UNIVERSITY OF CALIFORNIA,
IRVINE

Fine Grained Access Control Policies in Data Management Systems for Internet of Things Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Primal Pappachan

Dissertation Committee:
Sharad Mehrotra, Chair
Johann-Christoph Freytag
Faisal Nawab
Nalini Venkatasubramanian

2021

DEDICATION

To Kunjamma Chakkiyath (Ammachi) – My maternal grandmother!
കുഞ്ഞമ്മ ചക്കിയത്ത് (അമ്മച്ചി)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The biggest lesson I learned in my Ph.D. is that it takes a village. The PhD is mine to complete but I have leaned on many around me to get me across the finish line. I have always wanted to thank these folks and finally, I have the opportunity to do so. I am not sure where to begin.

Sharad Mehrotra: I am really happy and grateful that I had Sharad as my advisor. He has been wonderful in guiding me through my graduate school experience. His big smile and dad jokes help in lightening the heavy research discussions. I learned from him how to be patient with mentees and how to explain a complex concept in different ways. He balanced pushing me when I really needed it and giving me the freedom to guide my research on my own when I was making progress. He also helped me in becoming a better presenter of my own research and that of others. I admire his ability to stay focused during long research meetings (we had a lot of them) and ask good questions. I wish to emulate his enthusiasm and drive for research in my future career.

Thank you Sharad for your kindness, guidance, and patience.

Johann-Christoph Freytag: Christoph is an excellent collaborator and mentor. I learned a lot from the research discussions we had while formalizing the problem of scaling up policy enforcement in the Sieve project. When I started thinking about my committee members, Christoph's name is the first one that came into my mind after that of Sharad. He has always shown great enthusiasm about my research and was always willing to help in whatever way possible.

I would also like to thank **Nalini Venkatasubramanian** and **Faisal Nawab** for being part of my thesis committee. I have worked with Nalini on many projects and learned many things from her. I have always been impressed by her ability to bring in different perspectives to a research discussion. Her gracious hospitality during the yearly lab parties made all of them memorable experiences. After joining my thesis committee in Spring, Faisal has been a wonderful advocate of my research and has given me excellent suggestions on extensions for some of the work presented in this thesis.

Roberto Yus: I consider myself extremely lucky to have a friend like Roberto throughout my entire graduate school experience. He believed in me, pushed me, and helped me whenever I hit a roadblock in my PhD. He has been a great friend, collaborator, mentor, and ultimately a witness to my entire PhD. He also helped with proof-reading and editing this thesis. I learned so many things from him and I am a better student, researcher, and person because of our friendship. Thank you Roberto.

I would also like to thank my collaborators **Xi He** and **Shufan Zhang**. Xi has been an active collaborator and helped significantly in formalizing the security definitions of the work presented in Chapter 5. Shufan has been an amazing collaborator and friend since joining the project in early Spring. His hard work, dedication, and tenacity tremendously helped in swiftly getting from idea to implementation for the work presented in Chapter 5. I am very grateful to them both.

The **UCI ISG Group** is a brilliant group of academics and researchers who are always supportive of each other, be it attending each other's talks or having a Java City coffee chat. I am indebted for their friendship and making our lab space a fun place to be at. I would like to acknowledge some of them below with apologies for anyone I might have missed.

Dhrubajyoti Ghosh, who is always smiling and always readily joins me in lunch breaks. Every PhD journey is unique but Dhrub and I had the most similar experience. This helped in normalizing my experience. We always made sure that both of us were on track with all the Ph.D. requirements.
Sameera Ghayyur, who always asks pertinent questions and for weekend hikes.
Peeyush Gupta, who knew about all the DBMSs, and his quirky sense of humor.
Guoxi Wang, who is always willing to help and an inspiring collaborator to work with.
Xikui Wang, for being the best neighbor and having many fun 5-minute conversations.
Eunjeong Shin, purveyor of the best treats.
Georgios Bouloukakis, for his laughter and being a master of BBQs.
Abdul Alsaudi, for his questions and enthusiasm during group meetings.
Shantanu Sharma, capturer of the best moments.

I will miss you all and wish you all the best for the rest of your lives.

Outside the ISG group, I am grateful to have made some amazing friends who made my graduate life a wonderful one. **Anirudh Wodeyar** for the record of sharing the most number of hobbies with me. He is an amazing friend on whom I could always lean for support. I admire his non-judgemental attitude towards most things in life. He thoughtfully arranged for delivery of my favorite food items in the week of my defense. **Nitin Agarwal** for holding the best house parties during my time at UCI and was always up for a round of tennis in the evenings after a long day at work. **Rohit Zambre** for being a good friend and my virtual yoga buddy for almost 300 days. This yoga practice helped me stay grounded during the last stretch of my PhD. **Sumaya Almanee** for our cowalks, long voice memos and cute post-it notes. **Avinash Mohanakrishnan** for our monthly check-ins and long conversations on life and the pursuit of happiness. **Henna Manglani** for being my biggest cheerleader in the time I have known her. Her words of encouragement were a tremendous source of strength during the extremely stressful last phase of my Ph.D. She patiently listened to me when I needed it the most and offered relentless support through words of affirmation. She celebrates me and my achievements more than I do and made me overall more positive and less self-critical.

