum, thus facilitating the introduction of SELinux in already existing apps.

In our study we have also explored other design alternatives, in which the developer could explicitly state a domain transition in the code, wherever she needs it. Although this category of solutions would give the developers more control over domain transitions, it also has some drawbacks. First, the developer would be expected to enforce the isolation among source and target domains managing the multi-threaded scenario, and second, this design implies granting too many permissions to the app (e.g., `dyntransition`, `setcurrent` and read/write access to `selinuxfs`). Moreover, such solution would introduce a new Android API, that would be quite delicate and, if not used correctly, it might be difficult to control.

Files

Android applications aiming to create a file can use the `java.io.File` abstraction. Each file creation request that is generated is captured by the Android Runtime (ART) [22], and then converted into the appropriate syscall. The result is the creation of the target file, to which a security context inherited from the parent directory is assigned (see flow Ⓐ, Ⓑ of Figure 2.6). Since Android 9, the separation between files of different apps is enforced at MAC level (a unique context based on UID and SELinux category is assigned); however, all the files stored in the same app folder are labeled with the `app_data_file` type.

To make the most out of SELinux, SEApp complements Android with the implementation of a new service, which we called *restorecon* (to recall the SELinux *restorecon.c* tool). The *restorecon* service is spawned by *init* at boot, and works in its own SELinux domain. Its role is to create and label files as specified by the developer in the local `file_contexts`. To ease development, we implemented the new `android.os.File` abstraction, which exposes an interface equal to that of `java.io.File`, and transparently handles the call to our service. Figure 2.6 details the new control flow. A component running in a SEApp-enhanced process (highlighted in green in Figure 2.6) invokes `android.os.file`, and triggers a new file creation request (①). The new API first interacts with the *ServiceManager* (②) to get a handle of the *restorecon* service (③), then it interacts with the service using the AIDL [12] interface we defined for it, informing the *restorecon* of the target path (④). The *restorecon* service verifies whether the caller is the legitimate owner of the path, it reads the `file_contexts` file located at `/data/selinux/packageName` (⑤), and finally it creates the target file enforcing the correct labeling (⑥).

We also investigated three other implementation approaches: (i) change of the default security context inheritance behavior for the *ext4* filesystem, (ii) execution of the SELinux *restorecon* operation by the app, once the file is successfully created, and (iii) use of *restorecond* [32]. The first option would change the default behavior system-wide. As it might cause compatibility issues, we decided not to choose it. The second option is not ideal from a security standpoint, as it requires to grant the application too many permissions (e.g., `relabelfrom`, `relabelto`, as well as read/write access to `selinuxfs` to check the validity of the SELinux context). The third option refers to the use of *restorecond*, a system daemon that watches

(inodes of) a configurable list of files and checks that they are labeled as stated in the system `file_contexts`. Although it may realize the control, *restorecond* was meant for a few system files, therefore its performance would hardly scale, especially considering that SEApp needs to manage all files created by SEApp-aware apps. Another major issue is that this approach is exposed to race conditions, because there is a delay between file creation and its relabeling.

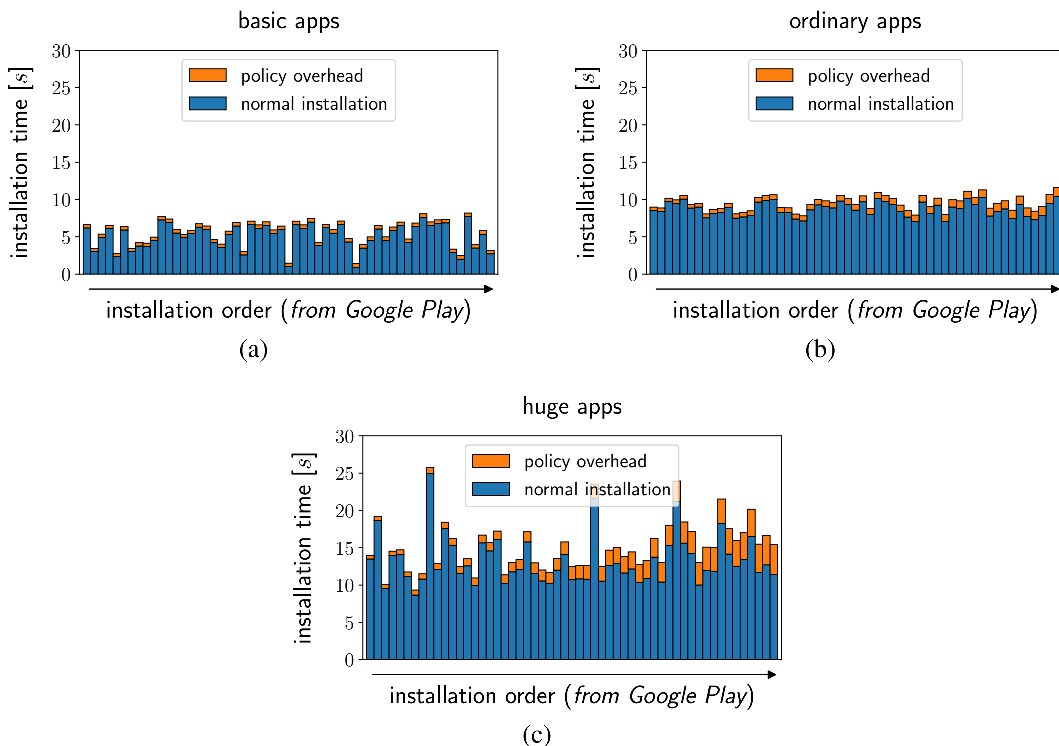


Figure 2.7: Installation time overhead for apps with different complexity

2.7 Experimental results

We now present a performance evaluation of SEApp. The experiments have been conducted on both Android 9 and 10, each with Linux kernel v4.9. However, all the measurements shown refer to Android 10 (release android-10.0.0_r41). The device used to run the tests is a Google Pixel 3 (*blueline*), in which the four *gold* cores frequency was set to 2.8 GHz, while the four *silver* ones were disabled. The change

in CPU configuration has been performed to reduce the variability of measures. The confidence intervals provided have an associated confidence level of 99%.

App installation

The introduction of dedicated app policies implies further steps to be executed at app installation time, as each SEApp module has to be validated, compiled, and loaded. To evaluate the impact on performance, we wrote dedicated tests to stress the installation procedure with multiple application samples.

To build representative samples of a typical consumer scenario, we first downloaded the 150 most popular free apps from Google Play (retrieved in October 2020) [104]. The apps were subsequently divided into three buckets: *basic*, *ordinary* and *huge* apps, according to the weighted normalized average of the .apk size, the number of Android activities and the number of services. Based on the bucket, each app was equipped with one of the following policy configurations: (i) *basic*, 1 domain and 1 type per policy module, (ii) *ordinary*, 10 domains and 25 types, and (iii) *huge*, 20 domains and 100 types. The rationale is that larger apps can gain considerable benefit from the use of a large policy. The *basic* configuration mimics how third-party apps are currently handled, but with some key improvements, as it permits to define the subset of services the domain can use, and it permits to enforce app isolation, not only based on MAC category, but also through the specification of its own type. The *ordinary* and *huge* policy configurations are meant to take full advantage of intra-app isolation and flexibility via the definition of multiple domains. Each test was repeated five times, measuring the time each package took to install. The measurements were done with the *nix *date* utility.

Test I. To measure the overhead caused by the presence of the policy module, we performed on device installation of each of the previously described app buckets (*basic*, *ordinary* and *huge*) via Android Debug Bridge (adb) [20].

The results of Test I are illustrated in Figure 2.7. In detail, it shows in blue (i.e., the lower part of the bar) the time required by the system to install the current package without the dedicated policy module, while in orange (i.e., the top of the bar) the overhead caused by the presence of the policy module. The data report that a limited overhead is associated with apps with *huge* policies, at most $3.59 \pm 0.04s$, while *basic* and *ordinary* policy configurations exhibit a negligible slowdown, never exceeding $1.22 \pm 0.02s$.

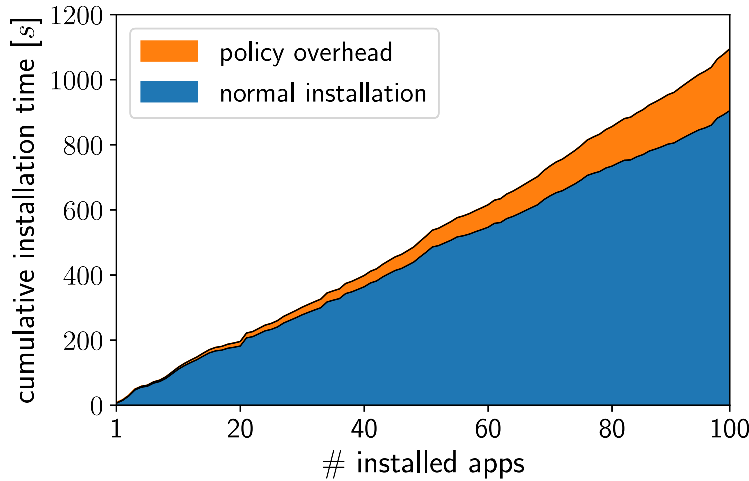


Figure 2.8: Cumulative install time overhead when installing the top 100 free apps on Google Play Store with our policies

Test II. To evaluate the overall impact of SEApp in a typical consumer scenario, we performed a test evaluating cumulative installations. At first, we repeated the installation of the top 100 apps on Google Play Store with the same policy configuration as in Test I (see Figure 2.8). In this case, we measured an overhead of $20.98 \pm 1,31\%$ on total installation time.

As explained in Section 2.6, each time a new application is installed, all policy fragments stored in the device have to be recompiled to produce the new binary policy. The installation time overhead then grows with the increase in the number of installed policy modules. To further analyze this aspect, we repeated the installation of the top 100 free apps adding to all the packages in three separate experiments the same *basic*, *ordinary*, and *huge* policy configurations. The experimental results illustrated in Figure 2.9, show that only the use of *huge* policy modules introduces a non-negligible overhead ($45.35 \pm 2.44\%$ on total installation time). However, this policy configuration simulates an edge case, as we do not expect to find 100 of them in a real scenario. To give a comparison, the *huge* policy declares 100 types; `public/file.te`, i.e., the file used to define all the file types of the system, declares 314 types in Android 10.

In Table 2.3 we report the sizes of the overall policies for the three scenarios considered in this experiment. We report the number of MAC types, the number of produced AV rules, and the overall size in KBytes of the binary policy.

policy	#types	#avrules	KB
system	1536	29228	596
system + 100 basic	1836	47028	867
system + 100 ordinary	6036	213228	3512
system + 100 huge	15536	417228	7064

Table 2.3: Policy size

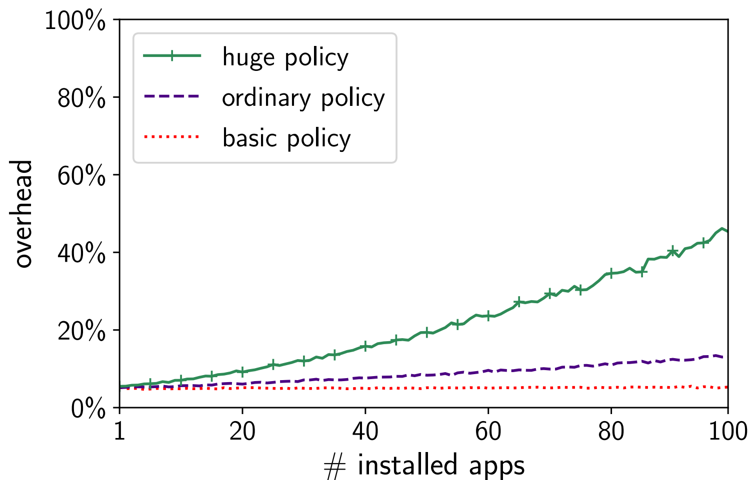


Figure 2.9: Install time overhead for the three policy sizes

Runtime performance

We now evaluate the runtime overhead for an app taking full advantage of SEApp. We focus on the creation of processes and files, as they are the entities directly affected by the changes made in the implementation. The data shown refer to the creation time of each resource. The measurements have been acquired via *System.nanoTime* and have been repeated 100 times for each test. Also, all outliers diverging more than 3 standard deviations from the mean have been suppressed.

Processes

As discussed in Section 2.6, in SEApp the creation of a process is originated from the request of execution of an Android component. Thus, the slowdown occurs between the request for the component and the execution of the method *onCreate*, which is the time interval subject to measurement. Our evaluation is limited to activities and services, as these are the components most used by developers. Our analysis showed identical behavior for broadcast receivers and content providers, the

other two components supporting the `android:process` attribute in the manifest.

Separate test cases have been identified based on the type of process that supports the component. We refer to *Local*, *Remote*, *Isolated* or *SEApp* components when we run components respectively in the current process, in another process, in another process with the `isolated_app` domain (using the *isolatedprocess* we described in Section 2.3), or in a package specific domain (declared in the app policy module). Furthermore, we cover *cold* and *warm* start scenarios. The *cold* start corresponds to the first time the application brings up the component, and the *warm* start to the subsequent times the app reuses a previously instantiated one.

	Cold start (ms)				**Warm start (ms)**			
Component	***Stock OS***		***SEApp***		***Stock OS***		***SEApp***	
	μ	σ	μ	σ	μ	σ	μ	σ
LocalActivity	39.102	1.094	38.689	0.980	21.052	6.046	18.685	5.001
RemoteActivity	123.468	3.176	124.649	3.526	15.722	2.682	15.933	3.256
SEApp Activity	-	-	127.356	3.542	-	-	15.188	2.394
LocalService	19.164	1.444	18.835	1.392	1.399	0.208	1.328	0.208
RemoteService	105.467	2.800	106.935	2.565	2.617	0.879	2.676	0.593
IsolatedService	103.923	2.425	104.260	3.727	-	-	-	-
SEApp Service	-	-	106.925	3.774	-	-	2.528	0.675

Table 2.4: Cold and warm start performance for activities and services

The results shown in Table 2.4 demonstrate that the performance of a stock version of the OS and SEApp are equivalent. Also, we observe that apps willing to benefit of the intra-app isolation feature get from the use of SEApp the same performance they would get from the use of remote components. Our approach also proves to outperform the *IsolatedService*, as the *isolatedprocess* option forces the creation of a new process every time an *IsolatedService* that was previously *unBind*-ed is activated. This introduces a slowdown of $102 \pm 1ms$ compared to the *SEAppService* warm start, which instead benefits from the system caching mechanism.

Files

Alongside the usual creation method, SEApp introduces in Android the possibility of creating files with a security domain defined by the app dedicated

`file_contexts`. Table 2.5 shows the time required to create a file, for each of the methods discussed. We observe no overhead on direct file creation, but the overall execution time becomes larger due to the invocation, as described in Section 2.6, of the *restorecon* service, which requires approximately $374 \pm 30\mu s$. This overhead only occurs at file creation and every subsequent operation on the file does not exhibit any performance degradation.

File creation		
Test	μ **(µs)**	σ **(µs)**
Stock OS	57.077	5.174
SEApp	60.696	6.782
SEApp + *restorecon*	431.472	109.494

Table 2.5: File creation performance

2.8 Related work

In traditional desktop operating systems significant effort has been spent in retrofitting legacy code for authorization policy enforcement leveraging MAC. An approach is to place reference monitor calls to mediate sensitive access locations through the use of static and dynamic analysis [97, 140]. An evolution of this solution is the multi-layer reference monitor [110], in which the MAC policy is enforced at different levels (e.g., application, OS, Virtual Machine Manager). Another approach is to identify integrity-violating permissions through the use of information-flow analysis [167].

Android's open source nature and popularity made it the target of careful security investigations (e.g., [5, 89, 90, 94]) and several proposals aiming at strengthen its security properties. In the following we discuss the ones that try to address app isolation and modularity, underlining the key differences with our methodology.

Our approach presents similarities with *Secure Application INTeraction (Saint)* proposed by Ongtang et al. in [142], in which the authors also try to address the issue of allowing developers to define policies that can be verified at both installation time and runtime, to better specify the permissions for each component of their app. However, since the paper has been published in 2010, *Saint* could not leverage SEAndroid [168], which was introduced later, thus the authors had to define their

own Android security middleware, which would not fit into the current Android architecture [137].

FlaskDroid [56] defines a versatile middleware and kernel layer policy language. It is based on Userspace Object Managers (USOMs), which control access to services, intents and data stored in Content Providers. However, FlaskDroid does not focus on intra-app compartmentalization, a central aspect in our proposal.

ASM [109] and *ASF* [47] promote the need for a programmable interface that could serve as a flexible ecosystem for different security solutions. The generality of these solutions, however, requires to introduce several changes to the current Android security model.

AppPolicyModules [44] is another proposal that allows app developers to create dedicated policy modules. The authors focus more on how apps could use SEAndroid to better protect their resources from the system and from other apps, paying limited attention to internal compartmentalization.

DroidCap [76] is a recent contribution proposed by Dawoud and Bugiel, in which the authors propose to replace Android's UID-based ambient authority (DAC) with per-process Binder object capabilities. The proposal is interesting as it permits to achieve security compartmentalization between different app components. To introduce capability-based access control on files, *DroidCap* had to integrate *Capsicum for Linux* [102] in Android. Overall, DroidCap is a nicely engineered solution, which shares similar objectives with ours, and the two could work in parallel as they do not interfere with each other. However, as our proposal relies on SELinux and SEAndroid, which are already part of the Android security framework, our architecture appears to be more aligned with the natural evolution of the Android ecosystem.

Boxify [48] is a virtualization environment for Android apps, which could be used to achieve a higher level of privacy and better control over app permissions. The authors also describe how their solution could be used to compartmentalize Ads libraries to reduce the risk of sensible information leakage. Yet, since the virtualization environment acts as a mediator between the applications and the system, it extends the set of trusted components the app has to rely on.

AFrame [187] and *CompARTist* [113] propose to compartmentalize third-party libs from their host app using a separate process with a dedicated UID. In *AFrame* the Android Manifest is modified with the introduction of library ad-hoc permis-

sions, while *CompARTist* uses compile time app rewriting. Both proposals do not extend the protection at the MAC level.

To summarize, the main differences that characterize our proposal are: (i) we propose a natural extension of the role of SELinux to apps leveraging what is already used to protect the system itself, thus minimizing the impact on it, and (ii) we empower the developers while limiting the amount of changes an application must undergo in order to take advantage of our solution.

2.9 Conclusions

This chapter proposed an extension to the current MAC solution (SELinux) already available in Android. Developers can use SELinux to define domains that are internal to their apps, in such a way that it is possible to leverage the modules that are already providing protection to the system. By mapping SELinux domains to activities and services, developers can limit the impact that a vulnerability has on the app processes and files. We described in the chapter the changes that we introduced into Android, and our experimental evaluation shows that the overhead introduced by our proposal is compatible with the additional security guarantees.

Availability

The implementation source and artifacts produced for the evaluation of our proposals are freely available at this URL: `https://github.com/matthewrossi/seapp`

The work in this chapter was supported in part by the EC within the H2020 Program under projects MOSAICrOWN, and by the 2015 Google Faculty Research Award Program.

Chapter 3

Data sanitization

SDS: scalable distributed data sanitization

Data sanitization (or anonymization) follows to the collection. Sanitization irreversibly alters the data so that a subject (referenced within it) cannot be identified, given a certain security parameter, while data remain practically useful. While sanitization is known to be a computationally demanding process, we aim at providing a solution for enabling a distributed anonymization over large collections of data.

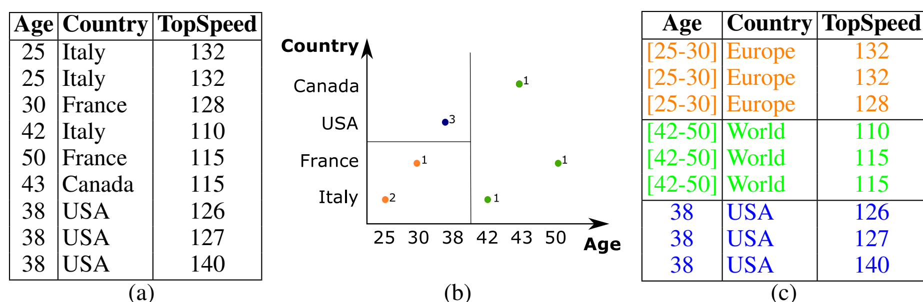
There are many techniques to sanitize a dataset. Examples are k-anonymity [65], ℓ-diversity [130], and Differential Privacy [68]. In this chapter, we focus on the ones that apply sanitization through *generalization* and *suppression*. Before replacing the datum with a less precise but semantically consistent version, these approaches cluster the data records into partitions based on their values. Intuitively, grouping similar values into the same partition permits to reduce the information loss induced by the generalization process. Our idea is that the same intuition can be used to identify an initial number of partitions, each subsequently sanitized by a dedicated network node. As this approach avoids data shuffling between the nodes, we expect a more predictable reduction in the total sanitization time when the number of nodes increases, with a limited degradation of the dataset produced in terms of quality (i.e., information loss).

We implement our solution as a multi-container application deployed with Docker. The experimental evaluation confirms that the approach is scalable with limited impact on information loss, even when the initial set of partitions is determined based on a sample of the dataset. The open source solution implementing this approach was awarded as the *Best Artifact* at *IEEE PerCom* 2021.

3.1 Introduction

Almost every object we use in our everyday life already is or is going to be smart, and equipped with sensors that constantly collect information about ourselves and the environment where we live (e.g., smart cars monitor the position of the car, engine configuration, tire pressure, etc.) [42]. Such data are valuable and may need to be shared with others (e.g., to design better solutions for autonomous driving) without, however, violating the privacy of the individuals to whom they refer.

Guaranteeing privacy in datasets containing possible identifying and sensitive information requires not only refraining from publishing explicit identities, but also obfuscating data that can leak (disclose or reduce uncertainty of) such identities as well as their association with sensitive information. k-anonymity [65, 163], extended with ℓ-diversity [130], offers such protection. k-anonymity requires generalizing values of the *quasi-identifier* attributes (i.e., attributes that leak information on respondent's identities exploiting linkage with external sources) to ensure each quasi-identifier combination of values to appear at least k times. ℓ-diversity considers each sensitive attribute in such operation so to ensure each combination of quasi-identifier values to be associated with at least ℓ different values of the sensitive attribute (see Figure 3.1(c)).



Age	Country	TopSpeed
25	Italy	132
25	Italy	132
30	France	128
42	Italy	110
50	France	115
43	Canada	115
38	USA	126
38	USA	127
38	USA	140

(a)

(b)

Age	Country	TopSpeed
[25-30]	Europe	132
[25-30]	Europe	132
[25-30]	Europe	128
[42-50]	World	110
[42-50]	World	115
[42-50]	World	115
38	USA	126
38	USA	127
38	USA	140

(c)

Figure 3.1: An example of a dataset (a), its spatial representation and partitioning (b), and a 3-anonymous and 2-diverse version (c), considering quasi-identifier `Age` and `Country` and sensitive attribute `TopSpeed`

While simple to express, k-anonymity and ℓ-diversity are far from simple to enforce, given the need to balance privacy (in terms of the desired k and ℓ) and utility (in terms of information loss due to generalization). Also, the computation of an optimal solution requires evaluating (based on the dataset content) which

quasi-identifying attributes generalize and how, and hence demands complete visibility of the whole dataset for operating the generalization steps. Hence, existing solutions implicitly assume to operate in a centralized environment. Such an assumption clearly does not fit pervasive systems where the amount of data collected is huge (there are widely circulating estimates that a smart car will upload to the cloud 25GB per hour). While scalable distributed architectures can help in performing computation on such large datasets, their use in computing an optimal k-anonymous solution requires careful design. In fact, a simple distribution of the load among workers would affect either the quality of the solution or the scalability of the computation (requiring expensive synchronization and data exchange among workers [41]).

In this chapter, we detail our scalable, efficient, and effective approach for the distributed enforcement of k-anonymity and ℓ-diversity requirements on large datasets. The solution is based on an adaptation of Mondrian [121], revised to operate without requiring knowledge of the complete dataset. Mondrian is a multidimensional algorithm that has established itself as an efficient and effective approach for achieving k-anonymity. Mondrian leverages a spatial representation of the data, mapping each quasi-identifier attribute to a dimension and each combination of values of the quasi-identifier attributes as a point in such a space. Mondrian then recursively cuts the tuples in each partition (the whole dataset at the first step) based on their values (lower/higher than the median) for a quasi-identifying attribute chosen at each cut. The algorithm terminates when any further cut would generate sub-partitions with less than k tuples, at which point values of the quasi-identifier attributes in a partition are substituted with their generalization. Figure 3.1(b) shows the spatial representation and partitioning of the dataset in Figure 3.1(a), where the number associated with each data point is the number of tuples with such values for the quasi-identifier in the dataset.

We have extended Mondrian designing a solution for partitioning data for distribution to workers without requiring knowledge of the whole dataset. We have implemented such an approach providing parallel execution on a dynamically chosen number of workers. The design of our partitioning approach aims at limiting the need for workers to exchange data, by splitting the dataset into as many partitions as the number of workers, which can independently run a revised version of Mondrian

on their portion of the data. The experimental evaluation shows that our solution provides scalability, while not affecting the quality of the computed solution.

3.2 Distributed anonymization

This section illustrates the architecture and working of our open source system (available at this link), supporting the distributed anonymization of large datasets.

Architecture

Figure 3.2 illustrates the architecture of our system, which includes two clusters: an *Hadoop Distributed File System* (HDFS) *cluster*, a well known and widely used solution for data storage and management, and a *Spark cluster* for data processing. Data are split in smaller blocks stored at *datanodes*. A *namenode* in the HDFS cluster manages the data stored at the datanodes and the access requests to them. For data processing, we have opted for *Spark* because it is a widely used engine for big data analytics that is fully compatible with the HDFS cluster. Among the nodes in the Spark cluster, one acts as *Spark Cluster Manager* and coordinates the work of the other nodes in the cluster, acting as *workers*.

The distributed SPARK anonymization application has been developed in Python to leverage the Pandas framework, which can be conveniently used for managing large data collections. The application is associated with a *Spark Driver*. The Spark Driver, which runs on the Spark Cluster Manager, is responsible for converting the application into a set of jobs that are then divided into smaller execution units, called tasks. The tasks are allocated to workers by the Spark Cluster Manager.

Distributed anonymization algorithm

The application operates in three steps (Figure 3.2): *pre-processing*, which partitions the dataset and distributes tasks to workers; *anonymization*, which anonymizes the dataset; *wrap-up*, which computes the information loss and collects other information related to the anonymization process.

Pre-processing. The first problem addressed consists in deciding how the dataset can be partitioned by the Spark Driver among the n available workers, in such a way that each worker can independently apply the anonymization algorithm on the

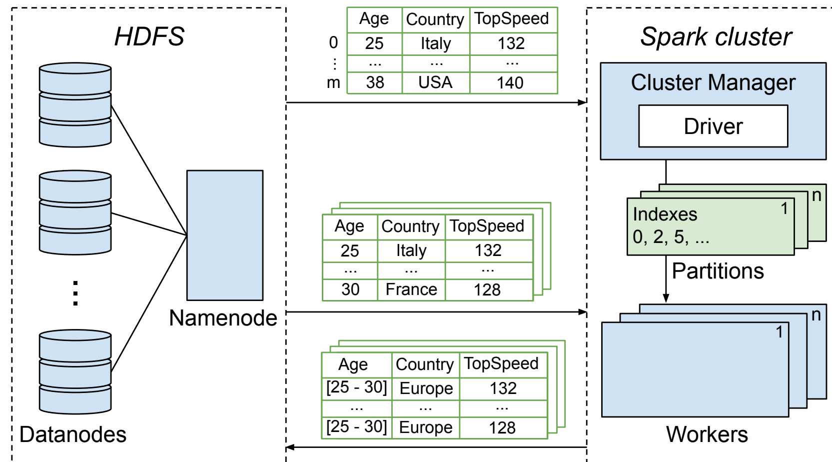


Figure 3.2: Architecture and working of the distributed anonymization system

portion of data assigned to it, without incurring in too much information loss. We first observe that, while a random partitioning of the dataset would work, it may increase the information loss. We therefore apply a strategy similar to the strategy used by the original Mondrian for creating sub-partitions: we first select an attribute of the quasi-identifier on which to partition the dataset and then create n partitions (one for each worker) depending on the values of the selected attribute. The attribute can be selected by applying different metrics (the tool supports maximum entropy, minimum entropy, and maximum span) that, however, require to have the dataset in main memory to determine the distribution of the quasi-identifying attributes' values. To overcome this problem, we operate on a *sample* of the dataset (whose size is a configuration parameter of the tool) that fits into the main memory of the Spark Driver. Based on the randomly extracted sample, the Spark Driver determines the most suitable attribute, and partitions the tuples in the dataset according to the n-quantiles. We note that, as confirmed by the experimental results (Section 3.3), operating on a sample of tuples for performing the first partitioning of the dataset does not affect the quality of the solution.

Anonymization. The Spark Cluster Manager assigns the task of anonymizing each partition determined in the pre-processing step to a worker, depending on different factors (e.g., the workload, the datanode where data are stored). To make the system scalable, our implementation forces each partition to be assigned to a different worker. Each worker then downloads from the HDFS datanodes its portion of the

dataset, and runs a revised version of Mondrian, without the need of interacting with the other workers. Our revised version of Mondrian differs from the original one in two aspects: *1)* the attribute selected for partitioning is determined by applying the same metric used in the pre-processing step; *2)* the partitioning is performed considering both the k-anonymity and ℓ-diversity requirements. When the partitions cannot be further divided without violating k-anonymity nor ℓ-diversity, the tuples in each partitions are generalized. Our tool implements different generalization strategies, suited for different kinds of data (e.g., ranges for numeric attributes, user-defined generalization hierarchies for categorical attributes). Before storing the anonymized portion of dataset back at the datanodes, each worker computes the information loss on its portion of the dataset and sends the result to the Spark Driver (see next step).

Wrap-up. To assess the quality of the anonymized dataset, the Spark Driver computes the information loss produced by our distributed anonymization algorithm. To this end, the Spark Driver combines the values of the information loss received from the workers. Such a combination is done depending on the information loss metric adopted. The tool supports two of the most common metrics, that are, the Discernibility Penalty (DP) and the Global Certainty Penalty (GCP) [188].

3.3 Experimental results

To assess the scalability of the approach and its limited impact on information loss, we have tested it over the IPUMS USA dataset [162], which has become a de-facto benchmark for anonymization solutions. The dataset includes 500,000 tuples. We assume the quasi-identifier to include attributes `State FIP Code`, `Age`, `Education Number`, `Occupation`, and the sensitive attribute to be `Income`. We have simulated a distributed environment using a single server through Docker containers. Each node in the architecture in Figure 3.2 runs in a different Docker container. The server is a 12 cores (24 threads) AMD Ryzen 3900X CPU, with 64 GB RAM and 2 TB SSD, running Ubuntu 20.04 LTS, Apache Spark 3.0.1, Hadoop 3.2.1, and Pandas 1.1.3. The distributed algorithm operates over workers equipped with 2GB of RAM and 1 CPU thread each. The centralized algorithm is single core, with no limitation on the use of the RAM.