ZotBins: My mentees brought a lot of joy to my graduate school experience. They are one of the strongest reasons that I continue the academic hustle to be a research faculty. Thank you Owen Yang, Joshua Cao, Jesse Chong, and every other member of ZotBins.

Family: My parents – **Pappachan and Meena** – have been supportive of my rather audacious plans to move to the furthest time zone possible and pursue graduate studies. I talk with them daily and their love and encouragement have been my daily supplements. My elder sister **Preema** set the way for me to do my bachelor's in Computer Science by doing it first and similarly my brother in law **Jopaul** by doing his Doctorate. I could always count on my younger sister **Priya** for support and our weekly venting sessions. I miss my two nephews dearly as I only get to meet them once a year since my move to the U.S. **Alan** and **Ryan** make me the happiest uncle in the world. Moving away from family was difficult but it was made easier by the kindness of the following folks in the US who adopted me as part of their family: **Biju, Roxy, Trisha, Brian,** and **Wilson Kidangan**.

I would also like to express my words of gratitude to the amazing staff at the Computer Science department – special shout out to Mary Carillo and Leslie Escalante – and the International Center. Thanks for the great work you do.

I would like to acknowledge some of my past mentors who believed in me and thus have directly contributed to me doing a PhD. **Pramode C.E.** was my teacher and mentor during undergrad. He inspired me to dive deep into programming and encouraged me to do projects outside the given curriculum. His joy for teaching was infectious and it inspired me to learn more. His support and encouragement directly resulted in me applying for Google Summer of Code program. **Jorge Silva** was my mentor during Google Summer of Code and introduced me to research in Computer Science. I learned a lot from my discussions with him during this time and understood how to think about a challenging problem and come up with practical solutions for it. He was always available to discuss ideas, brainstorm, and inspired to go well and beyond the project scope. **Anupam Joshi** and **Tim Finin** were my mentors during my Master's. They gave me the opportunity to be part of **Ebiquity** and work on interesting projects during my time at UMBC.

There are possibly many others I am missing here. I am lucky to have a strong supporting village. If you ever read this section or any part of my thesis, I owe you a boba drink so please do reach out (even if you don't like boba). This long journey would have been impossibly difficult to accomplish without your support.

The work reported in this thesis is supported by DARPA under agreement number FA8750-16-2-0021 and partially supported by the NSF grants 2032525, 1545071, and 1527536.

VITA

Primal Pappachan

EDUCATION

Doctor of Philosophy in Computer Science	**2021**
University of California, Irvine	*Irvine, CA*
Master of Science in Computer Science	**2014**
University of Maryland, Baltimore County	*Halethorpe, MD*
Bachelor of Technology in Computer Science and Engineering	**2011**
Government Engineering College, Thrissur	*Thrissur, Kerala*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2014–2021**
University of California, Irvine	*Irvine, California*
Graduate Research Assistant	**2013–2014**
University of Maryland, Baltimore County	*Halethorpe, MD*

TEACHING EXPERIENCE

Teaching Assistant	**2015–2019**
University of California, Irvine	*Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems **2020**
Proceedings of the Very Large Database Endowment (PVLDB)

REFEREED CONFERENCE PUBLICATIONS

Don't be a tattle tale: Preventing data leakages through data dependencies on access control protected data **Planned: Sept. 2021**
Pending Acceptance

Designing privacy preserving data sharing middleware for Internet of Things **Nov 2020**
3rd International SenSys+BuildSys Workshop on Data: Acquisition to Analysis (DATA 2020)

The ZotBins solution to waste management using Internet of Things **Nov 2020**
18th ACM International Conference on Embedded Networked Sensor Systems (SenSys 2020)

SemIoTic: Bridging the Semantic Gap in IoT Spaces **Nov 2019**
6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys 2019)

Towards Privacy-Aware Smart Buildings: Capturing, Communicating, and Enforcing Privacy Policies and Preferences **Jun 2017**
37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2017)

Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications. **Oct 2015**
1st International Workshop on Mobile Deployment of Semantic Technologies (MoDeST 2015) co-located with 14th International Semantic Web Conference (ISWC 2015)

Semantics for Privacy and Shared Context **Oct 2014**
Second International Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014)

SOFTWARE

Sieve https://github.com/primalpop/sieve
SIEVE is a general purpose middleware to support access control in DBMS that enables them to scale query processing with very large number of access control policies.

TIPPERS https://tippers.ics.uci.edu/

TIPPERS is a system that manages IoT smart spaces by collecting sensor data, inferring semantically meaningful information from it, and integrates different Privacy Enhancing Technologies to deal with privacy issues in IoT data management.

PE-IoT

PE-IoT, a system for orchestrating privacy-enhanced data flows that (a) provides users (data subjects) with capabilities to opt-in/opt-out in the data that is shared with the service providers and (b) enable data controllers to invoke a range of Privacy Enhancing Technologies (PETs) such as anonymization, randomization, and perturbation to transform data streams into their privacy preserving counterparts.

ABSTRACT OF THE DISSERTATION

Fine Grained Access Control Policies in Data Management Systems for Internet of Things Applications

By

Primal Pappachan

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Sharad Mehrotra, Chair

New technology domains, such as the Internet of Things (IoT), are adding a large number of new devices with Internet connectivity to the spaces where we work and live. These devices are accelerating the collection of user data at an unprecedented rate. On the other hand, new data privacy regulations are emerging all around the globe to protect people's privacy (such as the California Consumer Privacy Act CCPA, European General Data Protection Regulation GDPR, and Brazilian LGPD, among many others). These regulations have put forward stringent requirements on organizations on what should be done when user data is handled. Organizations have been scrambling to adapt their infrastructure in response to these regulations and many have been punished with hefty fines for improper handling of data[1].

Data Management Systems are at the core of organizations collecting such data and handle its capture, retention, processing, and sharing. To protect people's privacy, Data Management Systems require, among others, to be able to enforce individuals' privacy preferences. These issues become even more challenging given the scale at which data is captured in new domains such as the IoT. This thesis presents various solutions to support fine-grained privacy policies for data protection when dealing with upcoming IoT applications.

[1]According to the GDPR enforcement tracker (https://www.enforcementtracker.com/?insights).

In particular, this thesis presents a framework to enable people to communicate their privacy preferences/policies to smart spaces to address the challenge of *Policy-based Privacy-by-design Smart Spaces*. This includes a language which allows users to define who, and under which circumstances, can access their data collected by IoT systems. Supporting the definition of such user-defined fine-grained IoT policies in Data Management Systems can lead to scenarios where a large number of them have to be enforced in real-time. The thesis presents an approach to answer queries efficiently while enforcing a very large number (hundreds of thousands) of user policies to address the challenge of *Scalability of Policy Enforcement*. In modern DBMS and particularly in IoT settings, data exists at different semantic levels where in dependencies capture the constraints that exist within the data. This thesis presents an approach to *prevent leakages through various different dependencies on access controlled data* .

The prototypes built as part of the solutions for the above challenges have been integrated into two IoT Systems deployed at UC Irvine. The first is an IoT test bed entitled TIPPERS and the second is privacy preserving middleware called PE-IoT. These integrations show the feasibility of the approaches presented to specify and efficiently enforce privacy policies for supporting IoT applications.

Chapter 1

Introduction

"The fantastic advances in the field of electronic communication constitute a great danger to the privacy of the individual."

Chief Justice Earl Warren, *1963 Supreme Court opinion*

From smart cars to smart buildings and from activity bracelets to smart fridges, every object in our environment is increasingly being endowed with sensing, computing, communication, and actuation functionalities. The total number of Internet of Things (IoT) devices is expected to reach 43 billion units by 2023 which is an almost threefold increase from 2018[1].

This rapid growth is transforming many domains and one such domain is *smart spaces* where the IoT applications improve productivity, comfort, social interactions, safety, energy savings and more. As an example, modern HVAC (heating, ventilating, and air conditioning) systems come with functionalities that ties to beacons, presence sensors, cameras, and personal devices (e.g., smartphones carried by the building's inhabitants). One commonality to all these applications and scenarios is their reliance on the collection of personal and identifying data. Thus the promises of IoT comes at the cost of new and complex privacy challenges.

[1]https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things

The privacy challenges in IoT settings are due to some unique characteristics of the data management in this domain as well as the user experience of using an IoT application: 1) Data captured by IoT devices are highly granular including information about individuals which can reveal their location, habits, health status, and so on. These are things people consider private and would want to control their exposure. 2) In IoT environments, such as smart buildings, users are less likely to be aware of the technologies with which they might be interacting. Hence, users might not be aware of which data related to them is potentially captured and shared with others. 3) Users have no control over the management of data (e.g., data capture, retention, processing, and sharing) by IoT systems. This is because IoT devices are typically part of the infrastructure and cannot be directly controlled by individuals. These differences add to the complexity of meeting the privacy needs of users.

The privacy problem associated with management of user data has been recognized at the highest level, including in the form of guidelines developed by the OECD and reports from the Federal Trade Commission. Moreover, the massive data collection taking place in today's world has motivated the advent of stringent data privacy regulations, such as the European General Data Protection Regulation (GDPR) [3], the California Online Privacy Protection Act (CalOPPA) [2], and the Consumer Privacy Act (CCPA) [1]. This imposes legislative requirements that control how organizations manage user data including data collection and sharing transparency, data minimization, and data retention. More importantly, such new data protection regulations highlight the need to give a voice to individuals whose data is being collected. This implies enabling users to opt-in/out from different practices such as sharing of data with services or others.