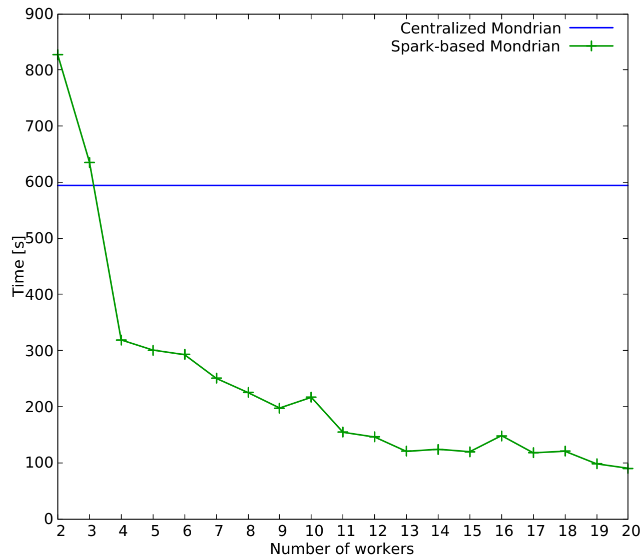


Figure 3.3: Execution time of the centralized version and distributed version varying the number of workers

The experiments aim at comparing *1)* the execution time and *2)* the information loss of our distributed approach with those of the centralized version of Mondrian.
Execution time. Figure 3.3 illustrates the execution time (in seconds) for computing a 3-anonymous and 2-diverse version of the IPUMS USA dataset. The figure shows the execution time of our distributed (Spark-based) Mondrian varying the number of workers between 2 and 20. The execution time of the distributed Mondrian decreases, as expected, when the number of workers grows with a saving with respect to the execution time of the centralized Mondrian that ranges from 46% to 85% when using more than 3 workers. This confirms the scalability of our distributed approach. It is interesting to note that the centralized Mondrian is more efficient than the distributed one when the number of workers is low (2 or 3 in our experiments). This is due to the constant initialization time paid by the distributed implementation for setting distribution and interoperation among workers, and by the different libraries used by the centralized implementation (NumPy) and by the distributed implementation (Spark APIs).
Information loss. We first observe that the information loss caused by distribution can be impacted by: *1)* the number of workers (and hence of partitions), and *2)* the size of the sample used to partition the dataset. Table 3.1 illustrates the average information loss (and its variance) obtained in 5 runs of the centralized and dis-

	100% (centralized)	0.01% sampling		
		5 workers	10 workers	20 workers
DP	$1.24e7$	$1.23e7$ ($\pm 4e5$)	$1.26e7$ ($\pm 4e5$)	$1.33e7$ ($\pm 1e5$)
GCP	6.44	6.47 (±0.08)	6.49 (±0.07)	6.46 (±0.10)

Table 3.1: DP and GCP information loss with 100% and 0.01% sampling

tributed (with 5, 10, and 20 workers) Mondrian for computing a 5-anonymous and 2-diverse version of the IPUMS USA dataset, assuming 0.01% and 100% sampling. In the table, 100% sampling corresponds to the centralized Mondrian, hence in our experiments the information loss is substantially not affected by distribution.

The results we obtained confirm that, as expected, information loss grows with the number of workers (i.e., values in DP and GCP lines in Figure 3.1 grow when moving from left to right), but the impact is negligible. Also, the results show that sampling has a very limited impact on information loss (i.e., values obtained with 0.01% sampling are slightly higher than the values obtained with 100% sampling). For instance, GCP increases of less than 2% when passing from the centralized version with 100% sampling to the distributed version with 20 workers and 0.01% sampling. DP has a similar trend.

We can then conclude that parallelization provides high scalability at a limited cost in terms of information loss.

3.4 Conclusions

This chapter proposed a distributed version of Mondrian that provides scalability without affecting information loss and leveraging an arbitrary number of independent workers.

Appendix B details the working of our distributed Spark anonymization application. Parameter settings, work distribution, and anonymization results are conveniently controllable via a web interface. The interface enables setting the parameters for privacy (i.e., k and ℓ) and distribution (i.e., number of workers) and provides a visual representation of the system working, as well as of the privacy and utility guarantees for different information loss metrics.

Availability

The implementation source of our proposals is freely available at this URL: `https://github.com/mosaicrown/mondrian`

The work in this chapter was supported in part by the EC within the H2020 Program under projects MOSAICrOWN and MARSAL, by the Italian Ministry of Research within the PRIN program under project HOPE, and by JPMorgan Chase & Co under project “k-anonymity for AR/VR and IoT/5G”.

Chapter 4

Data storage & processing

SecIdx: k-flat encrypted indexes for encrypted databases

In this chapter, we bring into focus the storage and processing stage. The goal is to enable provider-side evaluation of queries with guarantees that an *honest-but-curious* storage provider cannot access to the data content.

In this scenario, we concentrate on relational data and provide an approach that supports the evaluation of point and range conditions on multiple attributes. The approach we propose involves packing the tuples into blocks, which are subsequently stored to the cloud provider. To guarantee that the cloud provider cannot access to the data inside the blocks, a probabilistic encryption layer is applied. Blocks are indistinguishable for the cloud provider, and access to them is performed via a set of indexes. Protection against inferences from indexes is guaranteed by clustering tuples in blocks that are then mapped to the same index values, so to ensure collisions for individual attributes as well as their combinations. To produce such a clustering, while at the same time ensuring efficient query execution, we provide a spatial-based tuple partitioning algorithm. Query translation and processing require the client to store a compact map. The map is used to search the indexes efficiently.

To validate our approach, we conduct a series of experiments evaluating the query response time and client-storage requirements. The experiments confirm the effectiveness associated with our approach. The degradation of query response time, compared to a configuration where the data are stored in plaintext on a remote database, is very low, especially when the network link bridging the client and server is affected by at least 10 ms latency, and has a transmission rate of 100 Mbps.

4.1 Introduction

A lot of research has been dedicated in the past twenty years to the investigation of how cloud service providers can support the management of data with a guarantee that the cloud provider does not have access to the data content. The model that is often used in this scenario is the *honest-but-curious* [164] server, which assumes that the cloud provider correctly executes all the data access and manipulation operations issued by the customer, but may abuse the fact that it observes the data to have access to the customer information.

The classical solution to this threat is represented by the use of encryption, so that the control of the physical representation of the data does not give access to the information content, as long as the cloud provider does not have access to the encryption key. Unfortunately, a simple use of encryption may not offer real protection against the curious cloud provider. When direct deterministic encryption is applied, the encrypted data maintain the same distribution of the original data, and this allows an adversary to reconstruct the original values by comparing the distribution to that of known public data (e.g., first names, last names, zip codes).

The research and industry communities have dedicated significant effort to solve this problem, considering different lines of investigations. Possible approaches include: *i)* the use of searchable encryption (e.g., [51, 52, 147]), supporting the evaluation of conditions on encrypted data; *ii)* the use of trusted hardware components (e.g., Intel SGX [114]) and Oblivious Random Access Memory (ORAM) [75, 171] protocols at the server, offering a trusted execution environment residing at, but not accessible by, the server; *iii)* the use of Fully Homomorphic Encryption (FHE) schemes [96, 99, 139]. All these approaches represent valid alternatives, but in general, they suffer from a significant overhead (e.g., [40]), making them still not applicable in many practical scenarios.

The approach described in this chapter specifically targets relational data. To support the evalution of conditions, we propose the association with the encrypted data of metadata working as indexes. However, indexes may suffer from a possible exposure to inferences, as they might leak information on the values behind them. The vulnerability of indexes typically resides in the frequencies of their occurrences, which can bear relationship with the plaintext values. Frequencies of both individual attributes values as well as combinations of them can be exposed

to inference. We address these problems proposing a multidimensional index that guarantees perfect indistinguishability to an adversary with access to the data. This means that all the values of the attributes in the index (and their combinations) will appear in the index with the same number of occurences. To offer protection against an attacker monitoring the access to the data content, we cluster tuples into equally large unintelligible blocks (or groups) so that tuples referenced by the same index value are indistinguishable from one another (so are queries over them).

The consequence for such a protection is that tuples can only be retrieved by the client with the granularity of blocks. It is therefore important to carefully group tuples for indexing so to limit the degradation in query response time. While this can be trivial when only one attribute is to be indexed, it is far from being so (it is an NP-hard problem) when multiple attributes need to be indexed. The approach for index construction employs a spatial-based representation of tuples to be outsourced and an algorithm performing recursive cuts on such space, resulting in a partitioning of tuples for indexing. As confirmed by the experimental results detailed in Section 4.7, the proposal provides for effective and efficient query evaluation, enjoying limited overhead and limited storage requirements at the client side.

4.2 Basic concepts and problem statement

The work proposed in this chapter is framed in the context of relational database systems. We illustrate our approach with reference to the outsourcing of a relation r over schema $R(a_1, \ldots, a_m)$, with each attribute a_j defined over a domain $\mathtt{d}(a_j)$, $j = 1, \ldots, m$. Since our assumption is a honest-but-curious server, clearly tuples need to be encrypted for storage, the idea is to specify indexes over attributes and evaluate conditions on queries over such indexes. The problem is the definition of privacy-preserving indexes for enabling execution of queries involving evaluation of conditions over attributes, considering equality (i.e., $=$) conditions as well as range (i.e., $>, \geq, <, \leq$) conditions. As a running example, we consider the problem of outsourcing the relation in Figure 4.1(a), where queries may need to evaluate condition over attributes `State` (with domain the two-letter codes for states within the USA), and `Age`. A query we want to support is, for instance, "SELECT `Name`, `Age` FROM r WHERE `State`= Ca $\wedge$ `Age`>38", aimed at retrieving name and age of people older than 38 living in California.

	Name	Age	State
t_1	Abe	34	Ne
t_2	Bud	34	Tx
t_3	Coy	40	Ne
t_4	Doc	34	Wy
t_5	Edd	37	Ca
t_6	Fox	40	Ak
t_7	Gus	43	Ca
t_8	Hae	46	Ca
t_9	Isa	49	Oh
t_{10}	Jim	55	Wy
t_{11}	Ken	46	Mi
t_{12}	Luc	52	Tx

(a)

$i_{\texttt{Age}}$	$i_{\texttt{State}}$	enctuple
ϵ	α	t_1
ϵ	α	t_2
ϵ	α	t_3
ζ	β	t_4
ζ	β	t_5
ζ	β	t_6
η	γ	t_7
η	γ	t_8
η	γ	t_9
θ	δ	t_{10}
θ	δ	t_{11}
θ	δ	t_{12}

(b)

Figure 4.1: Plaintext relation (a), and its encrypted and indexed version (b)

Indexing the relation for external outsourcing while protecting confidentiality of its content means providing some coding for the attributes on which conditions need to be supported, so to enable evaluation of conditions while not exposing actual values to the storing server. Indexing must however be done carefully to ensure it does not leak information. For instance, while the coding protects actual value, a one-to-one correspondence between plaintext values and indexes clearly makes indexes exposed to frequency-based attacks (e.g.,, in our example, California is the only state with three occurrences and a one-to-one indexing would maintain such a property). Also, an order-preserving index to support range queries would maintain the order of values in the indexing, hence again leaking information which can enable reconstructing the values behind the indexes. Hence, indexes should not leak, in their values, any order.

Frequency-based attacks can be counteracted by destroying the frequency-based correlation between values and indexes. The extreme case for this is a 1-to-many correspondence (hence mapping different occurrences of the same value to multiple indexes) with no index values appearing more than once (e.g., the three occurrences of California would be mapped to three different indexes). While destroying frequencies in the index, such an approach would clearly introduce a significant overhead in query execution. An alternative approach to confuse frequencies is through collision, mapping different plaintext values to the same index, but the presence of high-occurring values would however still be exposed. A solution to the problem

above is to provide indexing while ensuring a complete flat occurrence of index values through both multiple index values for the same plaintext as well as collision, which is through a many-to-many correspondence between plaintext values and indexes. To provide effective protection, not only individual attributes, but also any combination of them, should enjoy a flat profile. Besides providing protection against static inference attacks, that can no longer exploit frequencies of index values or combination thereof, such a flattening provides protection also against dynamic observations from the storage provider, since tuples with the same index are indistinguishable from one another. The level of protection to dynamic observations is directly proportional to the number of tuples referenced by the same combination of index values. Figure 4.1(b) shows an example of such an indexing for the relation in Figure 4.1(a), where index values are represented with Greek letters, and the encrypted tuples are represented with a gray background. Indexes and combinations thereof enjoy a completely flat occurrences and therefore no static inference can be made of the values behind them. The challenge is producing a flattened index over multiple attributes.

4.3 Partitioning

Our approach for the construction of index values is based on a partitioning of the tuples in the original relation into groups with fixed cardinality. All tuples in the same group are then associated with the same combination of index values. The number of tuples that must be included in each group, denoted k, is a parameter that can be arbitrarily set. A larger k provides more protection against dynamic observations, but also increases the potential query execution overhead (see Section 4.7).

The first step of our approach is the clustering of tuples in groups of the same size, k. Since the cardinality of the relation may not necessarily be a multiple of k, we need to account for the remainders, which we accommodate by allowing clusters to include at most one tuple more than the k requested (as needed to fully cover the sets of tuples to be partitioned). Our definition of k-flat partition captures the partitioning of tuples, to produce a *maximal* flattening of groups with cardinality k as follows.

Definition 4.3.1 (k-flat partition)**.** *Let r be a relation. A k-*flat partition *of r, denoted $\mathcal{P}$, is a partition $\mathcal{P} = \{G_1, \ldots, G_p\}$ of tuples in r such that:*

1. $\forall G \in \mathcal{P}$*,* $k \leq \text{card}(G) \leq k+1$*;*

2. $p = \lfloor \text{card}(r)/k \rfloor$.

In Definition 4.3.1, the first condition expresses the requirement on the cardinality of the groups (allowing groups to have either k or $k+1$ tuples, this latter being needed to accommodate remainders), and the second condition dictates the number of groups to be the *maximum* among those that satisfy condition 1, or - equivalently - the number of groups with $k+1$ tuples to be *minimum*. By dictating the number of groups in the partition, condition 2 forces exactly $h = card(r)\ mod\ k$ of the groups to have $k+1$ tuples, while all the others will have k tuples. In other words, the condition rules out from consideration partitions which do not enjoy maximum flattening, or rather that have a number of groups of cardinality $k+1$ larger than the number of remainders to be accommodated. For instance, assume $\text{card}(r)$=231 and k=10. Condition 2 would accept only a partition composed of 23 groups (one of which composed of 11 tuples, all others being of 10 tuples) ruling out of consideration partitions composed of 22 groups (eleven of which composed of 11 tuples) or 21 groups (all with 11 tuples), which - although satisfying condition 1 - do not maximize the required flattening of $k = 10$.

Clearly, for a relation r to have a k-flat partition, the number of remainders to be accommodated (i.e., the extra tuples to allocate to groups) must be not greater than the number of groups composing the partition. For instance, trivially, no k-flat partition for $k = 10$ can exist for a relation with 23 tuples. In other words, with $h = card(r)\ mod\ k$ and $p = \lfloor \text{card}(r)/k \rfloor$, it must be that $h \leq p$, which is also a sufficient condition for a k-flat partitioning to exist, as stated by the following theorem.

Theorem 1 (Existence of a k-flat partition)**.** *Let* r *be a relation such that* $\text{card}(r) \geq k$*. A* k*-flat partition* $\mathcal{P}$ *of* r *exists iff* $h \leq p$*, with* h*=(*$\text{card}(r)$ *mod* k*) and* p*=*$\lfloor \text{card}(r)/k \rfloor$.

Proof. See Appendix C.2 □

Given a relation r and an integer k, we say that r is *valid* wrt the k-flat partitioning problem if a k-flat solution exists for r. This is captured by the following definition.

Definition 4.3.2 (Validity)**.** *Relation* r *is said to be* k-valid *for* k*-flat partition iff* $h \leq p$*, with* h*=(*$\text{card}(r)$ *mod* k*) and* p*=*$\lfloor \text{card}(r)/k \rfloor$.

While the observation in Theorem 1 may seem a non issue for the problem to be solved, where the cardinality of r is extremely large and k very small, it is an important aspect to take into account in the partitioning process, which, if not done properly, may easily degenerate. Our approach to compute a k-flat partition is via a process recursively cutting a relation in two groups at each step, until a k-flat partition is reached. To ensure that our recursive process terminates with the computation of a k-flat partition, we force the cut at each step to produce only k-valid relations and to not increase the number of groups with cardinality higher than k. We then introduce the notion of cut validity as follows.

Definition 4.3.3 (Cut validity)**.** *Let r be a relation and k be an integer. A cut (r_l,r_r), partitioning r in two groups, is* valid *iff both r_l and r_r are valid (Definition 4.3.2) and $h = h_l + h_r$, with h=($\mathrm{card}(r)$ mod k), h_l=($\mathrm{card}(r_l)$ mod k), and h_r=($\mathrm{card}(r_r)$ mod k)*

Intuitively, a cut is valid if the two relations resulting from it are k-valid, that is, a k-flat partition exists for them, and the total number of groups of cardinality $k+1$ is not increased by the cut. For instance, consider a relation composed of 233 tuples. A cut partitioning it in two relations of 23 and 200 tuples, respectively, is not valid due to the non validity of the first relation (which cannot have a 10-flat solution). Also, a cut partitioning it in two relations of 117 and 116 tuples, respectively, is not valid since, their k-flat solutions, having respectively seven and six groups of 11 tuples, cannot represent a k-flat solution for the original relation.

Since our problem is to group tuples for index construction, it is important not only to partition tuples as a k-flat partition to ensure flat indexing, but also to have indexes that perform well with respect to query execution. In general, a partitioning maintaining same or close values within the same group as much as possible behaves better, meaning it introduces less performance overhead in query execution, than an approach scattering such values in different groups. However, as already noticed, with multiple attributes involved, the problem is far from being trivial.

In the next sections we first describe how we take into consideration the values within tuples so to provide a partitioning performing well for query execution, and then we describe its tweaking to enforce partitioning to ensure k-flatness.

Recursive partitioning

Our approach to partitioning is based - at a very high level - on an algorithm similar to the one used by the Mondrian anonymization algorithm [122], while bearing some important differences. The similarity is the representation of the dataset in a *multidimensional space* and the enforcement of partitioning through recursive cuts. The differences, to accommodate the fact that we need to cluster tuples to produce flat indexing performing well for query evaluation (in contrast to cluster tuples for semantically meaningful generalizations), are mainly in the way cuts are determined, and the forcing on groups to provide flattening as per Definition 4.3.1. Our partitioning process works then in a multidimensional space, with one dimension for each attribute to be indexed, and where each tuple is the point in such a space where its coordinate values meet. The space appearing at the top in Figure 4.2 is the multidimensional representation of attributes `State` and `Age` for the tuples in Figure 4.1(a). We distinguish attributes to be indexed in two categories:

- *continuous* attributes (e.g., `Age` in Figure 4.1(a)), characterized by a total order relationship on their domain hence supporting range conditions;
- *nominal* attributes (e.g., `State` in Figure 4.1(a)), which do not have an order in their domain and thus supporting equality conditions only.

While the spatial representation conveys an order of values along a dimension, we maintain such an order fixed, and corresponding to the order dictated by the domain, only for continuous attributes, so that partitioning will cluster together same or close values. By contrast, we adjust the order in our spatial representation of nominal attributes as best suited for the process, as we elaborate next.

The partitioning process works cutting at each step the tuples along one dimension (attribute) in the space and recursively calling itself on each of the two produced subspaces. At each iteration, the dimension along which a cut is to be performed is chosen to be an attribute that enjoys the highest support (i.e., number of distinct values). If the attribute is a continuous attribute, the cut divides the tuples in two groups depending on their value wrt the median: values lower than or equal to the median in one group and values higher than the median in the other group. Should the median correspond to the maximum value for the attribute in the relation, the values equal to the median will be put in the second group (which would otherwise

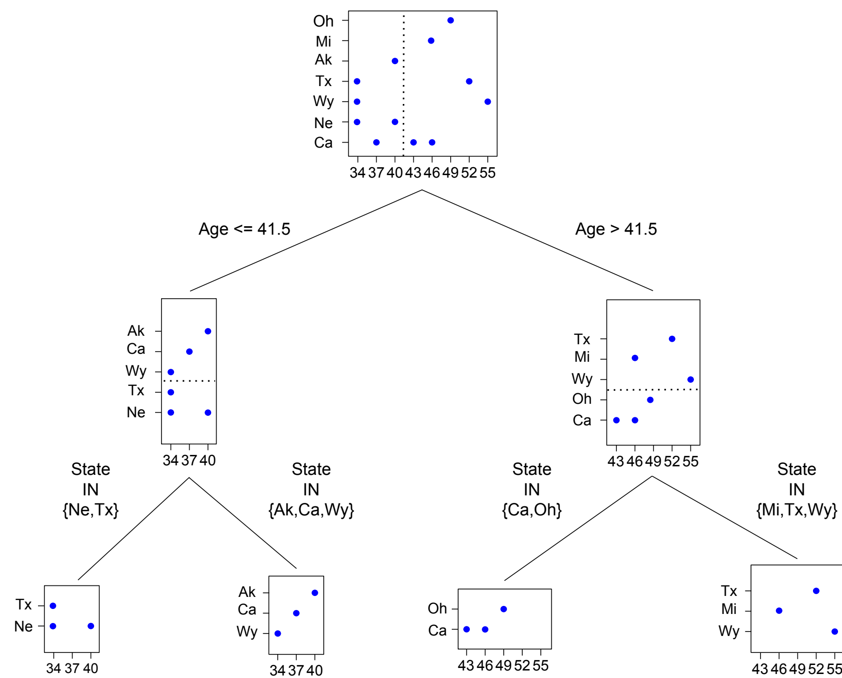


Figure 4.2: Graphical representation of the cuts performed by procedure Cut over the relation in Figure 4.1(a)

be empty) instead than the first one. If the attribute is a nominal attribute, the cut divides the tuples in two groups with a bin packing strategy, considering values of the attribute in decreasing order of their occurrences and placing tuples that have the value under consideration in the group that is smaller. Figure 4.2 illustrates the working of the partitioning process for our running example. The first cut operates on attribute `Age` (whose support is 8), splitting tuples in two groups, the left group has the tuples with age lower than or equal to the median (which is 41.5) and the right group has the tuples with age higher than the median. On each of the two spaces, the subsequent cut operates on attribute `State`, diving tuples in two groups with a bin packing strategy, considering state values in decreasing order of occurrences and placing tuples with such values in the group that is smaller. In the figure, the order of values in the state dimension has been rearranged (starting from the origin, they always appear in decreasing order of occurrences), to better graphically represent the cut. The resulting groups, reported at the bottom of Figure 4.2, have all cardinality of 3, and hence no further cut needs to be performed (as k is equal to 3 in the running example).

INPUT: (r, k) /* relation r to partition; global variable k */
OUTPUT: $\mathcal{P}$ /* k-flat partition $\mathcal{P}$ */
PARTITION(r)
1: **if** $\text{card}(r) \leq k + 1$ **then** $\mathcal{P} := \mathcal{P} \cup \{r\}$
2: **elseif** support(r) = 1 **then** /* all tuples over A are equal */
3: $p := \lfloor \text{card}(r)/k \rfloor$
4: $h := \text{card}(r)$ mod k
5: Let $\{G_1, \ldots, G_p\}$ be a partition of r in h groups of $k{+}1$ tuples
6: and $p{-}h$ groups of k tuples
7: $\mathcal{P} := \mathcal{P} \cup \{G_1\} \cup \ldots \cup \{G_p\}$
8: **else**
9: Choose $a \in A$ s.t. support($r[a]$) is maximum
10: (r_l, r_r) := **cut**(r,a)
11: **partition**(r_l)
12: **partition**(r_r)

Figure 4.3: Algorithm for computing a k-flat partition

Computing a k-flat solution

Our approach to compute a k-flat solution comprises three procedures: **Partition**, **Cut**, and **Check**. These are illustrated in Figures 4.3, 4.4 and 4.5 respectively.

Partition. Procedure **Partition** (Figure 4.3) performs the partitioning recursively calling itself and calling procedure **Cut** for performing the described cutting process, eventually determining the sets $\mathcal{P}$ composing the groups of the k-flat partition. When called, **Partition**(r) first evaluates the cardinality of r (line 1). If such a cardinality is no greater than $k + 1$ (meaning it is either k or $k + 1$), no further cut needs to be performed and r is added to $\mathcal{P}$. Else, if all the tuples in r have the same values (line 2), it simply splits the tuples in $\lfloor \text{card}(r)/k \rfloor$ groups each containing either k or $k + 1$ tuples (as per Definition 4.3.1). Otherwise (line 8) it picks an attribute a with the highest number of distinct values and calls procedure **Cut** to split the tuples in the relation along a's dimension, then recursively calling itself on the two partitions returned.

Cut. Called with a relation r and attribute a as parameters, procedure **Cut** (Figure 4.4) partitions the tuples in r based on the values of a, enforcing the process described in Section 4.3 depending on whether a is continuous (line 1) or nominal (line 10). After producing the two partitions r_l and r_r, it calls procedure **Check**,

CUT(r,a) /* cut relation r over attribute a and produces two valid relations r_l and r_r */
1: **if** a is continuous **then**
2: *med* := median($r[a]$) /* compute the median of a */
3: **if** *med* = max($r[a]$) **then**
4: r_l := $\{t \in r \mid t[a] < med\}$; r_r := $\{t \in r \mid t[a] \geq med\}$
5: **else** r_l := $\{t \in r \mid t[a] \leq med\}$; r_r := $\{t \in r \mid t[a] > med\}$
6: m := **check**(r,r_l.r_r)
7: **case** m **of** /* move $|m|$ tuples to produce two valid relations r_l and r_r */
8: > 0: Move m tuples with values for a closest to *med* from r_l to r_r
9: < 0: Move $|m|$ tuples with values for a closest to *med* from r_r to r_l
10: **else** /* a is nominal */
11: $\forall v \in r[a]$, c_v := count($r[a]$=v) /* count c_v to be priority of v */
12: Let Q be a max priority queue with the distinct values in $r[a]$
13: r_l := $\emptyset$; r_r := $\emptyset$
14: **while** NOTEMPTY(Q)
15: v := POP(Q)
16: **if** $\text{card}(r_l) < \text{card}(r_r)$ **then** r_l := $r_l \cup \{t \in r \mid t[a] = v\}$
17: **else** r_r := $r_r \cup \{t \in r \mid t[a] = v\}$
18: m := **check**(r,r_l.r_r)
19: **case** m **of** /* move $|m|$ tuples to produce two valid relations r_l and r_r */
20: > 0: Move m tuples with the minimum count from r_l to r_r
21: < 0: Move $|m|$ tuples with the minimum count from r_r to r_l
21: **return** r_l,r_r

Figure 4.4: Algorithm for cutting a k-flat partition

which checks if the cut is *valid* (if $m > 0$) or viceversa (if $m < 0$) to make the cut valid (it returns 0 if the cut is already valid). To maintain the quality of the computed cut, tuples to be moved from one partition to the other are those close to the median (if the cut was on a continuous attribute), or those with lower number of occurrences (if the cut was on a nominal attribute).

Check. Procedure **Check** (Figure 4.5) checks the validity of a computed cut and, in case the cut is not valid, returns the number of tuples to be moved from a partition to the other to make the cut valid while minimizing the number of tuples to be moved. The sign ($+$ or $-$) of the returned number indicates the direction of the movement: a positive number indicates that tuples need to be moved from r_l to r_r, while a negative number indicates that tuples need to be moved from r_r to r_l. Procedure **Check**, called with the original relation r and the partitions r_l and r_r

CHECK(r,r_l.r_r) /* r: original relation; r_l,r_r: left,right partition of r */
1: p := $\lfloor \text{card}(r)/k \rfloor$; h := $\text{card}(r)$ mod k
2: p_l := $\lfloor \text{card}(r_l)/k \rfloor$; h_l := $\text{card}(r_l)$ mod k
3: p_r := $\lfloor \text{card}(r_r)/k \rfloor$; h_r := $\text{card}(r_r)$ mod k
4: **if** ($h_l \leq p_l$) $\wedge$ ($h_r \leq p_r$) $\wedge$ (h_l+h_r=h) **then return** 0
5: **if** $p_l < 1$ **then** m_{rl} := [(k−h_l) + max(0,h−(p−1))]; **return** $-m_{rl}$
6: **if** $p_r < 1$ **then** m_{lr} := [(k−h_r) + max(0,h−(p−1))]; **return** m_{lr}
7: **if** $h_l \leq h$ **then** /* h_l+h_r=h */
8: **if** $h_l > p_l$ **then** /* r_l is not valid */
9: m_{lr} := $h_l - p_l$
10: **if** $p_r = 1$ **then return** m_{lr}
11: **else** m_{rl} := k−h_l + max(0, h−(p_r−1))
12: **else** /* r_r is not valid */
13: m_{rl} := $h_r - p_r$
14: **if** $p_l = 1$ **then return** $-m_{rl}$
15: **else** m_{lr} := k−h_r + max(0, h−(p_l−1))
16: **else** /* h_l+h_r>h */
17: m_{lr} := h_l − min(p_l,h)
18: m_{rl} := h_r − min(p_r,h)
19: **if** $m_{lr} < m_{rl}$ **then return** m_{lr} **else return** $-m_{rl}$

Figure 4.5: Algorithm for checking the validity of a cut

resulting from the cut as parameters works as follows. First, it computes the number of groups (p, p_l, p_r, resp.) and remainders (h, h_l, h_r, resp.) for each of them. (Note that, as per modulo theory, $h = (h_l + h_r)\ mod\ k$ and $h_l \leq h \Leftrightarrow h_r \leq h$.) It then considers the different cases that can occur. In their illustration, we point to their occurrences in the examples of Figure 4.6. First, it consider the particular cases, when the cut is already valid or one of the two relations does not have a sufficient number of tuples to form a group (i.e., it has less than k tuples). If the cut is valid (line 4, Figure 4.6(a)) then no tuple needs to be moved and 0 is returned. If the left relation does not have cardinality sufficient to form a group, that is, $p_l = 0$ (line 5), it returns the number of tuples to be moved to it to reach cardinality k and ensure validity of the cut. This is $k - h_l$, if the number of original remainder h to be accommodated is no higher than the number of groups ($p-1$) remaining in the right partition (e.g., Figure 4.6(a)); it is $k - h_l + 1$, otherwise (e.g., Figure 4.6(b)). Note that $k - h_l + 1$ comes from the formula in line 5 due to the fact that, since r is valid, h cannot be greater than p. If the right relation does not have cardinality sufficient

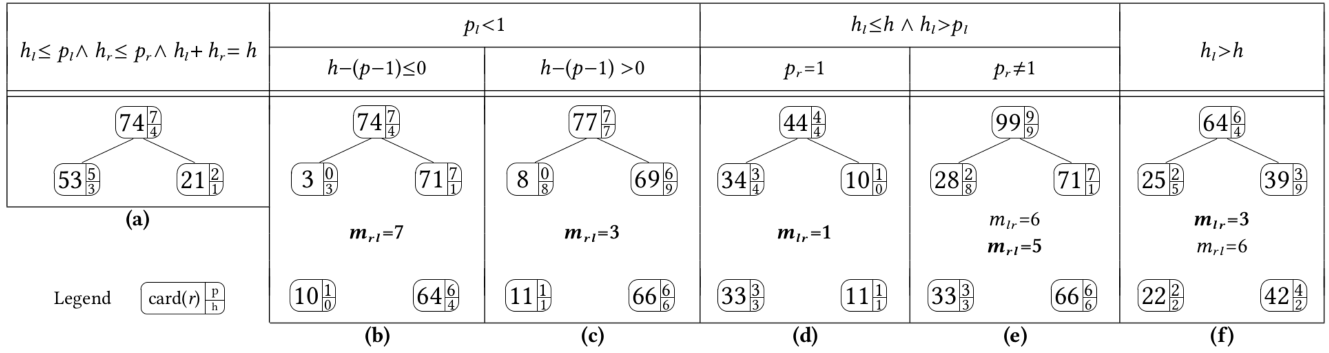
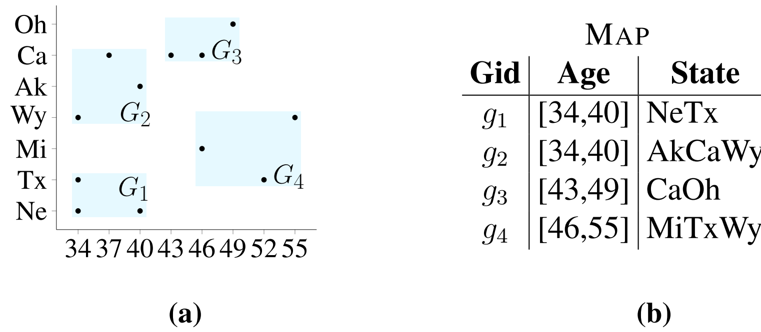


Figure 4.6: Examples of partitions for different cases of the **Check** procedure, k=10

to form a group, that is, $p_r = 0$ (line 6), the number of tuples to be moved from r_l is computed analogously. If none of the cases above occur (i.e., the cut is not valid and both partitions have at least a group), the algorithm proceeds considering each of the conditions that can make the cut not valid, and computes the minimum number of tuples to be moved from r_l to r_r (m_{lr}) or from r_r to r_l (m_{rl}) to make the cut valid. If no extra remainders have been generated, that is $h_l + h_r = h$ (line 7), then either $h_l{>}p_l$ or $h_r{>}p_r$. If $h_l{>}p_l$ (line 8), the number m_{lr} of tuples to move from r_l to r_r to make r_l valid is $h_l{-}p_l$ (line 9). If r_r has only one group, this is the only possible move, which is then returned (line 10, Figure 4.6(c)). Else (line 11, Figure 4.6(d)), the alternative move m_{rl} of tuples to move from r_r to r_l is computed as $k{-}h_l +$ max(0, $h{-}(p_r{-}1)$). Here, $k{-}h_l$ is the number of tuples for the cardinality of r_l to become multiple of k and $h{-}(p_r{-}1)$ the possible extra tuples to be assigned to it since otherwise r_r would not be valid. The case where $h_r{>}p_r$ (line 12), is treated analogously (lines 13-15). If condition in line 7 evaluated false, then (line 16) extra remainders have been generated, that is, $h_l{+}h_r{>}h$. The procedure then determines the (minimum) number of tuples to be moved from r_l to r_r or viceversa to make the cut valid. Note that, in this case, $p_l + p_r = p - 1$, and r_l and r_r might be valid or not. The number m_{lr} of tuples to be moved from r_l to r_r is h_l − min(p_l,h) (line 17, Figure 4.6)(e)). Intuitively, this is due to extra remainders generated with respect to the original h in r_l and/or remainders that cannot be accommodated in r_l (if $h_l > p_l$). The number m_{rl} of tuples to be moved from r_r to r_l is computed analogously (line 18, Figure 4.6)(f)). With the possible moves for the different cases computed, the procedures returns then the value among m_{lr} and m_{rl} that is lower, changing sign (−) if m_{rl} is to be returned to signal the direction of the movement.