Access control policies are a traditional mechanism used in data management to allow users to specify their privacy preferences with respect to usage of their data. In the case of IoT domain, wherein sensors continuously monitor individuals (e.g., continuous physiological monitoring by wearable devices, location monitoring both inside and outside buildings), data management systems need to provide users with mechanisms for finer control over who can access their data and for what purpose. Supporting such fine grained policies in data management systems raises several research challenges and this thesis focuses on addressing some of the important challenges that arise when building *policy-based privacy-by-design data management systems* for new scenarios such as the IoT.

The first challenge is that of *developing a privacy-by-design framework for IoT applications.* The first requirement for such a framework is to have mechanisms that can notify users about IoT data collection in a space and their relevant privacy options. New models and privacy concepts need to be developed to enable the smart space to communicate with its inhabitants about its data practices and for the inhabitants to be able to choose what data related to them can be shared with such services.

After capturing the privacy policies of the user, the underlying data management system should be capable of efficiently enforcing privacy policies and preferences from different users without loss of utility for the services that exist in the space. Thus the second challenge is that of *scaling enforcement of access control policies* in data management systems. In IoT settings, the set of policies becomes a dominant factor/bottleneck in the query processing due to their large numbers. This has been highlighted as one of the open challenges for Big Data management systems in recent surveys such as [32].

In many scenarios including IoT applications, there might be background knowledge available to an adversary who can utilize that as an inference channel to learn more about protected data. The third challenge involves *protecting access controlled data from leakages through data dependencies.* One such common form of background knowledge is dependencies that capture the various type of constraints that exist within the data. In the IoT domain, an example of such constraint is through an enrichment function which transforms the raw data collected from sensors and generates the derived data which is shared with application developers and service providers. Such leakages leads to violations of access control policies even when sensitive data is hidden.

Thesis Contributions

This thesis makes the following main contributions in the context of fine-grained access control in data management systems for supporting IoT applications:

- We introduce a framework where *IoT Assistants* capture and manage the privacy preferences of their users and communicate them to *privacy-aware IoT smart spaces*. Such a framework outlines necessary the important attributes required for this interaction such as: (1) the data collection and sharing practices associated with deployed sensors and services in smart buildings as well as (2) the privacy preferences to help users manage their privacy in such environments.

- We present Sieve, a layered approach of implementing Fine-Grained Access Control in existing DBMSs, that exploits a variety of their features (e.g., UDFs, index usage hints, query explain) to scale to a large number of policies. Given a query, Sieve exploits its context to filter the policies that need to be checked. It also generates *guarded expressions* that save on evaluation cost by grouping policies and exploit database indices to cut on read cost. Our experimental results demonstrate that existing DBMSs can utilize Sieve to significantly reduce query-time policy evaluation cost. Using Sieve DBMSs can support real-time access control in applications such as emerging smart environments.

- We study the leakages of access control protected data in data management systems through two important classes of data dependencies: 1) *Denial Constraints* and 2) *Provenance Based Dependencies.* Denial constraints are a general model of integrity constraints and can express commonly used constraints such as functional dependencies, conditional functional dependencies, and key constraints. We introduce *provenance based dependencies* which can capture the relationships between source data and derived data. Considering these dependencies as background knowledge to an adversary, we formally define how the information about a sensitive data could leak through them and methods to compute the leakage. Furthermore, we describe the rules to decide the non-sensitive data that should not be disclosed to prevent leakages of the access control protected data. We describe an algorithm which utilizes these rules and leakage computation to achieve the required deniability guarantees for all sensitive cells.

- Deploying policy-based mechanisms in real-world IoT data management systems brings about its own challenges such as: 1) Integration of such mechanisms on systems that can be deployed at the data controller side (e.g., network provider, water agency, building management) or at the user side; 2) Developing appropriate policy models and enforcement for different phases of the data management flow (i.e., capture, sharing, retention); and 3) Development of user-facing systems to enable them to specify their privacy preferences/policies for their enforcement. In this thesis, we show the integration of these *policy-based privacy-by-design* approaches in two different systems deployed in the real world. The first is an *IoT Testbed (called TIPPERS)* deployed at University of California, Irvine. The policy-based privacy-by-design mechanisms were incorporated into testbed by building various interfaces for users to input their policy preferences, Application Programming Interfaces (APIs) for service providers to query the data and finally the data management system backend to enforce these policies when answering these queries. The second is integration of a policy engine in the context of *Privacy Enhancing middleware (entitled PE-IoT)*. This middle-ware (a) Provides users (data subjects) with capabilities to opt-in/opt-out in the data flows that is shared with the service providers; and (b) Enables data controllers to invoke a range of Privacy Enhancing Technologies (PETs) such as anonymization, randomization, and perturbation to transform data streams into their privacy preserving counterparts.