4.4 Indexing

At the end of the partitioning process, each group in the k-flat partition contains (k or $k+1$) tuples, that must be mapped to the same combination of index values. For instance, Figure 4.7(a) shows, in the multidimensional spatial representation, the four groups (G_1, G_2, G_3, G_4) resulting from the partitioning process in Figure 4.3.



Gid	Age	State
g_1	[34,40]	NeTx
g_2	[34,40]	AkCaWy
g_3	[43,49]	CaOh
g_4	[46,55]	MiTxWy

Figure 4.7: Spatial representation of the running example and corresponding MAP

To define indexes to associate with encrypted tuples in groups supporting evaluation of conditions, we first introduce the *coverage* of an attribute in a group as the set of values of the attribute covered by the group.

Definition 4.4.1 (Coverage). *Let $\mathcal{P}$ be a k-flat partition of a relation r, $a \in A$ be an attribute to be indexed, and and G be a group in $\mathcal{P}$. The* coverage *of a in G, denoted $G[a]$, is defined as:*

- *$G[a]$=$[v_l,v_u]$, with v_l=min$\{t[a] \mid t{\in}G\}$ and v_u=max$\{t[a] \mid t{\in}G\}$, if a is a continuous attribute;*
- *$G[a]$=$\{t[a] \mid t{\in}G\}$, if a is a nominal attribute.*

The set of groups, together with their coverage for the different attributes to be indexed represents the MAP of the partition, which we formally define as follows.

Definition 4.4.2 (Map). *Let r be a relation, $A = \{a_1, \ldots, a_n\}$ be a set of indexed attributes, and $\mathcal{P}$=$\{G_1, \ldots, G_p\}$ be a k-flat partition of r. The* MAP *of $\mathcal{P}$ is the set of p tuples of the form $\langle G_i[gid], G_i[a_1], \ldots, G_i[a_n]\rangle$ such that $G_i[gid]$ is the identifier of partition $G_i{\in}\mathcal{P}$, and $\forall a_j \in A$, $G_i[a_j]$ is the coverage of a_j in G_i, $i = 1, \ldots, p$.*

In the following, we use notation $\textsc{Map}[a]$ as a shorthand for the multiset $\bigcup_{i=1}^{p} G_i[a]$ and $\textsc{Map}[gid]$ to denote the set of all gid of the groups in $\mathcal{P}$. For instance, Figure 4.7(b) show the $\textsc{Map}$ related to our running example. Here, $\textsc{Map}[\texttt{Age}]$ = $\langle$[34,40], [34,40], [43,49], [46,55]$\rangle$, and $\textsc{Map}[gid]$= $\langle g_1, g_2, g_3, g_4\rangle$.

Before outsourcing relation r, the tuples in r must be encrypted. Since our indexing approach associates the same combination of index values to all tuples belonging to the same group in $\mathcal{P}$, meaning that queries are supported at the level of a single group and not at the level of a single tuple, it is natural to physically store all tuples belonging to the same partition as a single encrypted block (see Section 4.6 for a description of the encryption process). Each encrypted block is then associated with a unique combination of index values, meaning that different encrypted blocks have different index values. Intuitively, this implies that for each indexed attribute $a{\in}A$ and for each group $G{\in}\mathcal{P}$, $G[a]$ will be mapped to the same index values. Since we want index values for different groups to be different, the generation of indexes must be done through a set of different indexing functions, one for each attribute on which we can define an index, such that they satisfy the following definition.

Definition 4.4.3 (Map indexing)**.** *Let* $\textsc{Map}$ *be a map over a k-flat partition $\mathcal{P}$ of relation r, and A be a set of indexed attributes. A* map indexing *over* $\textsc{Map}$ *is a set* $F{=}\{\iota_{pid}, \iota_{a_1}, \ldots, \iota_{a_n}\}$ *of functions such that:*

1. $\forall x \in \{A \cup gid\}$, ι_x:$\textsc{Map}[x] \rightarrow I_x$;
2. $\forall x, y \in \{A \cup gid\}$, $x \neq y \Rightarrow I_x \neq I_y$;
3. $\forall x \in \{A \cup gid\}$ *and* $\forall c, c' \in Map[x]$, $\iota_x(c) \neq \iota_x(c')$.

According to this definition, we have to define an index function for each index attribute a in A and for the gid attribute (condition 1). To guarantee that different groups in $\mathcal{P}$ are associated with different index values for all attributes of interest, the index functions must be defined over different co-domains (condition 2), and the same coverage of an attribute a in different groups must be associated with different index values (condition 3).

Two different strategies can be adopted for generating index values with index functions that satisfy Definition 4.4.3: *gid-based* strategy and *value-based strategy*.

With a gid-based strategy, we need only a single index function over attribute gid. In this case, each group in $\mathcal{P}$ is associated with a unique index value. The association between each group G_i and the value g_i must be stored by the client.

The value-based execution strategy is more complex, as we need to ensure that condition 3 is valid for different occurences of the same coverage of an attribute a. Even in this case, we associate to each occurence a unique index value, but we refer to it as a *token*. The peculiarity of tokens is that, contrary to the gid-based strategy, the association between each token value and coverage occurence must not be memorized by the client. In Section 4.6, we described how the client can deterministically retrieve all the tokens at runtime starting from a *seed* and the number of coverage occurences.

Figure 4.8 illustrates the encrypted and indexed relation corresponding to the relation in Figure 4.1(a).

Gid	$i_{\texttt{Age}}$	$i_{\texttt{State}}$	**encblock**
g_1	ϵ	α	$t_1t_2t_3$
g_2	ζ	β	$t_4t_5t_6$
g_3	η	γ	$t_7t_8t_9$
g_4	θ	δ	$t_{10}t_{11}t_{12}$

Figure 4.8: Physical representation of the relation in Figure 4.1(a)

4.5 Query translation

In the previous section, we have shown how an encrypted and indexed relation is created through a gid-based or a value-based indexing function. We now describe how a query formulated over the original plaintext relation can be translated into a query over the encrypted and index relation by transforming the conditions appearing in the original query into conditions on indexes. To this purpose, a client first maps the plaintext values into the corresponding index values, and then, after having computed this mapping, rewrites the original conditions into conditions that operate over the index values previously determined.

Mapping

The mapping of plaintext values into the corresponding index values depends on the indexing function used in the creation of the encrypted and indexed relation. Given a plaintext value v, called *target value*, for attribute a, the client identifies the groups containing the tuples associated with the target value for attribute a, and determines the index values associated with these groups. Intuitively, the target partitions are those for which value v is included in the coverage of a, and the corresponding index values are determined considering their coverages or gids, depending on whether the client has used a value-based or a gid-based approach, respectively. Formally, the coverages or gids are determined though the following *mapping functions*.

Definition 4.5.1 (Mapping functions)**.** *Let* MAP *be a map over a k-flat partition $\mathcal{P}$ of relation r, and $a{\in}A$ be an indexed attribute. A* mapping function *for attribute a is a function:*

- *map_a : $\mathtt{d}(a){\rightarrow}2^{\text{MAP}[a]}$ such that $\forall v \in \mathtt{d}(a)$, $map_a(v) = \langle c{\in}\text{MAP}[a] \mid v{\in}c\rangle$ (*value-based*);*
- *$id.map_a$: $\mathtt{d}(a){\rightarrow}2^{\text{MAP}[gid]}$ such that $\forall v \in \mathtt{d}(a)$, $id.map_a(v) = \{\text{id}{\in}\text{MAP}[gid] \mid v{\in}G_{id}[a]\}$ (*gid-based*);*

According to this definition, the client uses two different mapping functions. Function map_a for attribute a is used when the indexes have been generated through a value-based approach. The function maps a value v in the domain of a to the multiset of coverages that include it. For instance, consider the relation in Figure 4.1(a) and suppose that we are interested in retrieving all people of 39 years old. According to the MAP in Figure 4.9(b), $map_{\mathtt{Age}}(39) = \langle[34,40]\rangle$, meaning that the target tuples can be included in the groups with coverage [34,40] for attribute `Age`. Function $id.map_a$ for attribute a is used when the indexes have been generated through a gid-based approach. The function maps a value v in the domain of a to the set of gids of the groups whose coverages of a include v. For instance, with respect to the previous example, $id.map_{\mathtt{Age}}(39) = \{g_1, g_2\}$. Note that whenever $map_a(v)$ ($id.map_a(v)$, resp.) returns the empty set, we can immediately conclude that relation r does not have any tuple with value v for attribute a. The information that a client has to store to implement these mapping functions is therefore the informa-

$map_{\texttt{Age}}$
[34,40]
[43,49]
[46,55]

(a)

$map_{\texttt{State}}$	
Ak	AkCaWy
Ca	AkCaWy, CaOh
Oh	CaOh
Mi	MiTxWy
Ne	NeTx
Tx	MiTxWy, NeTx
Wy	AkCaWy, MiTxWy

(b)

Coverage	**Gid**
[34,40]	$\{g_1,g_2\}$
[43,49]	$\{g_3\}$
[46,55]	$\{g_4\}$

(c)

Coverage	**Gid**
AkCaWy	$\{g_2\}$
CaOh	$\{g_3\}$
MiTxWy	$\{g_4\}$
NeTx	$\{g_1\}$

(d)

Figure 4.9: Information stored at the client-side for representing functions $map_{\texttt{Age}}$/$id.map_{\texttt{Age}}$ (a)-(c) and $map_{\texttt{State}}$/$id.map_{\texttt{State}}$ (b)-(d)

tion that allows the efficient retrieval of all the coverages including the target value and their gids.

The index values corresponding to the target groups are generated by the client through the application of the same index function adopted for producing the encrypted and indexed relation. More precisely, let $I_a(v)$ be the set of index values associated with $v \in \mathrm{d}(a)$. Formally, $I_a(v)$ is defined as:

- $I_a(v)$=$\bigcup_{c \in map_a(v)} \iota_a(c)$ (value-based) or
- $I_a(v)$=$\bigcup_{id \in id.map_a(v)} \iota_{gid}(id)$ (gid-based).

For instance, with respect to the previous example, we have that $I_a(39)$ is equal to the token representations $\{\epsilon, \zeta\}$, with a value-based approach, and $I_a(39)$ is equal to $\{g_1,\ g_2\}$, with a gid-based approach and assuming, for simplicitly and without loss of generality, that the gids are also used as index values.

Map structures

To efficiently perform search operations over the map during query translation, we support different structures for the creation of maps.

For continuous attributes, a first solution consists in representing the support of MAP[a] as a list of elements [v_l, v_u] ordered with respect to the lower limit v_l or the upper limit v_u, depending on whether we also aim at efficiently supporting conditions of the form $a \leq v$ or $a \geq v$, respectively. In fact, for efficiently evaluating a condition $a \leq v$, the coverages including v are those sequentially accessed until

a coverage with the lower limit greater that v is found. Analogously, for efficiently evaluating a condition $a \geq v$, the coverages including v are the coverages that follow those sequentially accessed and with an upper limit that is lower that v. Note that if both types of conditions need to be efficiently supported, two copies of the ranges ordered as described can also be maintained. For instance, in Figure 4.9(a) three coverages are ordered in increasing order with respect to both their lower and upper limit.

An alternative solution for continuous attributes consists in using an *interval tree*, that is a balanced binary tree where each node stores at least one of the coverages to be represented. Being a balanced binary tree, the interval tree ensures that searches have a logarithmic cost in the number of coverages represented in the tree. Moreover, an interval tree is far more efficient compared to the solution based on two ordered lists for the evaluation of conditions of the form $v_l \leq a \leq v_r$, as it doesn't rely on a sequence of two sequential scans followed by an intersection to determine the resulting list of coverages (details in Section 4.6).

For each nominal attribute $a \in A$ instead, the client has to store, for each actual value v in the domain $\mathtt{d}(a)$ of attribute a, the coverages including it. At the logical level, this information can be seen as a set of entries of the form $\{(v,\{c\}) \mid c \in Map[a]$ and $v \in c\}$. Then , like for continuos attributes, the client stores the association between the coverages and their gids. For instance, Figures 4.9(b)-(d) illustrate the logical representation of the information stored at the client for implementing the map functions for attribute `State`. Figure 4.9(b) shows seven rows, one for each of the distinct values of attribute `State` appearing in the twelve tuples of the original relation in Figure 4.1(a). Figure 4.9(d) shows the correspondence between the coverages in Figure 4.9(b) and their gids.

Like for continuous attributes, we have implemented different solutions for representing the mapping function of nominal attributes. The first solution consists in using a *bitmap*, with a row for each nominal value and a column for each set in the support of $\text{M}{\small\text{AP}}[a]$. If a value v_i is included in the j-th coverage, the entry (i, j) of the bitmap is set to 1; 0, otherwise. Bitmaps can efficiently identify, for each value, the coverages including it. Another solution consists in using a *roaring bitmap*, which is a compressed representation of a bitmap. This representation is particularly convenient when bitmaps are sparse, meaning that several entries are 0

(which may happen frequently in our scenario since coverages are small compared to the size of the attribute domain).

4.6 Implementation

In the previous sections we have presented our proposal from a high level perspective. In this section, we detail our open-source implementation. Operations have been separated into preprocessing and runtime. Preprocessing operations involve transforming the initial dataset into a k-flat relation, encrypting it, and sending it to the storage provider. Runtime operation are instead associated with the translation and resolution of queries.

Preprocessing

Preprocessing is organized into four steps: 1) construction of the k-flat relation, 2) construction of the maps, 3) dataset wrapping, and 4) dataset outsourcing.

Construction of the k-flat relation. The construction of the *k*-flat relation is performed by a distributed application leveraging Apache Spark [37], and deployed with Docker [85]. The app is implemented in Python and starts on the master node. The master node cooperates with a configurable number of workers (each of them running in a dedicated container). This permits to efficiently execute the k-flat indexing algorithm on large datasets. Tuple shuffling is reduced to a minimun, as each of the workers processes a dedicated chunk of the initial relation. To speedup the Spark app we rely on Pandas [144], an open source data analysis library, and Arrow [175], a cross-language development platform for in-memory analytics providing a language-independent columnar memory format. To automatically build, install and run the application we instead leveraged Docker Compose [86].

A set of composable Python utilities implements steps 2, 3 and 4. The user can customize each intermediate step through a json file. Several options are available to the user. As example, the user can select the map to use to index each attribute a, and whether to use a gid-based (*group-id map*) or a value based (*token map*) indexing strategy. Moreover, the user can select the type of server-side backend. We elaborate more on this in the description of step 4.

Construction of maps. Starting from the k-flat relation, the client builds the local maps. For continuous attributes, range and interval-tree [57] maps are available; while for nominal attributes, set, bitmaps and roaring-bitmaps [173] maps can be used. The maps are characterized by different storage cost and runtime performance. For instance, our implementation of the range map occupies approximately a third of the space occupied by the interval-tree map, but the interval tree map is far more efficient when a closed range query is evaluated. Given N the number of ranges in the map, and M the number of ranges in the interval requested by a query, the range map needs $O(N)$ memory to retrieve the M ranges in the result, while the interval-tree map needs only $O(M)$. The maps are then completed storing the group-ids or the token seeds. As last operation in the construction of maps, the local maps are made persistent. Pickle [154] is used to serialize the maps to different bytestrings, then the bytestrings are encrypted, and finally transferred to file.

Dataset Wrapping. After the maps have been produced, the initial relation is prepared to be outsourced to the storage provider. For each attribute in the index, a group-id and/or a token is retrieved. Group-ids (the *gids*, e.g., g_1) are simply looked-up in the maps, while tokens (e.g., $\{\zeta, \epsilon\}$) are derived starting from the seed. The seed is expanded to 16 bytes and used as input to the AES symmetric encryption cipher configured in Cipher Block Chaining (CBC) mode. A 16 bytes salt (unique per-attribute), is used as Initialization Vector, and a robust key derivation function (Argon2id [4]) is used to retrieve the 16 bytes key. Multiple tokens can be produced at each encryption round, as each of them is extracted truncating the output of the cipher. The process is deterministic, and associated with a collision probability that can be approximated with the *birthday-paradox*, and that can be reduced increasing the length of the tokens (i.e., each token can be seen as a fixed length bit sequence, and compactly stored as an unsigned integer).
Immediately after the group-ids or tokens are retrieved, the initial relation is transformed into a dataframe. The records in the dataframe have the following schema: $(gid, i_1, ..., i_m, b)$ where gid is the group-id, $i_1, ..., i_m$ is the sequence of index values represented with tokens (if requested by the user), and b is the list of plaintext tuples in the group.
Each block b is then replaced with B, the unintelligible physical representation of the block. Encryption is applied according to the following procedure. b is serialized into a binary object using Pickle, it is padded so that the blocks

are physically indistinguishable in length, and after that it is encrypted through a non-deterministic Authenticated Encryption cipher. To encrypt the blocks we used XSalsa20-Poly1305, an Authenticated Encryption cipher implemented by `Libsodium` [126]. The use of a Message Authentication Code (Poly1305 [69]) permits to verify the integrity and authenticity of the blocks, as any attempt to tamper with a block (or a portion of it) is detected with decryption. Again, the 32 bytes key used by the cipher is derived using the Argon2id key derivation function. A unique 24 bytes high-entropy nonce is used for each block. Please notice that, since the blocks have been wrapped with a PRF (XSalsa20 [70]), and that they are indistinguishable in length, an adversary with a copy of the storage of the cloud provider learns nothing on the relation between blocks and index values (e.g., the attacker cannot infer which of the blocks are likely to be related to an index value).
Since tokens can be pre-computed, and group-ids are read-only, indexing and wrapping are executed in parallel up to the number of cores available to the client. This is achieved using the multiprocessing [153] Python library.

Dataset Outsourcing. We provide a separate container for the server. Based on the backend selected by the user, a container running PostgreSQL [176] or Redis [157] will be automatically deployed using Docker Compose. Hence, the indexed relation is outsourced to the server. Based on the setup selected by the client (i.e., the backend type and the group-ids or token based maps) there are four setup alternatives (or simply *configurations*):

- PostgreSQL with group-ids, the server stores a relation with the following schema $(\underline{gid}, B)$;
- Redis with group-ids, the server stores a hash set with gid as the key and B as the value;
- PostgreSQL with tokens, the server stores a relation with the following schema $(\underline{i_1, ..., i_m}, B)$;
- Redis with tokens, the server stores a primary hash set with gid as the key and B as the value, and a set of m secondary hash sets (i.e., auxiliary indexes), each with I_i as the key and gid as the value.

A representation of the information stored on the server based on the configuration selected by the user is illustrated in Figure 4.10.

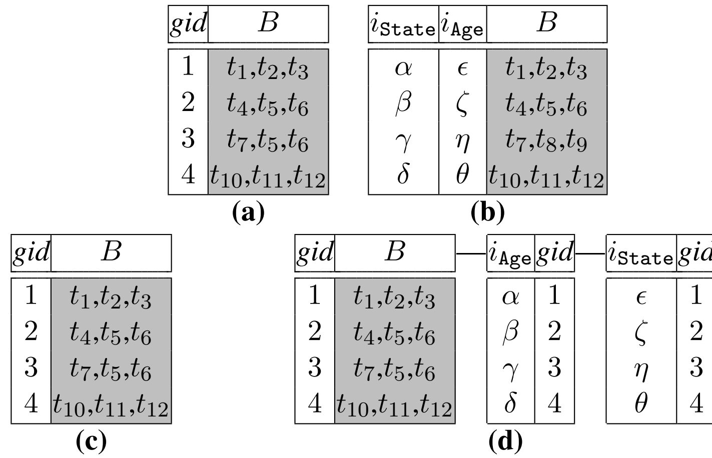


Figure 4.10: Server-side representations: (a) Relational DBMS with GIDs, (b) Relational DBMS with Tokens, (c) Key-value store with GIDs, and (d) Key-value store with Tokens.

Runtime

At runtime, the user deploys the container running the server and starts our client application, providing the master key as an input. The master key is used to decrypt and load the local maps from storage to memory. Immediately after that, an empty SQLite [169] in-memory database is initialized. At this point, the local environment is setup, and the user can submit queries to the client application.

Upon receiving a query, the client uses sqlparse [3] to parse it. Maps are used to translate each point or range value for all the attributes in the predicate list. Based on the setup and on the type of query (i.e., single or multi-column), the client can use different server execution strategies. With PostgreSQL, the query is simply rewritten and sent to the server using SQLAlchemy [49]; with Redis, the server is queried with Redis-py [138] in two ways:

1. group-id maps: the client queues a sequence of `get` operations to a pipeline, which is then run on the server only once;

2. token maps: the client submits to a server-side Lua script (registered during deployment) a hashmap mapping the attributes and tokens targeted by the query. The script first performs multiple lookups in the secondary hash sets

to determine the targeted gids, then it looks up the gids in the primary hash set and returns the blocks to the client.

In both cases (i.e., PostgreSQL and Redis), predicate intersection on the same attribute is solved at client-side, to reduce the data exchanged between client and server. When group-id maps are used, the intersection is evaluated on the client-side even when the query is on multiple attributes.

After the blocks are pulled from the server, the are decrypted (integrity and authenticity are verified by the cipher suite) and the padding removed. Then, the plaintext memory is de-serialized using Pickle (thus obtaining a list of tuples), and written to the SQLite in-memory DB with SQLAlchemy. Finally, spurious tuples are filtered executing the initial query on the SQLite instance.

4.7 Experimental results

The architecture proposed considers as critical resources the amount of storage used by the client to save the maps, and the performance impact that affects query execution due to the clustering of tuples. This section investigates these two aspects. Appendix C details how to use our open source solution to fully reproduce the experimental results presented.

Storage

The solution proposed permits to outsource an encrypted and indexed version of a relation to a storage provider. The ability to query the relation is untouched, as long as the client stores the maps locally. Hence, a first aspect to investigate is the resulting saving in storage.

The size of the maps is affected by three factors: *i)* the relation used, *ii)* the indexing strategy, and *iii)* the number of tuples in each block k. The nature of the relation (i.e., the number of tuples, the attributes, the value distribution of the attributes, and whether each attribute is continuous or nominal) cannot be changed, while we provide alternatives for *ii)* and *iii)*.

With regard to *ii)*, the indexing strategy, we described two alternatives:

1. the gid-based strategy, in which the mapping between each coverage and the related list of *gids* is materialized inside the map;

2. the value-based strategy, in which the map materializes only the starting seed and the number of token occurrences, for each coverage.

To evaluate the size of client-side maps we indexed three datasets extensively used in literature: the `usa2018` [180], the `usa2019` [181] and the `transactions` [117] dataset. The datasets collect 0.5M, 3.5M and 30M tuples, respectively. The number of tuples in the block k was set to 25. As we can see from the results shown in Table 4.1, our approach permits to limit client-side storage to 4.51%, 2.05% and 1.40% respectively of the original dataset size. Savings in storage further increase if we consider that, compared to a client-only solution, the client is no longer required to store locally additional resources as the DBMS installation files, the metadata, and the auxiliary DBMS indexes. We also observe that runtime token generation produces significant storage saving compared to the materialization of *gids*.

Dataset	**Size**	k	*Tokens*	*Group-ids*
usa2018	12 MB	25	4.51%	6.66%
usa2019	65 MB	25	2.10%	3.60%
transactions	1.5 GB	25	1.40%	2.10%

Table 4.1: Relative size of the client-side maps given the size of the initial dataset

We then conducted some experiments varying the number of tuples in each block k, to evaluate case *iii)*. The results are shown in Figure 4.11. We observe significant reductions in the size of the maps as k increases. However, the optimal value of k is obtained by considering the trade-off between storage requirements and runtime performance. We elaborate more on this in Section 4.7, showing how much the value of k affects query execution time.

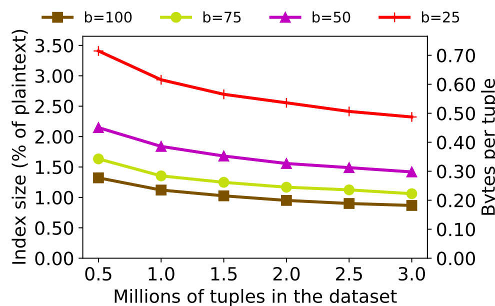


Figure 4.11: Map size for each sample of `usa2019`

Runtime

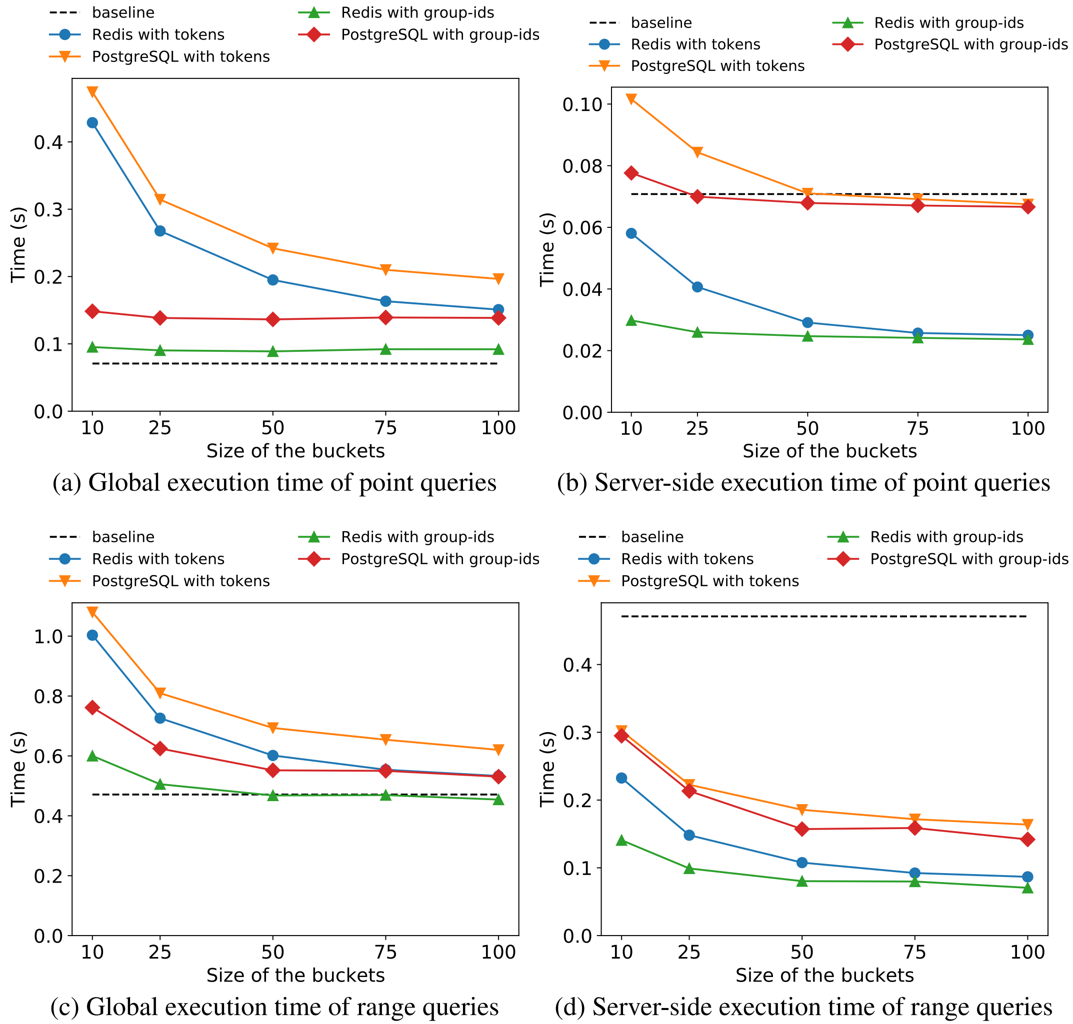
For users a crucial aspect in the adoption of the proposed approach is its performance. The clustering of tuples inevitably leads to an increase in the size of the results returned by a query. This increase does not directly map to a corresponding increase in query response time. Many elements come into play and experiments will show that in most cases the impact is limited; in some configurations we even see an improvement compared to a classical configuration where the data are stored in plaintext on a remote database.

Taking into account the two approaches to create the maps and index the relation (one with *gids*, and the second with *tokens*), and the two backend alternatives (PostgresSQL and Redis), we get four configurations. The aim of the runtime experiments is to evaluate the performance of each of them. The performance is always compared with the one of the *baseline*, which is a lightweight configuration used to execute the queries over the plaintext dataset. In this case, the server runs an instance of PostgreSQL, where the dataset is stored as a table. The client directly submits the query to the server (withouth performing any rewriting), the server executes the query, and the result is sent back to the client. No security protection is available when using the baseline.

We conducted several tests. Each test performs in sequence a sample of queries over the dataset. The dataset used is the `usa2019`. The queries used to build the sample are randomly extracted from a pool that globally contains 5,000 queries. The pool groups queries with the same *selectivity*, i.e., the fraction of tuples in the dataset that are part of the query result. In the experiments we consider queries with a selectivity of 10% of the dataset or less.

Single-column queries

In the first experiment we measure the performance of queries over a single column (`WAGP`). Network latency between the client and the server is set to 10 ms, and the transmission rate to 1 Gbps. The experiment is repeated 5 times (i.e., the 4 setups + the baseline), and for k equal to 10, 25, 50, 75, 100. The results are shown in Figure 4.12. We use the same color and marker for each setup across all experiments; a dashed line always represents the baseline. Each point in the plot represents the average execution time of the queries in the sample; standard



(a) Global execution time of point queries (b) Server-side execution time of point queries

(c) Global execution time of range queries (d) Server-side execution time of range queries

Figure 4.12: Average execution time of queries on WAGP column with a 10 ms latency.

deviation is not shown, because in most cases it is smaller than the size of each marker. In the left column (cases `a` and `c`) we show the *global* execution time, while in the right column (cases `b` and `d`) we show the *server-side* execution time. The global execution time measures the time required to get the result of the query (the `SQL` query is submitted to the client-side query executor, the query is parsed, index values are retrieved from the maps, the query is rewritten with gids or with the tokens, the query is submitted to the server, the query is executed on the server to retrieve the blocks, the blocks are sent to the client, block encryption is removed, plaintext tuples are serialized into a SQLite DB, and finally spurious tuples are

filtered by re-submitting the initial query); the server-side execution time measures only the time required by the server to run the query and then retrieve the blocks, i.e., the time from when the (rewritten) query is sent to the server until the result is received.

Several properties can be deduced from the analysis of the results in Figure 4.12. The most important one is that the global execution time (for each of the four setups) is comparable to (and sometimes smaller than) the baseline. Looking at cases 4.12b and 4.12d, which show the comparison between the server-side execution time of the various setup alternatives, we can see that the server execution times are aligned or well below the baseline for all the setups. On the other hand, if we compare cases *b* and *d* against *a* and *c*, we see that the server-side execution time of the baseline covers approximately the whole global execution time (as it should be expected, as the client in the baseline has no further processing to do). For the other setups, rewriting, decryption and the filtering of spurious tuples occupy most of the global time, leading those setups to offer in general a (slight) decrease in performance compared to the baseline.

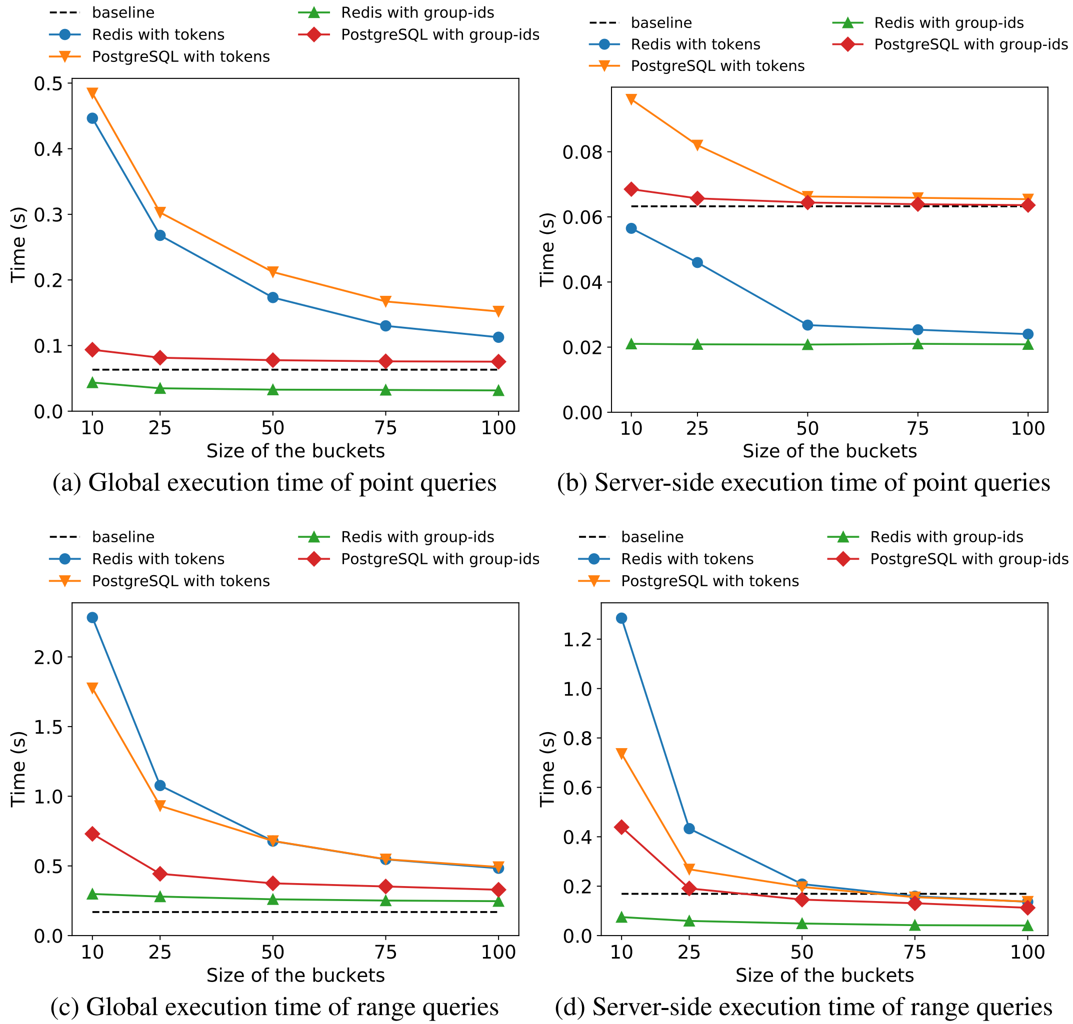
Another aspect is that the setups with gids are consistenly faster than the ones using tokens. This is shown by the delta between on one side the blue-circle and green-triangle curves, and on the other side the orange-triangle and the red-diamond ones. The difference in speed between the setups relying on the tokens and the ones using gids decreases when the size of the blocks increases, as larger blocks imply a smaller number of tokens.

We also observe that the global and server-side execution time decreases with larger blocks. With $k = 100$ we see most of the times the minimal execution time (a larger value of k corresponds to a smaller size of local maps), however large values of k increase the bandwidth requirements (more details about this in the following).

Lastly, we notice that point queries exhibit lower query processing times compared to the range ones, but this is justified by the larger size of the query results of range queries. The main focus of our analysis will be the comparison with the execution time of the baseline, rather than the absolute times.

Multi-column queries

In the second series of experiments we measure the performance of multi-column queries (selecion predicates on attributes `OCCP` and `WAGP`, a nominal and a contin-



(a) Global execution time of point queries (b) Server-side execution time of point queries

(c) Global execution time of range queries (d) Server-side execution time of range queries

Figure 4.13: Average execution time of queries on both OCCP and WAGP columns with a 10 ms latency.

uous). Again, the network latency between client and server is set to 10 ms, and the transmission rate to 1 Gbps (as in the previous experiment). We observe two distinct trends.

With point queries we observe significant improvements compared to the results shown for single-column point queries. This can be clearly seen comparing Figure 4.12a-b and Figure 4.13a-b. In both cases, the server-side execution time has identical behavior, but the global time is much smaller. There are two factors that cause this performance improvement: i) a smaller number of tokens is generated, and ii) the amount of data transmitted is lower. Factor (i) is proved by the difference

between the setups using gids and those with token maps. Factor (ii) is proved by the fact that server-side execution time dominates the global time for setups that use the group-ids map (this means that the time required to transfer the data between the client and the server is negligible compared to the server-side query execution time). This confirms that our approach is particularly effective for highly selective queries, like the queries considered in this experiment (the queries combine predicates over the two columns with a logical `AND`).

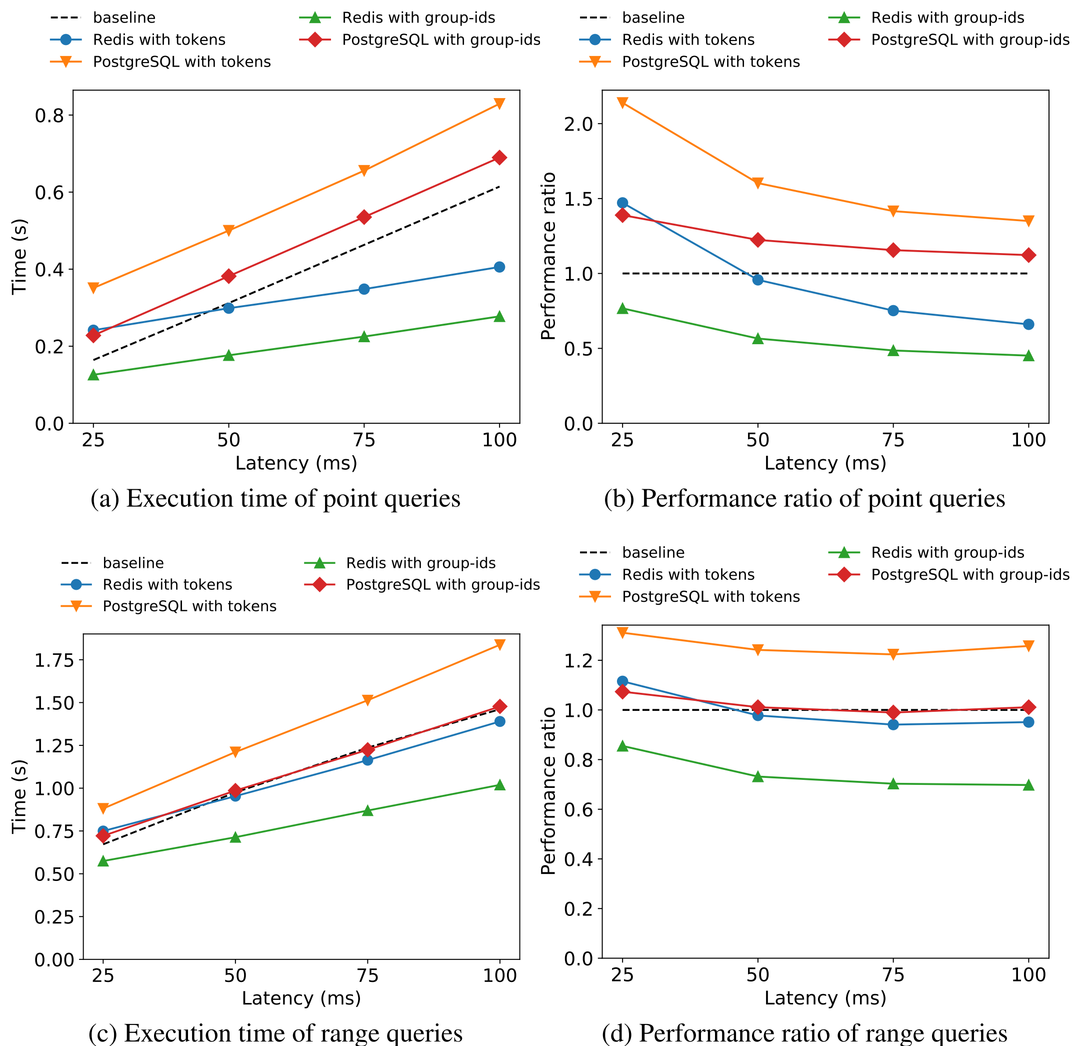
With regard to multi-column range queries, we observe similar performance between the baseline and the setups using group-ids maps (Figure 4.13c), while we notice a slight deterioration of performance when tokens are generated (overhead is approximately 2 for $k \geq 75$). Comparing Figure 4.13c and Figure 4.13d we notice that all the four setups spend a non negligible amount of time to delete spurious tuples (we will evaluate bandwidth degradation in the following).

Impact of latency

In most data outsourcing scenarios latency is a crucial parameter, with a direct impact on performance and usability. Geographical distance between client and server inevitably causes relatively high latency in the dialogue. To quantitatively measure the impact of latency, we repeated the previous tests (in this case the single-column query execution) varying the latency. The latency values used to conduct our experiments are 25, 50, 75, and 100 ms, corresponding to round-trip-times (RTTs) equal to twice the latency.[1] The transmission rate in these experiments is 1 Gbps. These latency values have been selected to mimic a variety of configurations, with the server located in the same geographic region or farther from the client. Group size k was instead set to 50. The experimental results are shown in Figure 4.14, which also depicts the results in terms of the ratio between the execution time of each setup and that of the baseline.

We observe that for both point and range queries, the performance ratio improves with the increase of latency. This is clearly expected when the ratio is greater than 1, as the addition of latency can be expected to create an additional fixed cost for all the setups and this reduces the impact of the overhead introduced by client-side processing. The experiments also show an improvement for config-

[1]The delay is set using `tc` [174], a utility bundled with `iproute2` [115] that permits to control the Kernel packet scheduler.



(a) Execution time of point queries

(b) Performance ratio of point queries

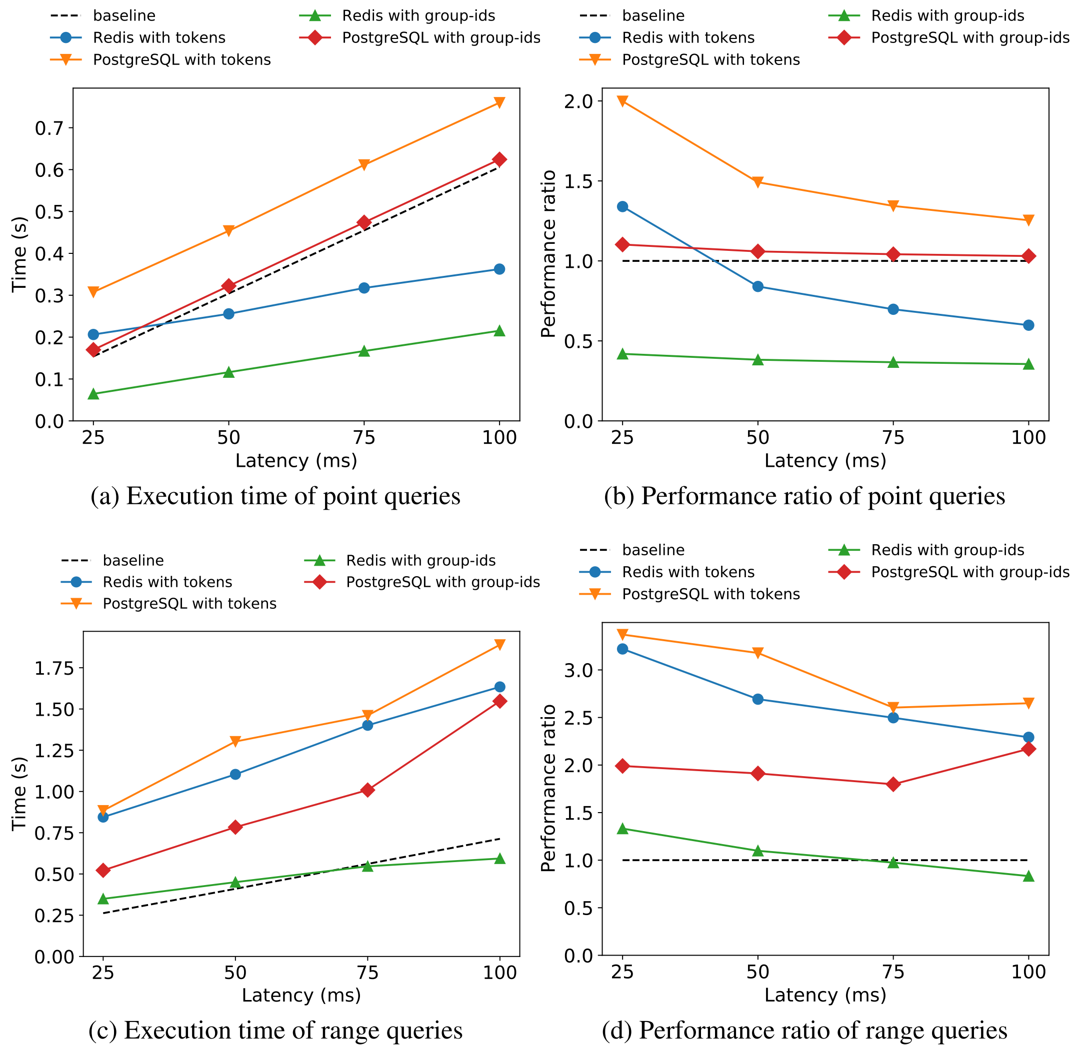
(c) Execution time of range queries

(d) Performance ratio of range queries

Figure 4.14: Execution time and performance ratio of queries on WAGP column with the addition of latency.

urations using Redis, with an increase in execution time that directly grows with the increase in latency, whereas all the configurations using PostgreSQL (including the baseline) exhibit a large impact of latency. This derives from the protocol used to access PostgreSQL in the server. Overall, the experiments demonstrate that the approach is particularly interesting when there is relatively high latency.

Similar behavior is exhibited by the same experiment run on multi-column queries, shown in Figure 4.15. In all these experiments we see that the performance ratio improves with the increase of latency, especially for the setups with Redis and the use of gids.



(a) Execution time of point queries

(b) Performance ratio of point queries

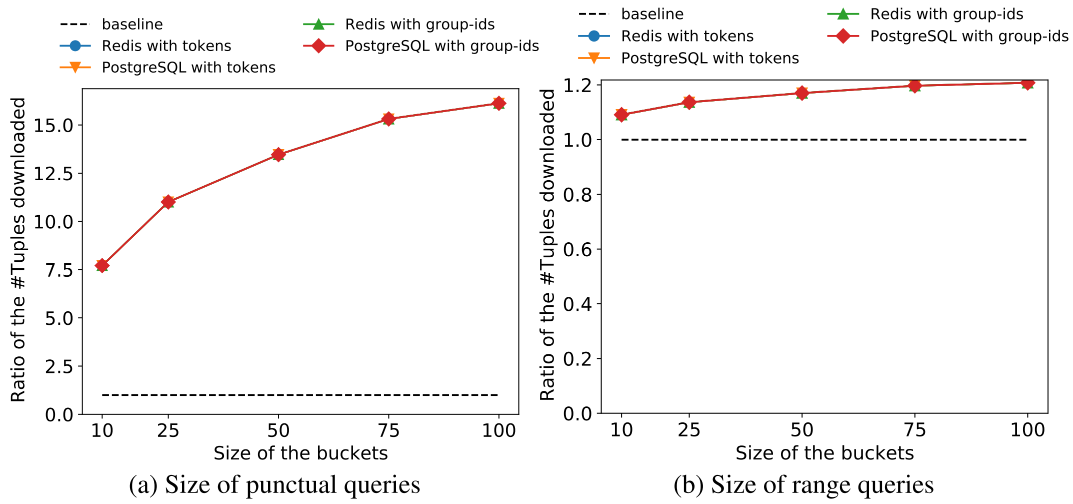
(c) Execution time of range queries

(d) Performance ratio of range queries

Figure 4.15: Average global execution time and performance ratios of queries on both OCCP and WAGP columns with the addition of latency.

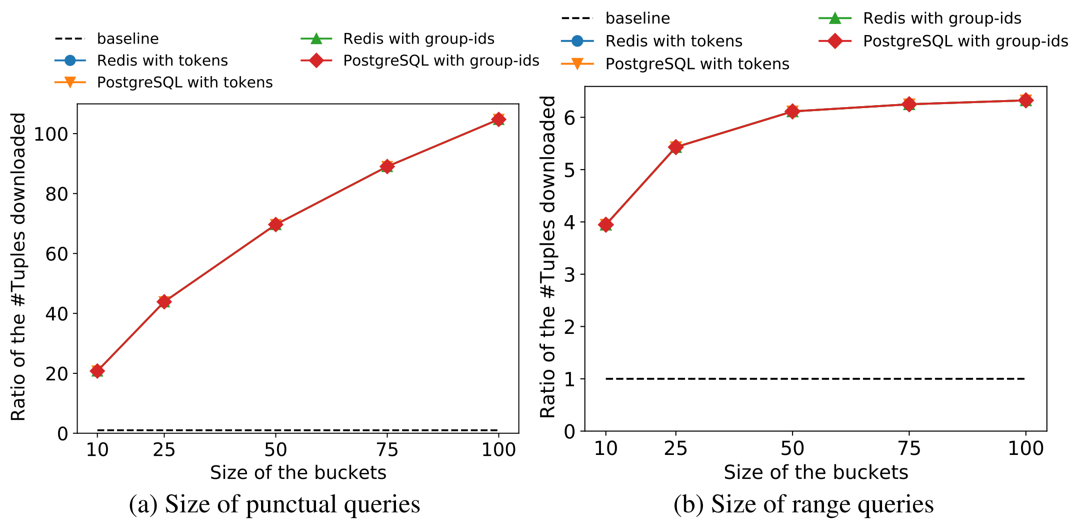
Data overhead

A natural consequence of using groups of tuples as an atomic entity in the dialogue between client and server is an increase in the amount of data returned in query execution. In Figure 4.16 we report the overhead introduced by the execution of the query workload, for point and range queries, when executing queries on the single column WAGP. Figure 4.17 reports the same measure for the multi-column queries considered before, on attributes OCCP and WAGP. As shown by the experiments, the data overhead is significant. This proves to have a limited impact on high-



(a) Size of punctual queries (b) Size of range queries

Figure 4.16: Overhead in the number of tuples downloaded with queries on the WAGP column.



(a) Size of punctual queries (b) Size of range queries

Figure 4.17: Overhead in the number of tuples downloaded with queries on both OCCP and WAGP columns.

bandwidth (1 Gbps) configurations like those used in the previous experiments, but it may have an economic and performance impact depending on the scenario where the technique is used. Many cloud providers monitor and make customers pay for the data they transfer outside of the cloud infrastructure. We expect that in some cases the security requirements are going to be the major need and the additional data transfer cost can be considered negligible. In scenarios where instead this data

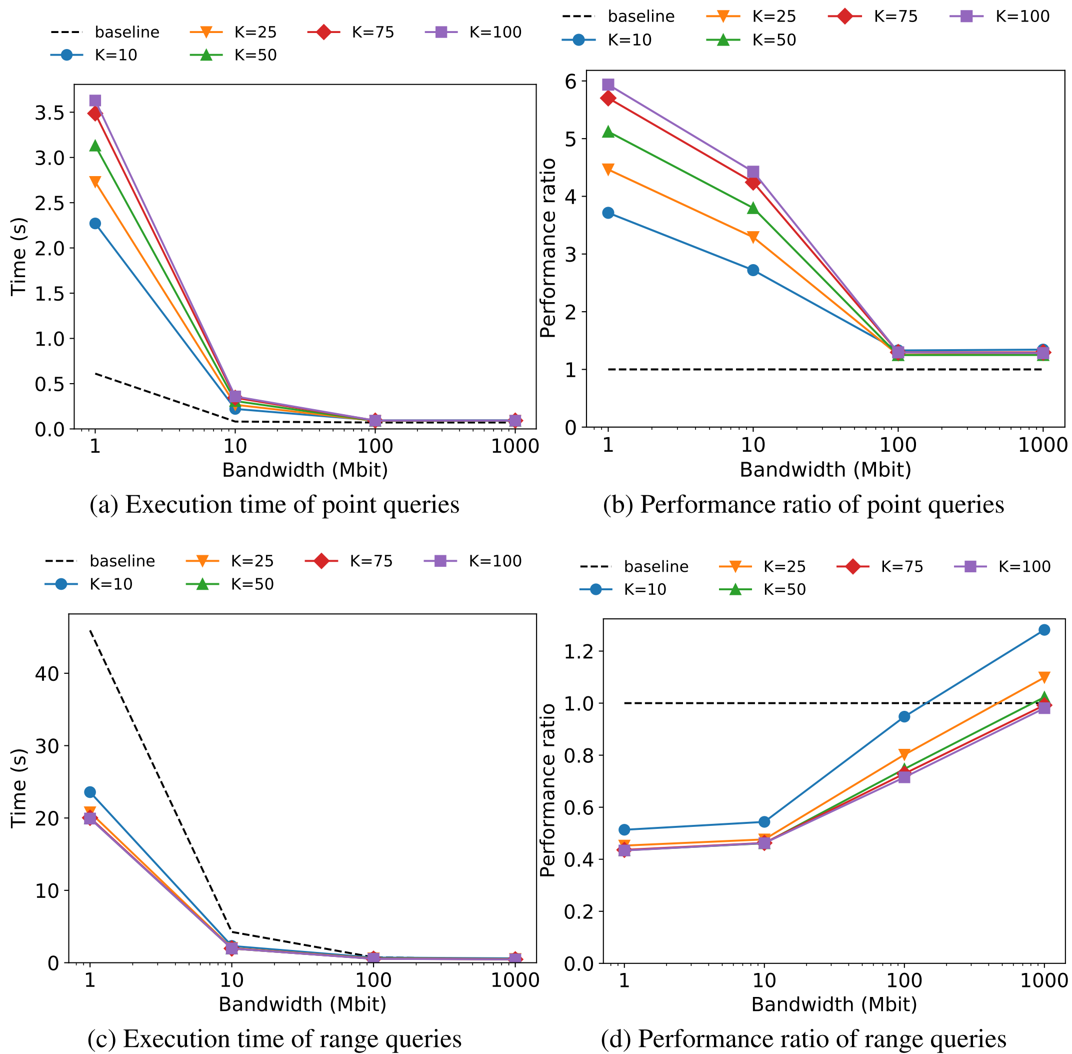
transfer cost is critical, the approach proposed in the paper meets some obstacles and it can only be used with relatively small k values. This overhead is significant only for point queries, which return a compact result. For range queries the overhead is a lot smaller, thanks to the organization of the groups in a way that keeps together tuples with similar values of the attributes.

With respect to the impact on performance of the increase in data transfer, we run a dedicated set of experiments reducing the bandwidth between client and server. Figure 4.18 and Figure 4.19, respectively for single-column and multi-column queries, report the execution times for configurations with varying values of k and bandwidth of 1 Mbps, 10 Mbps, 100 Mbps and 1 Gbps. The evolution of network technology is making available in most scenarios communication channels with bandwidth above 100 Mbits, but there are domains where limited bandwidth may still be a concern. The experiments confirm that on broadband network connections, the impact of the increased data transfer is limited, whereas for limited channels (around 1 Mbps) the size of the transferred data becomes the factor dominating the performance.

4.8 Related work

Several research efforts have addressed the problem of supporting queries on outsourced encrypted data through the definition of indexing techniques (e.g., [74, 106, 111, 112]). The definition of efficient solutions that are robust against inferences often depends on the specific queries to be supported (e.g., [182]). While sharing with our approach the goal of supporting queries over encrypted data, these solutions operate on a single attribute. Our approach instead is based on a multi-dimensional interpretation of the dataset that allows the definition of indexes over multiple attributes. The problem of indexing multidimensional datasets has been already considered and resulted in the definition of multidimensional indexes for supporting queries with conditions on multiple attributes (e.g., [184]). These solutions, however, differ from our since they define a single index for the whole set of attributes considered. Our approach instead defines an index with a component for each attribute.

Another approach to support the execution of queries over encrypted data involves the use of Searchable Symmetric (SSE) [53, 78, 148] and Order Preserving



(a) Execution time of point queries

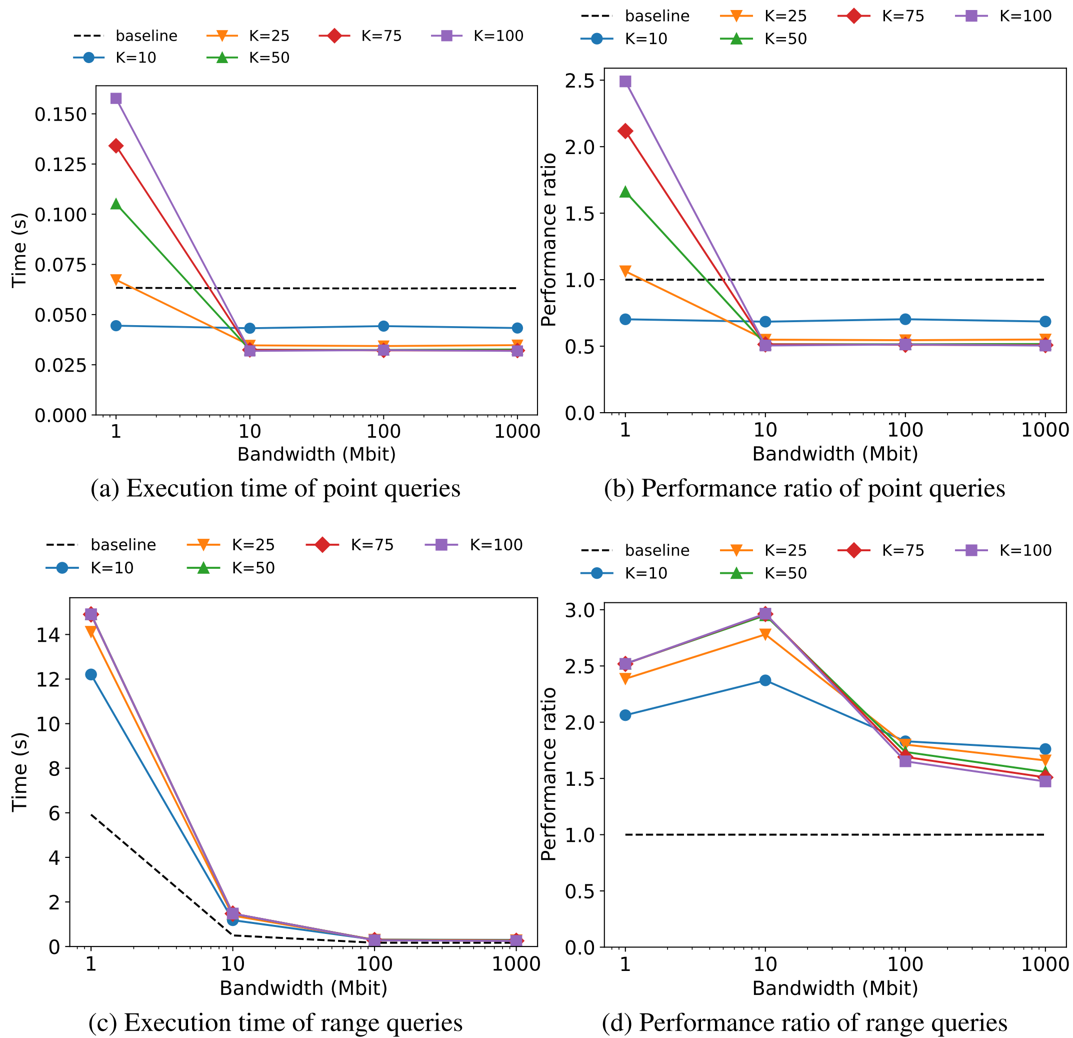
(b) Performance ratio of point queries

(c) Execution time of range queries

(d) Performance ratio of range queries

Figure 4.18: Execution time and performance ratio of queries on WAGP column with a latency of 10 ms and, varying K and bandwidth.

(OPE) [51, 150] Encryption. SSE is a form of deterministic encryption that permits to outsource data to a cloud storage provider, while maintaining the ability to selectively search over it. OPE is again a kind of deterministic encryption, but that preserves the numerical ordering of the plaintext data. Although these techniques bear some advantages (e.g., OPE enables the execution of functions like `MIN`, `MAX`, `COUNT` directly on encrypted data with a relatively low overhead), they must be applied with care due to the inevitable information leakage.



(a) Execution time of point queries (b) Performance ratio of point queries

(c) Execution time of range queries (d) Performance ratio of range queries

Figure 4.19: Execution time and performance ratio of queries on both OCCP and WAGP columns with a latency of 10 ms and, varying K and bandwidth.

A recent solution that permits to perform computations on encrypted data without first decrypting it is represented by Fully Homomorphic Encryption (FHE) [99, 139] schemes. The resulting computations are left in an encrypted form which, when decrypted, results in an identical output to the one produced performing the proper operation on the unencrypted data. Fully homomorphic encryption schemes are a promising solution. However, they result in an overhead too large to be considered practical in many scenarios. This does not impact only the processing time, but also the size of the encrypted data compared to the plaintext values. An increase in the size of data may be inconvenient if the user pays for the bandwidth consumed.

To completely revoke access to the data content to the cloud storage provider, researchers have also proposed to use Oblivious RAMs (ORAM) [100, 101, 143]. The schemes developed through this technique (e.g., [40, 61, 75, 171]) ensure optimal security guarantees, as the cloud provider no longer has the ability to access to the data, nor to monitor the access pattern. Unfortunately, the ability to answer queries depends on the availability of a safe environment, that could reside at but is not accessible by the cloud provider (e.g., Intel SGX [114]), or reside at the client. Despite the strong security guarantees, these solutions are characterized by a significant overhead.

Lastly, with regard to the relational database world, one notable implementation is CryptDB [151]. In CryptDB the data is wrapped applying multiple encryption layers (deterministic, order preserving, symmetric searchable, and finally probabilistic), an approach commonly referred to as *onion encryption*. Depending on the relational operator or the function to be executed on the data, each of the layers can be removed. A proxy is instead used to translate the queries so that they are compatible with the encrypted data, and the result of the queries is decrypted at client side. Despite this solution is characterized by a limited overhead, the proposal suffers from the drawbacks associated with the previous approaches (e.g., information leakage, the need to share decryption keys between the client and the server to remove some of the layers).

4.9 Conclusions

This chapter addressed the problem of outsourcing encrypted data to external providers and defining indexes over the data for enabling query execution. The multidimensional technique used to construct the index guarantees protection against inferences, while providing efficient query execution, with support for both point and range conditions. Our experimental evaluation on publicly available datasets confirms the the effectiveness of the approach.

Further details on the open source solution described in this chapter can be found in Appendix C, which details the command line user interface and how to fully reproduce the results shown in the chapter.

Availability

The implementation source of our proposals is freely available at this URL: https://github.com/unibg-seclab/secure_index

The work in this chapter was supported in part by the EC within the H2020 Program under projects MOSAICrOWN and MARSAL, by the Italian Ministry of Research within the PRIN program under project HOPE, and by JPMorgan Chase & Co under project “k-anonymity for AR/VR and IoT/5G”.

Chapter 5

Data Release

ITYT: scheduling the release of data at a future time

In the last part of this thesis we investigate the data release stage. In particular, we aim at providing a method to confidentially schedule the release of data to a future point in time. This primitive is commonly referred to as a *Time-Lock* in literature.

The solution typically adopted to implement a Time-Lock (or simply, TL) involves the use of cryptography. The data to be released (i.e., the *secret*) are often concealed into a cryptographic puzzle by a sender (i.e., the *owner*). The puzzle is then handed to a receiver, which needs to find a solution to recover the secret. Solving the puzzle requires to run a decryption procedure for a very long time. However, as resolution time is affected by many factors, puzzles may not suit all scenarios.