The rest of this thesis is organized as follows. Chapter 2 discusses the privacy regulations and reviews privacy enhancing technologies, and the history of access control policies in data management systems. Chapter 3 presents a framework for IoT smart spaces where users and IoT systems can communicate with each other using privacy policies. Chapter 4 presents a middle-ware that makes real-time enforcement of fine-grained access control policies feasible in IoT settings where there are large number of policies. Chapter 5 presents a model for studying leakages in access controlled data in the presence of various types of dependencies and describes algorithms to prevent these leakages. Chapter 6 shows the integration of some of the policy-based mechanisms into a IoT testbed and a privacy enhancing middle-ware. Finally, Chapter 7 concludes by summarizing the contributions and the possible future extensions to this work presented in this thesis.

Chapter 2

Related Work

"The best time to plant a tree was 20 years ago. The second best time is now."

Chinese Proverb

This chapter describes the privacy challenge that have been brought about by IoT. Then we review recent legislative efforts in data privacy protection, highlighting the recent European General Data Protection Regulation. As a mechanism to make systems enforce privacy requirements in those regulations, next the chapter reviews different privacy enhancing techniques. Finally, it delves into the usage of privacy policies, with an emphasis on access control policies, as a mechanism to realize the requirement of data regulations of enabling users define who, and for which purpose, can access their data. The related works corresponding to the contributions in this thesis will be discussed in their corresponding chapters.

2.1 The Privacy Challenge

Organizations today capture and store large volumes of personal data that they use for a variety of purposes such as providing personalized services and advertisement. Continuous data capture, whether it be through sensors embedded in physical spaces to support location-based services (e.g., targeted ads and coupons), or in the form of web data (e.g., click-stream data) to learn users' web browsing habits, has significant privacy implications [12, 17, 94].

The advent of the Internet of Things and instrumented physical spaces is increasing even further the amount of data about individuals collected. While the IoT holds many promises, it also gives rise to new and complex privacy challenges. Especially given that IoT sensor data, whose collection might seem innocuous, can lead to significant privacy risks for data subjects. Various studies have demonstrated that by observing electrical events and cell phone usage in a space it is possible to detect the whereabouts and daily activities of its residents [16, 71, 43]. Other studies, such as [66] have shown that collection of data from occupancy sensors (typically deployed in office buildings to automatically switch lights on/off) can lead to sensitive data of the individual such as whether they come in late to work, leave early, take long breaks, smoking habits, etc.

Data collected from Access Points in a smart space (e.g., WiFi or bluetooth), can also lead to privacy leakage. This data might also seem innocuous since it does not necessarily contain information about what the user connected to the AP was doing (e.g., what website the user was visiting) and, in general, associates events to a device rather than a user (e.g., the smartphone used to connect to the AP). However, by using simple classification rules on top of WiFi connectivity data and common sense background knowledge, we can correctly classify devices into profiles. For example, in a University building, knowing where the WiFi APs are located (e.g., a classroom, lounge areas, labs, faculty/staff offices) along with rules such as "undergraduate students typically are located in classrooms and stay in the building from 9am-4pm" or "staff members typically are location in staff offices and arrive to the building around 7am" can lead to highly accurate classifications. Figure 2.1 shows a result obtained after applying these rules to connectivity data captured in a University building by the TIPPERS system [75].

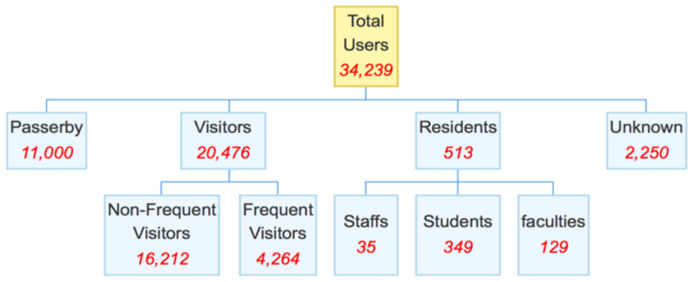


Figure 2.1: Classification of users based on WiFi Connectivity data.

Once this classification is performed, additional background knowledge could be used to associate devices to their potential users (e.g., a device classified as faculty that spends significant amount of time in the area that includes Prof. Smith's office probably belongs to him). Additionally, recent work has shown that background knowledge such as whether a specific room is public or private, along with historical connectivity data, could be used to improve the localization of a device/person [70]. This means that we can identify whether a device belongs to a specific individual and then determine, with a very fine-grained granularity, what rooms the person visited. Such information can lead to sensitive information such as who the person spends time with in the same space and/or whether the person visited sensitive spaces and with what frequency (e.g., counseling office, restrooms, religious spaces, etc.).

2.2 Data Privacy Regulations

The history of privacy rights and regulations starts long before the era of Big Data and IoT. As an example, the Supreme Court has found that the U.S. Constitution (which came into effect in 1789) does provide for a right to privacy in its First, Third, Fourth, and Fifth amendments[1]. In 1890, Warren and Brandeis articulated the Right to Privacy [19], a *"right to be let alone"*, which examined US laws to determine if they protected the privacy of the

[1]http://law2.umkc.edu/faculty/projects/ftrials/conlaw/rightofprivacy.html

individual. Warren and Brandeis were inspired to develop their work by the coverage of intimate personal lives by the press of the time.