To overcome this limitation we propose a novel way of implementing a TL. The basic idea is to split the secret data into *shares*, each handed to a different user (i.e., a decentralized network node). After the disclosure time set by the owner, the users cooperate according to a predefined protocol to recover the secret. The outcome of the protocol (as any attempt to tamper it) can be verified by a smart contract, which relies on the blockchain to measure the elapse of time. Security no longer follows from the trust for a single third-party, nor on any honesty assumption, but from the behavior of the users, whom are simply assumed to be rational.

To evaluate the approach we implement a prototype on top of the Ethereum blockchain. The prototype leverages secure computation protocols to avoid any single point of trust. Resiliency to attacks is analyzed with the help of economic game theory, in the context of rational adversaries. The experimental results demonstrate the low cost and limited resource consumption associated with this approach.

5.1 Introduction

Many real-world scenarios require disclosing a secret at a specific future point in time. For instance, this happens when we vote for elections or when we dispose our inheritance by will. In these circumstances we typically entrust a notary to keep a secret private until a future time, and then to publish it so that we are no longer needed for disclosure to happen. This however requires that the owner of the secret completely entrusts a single third-party (i.e., the notary).

Early proposals bound the recovery of the secret to the recovery of a key [156], which was split among a number of peers, distributing trust among many parties instead of entrusting a single one.

To completely remove the dependence on one or more trusted third-parties, cryptographers have been working on *timed-release cryptography* [136] proposing schemes that effectively replace notaries with Time-Locks (TL). Time-Lock *puzzles* [50, 132, 159] bind the recovery of the secret to the solution of a cryptographic puzzle. In this setting, a sending party (i.e., the secret's owner) can encrypt the secret so that the receiving party is required to perform multiple decryptions to recover it. Given the assumption that each decryption round takes the same amount of time, and that the number of rounds can be tuned by the sender during the encryption process, it is possible to protect a secret for arbitrarily long periods of time.

The first puzzle proposed by Rivest et al. [159] uses a trapdoor function so that anyone willing to recover the secret had to undergo a computing effort that was orders of magnitude larger compared to the one of the sending party. Also, since the authors imposed the decryption process to be executed in an inherently sequential order, their approach is the first example of a *Proof of Sequential Work* (PoSW) [66, 133].

An alternative to trapdoor functions is using *weak hash-chains* [54], by requiring anyone willing to obtain the secret to brute-force a chain of weak hashes. The use of a chain, rather than a single stronger hash, permits to reduce the variance of the decryption runtime. However, the decryption process remains parallelizable and, thus, the estimated disclosure time far less reliable. These approaches are often classified as *Proof of Work* (PoW) [88] algorithms.

Cryptographic puzzles avoid the need for a trusted party, yet two aspects make them impractical. First, the sending party has to make assumptions on future com-

puting power. This is far from trivial. As an example, Rivest's LCS35 time capsule [158], released in 1999 with an estimated decryption time of 35 years, was opened in 2019 after a decryption process of only 2 months on dedicated hardware [67]. Second, TL puzzles require the receiving party to run the decryption procedure for a long time, which poses a question about economic incentives.

The development of blockchains gives us new opportunities to implement time-locks. Indeed, a blockchain intrinsically defines the concept of time, which was one of the main reasons that led to the creation of cryptographic puzzles. It can also be used to persistently disclose secrets, as it is designed to resist modification of its data. However, the blockchain alone is not enough to deploy a TL because it does not offer any mechanism to keep information confidential. Recent proposals address the problem of dealing with secret data on public blockchains [1, 120] by splitting information among multiple users who have to cooperate to recover the secret information using a pre-defined protocol. The protocol is often programmed as a smart contract [172], which permits to verify the correctness of its outcome and to detect any undefined behavior. Although this approach could resemble the early implementations of timed-release cryptography, the security no longer follows from a single strong trust assumption (i.e., the trust to the notary or to a certification authority) but from the behavior of the users that take part to the protocol. If the users cooperate as intended, the outcome of the protocol will be successful, the secret will be recovered, and the TL function achieved as a consequence. To ensure that users cooperate as intended, several proposals (e.g., [125, 129]) reward them with economic incentives. This permits to analyze the protocols as extended form games whose outcome can be determined based on participants' expected utility.

In this chapter, we propose I Told You Tomorrow (ITYT), a practical and generic framework to implement time-locks using smart contracts that is based neither on trust assumptions nor on cryptographic puzzles. The basic idea of our approach is to first split a secret into shares using *threshold cryptography* (using Shamir's Secret Sharing [166]), and assign them to users so that no one can recover the secret unless k-*of*-n shares are available. To ensure the TL behavior, we rely on economic incentives and penalties enforced by a smart contract. The contract rewards users for revealing their share only after the disclosure time and penalizes any other misbehavior. As rewards and penalties are associated with the correct management of the shares, ITYT leverages *secure Multi-Party Computation* (sMPC) [189], which

ensures confidentiality and avoids the need for trusted users (including the owner of the secret).

Here we summarize our main contributions. We define a protocol that deploys Time-Locked secrets on the blockchain by leveraging an economic model in which every user (or coalition of them) has an expected negative payoff associated with possible misbehavior. We address the problems that arise when combining secure Multi-Party Computation and protocols based on economic incentives and penalties (e.g., use secure Multi-Party Computation protocols to break the protocol bypassing the smart contract's hashlocks). We implemented the protocol based on the Ethereum blockchain [185] and the FRESCO secure Multi-Party Computation framework [71], characterized by low overhead and limited cost of execution. Finally, we compare ITYT with other existing solutions, discussing the key advantages associated with our approach.

5.2 Background

We describe here a few concepts that are used in the design of the protocol.

Threshold cryptography Threshold cryptography [166] enables the owner of a secret to share it among a group of users. In a k-*of*-n threshold scheme, n shares of the secret are created and distributed among the parties. To reconstruct the secret, at least k different shares have to be combined. Hence, the advantages of threshold cryptography are (i) distribution of trust, and (ii) fault tolerance.

Secure Multi-Party Computation A *secure Multi-Party Computation (*sMPC) protocol [183,189] is a cryptographic protocol that allows multiple parties to jointly compute a function over their inputs while keeping them private. Current sMPC frameworks can execute binary or modular arithmetic algorithms computed among several parties (even with dishonest majority) by leveraging *semi-homomorphic encryption* [72,119] and *oblivious transfer* [118,155].

Smart contracts A blockchain is an append-only list of blocks linked together via cryptographic properties. The blocks are non-mutable and keep a permanent history of transactions. Smart contracts [172] are programming frameworks built

on top of blockchains. A smart contract permits to program tamper-proof protocols whose outcome is verifiable by the whole network. We rely on smart contracts to pay incentives, trigger penalties, and enforce the correct execution of our scheme without relying on a trusted party. Specifically, our approach makes use of the *time* and *hash* primitives, to conditionally execute actions based on time and submitted data (e.g., reward users participating in a successful execution of the protocol at protocol termination time).

Rational adversaries A malicious adversary [108] is someone who is willing to perform any action to attack a protocol. A rational adversary [105], instead, subverts the protocol only if it is economically convenient. Modeling the participants as rational enables the use of game theory concepts to analyze cryptographic protocols [8, 39, 58]. ITYT models each participant as rational.

5.3 The ITYT protocol

In this section, we overview the ITYT protocol, introducing the preliminary definitions, the roles of the participants, and the main functions.

ITYT is an instance of the *TL abstraction*: a mechanism that keeps a secret $\mathcal{S}$ private until its *disclosure time* t_d and publishes it afterward. ITYT implements TL by splitting the secret, provided by the *owner*, among several users named *shareholders* (each obtaining a share h), so that none of them can recover $\mathcal{S}$ before the disclosure time. To ensure that (i) each user keeps its share secret until t_d, and (ii) each user publicly discloses its share immediately after t_d, ITYT introduces a set of economic incentives and penalties. Thus, the TL function is achieved as a consequence of the rational economic behavior of the involved parties (see Section 5.4).

Definitions

Principals

We denote by $\mathcal{U}$ the set of users that take part in an instance of ITYT. Additionally, we denote by $\mathcal{SC}$ the set of the smart contract identifiers. We then denote by $\mathcal{P}$ the set of principals consisting of $\mathcal{U} \cup \mathcal{SC}$.

Wallets

Each principal $p \in \mathcal{P}$ is associated with a wallet $wlt(p)$, accessible only by p, that can be used to receive or issue payments.

Protocol parameters

In the following table we report the definition of all the parameters that characterize an instance of ITYT.

$\mathcal{S}$	secret
V	economic value assigned to the secret
n	number of shareholders
k	number of shares needed to reconstruct $\mathcal{S}$
h_i	share of the secret issued to the i-th shareholder
t_d	disclosure time
t_{term}	termination time
$F_{\mathcal{O}}$	fee deposited by the owner to use the service
$B_{\mathcal{H}}$	bid deposited by the shareholder to get a share
$R_{\mathcal{H}}$	reward paid to the shareholder in case of success
W_h	reward paid when whistleblowing a share
$W_{\mathcal{S}}$	reward paid when whistleblowing the secret

Smart contract state

The ITYT smart contract keeps track of the protocol status through the following data structures.

$[shares]$	array of shares submitted to the contract
$C_{\mathcal{S}}$	commitment of the secret (i.e., its hash)
$[C_h]$	array of share commitments
$state$	global state of the protocol
$[states]$	array of states for each share
$num_pending$	count of the pending shares
$num_disclosed$	count of the disclosed shares

Primitives

We now define the primitives used by the smart contract.

- `time()`: returns the current time as witnessed by the blockchain (generally defined in terms of block height).
- `hash(`d`)`: returns the result of the application of a chosen cryptographic hash function over the data d.
- `pay(`p_1, p_2, v`)`: transfers the amount v from $wlt(p_1)$ to $wlt(p_2)$.
- `initialize_sc(`$[params]$`)`: instantiates an ITYT smart contract, and deploys it to the blockchain. The primitive is executed by the owner and returns the smart contract identifier $sc \in \mathcal{SC}$.
- `generate_shares(`$\mathcal{S}, [users]$`)`: generates shares $h_1, \ldots, h_n$ and securely distributes them to the parties. The primitive guarantees that (i) the i-th shareholder is the only principal who learns the share h_i, and (ii) the owner learns only the commitment $hash(h_i)$, for each share. We discuss in Section 5.5 how our prototype implements this primitive by leveraging sMPC and secret sharing.

Roles

In ITYT, each user $u \in \mathcal{U}$ plays one of the following roles: *owner*, *shareholder*, and *whistleblower*.

Owner.

An owner $\mathcal{O}$ delegates the disclosure of a secret $\mathcal{S}$ to a time-lock at disclosure time t_d. The owner $\mathcal{O}$ configures and deploys the TL providing all the required parameters. In particular, $\mathcal{O}$ sets (i) the total number n of shares h of $\mathcal{S}$, (ii) the number k of shares needed to recover $\mathcal{S}$, (iii) the disclosure time t_d, and (iv) all the bids and the rewards that define the instance.

Shareholder.

A shareholder $\mathcal{H}$ is entrusted by the owner $\mathcal{O}$ to keep a share h of the secret $\mathcal{S}$ confidential until t_d, and to publicly disclose it afterward. In exchange for her service, $\mathcal{H}$ receives a reward $R_{\mathcal{H}}$ paid by the smart contract whenever the following two conditions hold: (i) her share h is disclosed only after t_d, and (ii) the secret $\mathcal{S}$ is not revealed before t_d.

Whistleblower.

A whistleblower $\mathcal{W}$ reports user misbehavior in return for payment. Whenever $\mathcal{W}$ captures a share h or the secret $\mathcal{S}$ before t_d, $\mathcal{W}$ can submit it to the contract and receive a reward.

Setup

In the early stage of ITYT, the owner initializes and deploys a smart contract sc on the blockchain using the `initialize_sc()` primitive. Along with the protocol parameters, she writes to the sc the identifiers of all the shareholders, which the owner has selected beforehand[1], that will take part in the TL instance. She also deposits to the contract the amount $F_{\mathcal{O}}$, a fee that will be used to pay the rewards at protocol termination time t_{term}.

After the contract is deployed, the owner and the shareholders jointly execute the `generate_shares()` primitive. As a result, the owner gets the hash of the secret and the hash of each share (i.e., the commitments), while each shareholder $\mathcal{H}$ gets her share along with the hash of the secret. The owner submits to the sc contract the commitments (Algorithm 5.1). Then, each shareholder reads the contract and checks whether her commitment matches what received from `generate_shares()`. If so, she agrees and deposits her bid $B_{\mathcal{H}}$ (Algorithm 5.2).

As soon as each shareholder has committed, the TL is activated (i.e., the global state is set to `LOCKED`). If any party refuses to commit, the funds already deposited are returned to their proprietaries and the instance setup aborts (as discussed in Section 5.5).

[1] How to randomly choose competing players in adversarial settings such as blockchains has been addressed in many literature works (e.g., [87, 141])

The reader may have noticed that in this setup, the secret shares are exposed to the shareholders prior to the activation of the TL (i.e., the shares are distributed before the state is set to `LOCKED`). In Section 5.5 we show how to overcome this issue using a key instead of the actual secret. We also point out that the verification of protocol parameters (especially the economic ones) can be done by the shareholders at commit time, as their values are publicly available in the contract.

$[params]$	protocol parameters of the ITYT instance
$\mathcal{S}$	secret
$\mathcal{O}$	user identifier of the owner
$[\mathcal{H}_1, \ldots, \mathcal{H}_n]$	list of shareholder identifiers

1: **procedure** $\textsc{Init}([params], \mathcal{S}, \mathcal{O}, [\mathcal{H}_1, \ldots, \mathcal{H}_n])$
2: $sc \leftarrow$ `initialize_sc` $([params])$ /* Create sc */
3: `pay` $(\mathcal{O}, sc, sc.F_{\mathcal{O}})$ /* Transfer owner's fee to sc */
4: $C_h \leftarrow$ `generate_shares` $(\mathcal{S}, [\mathcal{O}, \mathcal{H}_1, \ldots, \mathcal{H}_n])$
5: $sc.C_{\mathcal{S}} \leftarrow$ `hash` $(\mathcal{S})$ /* Set secret commitment */
6: $sc.state \leftarrow$ `PENDING`
7: **for** $i \leftarrow 1, n$ **do** /* Set share commitments */
8: $sc.C_h[i] \leftarrow C_h[i]$
9: $sc.states[i] \leftarrow$ `PENDING`
10: **end for**
11: $sc.num_pending \leftarrow n$
12: $sc.num_disclosed \leftarrow 0$
13: **return** sc /* The smart contract identifier */
14: **end procedure**

Figure 5.1: Protocol initialization algorithm (executed by the owner).

Actions

When the TL is active, the actions performed by users determine the status of the protocol. Each action is perfomed executing a smart contract function whose effects are public. Four actions are available to users: `WhistleblowShare`, `WhistleblowSecret`, `Disclose`, and `Withdraw`, with the last two reserved to shareholders.

`WhistleblowShare`: This action (Algorithm 5.3) enables the whistleblower $\mathcal{W}$ to report the misbehavior of a single shareholder. Whenever $\mathcal{W}$ obtains a share h_i, she submits it to collect a reward. If the commitment $sc.C_h[i]$ matches, the share

sc smart contract identifier
$\mathcal{H}_i$ user identifier of the i-th shareholder
precondition: $\mathcal{H}_i$ checks that `hash(`h_i`)` $= sc.C_h[i]$
1: **procedure** PARTICIPATE($sc, \mathcal{H}_i$)
2: **if** $sc.state =$ `PENDING` **and** $sc.states[i] =$ `PENDING` **then**
3: `pay(`$\mathcal{H}_i, sc, sc.B_{\mathcal{H}}$`)`
4: $sc.states[i] \leftarrow$ `BID`
5: $sc.num_pending - = 1$
6: **if** $sc.num_pending = 0$ **then**
7: $sc.state \leftarrow$ `LOCKED`
8: **end if**
9: **end if**
10: **end procedure**

Figure 5.2: Shareholder commitment algorithm

sc smart contract identifier
h_i the i-th share to be whistleblown
1: **procedure** WHISTLEBLOWSHARE(sc, h_i)
2: **if** $sc.state =$ `LOCKED` **and** `time()` $< sc.t_d$ **then**
3: **if** $sc.states[i] =$ `BID` **and** `hash(`h_i`)` $= sc.C_h[i]$ **then**
4: $sc.shares\,[i] \leftarrow h_i$
5: $sc.states[i] \leftarrow$ `WHISTLEBLOWN`
6: $sc.num_disclosed + = 1$
7: **if** $sc.num_disclosed = sc.k$ **then**
8: $sc.state \leftarrow$ `FAILED`
9: **end if**
10: `pay(`$sc, caller, sc.W_h$`)`
11: **end if**
12: **end if**
13: **end procedure**

Figure 5.3: Share whistleblow algorithm

whistleblow reward W_h is paid to the whistleblower and the shareholder $\mathcal{H}_i$ loses her reward $R_{\mathcal{H}}$. Additionally, if the number of whistleblown shares equals k, then the TL is marked as `FAILED` (i.e., no further actions allowed).

`WhistleblowSecret`: This action (Algorithm 5.4) enables the whistleblower $\mathcal{W}$ to prove the possession of the secret ahead of the disclosure time t_d, thereby reporting the misbehavior of a group of at least k shareholders. In detail,

$\mathcal{W}$ submits the secret $\mathcal{S}'$ to the contract. If the commitment $sc.C_{\mathcal{S}}$ matches, then the TL is marked as `FAILED` and the secret whistleblow reward $W_{\mathcal{S}}$ is paid to the whistleblower. Moreover, all the shareholders lose their bid, and the remaining smart contract funds are destroyed.

```
    sc      smart contract identifier
    S       the secret to be whistleblown
 1: procedure WhistleblowSecret(sc, S)
 2:     if sc.state = LOCKED and time() < sc.t_d then
 3:         if hash(S) = sc.C_S then
 4:             sc.state ← FAILED
 5:             pay(sc, caller, sc.W_S)
 6:         end if
 7:     end if
 8: end procedure
```

Figure 5.4: Secret whistleblow algorithm

```
    sc      smart contract identifier
    h_i     the i-th share
 1: procedure Disclose(sc, h_i)
 2:     if time() ≥ sc.t_d and time() < sc.t_term then
 3:         if sc.state = LOCKED and sc.states[i] = BID then
 4:             if hash(h_i) = sc.C_h[i] then
 5:                 sc.shares[i] ← h_i
 6:                 sc.num_disclosed += 1
 7:                 sc.states[i] ← DISCLOSED
 8:             end if
 9:         end if
10:     end if
11: end procedure
```

Figure 5.5: Share disclose algorithm

`Disclose`. After the disclosure time t_d, each shareholder $\mathcal{H}_i$ is required to publicly reveal its share h_i to enable the retrieval of the secret. The submission is successful if (i) the TL was not previously marked as `FAILED`, and (ii) `hash`(h_i) matches the commitment $sc.C_h[i]$, otherwise submission fails (as shown in Algorithm 5.5).

```
    sc          smart contract identifier
    h_i         the i-th share
 1: procedure WITHDRAW(sc, h_i)
 2:     if time() ≥ sc.t_term and sc.num_disclosed ≥ sc.k then
 3:         if sc.states[i] = DISCLOSED then
 4:             sc.states[i] ← WITHDRAWN
 5:             pay(sc, caller, R_H)
 6:         end if
 7:     end if
 8: end procedure
```

Figure 5.6: Reward withdraw algorithm

`Withdraw.` Conditionally to the outcome of the TL instance, and immediately after the termination time t_{term}, the shareholders are authorized to claim their rewards. Rewards are paid to all shareholders that correctly completed the disclosure procedure. Algorithm 5.6 illustrates how shareholders can request to withdraw their reward.

5.4 Economic model

ITYT models each participant as rational. The interest of the involved parties in the secret is represented by an economic value V associated with it. The use of an economic value permits to analyze the behavior of the parties involved in the protocol, under the assumption that each wants to maximize her reward. In this section we illustrate how to constrain the economic parameters to push rational actors to strictly adhere to the protocol, hence achieving the desired TL function. Since the participants could form alliances, we focus on the behavior of groups of users, considering a generic coalition $\mathcal{M}$ of users that team up to recover the secret ahead of disclosure time.

We provide a description of the structure, the strategy and the collective payoff users can gain as members of $\mathcal{M}$. To ensure that breaking the TL is not profitable, we develop a set of constraints guaranteeing that the attack cost is greater than the maximimun achievable payoff. To this end, we focus on the best attack scenario, that is, the one in which a coalition ideally completes its entire strategy, maximizing the payoff, with no interference by other groups of users.

Single-user constraints. Before going into details, we set some constraints to discourage misbehavior of single users. As the shareholder takes part to the protocol to get a payoff, her expected reward $R_{\mathcal{H}}$ has to be greater than the bid $B_{\mathcal{H}}$ paid to get the share. Moreover, the reward W_h paid to a share whistleblower has to be lower than $B_{\mathcal{H}}$, as each shareholder is expected to keep her share confidential before t_d. Then, to incentivize each shareholder to report misbehavior, $W_{\mathcal{S}}$ has to be greater than $R_{\mathcal{H}}$. Furthermore, since the owner is a priori able to perform the `WhistleblowSecret` action, the owner's fee to get the service $F_{\mathcal{O}}$ must be greater than the secret whistleblow bonus $W_{\mathcal{S}}$ (since the owner is considered rational). Inequality 5.1 captures, in a single expression, all the considerations made so far.

$$W_h < B_{\mathcal{H}} < R_{\mathcal{H}} < W_{\mathcal{S}} < F_{\mathcal{O}} \tag{5.1}$$

In the following, we address how a coalition of shareholders could try to break the TL behavior based on when the recostruction of the secret is performed. For the sake of clarity, Section 5.4 does not take into account the `WhistleblowShare` action, which is discussed separately in Section 5.4. Finally, Section 5.4 addresses how to constrain the fee paid by the owner.

Prevent the reconstruction of the secret

Rational shareholders will consider if it is worth breaking the TL ahead of t_d or not. To perform a successful attack, a shareholder has to team up with other $k-1$ shareholders to recover $\mathcal{S}$. In such an event, the coalition $\mathcal{M}$ would earn the most by monetizing $\mathcal{S}$ and then by performing the `WhistleblowSecret` action, getting an expected payoff of $V + W_{\mathcal{S}}$.[2] The alternative, i.e., $\mathcal{M}$ does not break the TL and submits the k shares after t_d, would lead to an expected payoff equal to the sum of the rewards. To make the second alternative more advantageous, we could require: $k \cdot R_{\mathcal{H}} > V + W_{\mathcal{S}}$. Yet, in order to earn the rewards, the coalition $\mathcal{M}$ should wait until termination time, whereas $V + W_{\mathcal{S}}$ could be collected earlier. For this reason, we use a stricter formulation of the constraint based on the cost already paid by $\mathcal{M}$ to participate in the protocol:

$$k \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}} \tag{5.2}$$

[2]The coalition $\mathcal{M}$ could setup an additional external contract with the buyer to be sure to gain both V and $W_{\mathcal{S}}$.

Constraint (5.2) addresses secrecy (time $t < t_d$), however, when the disclosure time expires, the constraints are used to promote the release of the secret. However, a coalition of $n-k+1$ shareholders could wait for $k-1$ others to submit the shares and then lead the TL instance to a stall by refusing to submit their own shares. To prevent them from waiting for a buyer of $\mathcal{S}$ and distribute the collective payoff, we introduce the constraint:

$$(n-k+1) \cdot R_{\mathcal{H}} > V \tag{5.3}$$

Here the contribution of the term $W_{\mathcal{S}}$ disappears, as the role of whistleblower is no longer admissible after t_d. Inequality 5.3 promotes the disclosure of the shares, as shareholders are rewarded only in case the TL terminates successfully (i.e., at least k shares have been submitted succesfully before termination time t_{term}).

Impact of share whistleblowing action

The `WhistleblowShare` action increases the number of available strategies. Indeed, a coalition could maximize the payoff by submitting to the contract some of the share it holds before performing the `WhistleblowSecret` action. However, as share whistleblowing implies writing to the blockchain, the public event may trigger strategies of other participants. Thus, the extra revenue $W_{er} = j^o \cdot W_h$ can be gained, where j^o stands for the optimal number of shares to be submitted that do not enable other coalitions' strategies. To address this case, we formulate a stricter version of Inequality (5.2):

$$k \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}} + W_{er} \tag{5.4}$$

The optimal number of shares a coalition $\mathcal{M}$ could submit before incurring into penalties or favor other participants, is function of the economic amounts $B_{\mathcal{H}}$, V and $W_{\mathcal{S}}$, and parameters n and k. There are two cases: (a) multiple coalitions are able to recover the secret, and (b) independently from the ratio between the economic amounts there is only one coalition holding at least k shares.

In the first case (a), when i shares have been whistleblown by coalition $\mathcal{M}$, a quiescent coalition $\mathcal{M}'$ formed by $k-i$ shareholders would gain the ability to recover the secret having paid only $(k-i) \cdot B_{\mathcal{H}}$ to get its shares. Therefore, the coalition $\mathcal{M}$ performing the submissions needs to determine the optimal number of shares j^o to be whistleblown so that $\mathcal{M}'$ does not end up having a positive payoff.

To compute it $\mathcal{M}$ can solve:

$$j_a^o = \max_i \{i|(k-i) \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}},\ i \in 1, \dots, k-1\}$$

In the second case (b), the k-shareholders coalition $\mathcal{M}$ is the only one able to recover the secret, since $k > \lfloor n/2 \rfloor$. This condition holds until $2k - n - 1$ shares are submitted to the contract. Note this number of submissions could be smaller compared to the one identified in case (a). So, $\mathcal{M}$ can compute j^o by:

$$j_b^o = \max \{2k - n - 1;\ j_a^o\}$$

In both cases, under the assumption of rational agents, the coalition $\mathcal{M}$ can submit j^o shares while still being sure that no other smaller coalition will break the TL. Inequality (5.4) ensures this strategy is associated with a negative payoff.
A graphical representation of the constraints is shown in Figure 5.7.

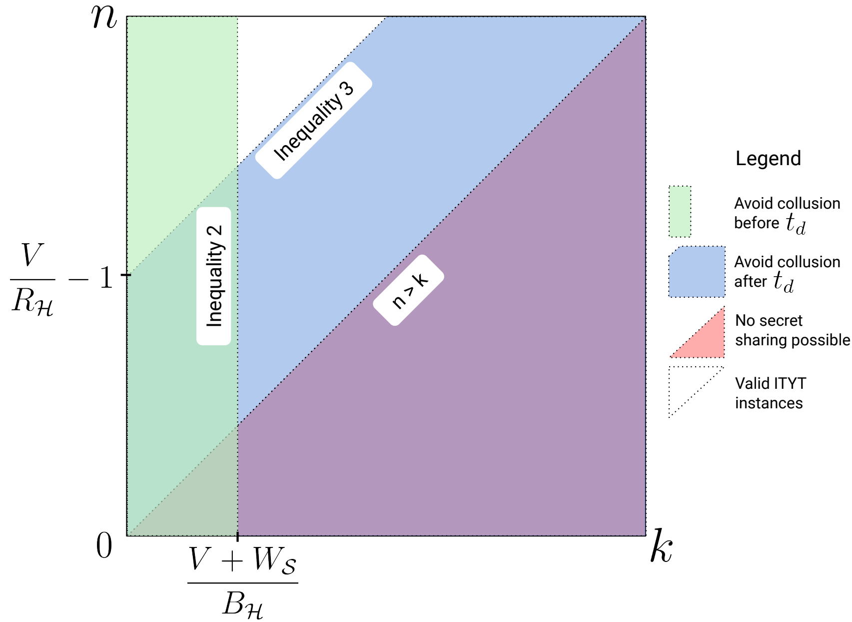


Figure 5.7: Ityt constraints representation

Rewards and bonuses

Now that we have introduced how to constrain the amounts to prevent misbehavior, we discuss some additional requirements to consider an instance of the ITYT protocol well-formed.

In the typical scenario, rational users will strictly adhere to the protocol. To accommodate for this case, the fee paid by the owner has to be enough to remunerate the shareholders:

$$F_{\mathcal{O}} \geq n \cdot (R_{\mathcal{H}} - B_{\mathcal{H}}) \tag{5.5}$$

In case of failure (i.e., the secret has been recovered before t_d), at least $k-1$ shares and the secret have been submitted to the contract. To ensure the smart contract has enough currency to pay the whistleblower bonuses, we impose the following constraint:

$$F_{\mathcal{O}} + n \cdot B_{\mathcal{H}} \geq (k-1) \cdot W_h + W_{\mathcal{S}} \tag{5.6}$$

In case less than $k-1$ shares are submitted to the contract before disclosure time, inequalities (5.5-5.6) still hold, since $B_{\mathcal{H}} > W_h$.

The constraints (5.1-5.6) must all hold for any well-formed ITYT instance. They determine the acceptance area for the economic amounts. The owner may desire to minimize the fee $F_{\mathcal{O}}$, while the shareholders may desire to maximize the profit $R_{\mathcal{H}} - B_{\mathcal{H}}$. In Table 5.1 we show three sample configurations obtained by constraint programming. We highlight that the fee paid by the owner is less than the value of the secret, which is a desirable property.

V	k	n	W_h	$B_{\mathcal{H}}$	$R_{\mathcal{H}}$	$W_{\mathcal{S}}$	$F_{\mathcal{O}}$
1	10	20	0.0031	0.1122	0.1153	0.1184	0.1216
1	15	30	0.0013	0.0717	0.0730	0.0743	0.0756
1	20	50	0.0006	0.0527	0.0533	0.0538	0.0544

Table 5.1: Sample configurations (economic amounts are expressed as ratio of V)

5.5 Implementation

In this section, we illustrate how to implement ITYT leveraging existing frameworks. Specifically, Section 5.5 details the ITYT smart contract, while Section 5.5 explains how to use sMPC to implement the share generation primitive.

Smart contract implementation

We designed ITYT as a finite state machine (FSM) within an Ethereum smart contract whose functions match the actions available to users. If successful, each action

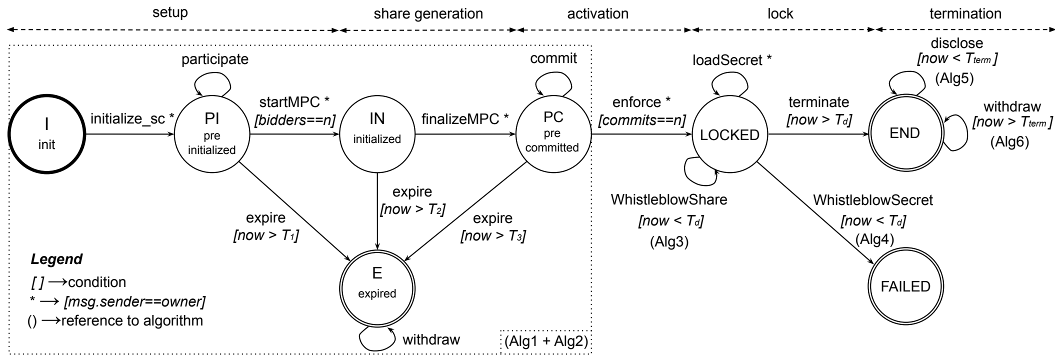


Figure 5.8: State machine representing the valid state transitions of the ITYT protocol. Each transition name maps to an action (an Ethereum smart contract function) that can be invoked by participants to update the state. Square brackets state additional conditions that must be met to consider the transition valid

entails a write to the blockchain and possibly a change of the global state. The FSM state regulates the actions available to users. Additionally, some actions are reserved to the owner. Figure 5.8 depicts a simplified version of the state machine that shows only valid transitions. It can be divided into five macro phases: (i) `setup`, in which the owner has to deploy the contract and the shareholders subscribe to it; (ii) `share generation`, that involves *off-chain* operations (i.e., not directly performed by smart contract functions) to confidentially split the secret; (iii) `activation`, in which the shareholders attest they have received their shares and give their go-ahead; (iv) `lock`, in which the shareholders keep the shares confidential; and (v) `termination`, where the secret is finally disclosed.

Setup Initially, the owner deploys a smart contract instance of ITYT to the Ethereum blockchain [185]. Then, she calls the contract initialization primitive, transfers $F_{\mathcal{O}}$ to the contract, and specifies all the parameters as detailed in Section 5.3. This advances the global state to PRE_INITIALIZED. Afterwards, each shareholder subscribes to the contract by depositing the proper amount of Ether that corresponds to the bid $B_{\mathcal{H}}$ and invoking the contract function `participate`, as part of Algorithm 5.2. After all the shareholders have deposited the bid, the owner executes the `startMPC` function, which advances the state to `INITIALIZED`.

Share generation At this point, owner and shareholders cooperate to generate the shares. Since the economic penalties have not been activated yet, a random

key $\mathcal{K}$ is used in place of the secret. From the shareholder's perspective there is no difference, as rewards and penalties are now associated with the management of $\mathcal{K}$, but from the owner perspective, the use of $\mathcal{K}$ avoids the exposure of $\mathcal{S}$ until TL activation (see Figure 5.9). Only after that, the owner will write to the contract an encrypted version (i.e., the ciphertext) of the secret, $CT = \mathcal{E}nc_{\mathcal{K}}(\mathcal{S})$. The share generation primitive returns to the owner the commitment of the key $C_{\mathcal{K}}$, along with the commitments of all the shares $\{C_1, ..., C_n\}$; while each shareholder gets her share h_i and the commitment of the key. Further details on the sMPC primitive are reported in Section 5.5.

Activation The owner calls the function `finalizeMPC` and updates the contract with the output received from the shares generation primitive, turning the global state to `PRE_COMMITTED`. Each shareholder verifies the share commitment value written to the contract, if it matches the one obtained from the sMPC (i.e., the owner did not tamper it), then she invokes the `commit` function. After all the participants have given their go-ahead, the owner executes `enforce`, activating the TL (i.e., `LOCKED` state). The economic incentives and penalties are activated as a consequence.

Lock Before disclosure time t_d, it is only possible to: (i) whistleblow a share, and (ii) whistleblow the key $\mathcal{K}$. To whistleblow a single share, a user can call `WhistleblowShare` submitting h' (Algorithm 5.3). If the commitment matches, then W_h is immediately paid to the whistleblower. The whistleblow of a share is permitted only k times, as the *k-th* submission leads to the global state `FAILED`. To whistleblow the key, a user can call the `WhistleblowSecret` function submitting $\mathcal{K}'$ (Algorithm 5.4). If its commitment matches $C_{\mathcal{K}}$, then $W_{\mathcal{S}}$ is paid to the whistleblower, and the protocol is marked as `FAILED`. When the protocol fails, the remaining amount is no longer withdrawable.

Termination If protocol is not marked as `FAILED` at time t_d, the shareholders can invoke the `disclosure` function to submit their share (Algorithm 5.5). The disclosure is successful if the share matches the corresponding commitment and it was not published before. The reward can be withdrawn by each shareholder that correctly disclosed her share after the protocol has terminated successfully. To claim

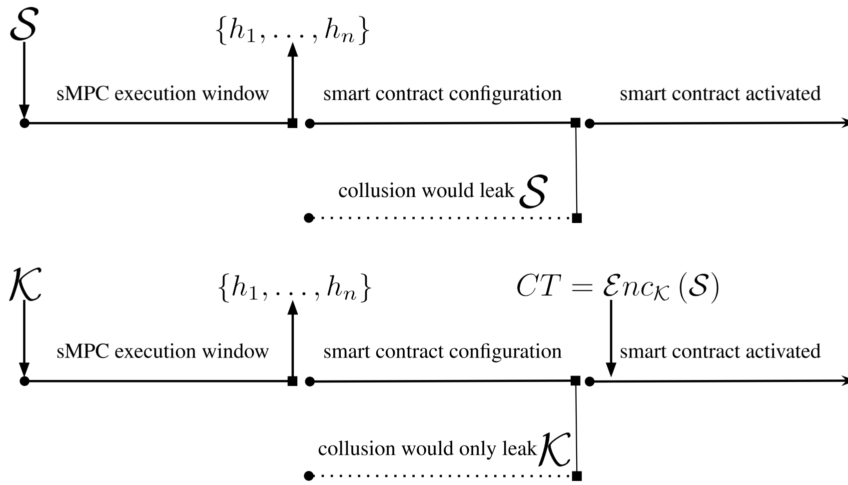


Figure 5.9: Avoid the exposition of $\mathcal{S}$ before sc activation

the reward, the user can call the function `withdraw` (Algorithm 5.6). We remark that there is no need to materialize $\mathcal{K}$ in the contract, as all the valid shares will be permanently accessible. Anyone can recover $\mathcal{K}$ just by assembling the shares (e.g., using Lagrange's interpolation in the case of secret sharing).

Share generation and distribution

In this section, we describe how to implement the `generate_shares` function by using secure Multi-Party Computation frameworks.

In the sMPC setting, each party joins the protocol as a network host. Each ITYT user is then provided with a virtual machine containing an application able to communicate via network following a pre-defined protocol. The application is implemented using FRESCO [7, 71], a FRamework for Efficient and Secure COmputation that aims to ease the development of prototypes based on secure computation. FRESCO offers several secure computation techniques, referred to as suites. Depending on the model of computation, each of them is classified into binary or arithmetic. The arithmetic suites permit to efficiently perform additions and multiplications on values that are defined over a finite field, a desirable feature for protocols like ITYT that rely on Secret Sharing. SPDZ [73] is the arithmetic suite we selected. In addition to high performance, SPDZ also ensures protection against *active adversaries* that can deviate in arbitrary ways. This grants ITYT the ability to securely open (i.e., release) partial results of the computation only to some of

the parties executing the sMPC protocol, enabling us to send each share only to the legitimate shareholder. To further increase performance, MASCOT [118] was used as SPDZ pre-processing strategy.

To execute the share generation primitive, owner and shareholders start the sMPC application. The owner inputs the random 128 bits key $\mathcal{K}$, together with the total number of shareholders n, and the reconstruction threshold k. Each shareholder, instead, submits only a random 128 bits seed. The sMPC selects k of the n seeds received, using it as the $a_1, \ldots, a_k$ coefficients of the Shamir Secret Sharing polynomial [166], while $\mathcal{K}$ is interpreted as a_0. Then, the sMPC generates n random x values of 128 bits and computes the associated y coordinates using the Horner's method, which permits to evaluate a polynomial of degree k with only k multiplications and k additions. Each i-th share is built as the concatenation of the xy coordinates $h_i = x_i || y_i$. To compute the commitments of the key and the shares, we used MiMC [6], a cryptographic primitive characterized by low multiplicative complexity implemented by FRESCO and compatible with SPDZ. Finally, dedicated output is opened to the parties: the owner gets the commitment C_i of any share generated, while each shareholder gets her share h_i, the commitment of the key $C_{\mathcal{K}}$, n and k.

5.6 Discussion

In this section, we discuss how ITYT ensures the methods to report misbehavior are not bypassable, and how ITYT mitigates denial of service (DOS) and prevents deadlocks.

Misbehavior detection

The economic penalties ITYT relies on are triggered when there is a user that is able to prove someone else's misbehavior, and this happens for example when a share is released improperly. Up to now, we have considered secure Multi-Party Computation as a mean to securely deliver dedicated output to users that take part in the ITYT protocol (i.e., to generate and distribute the shares confidentially). However, sMPC can also be used to subvert the protocol, as it enables a group of parties to jointly compute a function while keeping their input confidential. Indeed, a coalition

of shareholders could use it to recover the secret ahead of disclosure time without leaking any share nor the key, thus bypassing smart contract commitments. This is an interesting scenario as it applies to most protocols played by rational users that involve rewards and penalties (e.g., [149]).

In our setting this happens when there is a coalition that is able to recover $\mathcal{S}$ without releasing $\mathcal{K}$, thus preventing anyone to perform the `WhistleblowSecret` action. To do that, the coalition inputs to the sMPC protocol k shares along with the ciphertext CT (which is publicly available as it is written to the contract by the owner). Then, the protocol performs the reconstruction of the Shamir polynomial, recovers $\mathcal{K}$, and extracts the secret by $\mathcal{S} = \mathcal{D}ec_{\mathcal{K}}(CT)$, opening it to the parties as the only result.

There are two alternatives to prevent this attack: (i) use an encryption scheme vulnerable to the *Known Plaintext Attack*, and (ii) use an encryption scheme that is practically incompatible with the sMPC setting. As an example of (i), with the *One-Time Pad*, given two among $\{CT, \mathcal{S}, \mathcal{K}\}$ the third is implied; then, a coalition of shareholders cannot avoid to release $\mathcal{K}$ by recovering $\mathcal{S}$. The drawback of using *OneTimePad* is that $|\mathcal{S}| = |\mathcal{K}|$ by construction. This limitation can be overcome by selecting an encryption scheme that satisfies (ii).

DOS and deadlock prevention

A denial of service attack is performed by users that participate in multiple ITYT instances and refuse to deposit their bids, to commit, or to correctly execute the sMPC protocol. To mitigate these kinds of disruptions it is possible to introduce a reputation system. However, this requires to discriminate with high accuracy between misbehaving users and users that follow the protocol as intended. Therefore, we decided to include an additional step in the FSM *setup* phase, in which all participants (including the owner) are required to pay an additional small service pawn that will be returned only at activation time. It has been proven that the introduction of a small fee to access a service can mitigate many DOS attacks [128, 134].

Any other misbehavior, malfunction, or network error could result in a failure to meet the *setup* time threshold set by the owner. The deadlock, to which the protocol leads to, can be managed introducing the final state `EXPIRED` (see Figure 5.8). In this state, all the participants are allowed to withdraw their funds locked by the contract, except for the small service pawn (as the TL was never activated).

5.7 Experimental results

Our experimental analysis is organized into: (i) smart contract deployment and testing, and (ii) simulation of sMPC network protocols. The tests have been executed on a dual Intel Xeon E5 server with 256 GB memory and 512 GB SSD drive running Ubuntu 20.04 LTS. The size of the shares was set to 256 bits.

Smart contract A preliminary version of the smart contract was built with FSolidM [135], a tool that automatically generates Ethereum smart contracts code from high-level Finite State Machine representations. To deploy, test, and debug the contract generated, we relied on Brownie [107], a Python framework that allows us to create wallets, inspect transactions and automatize tests. To provide such functionalities, Brownie interacts with Ganache [2]: a personal Ethereum blockchain used to facilitate development.

The experiments mainly focused on estimating the execution cost of each ITYT instance. The cost is measured in gas, the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Table 5.2 shows, depending on the number of participants, the gas required to run each ITYT function.

Function	Gas		
	$n = 2$	$n = 5$	$n = 10$
`participate`	89545	90308	100207
`startMPC`	50399	50399	50399
`finalizeMPC`	122096	191666	307617
`commit`	56834	59597	69496
`enforce`	50425	50425	50425
`loadSecret`	51367	51367	51367
`WhistleblowShare`	125492	125676	125715
`WhistleblowSecret`	121587	121587	121587
`terminate`	29657	29657	29657
`disclose`	125492	125676	125715
`withdraw`	42634	48709	62451

Table 5.2: Gas cost for each smart contract function with $k = 2$

sMPC First, we implemented a sMPC protocol compliant with the description in Section 5.5. We refer to this version as single-phase. Each party was provided with

a different application acting either as client or server, and the network communication round trip time (RTT) was set to 10 ms. As it is illustrated in Figure 5.10a, strictly adhering to this protocol leads to a sudden performance degradation when the number of shareholders increases. This is because computing the MiMC primitive among several participants is highly affected by network latency (the parties have to exchange several messages to carry out even simple operations in the sMPC setting). To improve performance, we implemented a two-phase algorithm composed by two sMPC protocols: *Step 1* and *Step 2*.

Step 1. An n-to-n sMPC is jointly executed by all participants. The owner inputs the random 128 bits key $\mathcal{K}$, together with n and k. Each shareholder submits only a 128 bits seed. As detailed in Section 5.5, the sMPC selects the coefficients to generate the Shamir polynomial and computes the shares. Finally, the output is opened to the parties: the owner gets the polynomial coefficients of $f(x)$, while each shareholder gets her share h_i, n and k.

Step 2. A 1-to-1 sMPC is computed between the owner and each i-th shareholder. The owner inputs $f(x)$, while the shareholder inputs her $(x_i; y_i)$ coordinate. If $(x_i; y_i)$ belongs to $f(x)$, commitments are produced. The owner gets the commitment of the share C_i, while the shareholder gets the commitment of the key $C_{\mathcal{K}}$.

The difference between the single-phase and the two-phase sMPC lies in the evaluation of the MiMC primitive. Unlike the single-phase version, the two-phase solution separates the generation of the shares from the production of commitments. It follows that the first step can be carried out even in scenarios with several participants, as it is not computationally intensive, whereas the second step, which instead is computationally intensive, is always performed among two users. The comparison between the two sMPC protocol variants is shown in Figure 5.10a. More details about the two-phase execution time and memory consumption for each participant, in case of polynomial of higher degree, are illustrated in Figures 5.10b and 5.10c, respectively.

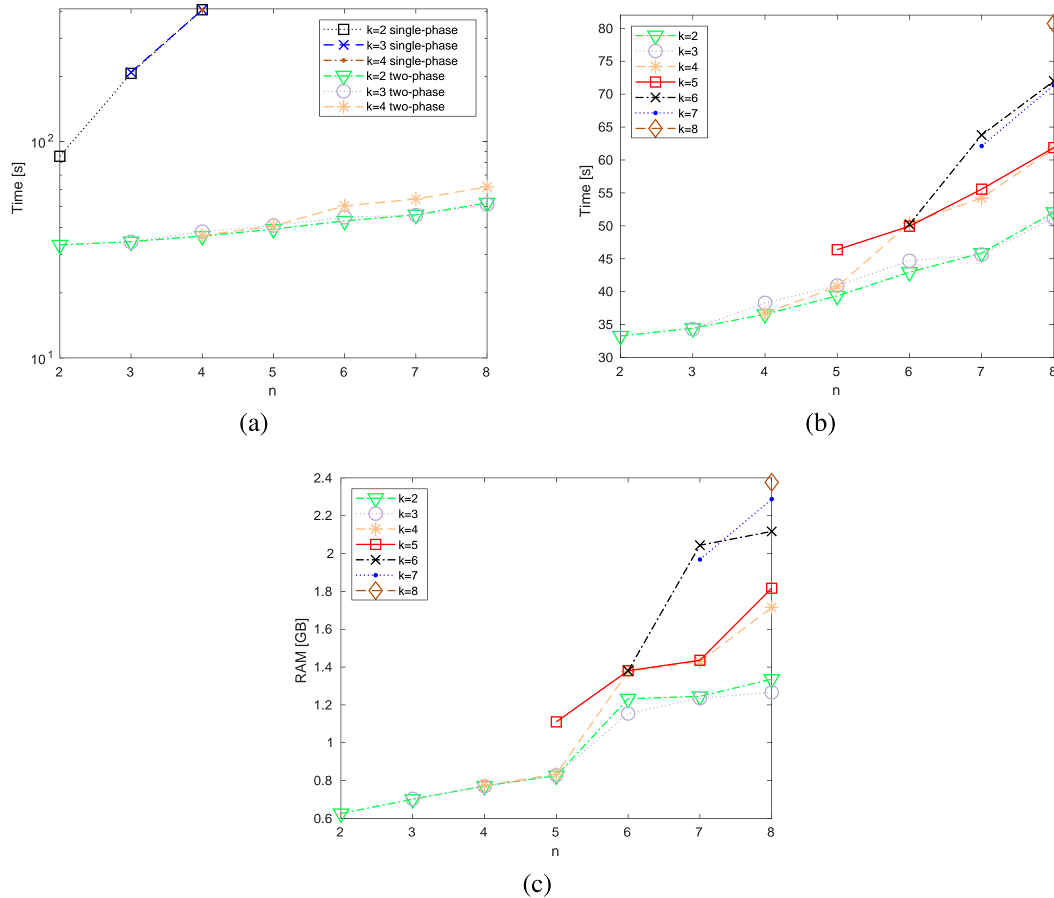


Figure 5.10: sMPC protocol time and memory consumption: (a) Single-phase vs two-phase sMPC execution, (b) Two-phase time consumption, (c) Two-phase memory consumption

5.8 Related work

The use of cryptography to solve the problem of unveiling private data at a specific time in the future was first envisioned in 1993 by Timothy May [136]. Since then, many researchers have proposed solutions to this problem. Based on the assumptions and technologies used, we can classify the proposals into four main categories.

Trust and Honesty — Chan et al. [60] and Cheon et al. [64] proposed single point of trust schemes, in which the owner encrypts the secret using public-key cryptograpyhy, and relies on a trusted time-server to release the private decryption key in the future. Rabin et al. described Time-Lapse Cryptography [145, 156] that overcomes the single point of trust assumption by splitting the single authority into

a group of users that have to cooperate to release the keys. Li et al. [124] proposed a solution that relies on Distributed Hash Tables to route the secret among peers. However, these proposals entail the peers to be honest as they do not consider the possible economic benefits that the parties would obtain by colluding.

Time-Lock Puzzles — They require the recipient to solve an inherently sequential mathematical puzzle to prove the elapse of time. Starting from Rivest et al. seminal work [159], many other puzzles have been proposed [50,66,132]. All these techniques require to run the decryption procedure for a long time, and to make assumptions on future computing power.

Smart Contracts — Similarly to our proposal, the third category leverages smart contracts [172] to replace the trusted party. Kimono [59] and Keep Network [129] rely on threshold cryptography to split the secret among participants that can earn a remuneration by keeping their shares private until disclosure time. However, they do not introduce security deposits, thus failing at preventing misbehavior. Li et al. proposal [125] overcomes some of these limitations by modeling the protocol as an extensive-form game with imperfect information [123]. Yet, as each peer is a single point of failure, and as the owner has perfect information about the shares, they require every participant to pay a security deposit that exceeds the value of the secret, limiting the applicability of the protocol. Compared to our solution, all the proposals in this category do not consider that coalitions of users can reconstruct the secret ahead of disclosure time inside an sMPC protocol without exposing the shares, thus effectively avoiding penalties and safeguarding remunerations.

Witness Encryption — This category of solutions leverages witness encryption [98], in which the sender can encrypt a message so that it can only be opened by a recipient who knows a witness to an NP relation. Liu et al. [127] showed how to construct a computational reference clock from large public computations, such as those made by the Bitcoin network, and couple it with witness encryption to achieve a TL encryption mechanism. Yet, this proposal relies on the availability of a practical witness encryption scheme.

Other Contributions — Several recent proposals address the problem of dealing with secret data on public blockchains. Enigma [191] and Hawk [120] leverage sMPC to allow multiple actors to execute an algorithm on private inputs and store the proof of correct execution on the blockchain. However, these proposals require the data holder to actively participate in the computation, thus they cannot

be used to solve the problem of data disclosure at a future point in time. *Proof of Elapsed Time* (*PoET*) is a network consensus algorithm often used in permissioned blockchains, like Hyperledger Sawtooth [1, 63], that avoids wasting computational resources by using a fair lottery system run inside a Trusted Execution Environment (TEE), such as Intel SGX. Each participant runs an algorithm in the TEE that waits for a random amount of time, thus proving the elapse of time without the need of PoW. Even if this approach resembles ITYT, as it prevents cheating on the chosen time, it is not able to store secret data. Another recent contribution, *Bitcoin Lightning Network* [149], shows how economic constraints enforced by TL primitives can be successfully integrated with blockchains. Lightning Network can be used to instantly exchange bitcoins among peers by using off-chain transactions while effectively preventing misbehavior.

5.9 Conclusions

This chapter presented I Told You Tomorrow (ITYT), a practical framework that leverages the rationality assumption to deploy Timed-Locked secrets on the blockchain. In contrast to other Time-Lock mechanisms, ITYT does not rely on a trusted third-party, neither it requires a receiving party to run a decryption algorithm until disclosure time, nor it demands guessing about future computing power. The implementation and experimental evaluation show the low cost and limited resource consumption associaced with our approach.

Further details on the open source solution described in this chapter can be found in Appendix D. There, we provide information on how to solve the economic model presented in Section 5.4 by using constraint programming, how to test the implementation of the smart contract, and finally, how to run the `generate_shares` primitive between many parties using sMPC.

Availability

The implementation source of our proposals is freely available at this URL: `https://github.com/unibg-seclab/ityt`

The work in this chapter was supported in part by the EC within the H2020 Program under project MOSAICrOWN.

Chapter 6

Conclusions and future work

In this thesis, we presented novel techniques to support the secure collection, sanitization, storage, processing and release of data.

With regard to the data collection stage, we proposed an approach to strengthen the security of mobile applications. Thanks to the additional security functions introduced in the Android Open Source Project, an application developer can compartmentalize the application to reduce access to confidential files, and to implement the least privilege principle on a per-process basis. This also permits to reduce the impact of a number of common app security vulnerabilities.

With reference to sanitization, we proposed an approach that enables the anonymization of large collections of sensors data. Based on the needs of the user, anonymization can be applied in parallel on a varying number of network nodes. While a large number of nodes greatly reduces the total sanitization time, the experimental evaluation reported very limited degradation on the quality of the results.

To support the storage and processing stages, we proposed an approach to execute point and range queries over encrypted data. The technique proposed ensures perfect indistinguishability to an attacker that gains access to the data (i.e., an attacker able to perfom a dump of the memory). It also offers protection against an attacker able to monitor the access to the memory, in a measure directly proportional to the parameter k, the number of records stored within the block. The experimental analysis showed that our solution exhibits a low overhead (in terms of overall query execution time) compared to the one that operates on plaintext data (which runs without any confidentiality guarantee), when the network latency is greater than 10

ms, and the transmission rate is at least 100 Mbps. This is a common situation when the cloud service provider is located in the same regional area of the client.

Finally, we presented an approach to deploy Time-Locked secrets on the blockchain and schedule the release of data. The proposal leverages an economic model which ensures that breaking the TL is not profitable. This is used to push rational actors to strictly adhere to the protocol, hence achieving the desired TL fuction. Our proposal operates without the need for a Trusted Third-Party.

6.1 Future Work

We conclude the thesis with a discussion of the future work that can be done in the four areas presented.

Data collection – Chapter 2 illustrated an approach to enable internal app compartmentalization. A potential obstacle to the adoption of SEApp is the usability by app developers. Although the use of macros greatly simplifies the development of the policy module, the `sepolicy.cil` file still needs to be written by hand. The availability of an Android Studio plugin that derives the policy module from a set of pre-defined `AndroidManifest.xml` flags can further facilitate the developer. The plugin could also be used to inspect the policy module and enforce part of the checks executed by the *PolicyModule-Validator*.

Data sanitization – Chapter 3 presented a scalable approach to apply sanitization to large collections of sensors data. As a future work, we plan to extend the set of criteria used to perform the cut and determine the partitions. With the availability of new cut criteria we expect to have a slight performance degradation but at the same time to reduce information loss.

Data storage & processing – Chapter 4 detailed an approach to support the execution of point and range queries over encrypted data. As a future work, we plan to implement a block rotation strategy to support the insertion, deletion and update of records. The challenge is to keep the index balanced and the client-side maps compact, without significantly increase the average number of spurious tuples pulled from the server to solve a query.

Data release – Chapter 5 presented a novel way to implement Time-Locked secrets. As a future work, we plan to devise a strategy to increase the size of the key. To do that, we are considering the use of homomorphic encryption for the share generation primitive.

Acknowledgments

The research documented in this thesis was supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo).

I would like to thank Prof. Stefano Paraboschi for the counsel given during this journey. A special thank goes to my former and present colleagues Enrico Bacis, Marco Rosa, Matthew Rossi and Gianluca Oldani, with whom I had the opportunity to grow by sharing this experience.

Bibliography

[1] Hyperledger Sawtooth. https://sawtooth.hyperledger.org, 2018.

[2] Ganache – personal blockchain for ethereum development. https://github.com/trufflesuite/ganache, 2019.

[3] A. Albrecht. sqlparse. https://github.com/andialbrecht/sqlparse, 2021.

[4] A. B. and D. Dinu and D. Khovratovich and S. Josefsson. The memory-hard Argon2 password hash and proof-of-work function. https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/, 2021.

[5] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons learned from Android security research for appified software platforms. In *IEEE S&P*, 2016.

[6] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, 2016.

[7] Alexandra Institute. FRESCO - a framework for efficient secure computation. https://github.com/aicis/fresco, 2019.

[8] J. Alwen, C. Cachin, O. Pereira, A. Sadeghi, B. Schoenmakers, A. Shelat, and I. Visconti. Summary report on rational cryptographic protocols, 2007.

[9] Android. Google Play Protect. https://www.android.com/play-protect/, 2021.

[10] Android Developers. adb install. https://developer.android.com/studio/command-line/adb#move, 2021.

[11] Android Developers. Android App Bundles. https://developer.android.com/platform/technology/app-bundle, 2021.

[12] Android Developers. Android Interface Definition Language. https://developer.android.com/guide/components/aidl, 2021.

[13] Android Developers. android:isolatedProcess. https://developer.android.com/guide/topics/manifest/service-element#isolated, 2021.

[14] Android Developers. Bound services overview. https://developer.android.com/guide/components/bound-services#Creating, 2021.

[15] Android Developers. isolated_app.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/isolated_app.te, 2021.

[16] Android Open Source Project. Enable per-user isolation for normal apps. https://android.googlesource.com/platform/external/sepolicy/+/a833763ba04147e840fd054b613f759395bada35, 2014.

[17] Android Open Source Project. SELinux for Android 8.0. https://source.android.com/security/selinux/images/SELinux_Treble.pdf, 2017.

[18] Android Open Source Project. Android 9 release notes. https://source.android.com/setup/start/p-release-notes#per-app_selinux_sandbox, 2018.

[19] Android Open Source Project. ActivityManagerService. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/am/ActivityManagerService.java, 2021.

[20] Android Open Source Project. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb, 2021.

[21] Android Open Source Project. Android Permissions. https://developer.android.com/guide/topics/permissions/overview, 2021.

[22] Android Open Source Project. Android Runtime. https://developer.android.com/guide/platform#art, 2021.

[23] Android Open Source Project. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2021.

[24] Android Open Source Project. Binder. https://developer.android.com/reference/android/os/Binder, 2021.

[25] Android Open Source Project. Implementing SELinux. https://source.android.com/security/selinux/implement, 2021.

[26] Android Open Source Project. init. https://android.googlesource.com/platform/system/core/+/refs/heads/master/init/main.cpp, 2021.

[27] Android Open Source Project. installd. https://android.googlesource.com/platform/frameworks/native/+/refs/heads/master/cmds/installd/, 2021.

[28] Android Open Source Project. Intent and intent filters. https://developer.android.com/guide/components/intents-filters, 2021.

[29] Android Open Source Project. Mounting partitions early. https://source.android.com/devices/architecture/kernel/mounting-partitions-early, 2021.

[30] Android Open Source Project. PackageManagerService. https://android.googlesource.com/platform/frameworks/

base/+/refs/heads/master/services/core/java/com/android/server/pm/PackageManagerService.java, 2021.

[31] Android Open Source Project. Policy compatibility. https://source.android.com/security/selinux/compatibility, 2021.

[32] Android Open Source Project. restorecond service. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/restorecond/restorecond.service, 2021.

[33] Android Open Source Project. secilc. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/secilc/, 2021.

[34] Android Open Source Project. SELinuxMMAC. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/SELinuxMMAC.java, 2021.

[35] Android Open Source Project. untrusted_app_all.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/untrusted_app_all.te, 2021.

[36] Android Open Source Project. Zygote. https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/os/Zygote.java, 2021.

[37] Apache Spark. Apache Spark. https://spark.apache.org/, 2021.

[38] Ars Technica. The Android 11 interview. https://arstechnica.com/gadgets/2020/09/the-android-11-interview-googlers-answer-our-burning-questions/, 2020.

[39] G. Asharov, R. Canetti, and C. Hazay. Toward a game theoretic view of secure computation. *IACR*, 2011.

[40] G. Asharov, I. Komargodski, W. Lin, E. Peserico, and E. Shi. Optimal oblivious parallel ram. *IACR*, 2020.

[41] F. Ashkouti, K. Khamforoosh, and A. Sheikhahmadi. DI-Mondrian: Distributed improved Mondrian for satisfaction of the ℓ-diversity privacy model using Apache Spark. *Information Sciences*, 2021.

[42] E. Bacis, S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Livraga, S. Paraboschi, M. Rosa, and P. Samarati. Multi-provider secure processing of sensors data. In *IEEE PerCom (PerCom Workshops)*, 2019.

[43] E. Bacis, D. Facchinetti, M. Guarnieri, M. Rosa, M. Rossi, and S. Paraboschi. I told you tomorrow: Practical time-locked secrets using smart contracts. In *ARES*, 2021.

[44] E. Bacis, S. Mutti, and S. Paraboschi. AppPolicyModules: Mandatory access control for third-party apps. In *ASIACCS*, 2015.

[45] E. Bacis, S. Mutti, and S. Paraboschi. Policy specialization to support domain isolation. In *SafeConfig*, 2015.

[46] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *CCS*, 2016.

[47] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *ACSAC*, 2014.

[48] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. V. Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *USENIX Security*, 2015.

[49] M. Bayer. SQLAlchemy. `https://pypi.org/project/SQLAlchemy/`, 2021.

[50] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ITCS*, 2016.

[51] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2009.

[52] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 2015.

[53] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 2015.

[54] G. Branwen. Time-lock encryption. `https://www.gwern.net/Self-decrypting-files`, 2018.

[55] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[56] S. Bugiel, S. Heuser, and A. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[57] C. L. Halbert and K. Tretyakov. `intervaltree`. `https://pypi.org/project/intervaltree/`, 2021.

[58] P. Caballero-Gil, C. Hernández-Goya, and C. Bruno-Castañeda. A rational approach to cryptographic protocols. *CoRR*, 2010.

[59] F. M. Celebi, P. Fletcher-Hill, G. Kaemmer, and D. Que. Kimono time capsule. `https://kimono.network`, 2018.

[60] A. Chan and I. Blake. Scalable, server-passive, user-anonymous timed release cryptography. In *ICDCS*, 2005.

[61] H. Chen, I. Chillotti, and L. Ren. Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[62] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid policies: Combining MAC and DAC in Android. In *ACSAC*, 2017.

[63] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi. On security analysis of Proof-of-Elapsed-Time (PoET). In *SSS*, 2017.

[64] J. Cheon, N. Hopper, Y. Kim, and I. Osipkov. Timed-release and key-insulated public key encryption. In *FC*, 2006.

[65] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-Anonymity. In *Secure Data Management in Decentralized Systems*. 2007.

[66] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In *EUROCRYPT*, 2018.

[67] A. Conner-Simons. Programmers solve MIT's 20-year-old cryptographic puzzle. `https://www.csail.mit.edu/news/programmers-solve-mits-20-year-old-cryptographic-puzzle`, 2019.

[68] D. Cynthia. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*, 2006.

[69] D. J. Bernstein. The poly1305-aes message-authentication code. In *International workshop on fast software encryption*, 2005.

[70] D. J. Bernstein. Extending the salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop*, 2011.

[71] I. Damgård, K. Damgård, K. Nielsen, P. Nordholt, and T. Toft. Confidential benchmarking based on multiparty computation. In *FC*, 2017.

[72] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[73] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *ESORICS*, 2013.

[74] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of ACM CCS*, 2003.

[75] J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. In *USENIX*, 2014.

[76] A. Dawoud and S. Bugiel. DroidCap: OS support for capability-based permissions in Android. In *NDSS*, 2019.

[77] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization. In *PerCom 2021*, 2021.

[78] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical private range search revisited. In *Proc. of ACM SIGMOD*, 2016.

[79] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. Gunter. Free for all! Assessing user data exposure to advertising libraries on Android. In *NDSS*, 2016.

[80] S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Artifact: Scalable distributed data anonymization. In *IEEE PerCom (PerCom Workshops)*, 2021.

[81] S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. k-flat secure indexing for encrypted databases. In *Under submission*, 2021.

[82] S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Multi-dimensional indexes for point and range queries on outsourced encrypted data. In *IEEE GlOBECOM*, 2021.

[83] S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization. In *IEEE PerCom (PerCom Workshops)*, 2021.

[84] M. Diamantaris, E. Papadopoulos, E. Markatos, S. Ioannidis, and J. Polakis. REAPER: Real-time app analysis for augmenting the Android permission system. In *CODASPY*, 2019.

[85] Docker inc. Docker. `https://www.docker.com/`, 2021.

[86] Docker inc. `Docker-compose`. `https://docs.docker.com/compose/`, 2021.

[87] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *SIGSAC*, 2017.

[88] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1993.

[89] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security*, 2011.

[90] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE S&P Magazine*, 2009.

[91] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions demystified. In *CCS*, 2011.

[92] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[93] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? The impact of copy paste on Android application security. In *IEEE S&P*, 2017.