Technological advances, such as computers, have enabled governments and corporations to capture and retain large amounts of individuals data since the 1950s. Society has been discussing the potential impact of this reality to people's privacy since then. As a consequence, different data protection/privacy regulations have been enacted. The federal state of Hesse (Germany) passed the first data protection law in the world in 1970 which became into force in 1978 [88]. With the advent of the Internet, e-commerce, and smartphones, the collection of data has become more pervasive which has been a catalyst for newer regulations which are more specific in their requirements and stricter in their enforcement (see Figure 2.2 for a chronology of new adopted regulations in the last 3 years).

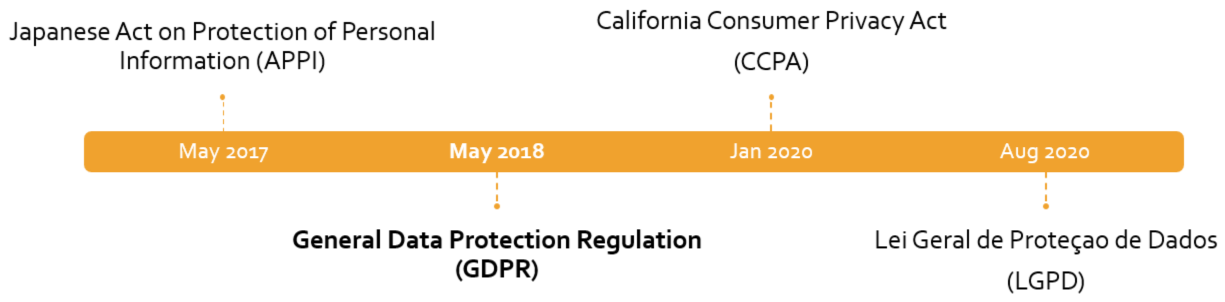

Japanese Act on Protection of Personal Information (APPI)
California Consumer Privacy Act (CCPA)
May 2017
May 2018
Jan 2020
Aug 2020
General Data Protection Regulation (GDPR)
Lei Geral de Proteçao de Dados (LGPD)

Figure 2.2: Chronology of privacy regulations.

These new data protection/privacy regulations are emerging all over the world. As of December 2020, over 130 countries and self-governing jurisdictions and territories have adopted national laws and almost 40 countries and jurisdictions have pending bills or initiatives (see Figure 2.3 extracted from [13]). Countries with no national law/initiatives, in some cases, have regional or state regulations. For instance, in the US, the state of California enacted the Consumer Privacy Act (CCPA) [1] in January 2020. The introduction of stringent privacy laws, which impose legislative requirements that control how organizations manage user data, has stimulated the need for a redesign of data processing systems. Regulatory compliance is challenging since it involves additional processing overheads. Moreover, implementing the required functionality is often in conflict with the design and operation of modern systems where persistence of data is inherent (e.g., storing data forever, reusing data indiscriminately, etc.) [92].

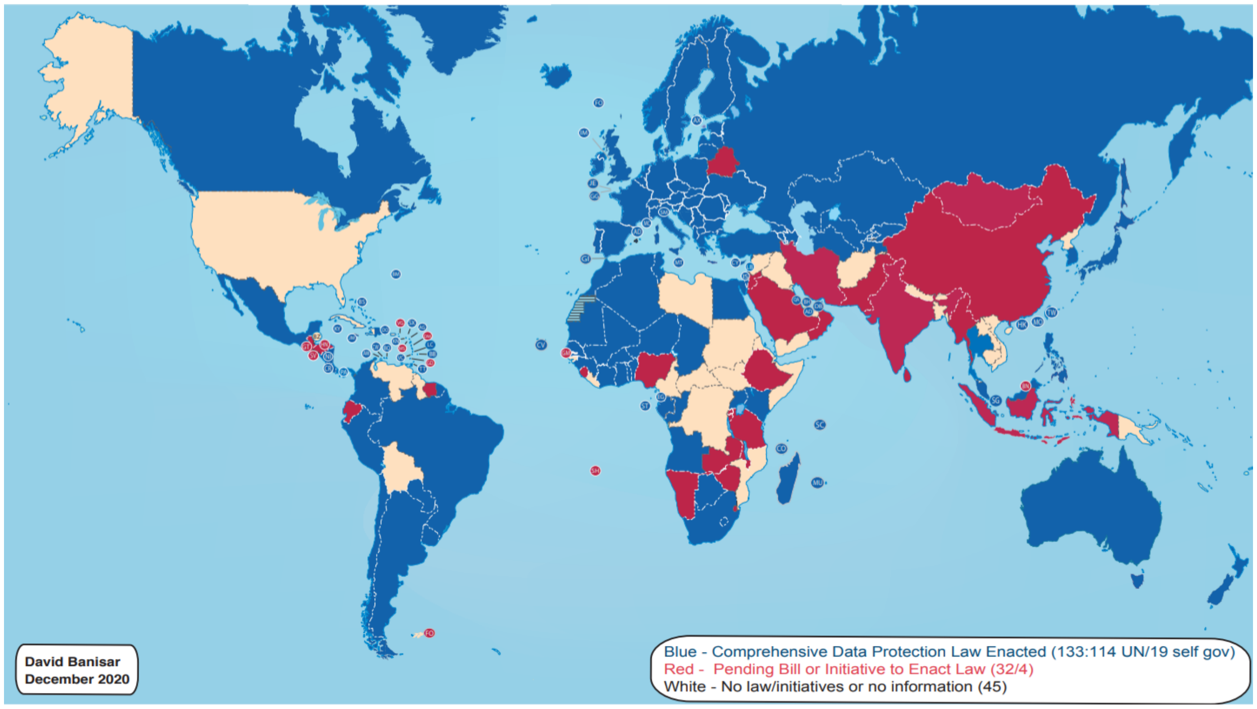

David Banisar
December 2020
Blue - Comprehensive Data Protection Law Enacted (133:114 UN/19 self gov)
Red - Pending Bill or Initiative to Enact Law (32/4)
White - No law/initiatives or no information (45)

Figure 2.3: National comprehensive data protection/privacy laws and bills in 2020.

Rest of the section analyzes one of these pioneer recent stringent privacy laws: The European General Data Protection Regulation (GDPR) [3]. GPDR was proposed on April 14, 2016 and came into effect on May 25, 2018. GPDR is a set of unified data protection rules in all 27 member states of the European Union. It replaces the previous Data Protection Directive. Unlike the Data Protection Directive, GDPR is a law (i.e., other national laws are not required). GPDR is applicable to all services offered within the EU (regardless of where the company is located).

GDPR establishes privacy and protection of personal data as a fundamental right. It includes 99 legal articles and 173 Recitals that regulate the collection, processing, protection, transfer, and deletion of personal data. GDPR(Article 4) defines personal data as

Any information relating to an identified or identifiable natural person ('data subject') meaning someone who can be identified, directly or indirectly, in particular by reference to an identifier.

It grants Rights to People for protection and privacy of their data. It assigns Responsibilities to Companies for safe and responsible collection and processing of data. Companies violating these regulations could face serious consequences for non-compliance with Max Penalty of 4% of global revenue or €20 million (≈$23 million), whichever is greater. Since July 2018, and as of June 2021, there have been 589 violations of GDPR regulations by companies all around the world punished for a total sum of fines amounting to €279 million (≈$331 million)[2].

Specific to data management, GDPR outlines the rights to users (data subjects) and responsibilities to organizations (data controllers) summarized in Table 2.1. With respect to users, for instance, the GDPR states the following in Article 14:

"[...] the controller shall provide the data subject with the following information necessary to ensure fair and transparent processing in respect of the data subject: [...] the existence of the right to request from the controller access to and rectification or erasure of personal data or restriction of processing concerning the data subject and to object to processing as well as the right to data portability;"

Table 2.1: Privacy Design Requirements for IoT

Article Name (Article #)	Design Requirement for IoT
Purpose limitation (5)	User data must be only collected and processed for specific purposes
Secure infrastructure (32)	Best effort should be made for implementing appropriate data security
Right to object (14)	Users should explicitly allow sharing of their data with others
Right to be Forgotten (5)	Data collected must be not stored indefinitely
Proof of Compliance (30)	Audit logs of all operations must be stored to demonstrate compliance
Data protection by design and by default (25)	Controller shall minimize the amount of data to be collected and processed

[2]According to the GDPR enforcement tracker (https://www.enforcementtracker.com/?insights).

This means that controllers have to not only provide transparency about how the data is used but also about the mechanisms in place for users to opt-in/out of different aspects of this processing. With respect to organizations, the GDPR states, among others that collection of data shall be minimized, cannot be done for unknown purposes, and that systems should have a more privacy-by-design implementation.

Hence, GDPR mandates data handlers to answer questions such as *"what data is captured?", "where is data stored?", "how is data analyzed?", "who has access to data?"* to manage data in a compliant manner. Additionally, it requires handlers of data to be transparent about the management of data and implement user preferences with respect to it. This translates on a requirement for enhancements to current data management systems to, among others, articulate collection purpose, enable people to opt-in/out from their data being shared with others, maintain audit logs, and support erasure. These form the design requirements to make today's systems run by data controllers regulation compliant. Hence, a key requirement for organizations/services to collect and to use an individual's data, is to adopt the principle of *choice and consent* [64][3]. At the core of this principle is the support for mechanisms to enforce people's choices to preserve their privacy. Particularly, user-facing mechanism to enable people to express their preferences.

2.3 Privacy Preservation

Due in part to the recently adopted privacy regulation laws, there has been a significant recent interest in developing technologies that ensure individual's privacy. In the literature, a diverse range of *Privacy Enhancing Technologies (PETs)* have been proposed that allow manipulating data in a privacy preserving manner. These PETs are based on different underlying mechanisms (such as removing personally identifiable information, introducing noise,

[3]Currently, such organizations typically follow the principle of *notice* wherein they inform the user about data collection, but may not support mechanisms to seek and enforce consent.

encryption, controlling access to data, etc.) to prevent the leakage of sensitive information about an individual. The main differences among such PETs are whether they are user oriented, provided privacy guarantees, underlying assumptions about the adversary, and release of aggregate vs. individual level data. An excellent survey of different techniques for privacy preserving data publishing techniques can be found in [47].