[94] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the security and engineering implications of finer-grained access controls for Android developers and users. In *DIMVA*, 2015.

[95] Free Software Foundation. GNU M4. `https://www.gnu.org/savannah-checkouts/gnu/m4/manual/m4-1.4.18/index.html`, 2016.

[96] G. Craig, and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. 2011.

[97] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE S&P*, 2006.

[98] S. Garg, C. Gentry, A. Sahai, and B. Waters. Witness encryption and its applications. In *STOC*, 2013.

[99] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM symposium on Theory of computing*, 2009.

[100] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM symposium on Theory of computing*, 1987.

[101] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[102] Google. Capsicum object-capabilities on Linux. `https://github.com/google/capsicum-linux`, 2017.

[103] Google Play Protect. Android app vulnerability classes: A whirlwind overview of common security and privacy problems in Android apps. `https://static.googleusercontent.com/media/www.google.com/en//about/appsecurity/play-rewards/Android_app_vulnerability_classes.pdf`, 2021.

[104] Google Play Store. Android top apps. `https://play.google.com/store/apps/top`, 2021.

[105] A. Groce, J. Katz, A. Thiruvengadam, and V. Zikas. Byzantine agreement with a rational adversary. In *ICALP*, 2012.

[106] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of ACM SIGMOD*, 2002.

[107] B. Hauser. Introducing Brownie: A python framework for testing, deploying and interacting with ethereum smart contracts. `https://medium.com/hyperlink-technology/introducing-brownie-a763859409ca`, 2019.

[108] C. Hazay and Y. Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries, 2010.

[109] S. Heuser, A. Nadkarni, W. Enck, and A. Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security*, 2014.

[110] R. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *SACMAT*, 2010.

[111] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, 2012.

[112] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proc. of Int. Conf. on Very large data bases-V. 30*, 2004.

[113] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. In *CCS*, 2017.

[114] Intel. Software Guard Extensions. `https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html`, 2021.

[115] `iproute2`. Ubuntu man pages. `https://launchpad.net/ubuntu/focal/+package/iproute2`, 2021.

[116] J. Vander Stoep. ioctl command whitelisting in SELinux. `http://kernsec.org/files/lss2015/vanderstoep.pdf`, 2015.

[117] Kaggle. `Acquire Valued Shoppers Challenge, transactions dataset`. `https://www.kaggle.com/c/acquire-valued-shoppers-challenge/data?select=transactions.csv.gz`, 2014.

[118] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *SIGSAC*, 2016.

[119] M. Keller, V. Pastro, and D. Rotaru. Overdrive: making SPDZ great again. In *EUROCRYPT*, 2018.

[120] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P*, 2016.

[121] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proc. of ICDE*, 2006.

[122] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proc. of ICDE*, 2006.

[123] K. Leyton-Brown and Y. Shoham. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis lectures on artificial intelligence and machine learning*, 2008.

[124] C. Li and B. Palanisamy. Timed-release of self-emerging data using distributed hash tables. In *ICDCS*, 2017.

[125] C. Li and B. Palanisamy. Decentralized release of self-emerging data using smart contracts. In *SRDS*, 2018.

[126] Libsodium. Libsodium Library. `https://github.com/jedisct1/libsodium`, 2021.

[127] J. Liu, T. Jager, S. Kakvi, and B. Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 2018.

[128] G. Loukas and G. Öke. Protection against denial of service attacks: A survey. *The Computer Journal*, 2010.

[129] M. Luongo and C. Pon. The Keep network: A privacy layer for public blockchains. `https://keep.network/whitepaper`, 2019.

[130] A. Machanavajjhala, J. Gehrke, and D. Kifer. ℓ-diversity: Privacy beyond k-anonymity. In *Proc. of ICDE*, 2006.

[131] K. MacMillan, C. Case, J. Brindle, and C. Sellers. SELinux Common Intermediate Language motivation and design. `https://github.com/SELinuxProject/cil/wiki`, 2020.

[132] M. Mahmoody, T. Moran, and S. Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO*, 2011.

[133] M. Mahmoody, T. Moran, and S. Vadhan. Publicly verifiable proofs of sequential work. In *ITCS*, 2013.

[134] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating distributed denial of service attacks with dynamic resource pricing. In *ACSAC*, 2001.

[135] A. Mavridou and A. Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. *ArXiv*, 2017.

[136] T. May. Timed-release crypto. `http://cypherpunks.venona.com/date/1993/02/msg00129.html`, 1993.

[137] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich. The Android platform security model. *arXiv*, 2019.

[138] A. McCurdy. Redis-py. `https://pypi.org/project/redis/`, 2021.

[139] Microsoft. Microsoft SEAL, 2021. https://www.microsoft.com/en-us/research/project/microsoft-seal.

[140] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *CCS*, 2012.

[141] M. Nojoumian, A. Golchubian, L. Njilla, K. Kwiat, and C. Kamhoua. Incentivizing blockchain miners to avoid dishonest mining strategies by a reputation-based paradigm. In *ICIC*, 2019.

[142] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC*, 2009.

[143] R. Ostrovsky. Efficient computation on oblivious rams. In *ACM Symposium on Theory of Computing*, 1990.

[144] pandas-dev. pandas. `https://pandas.pydata.org/`, 2021.

[145] D. Parkes, M. Rabin, S. Shieber, and C. Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. *ECRA*, 2008.

[146] P. Pearce, A. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS*, 2012.

[147] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad. Searchable symmetric encryption: Designs and challenges. 2017.

[148] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad. Searchable symmetric encryption: Designs and challenges. *ACM CSUR*, 2017.

[149] J. Poon and T. Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. `https://www.bitcoinlightning.com/wp-content/uploads/2018/03/lightning-network-paper.pdf`, 2016.

[150] A. P. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[151] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP*, 2011.

[152] A. O. S. Project. PackageParser. `https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageParser.java`, 2021.

[153] Python Software Foundation. `multiprocessing` - Process-based parallelism. `https://docs.python.org/3/library/multiprocessing.html`, 2021.

[154] Python Software Foundation. `pickle` - Python object serialization. `https://docs.python.org/3/library/pickle.html`, 2021.

[155] M. Rabin. How to exchange secrets with oblivious transfer. *IACR*, 2005.

[156] M. Rabin and C. Thorpe. Time-lapse cryptography. Technical report, 2006.

[157] Redis Ltd. Redis. `https://redis.io/`, 2021.

[158] R. Rivest. Description of the LCS35 time capsule crypto-puzzle. `https://people.csail.mit.edu/rivest/lcs35-puzzle-description`, 1999.

[159] R. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. Technical report, 1996.

[160] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi. Seapp: Bringing mandatory access control to android apps. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[161] S. Ruggles et al. IPUMS USA: Version 10.0 [dataset], 2020. https://doi.org/10.18128/D010.V10.0.

[162] S. Ruggles, S. Flood, R. Goeken, J. Grover, E. Meyer, J. Pacas, and M. Sobek. IPUMS USA: Version 10.0 [dataset], 2020.

[163] P. Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 2001.

[164] P. Samarati and S. De Capitani di Vimercati. Cloud security: Issues and concerns. In *Encyclopedia on cloud computing*. 2016.

[165] R. Sandhu and P. Samarati. Authentication, access control, and audit. *CSUR*, 1996.

[166] A. Shamir. How to share a secret. *Communications of the ACM*, 1979.

[167] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS*, 2006.

[168] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, 2013.

[169] SQLite Consortium. SQLite. `https://www.sqlite.org/index.html`, 2021.

[170] Statista. Most popular installed ad network software development kits (SDKs) across Android apps worldwide as of September 2020. `https://www.statista.com/statistics/1035623/leading-mobile-app-ad-network-sdks-android/`, 2020.

[171] E. Stefanov, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Computer & communications security*, 2013.

[172] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[173] T. Cornebize. `pyroaring`. `https://pypi.org/project/pyroaring/`, 2021.

[174] `tc`. Ubuntu man pages. `https://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html`, 2021.

[175] The Apache Software Foundation. Apache Arrow. `https://arrow.apache.org/`, 2021.

[176] The PostgreSQL Global Development Group. PostgreSQL. `https://www.postgresql.org/`, 2021.

[177] The SELinux Project. Type Enforcement. `https://selinuxproject.org/page/NB_TE`, 2015.

[178] The SELinux Project. libselinux. `https://github.com/SELinuxProject/selinux/tree/master/libselinux`, 2021.

[179] Unity. Unity Ads. `https://unity.com/solutions/unity-ads`, 2021.

[180] U.S. Bureau of the Census. `Public Use Microdata Sample. Individual dataset of all US. 1-Year version of ACS 2018.` `https://www2.census.gov/programs-surveys/acs/data/pums/2018/1-Year`, 2019.

[181] U.S. Bureau of the Census. `Public Use Microdata Sample. Individual dataset of all US. 1-Year version of ACS 2019.` `https://www2.census.gov/programs-surveys/acs/data/pums/2019/1-Year`, 2019.

[182] H. Van Tran, T. Allard, L. d'Orazio, and A. El Abbadi. FRESQUE: A scalable ingestion framework for secure range query processing on clouds. In *Proc. of EDBT*, 2021.

[183] M. von Maltitz and G. Carle. A performance and resource consumption assessment of secret sharing based secure multiparty computation. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2018.

[184] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li. Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *Proc. of ACM ASIACCS*, 2014.

[185] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[186] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman. Linux Security Module framework. In *Ottawa Linux Symposium*, 2002.

[187] Z. Xiao, A. Amit, and D. Wenliang. AFrame: Isolating advertisements from mobile applications in Android. In *ACSAC*, 2013.

[188] J. Xu, W. Wang, J. Pei, X. Wang, B. Shi, and A.-C. Fu. Utility-based anonymization for privacy preservation with less information loss. *ACM SIGKDD Explorations Newsletter*, 2006.

[189] A. Yao. Protocols for secure computations. In *SFCS*, 1982.

[190] Zerodium. Zerodium - The leading exploit acquisition platform. `https://zerodium.com`, 2021.

[191] G. Zyskind, O. Nathan, and A. Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv*, 2015.

Appendix A

SEApp documentation

This chapter gives a technical demonstration of the security measures introduced by SEApp. The description is based on the showcase app presented in Section 2.3. We show that: (1) the showcase app can operate without a policy module; in this mode, its vulnerabilities can be exploited; (2) the showcase app can also operate with the policy module listed in Appendix A.4 and use the services offered by SEApp; in this mode, the internal vulnerabilities are no longer exploitable.

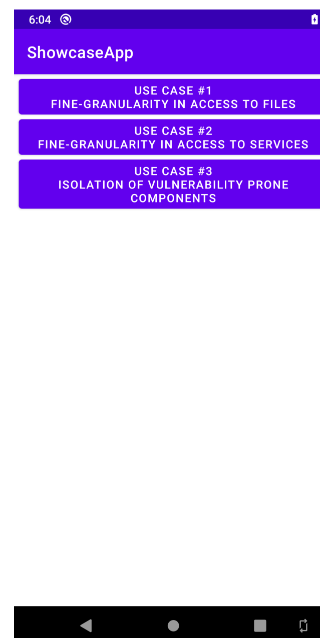


Figure A.1: MainActivity

The showcase app has a minimal structure. Its entry point is the *MainActivity* (Figure A.1), which is associated with the *core_logic* process. From the *MainActivity* it is possible to send a *startActivity* intent to one among *UseCase1Activity*, *UseCase2Activity* and *UseCase3Activity*; the entry points of use cases 1, 2 and 3, respectively. For each entry point *Zygote* starts a dedicated process and, according to the content of the `seapp_contexts` (in Listing A.1), assigns its specific domain (`user_logic_d` to UC#1, `ads_d` to UC#2, `media_d` to UC#3). A dedicated description of each use case follows.

A.1 Use case 1

In this use case we demonstrate how an app could benefit from the fine-granularity access to files. In particular, we show how the *UseCase1Activity*, suffering of a path traversal vulnerability, cannot be exploited when the app is associated with a properly configured policy module. According to the Google Play Protect report on common application vulnerabilities [103], unsanitized path names that lead to path traversal are a primary source of problems in applications.

UseCase1Activity is quite simple: it displays the content of a file given its relative path through an intent (Figure A.2a). While this may be fine when the intent comes from trusted components, the activity supports also implicit intents coming from untrusted sources. This makes the vulnerability easily exploitable by an attacker targeting the confidential files written by the *core_logic* components.

In our setup phase, we leverage *MainActivity* to create an internal directory structure by using the `android.os.File` abstraction, which sets file and directory context upon its creation (see Section 2.6). Two directories are created: `user/` and `confidential/`; inside both folders a file `data` is saved.

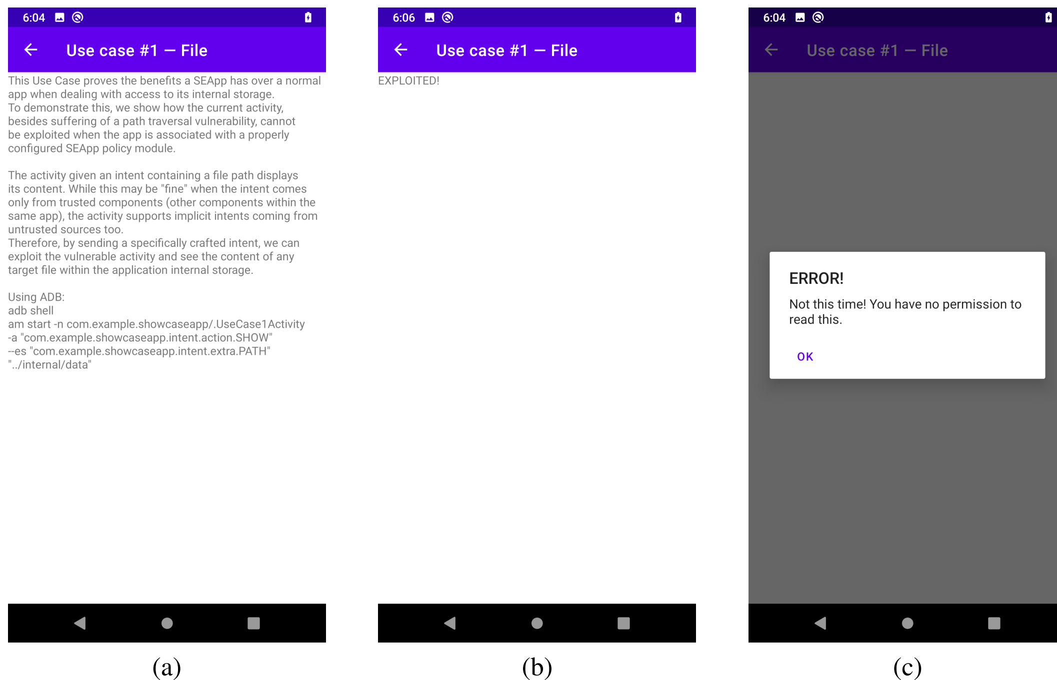
To test this use case, we first start *UseCase1Activity*, then we send an intent to "confuse" *UseCase1Activity* into showing us the content of `confidential/data`. This can be done via ADB with the command:

```
adb shell am start
-n com.example.showcaseapp/.UseCase1Activity
-a "com.example.showcaseapp.intent.action.SHOW"
--es "com.example.showcaseapp.intent.extra.PATH" "../confidential
 /data"
```

When the policy module is not availableg, all app internal files are flagged with `app_data_file` and every app component executes within the `untrusted_app` domain, which holds read access to `app_data_file`. As a consequence the vulnerability is successfully exploited and *UseCase1Activity* shows the content of the `confidential/data` file (Figure A.2b).

Instead, when the policy module is available, the file `confidential/data` is flagged with `confidential_t`, as indicated in line 2 in `file_contexts` (see Listing A.2). Since no permission is granted on `confidential_t` in the `sepolicy.cil` to `user_logic_d`, any access to the file `confidential/data` by *UseCase1Activity* is blocked by SELinux (Figure A.2c). The following de-

nial is written to the system log: *denied* `search` *to* `user_logic_d` *domain on* `confidential_t` *type* (Figure A.3). The `confidential` directory cannot then be accessed despite the exploitation of the path traversal vulnerability.



(a) (b) (c)

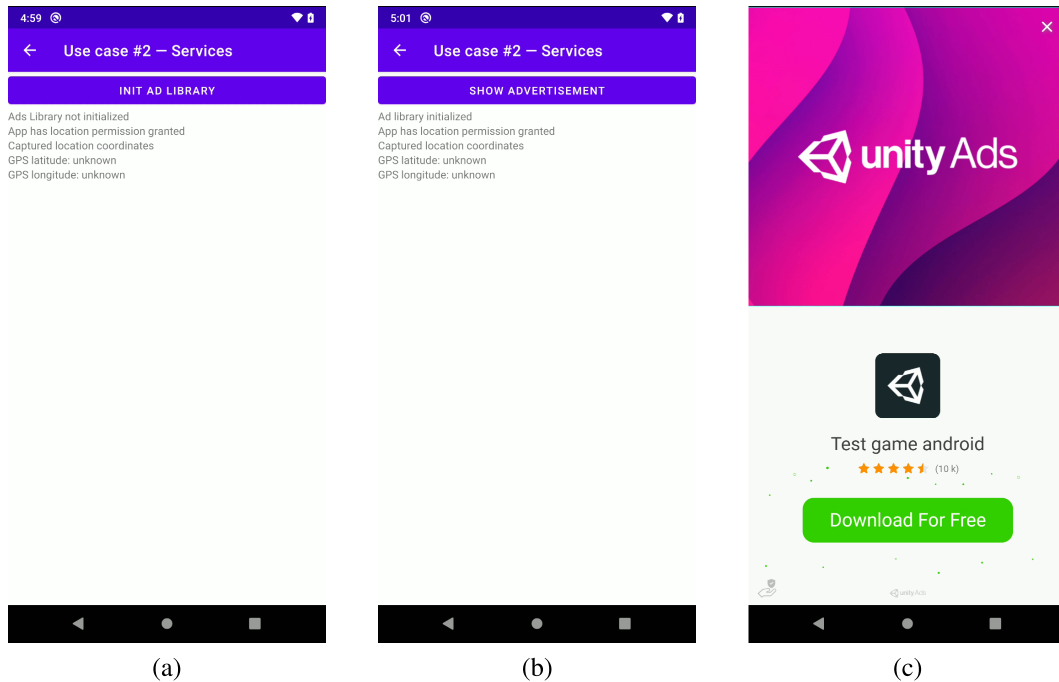
Figure A.2: Use case 1 views

```
01-16 16:39:50.563  4201  4201 W eapp:user_logic: type=1400 audit(0.0:29): avc: denied { search } for name="confidential" dev="sda13" i
no=1968083 scontext=u:r:com_example_showcaseapp.user_logic_d:s0:c153,c256,c512,c768 tcontext=u:object_r:com_example_showcaseapp.confide
ntial_t:s0:c153,c256,c512,c768 tclass=dir permissive=0 app=com.example.showcaseapp
```

Figure A.3: Use case 1 logcat

A.2 Use case 2

In this use case we show how to confine an Ad library into an ad-hoc process, with guarantees that it cannot abuse the access privileges granted to the whole application sandbox by the user. To do that, we deliberately inject, in the same process the library is executed, a malicious component (which is directly invoked by the library) that tries to capture the location when the permission *ACCESS_FINE_LOCATION* is granted to the app. The Ad library used is Unity Ads [179], which according to [170] in 2020 was used by 11% of apps that show ads.



(a) (b) (c)

Figure A.4: Use case 2 views

In this case the library is invoked by *UseCase2Activity* (Figure A.4a), and according to line 3 of the `seapp_contexts`, both the activity and the components created by the library are executed by *Zygote* in a process labeled with `ads_d`. To interact with the Ad library, *UseCase2Activity* instances a `UnityAdsListener`. After the Ad initialization (including the registration of the listener) and displaying the Ad to the user (Figures A.4b-c), the Ad framework invokes the listener callback method `onUnityAdsFinish`, which executes the malicious routine `captureLocation`. The routine probes the app permissions; if *ACCESS_FINE_LOCATION* is granted to the app, the malicious component retrieves through the *servicemanager* a handle to the `LocationManager`, and registers to it an asynchronous listener to capture GPS location (Figure A.5).

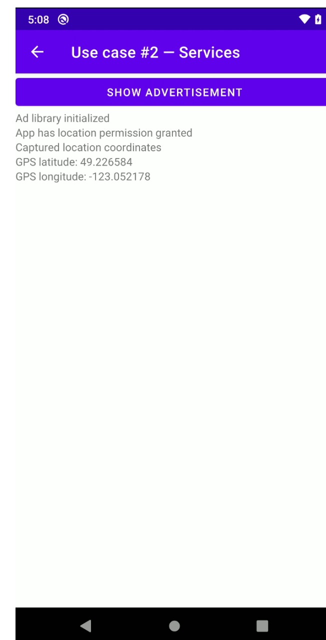


Figure A.5: UC2 exploit

We show that when the policy module is enforced by SEApp, the malicious component cannot access the GPS coordinates. This is because the component is executed in the same process of the library, which is labeled with `ads_d`. If we look at the `sepolicy.cil` (lines 43-50), `ads_d` is not granted access to the SELinux type `location_service`, so the malicious routine cannot retrieve and therefore connect to the *location_service*. The following denial is written to the system log: *denied* `find` *on* `location_service` *to the* `ads_d` *domain* (Figure A.6). As a result, the malicious component is terminated by the *ActivityTaskManager* (Figure A.7).

The Ad library was included in the app as an *.aar* archive. To confine it, no modification was necessary, only the use of `AndroidManifest.xml` and `sepolicy.cil` was required.

```
01-15 17:01:57.788   824   824 I /vendor/bin/hw/android.hardware.health@2.0-service.wahoo: SRAM data: 3699000
01-15 17:01:58.325  6735  6735 D showcaseapp:ads_d: Capturing location - setup location listener
01-15 17:01:58.325  6735  6735 D showcaseapp:ads_d: Capturing location - trying to get a location manager handle
01-15 17:01:58.325   613   613 E SELinux : avc:  denied  { find } for service=location pid=6735 uid=10148 scontext=u:r:com_example_showcaseap
p.ads_d:s0:c148,c256,c512,c768 tcontext=u:object_r:location_service:s0 tclass=service_manager permissive=0
```

Figure A.6: Use case 2 logcat - SELinux denial

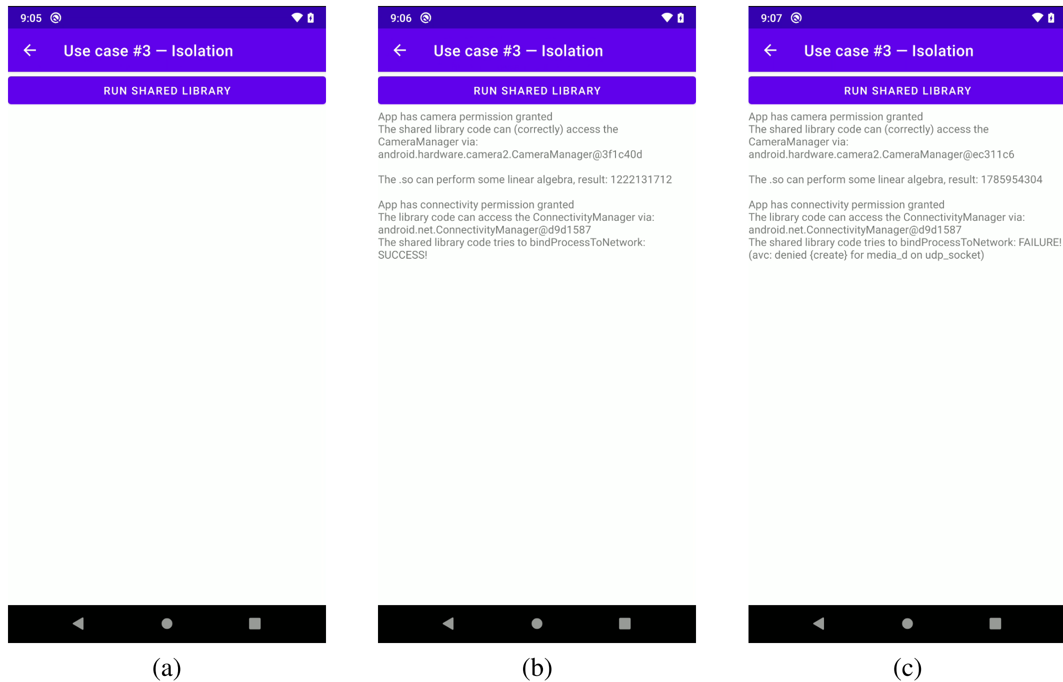
```
01-15 17:01:58.334  1276  1498 W ActivityTaskManager:   Force finishing activity com.example.showcaseapp/.UseCase2Activity
01-15 17:01:58.334  1276  6894 I DropBoxManagerService: add tag=data_app_crash isTagEnabled=true flags=0x2
01-15 17:01:58.342  6735  6735 I Process : Sending signal. PID: 6735 SIG: 9
01-15 17:01:58.380  1276  1501 W InputDispatcher: channel '135415b com.example.showcaseapp/com.unity3d.services.ads.adunit.AdUnitActivity (se
rver)' ~ Consumer closed input channel or an error occurred.  events=0x9
01-15 17:01:58.380  1276  1501 E InputDispatcher: channel '135415b com.example.showcaseapp/com.unity3d.services.ads.adunit.AdUnitActivity (se
rver)' ~ Channel is unrecoverably broken and will be disposed!
01-15 17:01:58.384  1276  1501 W InputDispatcher: channel 'ebf97cb com.example.showcaseapp/com.example.showcaseapp.UseCase2Activity (server)'
 ~ Consumer closed input channel or an error occurred.  events=0x9
01-15 17:01:58.385  1276  1501 E InputDispatcher: channel 'ebf97cb com.example.showcaseapp/com.example.showcaseapp.UseCase2Activity (server)'
 ~ Channel is unrecoverably broken and will be disposed!
```

Figure A.7: Use case 2 logcat - Activity termination

A.3 Use case 3

In this use case we show how to confine a set of components, which rely on a high performance native library written in C to perform some task. Our goal is to demonstrate that the context running the native library code is prevented to access the network, even when the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app sandbox.

The native library is invoked by *UseCase3Activity* (Figure A.8a), which, according to line 4 in the `seapp_contexts`, is executed in a process labeled with `media_d` by *Zygote*. The call to the library is performed via JNI. Its job is to connect to the *camera_service* and take a picture. Since the app is granted the *CAMERA*



(a) (b) (c)

Figure A.8: Use case 3 views

permission, the native library code (legitimately, line 53 in the `sepolicy.cil`) connects to the `CameraManager`.

Since the native library performs image processing, we do not want it to access the network. However, the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app, as they are required by the Ads framework. Thus, when the policy module is not available, the native library can connect to the `ConnectivityManager` and successfully bind the current process to the network (Figure A.8b). Instead, when the policy module is enforced by SEApp, since `media_d` was granted only the basic app permissions (line 11 in `sepolicy.cil`), the connection to the network is forbidden (Figure A.8c). This happens because binding a process to the network is associated with opening a network socket, an operation not permitted by SELinux without the required permissions. The following denial is written to the system log: *denied* `create` *on* `udp_socket` *to* `media_d` *domain* (Figure A.9).

```
01-15 21:30:04.752 11462 11462 W owcaseapp:media: type=1400 audit(0.0:111): avc: denied { create } for scontext=u:r:com_example_showcaseapp.me
dia_d:s0:c152,c256,c512,c768 tcontext=u:r:com_example_showcaseapp.media_d:s0:c152,c256,c512,c768 tclass=udp_socket permissive=0 app=com.exampl
e.showcaseapp
```

Figure A.9: Use case 3 logcat - SELinux denial

This use case, besides showing how SEApp confines a native library, also demonstrates the power and simplicity of the macro, as adding the line `(call md_netdomain (media_d))` to the policy module grants to `media_d` the needed permissions to access the network. The application developer is thus not required to know or understand the internal SELinux policy in order to leverage this functionality.

The isolation properties introduced by SEApp applies also to other common security problems presented in [103]. Just to mention one, SEApp can mitigate the impact of incorrect sandboxing of a scripting language.

A.4 Showcase app policy module

Here are reported the showcase app policy module files.

```
1 user=_app seinfo=showcase_app domain=com_example_showcaseapp.
  core_logic_d name=com.example.showcaseapp:core_logic levelFrom=
  all
2 user=_app seinfo=showcase_app domain=com_example_showcaseapp.
  user_logic_d name=com.example.showcaseapp:user_logic levelFrom=
  all
3 user=_app seinfo=showcase_app domain=com_example_showcaseapp.
  ads_d name=com.example.showcaseapp levelFrom=all
4 user=_app seinfo=showcase_app domain=com_example_showcaseapp.
  media_d name=com.example.showcaseapp:media levelFrom=all
```

Listing A.1: showcase app seapp_contexts

```
1 .*                   u:object_r:app_data_file:s0
2 files/confidential   u:object_r:com_example_showcaseapp.
  confidential_t:s0
3 files/ads_cache      u:object_r:com_example_showcaseapp.ads_t:s0
```

Listing A.2: showcase app file_contexts

```
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <policy><signer signature="SIGNATURE">
3 <package name="com.example.showcaseapp">
4 <seinfo value="showcase_app"/></package>
5 </signer></policy>
```

Listing A.3: showcase app mac_permissions.xml

```
1 (block com_example_showcaseapp
2 ; creation of domain types
3 (type core_logic_d)
4 (call md_untrusteddomain (core_logic_d))
5 (type user_logic_d)
6 (call md_appdomain (user_logic_d))
7 (type ads_d)
8 (call md_appdomain (ads_d))
9 (call md_netdomain (ads_d))
10 (type media_d)
11 (call md_appdomain (media_d))
12 (typeattribute domains)
13 (typeattributeset domains (core_logic_d user_logic_d ads_d
   media_d))
14 ; creation of file types
15 (type confidential_t)
16 (call mt_appdatafile (confidential_t))
17 (type ads_t)
18 (call mt_appdatafile (ads_t))
19 ; bounding the domains and types
20 (typebounds untrusted_app core_logic_d)
21 (typebounds untrusted_app user_logic_d)
22 (typebounds untrusted_app ads_d)
23 (typebounds untrusted_app media_d)
24 (typebounds app_data_file confidential_t)
25 (typebounds app_data_file ads_t)
26 ; grant core_logic_d access to confidential files
27 (allow core_logic_d confidential_t (dir (search write add_name)))
28 (allow core_logic_d confidential_t (file (create getattr open
   read write)))
29 ; grant ads_d access to ads_cache files
30 (allow ads_d ads_t(dir(search write add_name)))
31 (allow ads_d ads_t(file(create getattr open read write)))
32 ; minimum app_api_service subset
33 (allow domains activity_service (service_manager (find)))
34 (allow domains activity_task_service (service_manager (find)))
35 (allow domains ashmem_device_service (service_manager (find)))
36 (allow domains audio_service (service_manager (find)))
37 (allow domains surfaceflinger_service (service_manager (find)))
38 (allow domains gpu_service (service_manager (find)))
39 ; grant core_logic_d the needed permissions
```

```
40 (allow core_logic_d restorecon_service (service_manager (find)))
41 (allow core_logic_d location_service (service_manager (find)))
42 ; grant ads_d access to unity3ads needed services
43 (allow ads_d radio_service (service_manager (find)))
44 (allow ads_d webviewupdate_service (service_manager (find)))
45 (allow ads_d autofill_service (service_manager (find)))
46 (allow ads_d clipboard_service (service_manager (find)))
47 (allow ads_d batterystats_service(service_manager (find)))
48 (allow ads_d batteryproperties_service (service_manager (find)))
49 (allow ads_d audioserver_service (service_manager (find)))
50 (allow ads_d mediaserver_service (service_manager (find)))
51 ; grant media_d the needed permissions
52 (allow media_d autofill_service (service_manager (find)))
53 (allow media_d cameraserver_service (service_manager (find))))
```

Listing A.4: showcase app sepolicy.cil

Appendix B

SDS documentation

This appendix details the artifact associated with the approach presented in Chapter 3, and how to reproduce the experimental results.

B.1 Requirements

The deployment of the artifact requires a machine having a CPU with at least one physical core and at least 2 GB of RAM for each worker. The install procedure was tested on Ubuntu 20.04 LTS, with the default packages installed: `make, git, zip, gzip, python3, python3-venv, gnuplot`.
Install the required depencencies according to the following procedure:

- Install and set up `docker` and `docker-compose`
  ```
  sudo apt install docker
  sudo apt install docker-compose
  ```
- Add the current user to the `docker` group
  ```
  sudo usermod -aG docker <USER>
  ```
- Reboot the system
- Check that the following commands run without root privileges
  ```
  docker run hello-world; docker-compose -version
  ```

B.2 Deployment

To deploy the artifact execute the following steps.

1. Clone the repository
   ```
   git clone --depth 1 \
   --branch percom2021_artifact \
   https://github.com/mosaicrown/mondrian.git
   ```
2. Verify that all the software requirements illustrated previously are installed
   ```
   make
   ```
3. Pull and build a copy of the Docker images necessary to the artifact
   ```
   make start
   ```

Table B.1 lists the docker containers started by the artifact.

Container	URL
Hadoop Namenode	http://localhost:9870
Hadoop Datanode	http://localhost:9864
Spark History Server	http://localhost:18080
Spark Cluster Manager	http://localhost:8080

Table B.1: Docker containers and URLs associated

B.3 Usage

The artifact implements the centralized and the distributed version of the Mondrian algorithm. The artifact is complemented with a web UI that can be deployed running command `make ui`. The web UI is available at `http://localhost:5000` and can be used to run customized experiments. This section details the use of the artifact and the steps to reproduce the experimental results presented in [77].

The experiments in Chapter 3 used a sample from the IPUMS USA dataset [161]. The dataset is available at `https://ipums.org/`, together with a detailed guide for its download. To extract the sample to anonymize please go to IPUMS website `https://usa.ipums.org/usa/` and click on "Get Data". Then, select the attributes of interest (harmonized variables `State FIP Code`, `Age`, `Education Number`, `Occupation`, and `Income` in our experiments) and add

them to the cart. For your convenience, you can use the direct links at `https://github.com/mosaicrown/mondrian#usa-2018-dataset` (each variable name is a link that redirects to the page at ipums.org that permits to add the variable to the cart). Select the sample of interest (among USA samples, *2018 ACS* in our experiments) and create your data extract. To customize the sample size, set parameter *Persons* (in the experiments *Persons* is set to 510, to obtain a dataset with at least 500,000 tuples). Among the formats available for downloading the dataset, select the csv format and save the downloaded gzip archive in the root folder of the project, with name *usa_<extract_number>.csv.gz*.

The procedure to run the experiments has been automated and can be started running command `make artifact_experiments` from the root folder of the project. The procedure operates as follows:

1. it cleans the test environment stopping every Docker container that is still running and removing from HDFS the results produced by the previous runs;
2. it extracts the sample of IPUMS USA dataset to be anonymized from the archive and copies it to the *Spark Driver* volume;
3. it runs the centralized and distributed version of the Mondrian algorithm (see below), and measures the execution time and information loss, storing the results with the following directory structure:
```
mondrian/
|-- percom_artifact_experiments/
|-- |-- results/
|-- |-- |-- runtime_results_<TIMESTAMP>/
|-- |-- |-- loss_results_<TIMESTAMP>/
```
4. it shuts down all the containers except the *Spark History Server*, which remains available to keep track of the previous runs of the artifact.

Centralized version. The centralized version of Mondrian corresponds to the baseline of the experimental results in Chapter 3. The execution of the algorithm can be monitored through the messages showed on the terminal, which reports:

1. the schema and the first few tuples of the input dataset;
2. each decision taken by Mondrian to cut the dataset;

3. the schema and the first few tuples of the anonymized dataset;

4. a summary of the information loss measures and the execution time of the algorithm.

The anonymized dataset is in folder *local/anonymized*.

Distributed version. Given a number n of workers available in the distributed system, the artifact performs the following steps to execute the distributed version of the Mondrian algorithm:

1. start all the Docker services, initialize HDFS, and submit to the *Spark Driver* our Spark Application;

2. recover the dataset from HDFS and show its structure;

3. retrieve the n-quantiles of the best-scoring attribute of the dataset, showing the score used to decide the optimal cut and the size of the partitions;

4. show the first few tuples of the dataset, complemented with a new attribute containing the id of the quantile to which each tuple belongs and hence the worker to which the tuple is assigned;

5. anonymize the dataset;

6. show the first few tuples of the anonymized dataset, with a summary of the execution time.

The anonymized dataset is in folder *distributed/anonymized*.

B.4 Results

This section details the results presented in Chapter 3. To reproduce the results, the artifact has to be deployed on a machine equipped with a CPU with 20 logical cores, 40 GB of RAM, and 15 GB of free disk space on an SSD.

Execution time. This experiment measured the execution time when computing a 3-anonymous and 2-diverse version of a sample of the IPUMS USA dataset. The results of the experiments are stored in folder `runtime_results_<TIMESTAMP>`.

First, the artifact runs the centralized version of the Mondrian algorithm. The results are saved in file *centralized_results.csv*. Then, the artifact runs the distributed (Spark-based) version of the Mondrian algorithm, varying the number of workers from 2 to 20. The results are saved in file *spark_based_results.csv*. Besides generating the .csv files with the execution time of the centralized and distributed versions of the algorithm, the artifact plots these results generating file *comparison.pdf*. Note that the absolute times obtained running our artifact may slightly differ from the ones in Figure 3.3 of Chapter 3, due to the differences in the hardware of the machine used.

bf Information loss. This experiment measured the information loss when computing a 5-anonymous and 2-diverse version of a sample of the IPUMS USA dataset. The results of the experiments are stored in folder `loss_results_<TIMESTAMP>`. The artifact first runs the centralized version of the Mondrian algorithm, storing the results in file *centralized_results.csv*. Then, it runs the distributed version (with 5, 10, and 20 workers), using a sample including 0.01% of the dataset to determine the most suitable attribute and compute the n-quantiles (with $n = 5$, $n = 10$, and $n = 20$, respectively) for partitioning the dataset among the workers. The results obtained from five runs of the distributed version of the algorithm are stored in file *spark_based_results.csv*. The artifact also generates file *loss_table.csv*, which reports the average and the variance (in the form $\mu \pm \sigma$) of the results in file *spark_based_results.csv*. Note that, since the sample of IPUMS USA dataset is randomly extracted at each download, it may be different from the one used in our experiments and consequently the results might be slightly different from the ones in Table 3.1 of Chapter 3. We expect the results to have a similar trend, confirming the limited impact of sampling on information loss.

Appendix C

SecIdx documentation

This chapter provides further details on the implementation of SecIdx. It also gives the proofs of the theorems reported in Chapter 4.

C.1 Artifact

SecIdx is a Python library implementing the approach detailed in Chapter 4. The library provides several utilities. The utilities permits to: 1) build a k-flat relation, 2) construct the encrypted index with flat distribution and the client-side maps, and 3) translate and resolve queries at runtime. In the repository, the user can also find a complete system showing how to leverage SecIdx to store encrypted databases on PostgreSQL and Redis, and how to query them. The system is implemented as a multi-container Docker application. A dedicated set of Makefile targets is used to interact with it.

Requirements

To deploy the artifact we recommend a machine having a CPU with at least one physical core and 2 GB of RAM for each container. The install procedure was tested on Ubuntu 20.04 LTS, with the default packages installed: `make, git, python3, python3-venv`.

Install the required depencencies according to the following procedure:

- Install and set up `docker` and `docker-compose`
```
sudo apt install docker
sudo apt install docker-compose
```
- Add the current user to the `docker` group
```
sudo usermod -aG docker <USER>
```
- Reboot the system
- Check that the following commands run without root privileges
```
docker run hello-world; docker-compose -version
```

Submodules setup

To build the k-flat relation, SecIdx relies on an evolution of the approach presented in Chapter 3. The artifact detailed in Appendix B is automatically imported as a submodule of the SecIdx repository (i.e., the main module). By default, a `git clone` of the main module doesn't fetch the additional material provided by the submodules. Thus, after the main repository is cloned, the submodules must be downloaded and configured properly. In the `Makefile` we provide a target named `submodule_setup` to automatically pull the submodule and checkout to a fixed commit in its history. On top of that commit, a patch containing the changes introduced to build the k-flat relation properly is applied. This is achieved with a second target named `apply_patch`, which is also provided in the `Makefile`.

Workflow

To enable the k-flat secure indexing of an encrypted dataset, the user is required to perform four steps:

1. invoke the enhanced submodule to transform the original dataset into a k-flat relation;

2. build the encrypted index and the client-side maps on top of the k-flat representation;

3. wrap the k-flat representation with probabilistic encryption and enumerate the blocks;

4. outsource the encrypted relation and the index to a cloud storage provider (this is performed sending the relation to the proper container via network).

Upon sending a query to the backend, the client:

- parses the query converting it into an AST representation;
- uses the local maps to rewrite the query;
- sends the query to the server leveraging different execution strategies, based on the current setup.

When receiving the response, the client:

1. decrypts the encrypted blocks pulled from the server, retrieving the original plaintext rows;
2. streams the rows into a temporary table saved in a local SQLite instance;
3. filters spurious tuples by re-running the initial query using SQLite;
4. returns the results of the query to the user as if it was run against a plaintext database.

Deployment and usage

Artifact deployment is automatically performed when one of the utilities provided by the library is used. This means that, each time the user runs a `Makefile` target, the local environment in which the library is deployed, is automatically installed (or updated) by a number of Makefile prerequisites. This also applies to the containers; if the container is not available locally, a prebuilt container image is immediately downloaded from the network, customized installing the required dependencies, and finally deployed with `docker-compose`, just before the test requested by the user is performed. In the following, we demonstrate each of the steps described in the workflow section.

Construction of the k-flat relation

A prerequisite to construct the k-flat relation is to configure the submodule as explained previously. Running the command `make usa2018` from the terminal, the *pums 2018* dataset is downloaded from the `census.gov` website, unzipped and transformed into a k-flat relation using the submodule. In detail, the tool builds

and installs the container images required to run the submodule, the containers are started and the number of instances scaled with `docker-compose`, the indexing algorithm run, and finally the k-flat relation saved locally, in a csv format file. Figure C.1 shows the differences between the k-flat and the original relation.

```
dariofad@greymon:~/git/git_unibg/secure_index/datasets/usa2018$ head -n 10 usa2018_25.csv usa2018.csv
==> usa2018_25.csv <==
INDEX,STATEFIP,AGE,OCC,INCTOT,GID
228590,"{12,13,18,27,30,40,53,55,9}",[54-61],"{1108,2100,310,3255,420,5120,5320,565,6200,6600,7640}",[7200-9800],18607
195003,"{12,17,34,41,42,44,53,9}",[36-46],"{3255,3602,4920,5010,520,6515,6600,710,7640,8920}",[15000-17000],19807
162430,"{12,13,17,24,27,29,4,48,49,51,53}",[61-74],"{1007,4030,4055,4130,5240,5410,810,9121,9600}",[27900-29800],17088
175308,"{12,25,27,37,39,41,47,55,9}",[37-43],"{3322,350,3640,4251,4252,530,710,735}",[46000-51000],167
235664,"{29,4,53,54,55,6}",[29-33],"{1032,2100,230,520,6442,8640,8800}",[40000-42000],18480
187870,"{12,18,34,37,42,51,53}",[33-42],"{2850,335,3545,410,51,5110}",[45000-50000],8133
36828,"{17,25,36,47,8,9}",[36-43],"{440,5260}",[165000-200000],9469
388665,"{12,24,25,42,47,51,53}",[30-40],"{1541,2011,2300,3640,4020,440,5600,6520,725}",[30900-32000],16228
336316,"{13,20,42,45,48,49,55,8}",[20-26],"{2060,3424,3647,4010,530,710,7640,8990}",20000,15134

==> usa2018.csv <==
INDEX,STATEFIP,AGE,OCC,INCTOT
0,4,33,3802,28000
1,36,55,5610,17000
2,42,45,5420,30000
3,22,45,4965,51000
4,42,27,5240,25000
5,6,63,10,60000
6,36,40,800,90000
7,48,56,6355,55000
8,17,65,2100,120000
```

Figure C.1: Differences between an initial dataset and its 25-flat representation

Construction of the encrypted index and the client-side maps

After the k-flat relation has been computed, the user can start building the encrypted index, the client-side maps, and the encrypted k-flat relation. To demonstrate this workflow, the user must run the `preprocess` target. The user can customize its behavior editing the Makefile variables `INPUT`, `OUTPUT`, `TYPE`, `MAPPING`, and `ANONYMIZED`. The `TYPE` variable is the most important one, and allows the user to specify, for each column in the dataset, the structure to use when creating the local maps, and whether to use tokens (i.e., the value-based strategy). As an example, running the command `make preprocess TYPE=config/usa2018/runtime.json`, runs a test on the default *pums 2018* dataset, with runtime token generation, and the bitmap, range, roaring and range maps for the columns STATEFIP, STATE, OCC, and INCTOT, respectively. Figure C.2 shows the log of the test.

Outsourcing the relation to the storage provider

To conclude the preprocessing stages, the user has to upload the encrypted index and the encrypted k-flat relation to the storage provider. This can be done with

```
dariofad@greymon:~/git/git_unibg/secure_index$ make preprocess TYPE=config/usa2018/runtime.json

[*] CREATE GENERALIZATIONS TO TOKENS MAP
/home/dariofad/git/git_unibg/secure_index/.direnv/python-3.8.10/bin/python script/create_mapping.py --enc
 --password password --type config/usa2018/runtime.json datasets/usa2018/usa2018_25.csv mapping.enc
[*] Read anonymized dataset
[*] Remove duplicates to speed up mapping creation
[*] Map STATEFIP using bitmap.
[*] Map AGE using range.
[*] Map OCC using roaring.
[*] Map INCTOT using range.

[*] WRAP DATASET
/home/dariofad/git/git_unibg/secure_index/.direnv/python-3.8.10/bin/python script/wrap.py --pad --passwor
d password datasets/usa2018/usa2018.csv datasets/usa2018/usa2018_25.csv mapping.enc datasets/wrapped/usa2
018.csv
[*] Read plain dataset
Read plain dataset:           0.097s
[*] Read anonymized dataset
Read anonymized dataset:      0.319s
[*] Map STATEFIP generalizations to their tokens
Retrieve generalizations:     0.099s
Retrieve tokens:              0.163s
Collisions detection:         0.003s
Create t_mapping:             0.002s
Create t_mapping_idx:         0.002s
[*] Map AGE generalizations to their tokens
Retrieve generalizations:     0.000s
Retrieve tokens:              0.056s
Collisions detection:         0.003s
Create t_mapping:             0.000s
Create t_mapping_idx:         0.000s
[*] Map OCC generalizations to their tokens
Retrieve generalizations:     0.188s
Retrieve tokens:              0.121s
Collisions detection:         0.003s
Create t_mapping:             0.003s
Create t_mapping_idx:         0.001s
[*] Map INCTOT generalizations to their tokens
Retrieve generalizations:     0.001s
Retrieve tokens:              0.084s
Collisions detection:         0.005s
Create t_mapping:             0.001s
Create t_mapping_idx:         0.001s
Auxiliary stuff:              0.000s
[*] Compute maximum size of the pickle serialization
Maximum size:                 3.274s
[*] Wrap dataset
Wrapping:                     3.566s
[*] Checking correctness of the indexes (wrapping)
[*] Write dataset
Writing:                      0.143s
```

Figure C.2: Example of construction of the encrypted index, the client-side maps, and the encrypted k-flat relation

the `upload` target. The default backend alternative is PostgreSQL, to select Redis instead, the user can run the `upload_kv` target. If not available locally, the selected backend alternative is immediately installed in a dedicated container. The container is deployed and the data uploaded to it via network. Figure C.3 details the upload to the containerized PostgreSQL DBMS.

Runtime tests

To demonstrate how the system works at runtime, the user can run the `query` target. In this case a set of test queries is submitted to the client-side backend. The queries

```
dariofad@greymon:~/git/git_unibg/secure_index$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS         PORTS
                                  NAMES
a25473b5304c   secure_index_postgres    "docker-entrypoint.s…"   7 seconds ago   Up 6 seconds   0.0.0.0:54
32->5432/tcp, :::5432->5432/tcp   secure_index_postgres_1
dariofad@greymon:~/git/git_unibg/secure_index$ make upload
[*] Starting PostreSQL database.

docker-compose up -d postgres
secure_index_postgres_1 is up-to-date

[*] UPLOAD DATASET TO POSTGRESQL DATABASE
/home/dariofad/git/git_unibg/secure_index/.direnv/python-3.8.10/bin/python script/upload.py datasets/wrap
ped/usa2018.csv postgresql://postgres:mysecretpassword@localhost:5432/postgres
[*] Read datasets/wrapped/usa2018.csv dataset
[*] Upload datasets/wrapped/usa2018.csv as wrapped table
[*] Create primary key on wrapped using "STATEFIP","AGE","OCC","INCTOT" columns
[*] Create UNIQUE index on wrapped using STATEFIP
[*] Create UNIQUE index on wrapped using AGE
[*] Create UNIQUE index on wrapped using OCC
[*] Create UNIQUE index on wrapped using INCTOT
[*] Create UNIQUE index on wrapped using "INCTOT","OCC","AGE","STATEFIP"
```

Figure C.3: Dataset upload to the storage provider

are translated leveraging the local maps, and the blocks of interests are pulled from the containerized server. Finally, the plaintext rows are extracted from the blocks, the spurious tuples filtered by an in memory SQLite DB, and the result of the query sent to the user. The process is detailed in Figure C.4.

```
dariofad@greymon:~/git/git_unibg/secure_index$ make query | tail -n 6
secure_index_postgres_1 is up-to-date
[*] SELECT COUNT(*) FROM wrapped WHERE 90< "AGE" ORDER BY "AGE"

SELECT "EncTuples" FROM wrapped WHERE "AGE" IN (VALUES (4913692690168922l),(1020602504668320 42),(25518491
0992796335),(3099427427434628 52),(5326753138741550 84),(5390652440022407 73),(6887769314850628 74),(74528659
5831085895),(7553911265255138 01),(1012530011142831715),(1062417987451812723),(1088677205396610606),(10933
37439726165631),(1112371088196151081),(1174421813664102640),(1210984378902734082),(1262744970023964481),(
1415934980676354362),(1441750825607797453),(1468264862897916786),(1564459109010468202),(17446600560941872
09),(1889778985719538986),(1935845926920966055),(1993379224456704892),(2076172780833756808),(216293149034
6318447),(2370891556014212329),(2433736902455480233),(2504270361209049265),(2521659799759670394),(2738862
988601140521),(2822832755452328910),(2852616278125082598),(2918924155699387374),(2937247555753798095),(29
40240885019417768),(2942654123642218066),(3013120663475055777),(3291072508518324202),(3334097363079307652
),(3352269946834706889),(3377880404039054298),(3399953347718575695),(3518592878417909047),(35512978174039
32835),(3753565150514603134),(3869804400392434168),(3888246854132821730),(3919973435090336523),(394704462
8212761243),(3956997931411442746),(4032000629538537805),(4237779913789206025),(4283157852603991647),(4377
665830275163086),(4381764332641276368),(4427538685432796383),(4453167916619357605),(4491443576489655956),
(4516439666314594779),(4619713464573175985),(4641451618548462759),(4802674999937716059),(4862207952449499
692),(4892793523684426986),(4926802076579031177),(4951117510184662856),(4966486303525535461),(50662999734
12436464),(5091518972687440255),(5213224175035908358),(5402870826550909124),(5430114804293545240),(551659
4786741923029),(5523770930518618460),(5536904124180837121),(5566133393106381609),(5569680430450328512),(5
630945983439545780),(5660104692975520131),(5671179665317316825),(5701190369220221963),(58174710643550467 1
5),(5876374337396970033),(5963064659748842464),(5980798123053855277),(6058392342360277858),(6087999373456
191161),(6263257029563178640),(6322103178524860986),(6379760690545781363),(6390380806513709025),(65717872
49952287994),(6571983123942018828),(6622847074582607046),(6666339472998028073),(6677627233419367306),(670
9565267183753676),(6758073264593381154),(6921778761591248665),(7033544634171507400),(7098646963791153263)
,(7143808990676241387),(7279389102344965563),(7320300389191153544),(7356802090624927136),(738392434475405
2789),(7384878187936201443),(7521642706690194077),(7541988901426880253),(7629324972107035615),(7733978303
300580741),(7859327073075587867),(7982059578124066314),(8045372011359046916),(8144507926540095757),(82296
94467207485261),(8364233946546921519),(8582832736249592026),(8588938322961955696),(8730175067245125612),(
8744821800092333767),(8880017368797427409),(9092998174271727336))

   COUNT(*)
0       137
```

Figure C.4: Translation and resolution of a runtime query

Reproducibility and visualization

The results reported in the experimental evaluation (Section 4.7) can be reproduced running the command `make test`. Running the experiments takes a few days on our hardware. However, the user can recreate all the figures shown in Chapter 4 starting from the data attached to the repository. This can be done running the `make visualization` command. As an example, Figure C.5 compares the theoretical and experimental token collision probability measured in our test.

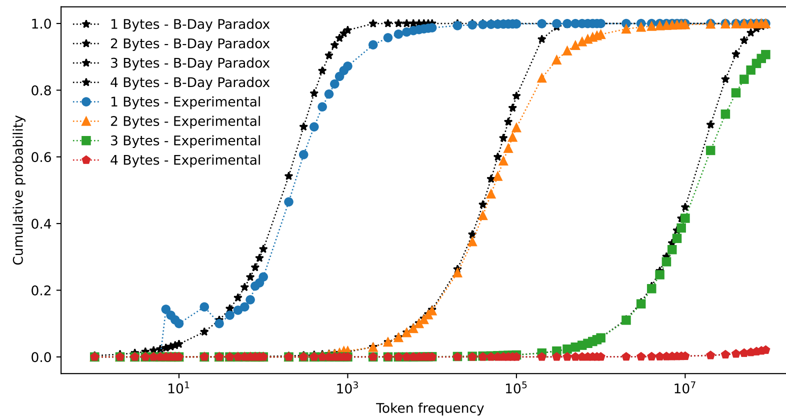


Figure C.5: Theoretical versus experimental token collision probability varying the number of tokens and their size

C.2 Proofs

Theorem 2 (Existence of a k-flat partition)**.** *Let r be a relation such that* $\mathrm{card}(r) \geq k$*. A k-flat partition $\mathcal{P}$ of r exists iff $h \leq p$, with h=(*$\mathrm{card}(r)$ *mod k) and p=*$\lfloor \mathrm{card}(r)/k \rfloor$.

Proof.
k-flat partition $\mathcal{P} \implies h \leq p$
By assumption, $\mathcal{P}$=$\{G_1, \dots, G_p\}$, with $p = \lfloor \mathrm{card}(r)/k \rfloor$. Let n_{k+1} be the number of partitions in $\mathcal{P}$ with k+1 tuples. Then, $0 \leq n_{k+1} \leq p$. By definition of k-flat partition, $\mathrm{card}(r) = k \cdot (p - n_{k+1}) + (k+1) \cdot n_{k+1} = k \cdot p + n_{k+1}$. By the quotient remainder theorem, when we divide $\mathrm{card}(r)$ by k, there exists unique integers p, h s.t. $\mathrm{card}(r)$=$k \cdot p + h$ and $0 \leq h < k$. Then, n_{k+1}=h.

k-flat partition $\mathcal{P} \Longleftarrow h \leq p$.
Let $\mathrm{card}(r)$=$k{\cdot}p{+}h$. Tuples in r can be partitioned in p groups of k tuples each. Then, since by assumption $h \leq p$, each of the remaining h tuples can be inserted into h of these p partitions, one tuple for each partition. In this way, by construction we have a partition with p groups in total s.t. $p{-}h$ groups have k tuples each, and h groups have $k{+}1$ tuples each. The partition is then a k-flat partition of r. □

Appendix D

ITYT documentation

This chapter provides further details on the implementation of ITYT. It also gives some instructions to fully reproduce the results presented in the experimental evaluation (Section 5.7).
ITYT repository is structured into three directories:

- `model`, that provides the implementation of a Z3-based solver of the economic model presented in Section 5.4;
- `contract`, that provides the implementation of the smart contract presented in Section 5.5 (with some extensions) alongside with some scripts to facilitate simulation with Brownie;
- `smpc`, that provides the implementation of three secure Multi-Party computation protocols based on FRESCO.

D.1 Model

This directory contains a Python script that can be used to find a solution to the ITYT Economic Model. To this end, the script leverages Z3, a theorem prover from Microsoft Research. To install the required dependencies and run the tool just navigate the filesystem to the current directory and run the `make` command. Custom input can be specified overriding the variables `N`, `K` and `V`. The process is detailed in Figure D.1.

```
dariofad@greymon:~/git/git_unibg/ityt/model$ make
venv/bin/python PoV.py --k 15 --n 30 --v 1 --format grid

[*] k = 15; n = 30; V = 1

[*] RESULTS

+-----+-----+--------+--------+--------+--------+--------+--------+--------+---------+
|   k |   n |      V |     Wh |     Bh |     Rh |     Ws |     Fo |   Fo/V |   Rh/Bh |
+=====+=====+========+========+========+========+========+========+========+=========+
|  15 |  30 | 1.0000 | 0.0013 | 0.0717 | 0.0730 | 0.0743 | 0.0756 | 0.0756 |  1.0179 |
+-----+-----+--------+--------+--------+--------+--------+--------+--------+---------+
```

Figure D.1: Solving the model from command line using the Z3 solver

Table D.1 reports a list of mappings that bind each solver variable to the related economic amount, as described in Chapter 5.

Solver	**ITYT**	**Description**
`V`	V	economic value assigned to the secret
`N`	n	number of shareholders
`K`	k	number or shares required to reconstruct the secret
`Rh`	$R_{\mathcal{H}}$	shareholder reward
`Bh`	$B_{\mathcal{H}}$	shareholder bid
`Fo`	$F_{\mathcal{O}}$	fee paid by the owner to get the service
`Ws`	$W_{\mathcal{S}}$	reward paid when whistleblowing the secret
`Wh`	W_h	reward paid when whistleblowing a share

Table D.1: Parameters and economic amounts constrained by the solver

D.2 Contract

This directory provides the implementation of a smart contract compatible with the ITYT framework. The implementation relies on MiMC to produce the commitments. To compile the smart contract with Solidity, a library implementing the MiMC primitive needs to be included in the directory contracts as a file named `MiMC.sol`. An example of such library implementation can be found at this link.

Dependencies

The following install procedure has been tested on Ubuntu 20.04.

- Install `solc`:
  `sudo snap install solc`

- Install `ganache-cli`:
  ```
  sudo apt install nodejs
  sudo apt install npm
  sudo npm install -g ganache-cli
  ```

- Install Python dependencies and the required Solidity compiler:
  ```
  python3 -m venv venv
  source venv/bin/activate
  pip install -r requirements
  python -c ``import solcx; solcx.install_solc('v0.5.6')''
  ```

Test

To deploy, test, and debug the contract generated the tools uses Brownie, a Python framework that allows to create wallets, inspect transactions and automatize tests. Running `brownie test` the script `test_proto_exec.py` is executed. Based on the (n, k) values specified, the script runs multiple smart contract simulations. The `-G` option can be used to display the gas consumed by each call to function. A simulation of the decentralized protocol is shown in Figure D.2.

```
(venv) dariofad@greymon:~/git/git_unibg/private-ityt/contract$ brownie test
Brownie v1.14.2 - Python development framework for Ethereum

======================================== test session starts =========================================
platform linux -- Python 3.8.10, pytest-6.0.1, py-1.10.0, pluggy-0.13.1
rootdir: /home/dariofad/git/git_unibg/private-ityt/contract
plugins: eth-brownie-1.14.2, xdist-1.34.0, forked-1.3.0, hypothesis-5.41.3, web3-5.11.1
collected 1 item

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul --mnemonic brown
ie'...

tests/test_proto_exec.py .                                                                      [100%]

======================================== 1 passed in 24.63s =========================================
Terminating local RPC client...
```

Figure D.2: Simulation of the decentralized protocol using Brownie

D.3 sMPC

This directory provides the implementation of the secure Multi-Party Comupation protocol designed in the ITYT framework. The protocol comes in two flavors:

- the *single-phase* version, which permits to compute the shares and the commitments in a single round jointly executed by the Owner and the *n* Shareholders;
- and the *two-phase version*, which separates the production of shares and commitments in two steps (*step 1* and *step 2*, respectively). Step 1 is jointly executed by the Owner and the n Shareholders, while step 2 is executed 1-to-1 by the Owner and each Shareholder.

Dependencies

On Ubuntu 20.04, the only prerequisite is the project management tool `maven`. It can be installed running on the console: `sudo apt install maven`

Compile and run

The functions are exposed to the user as `Make` targets. Running each of the available targets (listed below), the additional libraries are automatically downloaded by `maven`, the tool is compiled, installed and run. Here a brief description of the execution workflow.

Each Party is represented by a server. To each server a dedicated port (starting from 8080) is associated. The application, logfiles, and results related to each server are stored in the dedicated folder `server/serverX`, where `X` stands for the identifier of the server (`X`=1 is always associated with the Owner). Please note that the content under server can be wiped at each run (do not edit or create the directories related to the server, they are automatically created at runtime).
Table D.2 lists the targets available.

Target	**Description**
`runSP`	Simulates the *single-phase* version of the protocol
`runTPs1`	Simulates the *two-phase* step 1 version of the protocol
`runTPs2`	Simulates the *two-phase* step 2 version of the protocol

Table D.2: List of targets available

Each simulation can be customized overriding the variables in the `Makefile` accordingly. Table D.3 details each variable.

target	Description
`N`	The nof users running the *n-to-n* protocol
`K`	The degree of the polynomial + 1
`ARGS`	Common input supplied to the mpc by multiple parties
`ARGS[i]`	Input to the mpc supplied by the *i-th* party

Table D.3: Variables to customize the sMPC experiment

Figure D.3 shows the output received by parties 1 (i.e., the Owner) and 2 (i.e., the first shareholder) executing the *single-phase* protocol version. The Owner gets the commitment of the shares and of the key, while the shareholder gets her share and the commitment of the key.

```
(venv) dariofad@greymon:~/git/git_unibg/private-ityt/smpc$ cat servers/server1/log.txt servers/server2/lo
g.txt
15:11:38.399 [SCE-1] INFO dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - Running applica
tion: it.unibg.seclab.ityt.SinglePhase@61d9a19f using protocol suite: dk.alexandra.fresco.suite.spdz.Spdz
ProtocolSuite@6f1e53d4
[*] Step 1.1 - reading inputs and rnd polynomial coefficients...
[*] Step 1.2 - evaluating f(x) ...
[*] Step 1.3 - computing commitments using MiMC ...
[*] Step 1.4 - opening results to parties ...
15:11:52.533 [SCE-1] DEBUG dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - Evaluator done
. Evaluated a total of 1639 native protocols in 328 batches.
15:11:52.533 [SCE-1] INFO dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - The application
 it.unibg.seclab.ityt.SinglePhase@61d9a19f finished evaluation in 14129 ms.
15:11:52.534 [main] INFO it.unibg.seclab.ityt.SinglePhase - SinglePhaseResult {
  x: [null, null]
  y: [null, null]
  Cs: [7257021595259153262910960317193000740 5||13898934511104425000191531506001965930 9, 16350459915619964
9126741741353426129 89||18324600339851035233310992223692497804 5]
  Ck: 3328170873308519413540241289471750484 61
  n: 3
  k: 2
}
15:11:38.399 [SCE-1] INFO dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - Running applica
tion: it.unibg.seclab.ityt.SinglePhase@1585bf2d using protocol suite: dk.alexandra.fresco.suite.spdz.Spdz
ProtocolSuite@1b4253ed
[*] Step 1.1 - reading inputs and rnd polynomial coefficients...
[*] Step 1.2 - evaluating f(x) ...
[*] Step 1.3 - computing commitments using MiMC ...
[*] Step 1.4 - opening results to parties ...
15:11:52.533 [SCE-1] DEBUG dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - Evaluator done
. Evaluated a total of 1639 native protocols in 328 batches.
15:11:52.533 [SCE-1] INFO dk.alexandra.fresco.framework.sce.SecureComputationEngineImpl - The application
 it.unibg.seclab.ityt.SinglePhase@1585bf2d finished evaluation in 14126 ms.
15:11:52.534 [main] INFO it.unibg.seclab.ityt.SinglePhase - SinglePhaseResult {
  x: [29388742192630112757211571860771653265 4, null]
  y: [29388742192630112757211571860771653274 2, null]
  Cs: [null, null]
  Ck: 3328170873308519413540241289471750484 61
  n: 3
  k: 2
}
```

Figure D.3: *Single-phase* protocol execution for N = 2