- Release of *statistics* on the data instead of individual records such as mean, min, maximum, etc. This is typically used to study demographics of a population and hides, up to some extent, information about the participants (e.g., US Census).
- Release of *Predictive models* which can be outputs of, for instance, classifiers or other mechanisms based on machine learning (e.g., Sentiment Analysis [74]).
- Release of data after *deidentification* where information is shared after removing any personally identifying information (e.g., medical records) [95]. Another example is randomization of personalized identifiers over time for release of individual level data. Randomization although does not provide a formal guarantee, it is practical and simplistic and has been used to offer sufficient privacy (e.g., COVID-19 Alert app for contact tracing by Apple and Google [5]).
- Release of data based on *cryptographic* techniques (e.g., secure multi-party computation [55], homomorphic encryption [50], etc.) where the goal is to control and minimize the information the adversary can obtain.
- Release of *differential private* [40] data. Differential privacy gives a mathematically rigorous worst-case bound on the maximum amount of information that can be learned about an individual's data from the output of a computation. It assumes a very strong attacker who knows about all but one record in the data. The privacy parameter ϵ is used to control the privacy level where lower ϵ means higher privacy.

The techniques to implement privacy in data management can be classified based on whether they allow *sharing aggregated information* versus *sharing individual records*. In this section, we will look at examples of techniques in first category. Access control policies which are used for controlling sharing of individual records will be looked in the following section. We also include below another possible classification of the previously described PETs based on properties that can influence a data management system design to enable their seamless integration [53]:

- *Stateful vs stateless*: Stateful PETs maintain a "state" that is shared between events and therefore past events can influence the way current events are processed. An example of a stateful PET is Randomization of personal identifiers in a sensor stream. Personal identifiers of users are replaced with randomly generated identifiers after every t time unit. During each window of size t, all the events capturing a single user use the same randomly generated identifier (i.e., state). In a stateless PET, past events do not influence the current events thus does not require maintenance of state.
- *Blocking vs non-blocking*: Blocking PETs typically require processing the entire input before an output can be delivered. An example of a blocking PET is a Differentially Private Laplace mechanism for releasing aggregate statistics over time. A non-blocking PET does not need to wait of all the tuple in a window to arrive before applying the PET. The tuples are processed as they come. An example of a non-blocking PET is Randomization of personal identifiers in a sensor stream.
- *Negotiable vs Non-negotiable*: A data product using PET which is non-negotiable is made available to service providers with a fixed privacy model. An example is a data product consisting of a deferentially private sensor stream based on a fixed ϵ. Such data products come with strong privacy guarantees but may not provide any bound on utility of the data product to the end-application/service. A different model is to support data products with negotiable PETs. In such a case, the data product may come with

strong privacy properties, but the service provider is capable of negotiating with the data owner about the level of noise/anonymization added to the data product if the data product is unable to meet the utility goals of the applications. Negotiable PETs, specially, in the context of Differential Privacy, is new emerging concept - traditionally, Differential Privacy has explored algorithms to optimize utility with strict privacy constraints. Negotiable privacy offer more flexibility by making utility more central to the way data products are produced for sharing. In particular, it shifts the privacy-utility trade-off problem from optimizing utility given privacy constraint to that of optimizing privacy (i.e., minimizing the privacy loss) given utility constraint [48, 52]. In negotiable PET, the service provider can request for data product with sliding scale privacy based on the demonstration of need for accuracy.

The adaptation of a specific PET to a data management system also depends on the application context. For instance, in the case of IoT streaming sensor data, several PETs have been specifically designed based on the previous mechanisms [23, 85, 68, 25, 51].

2.4 Access Control and User Privacy Policies

The focus of this thesis is on *Access Control Policies* which control who has access to what data and under what conditions (see Figure 2.4 for a high-level architecture of a system based on access control). Access Control Policies, or simply *policies* from now on, have been around since computers could be accessed by more than one person at a time with time-sharing. In the following, we summarize the history of policies and their usage in DBMS[4].

David Elliott Bell and Len La Padula laid the foundations for Access Control Models in 1972-1975 with the Bell-La Padula model [14]. They defined objects (i.e., the generalization

[4]This summary was inspired by the excellent blog post "In Search for Perfect Access Control" (https://goteleport.com/blog/access-controls/).

of all things that could hold information), subjects (i.e., users accessing or requesting access to the objects), and access modes (i.e., read, write, append, and execute). Subject's access to objects was restricted by the access modes in the form of access triples {*subject, object, access*}. Both subjects and objects were assigned security levels and a subject could only access an object if their security level *dominated* that of the object. To prevent unwanted disclosures of classified items, the Bell-La Padula model only allowed *trusted subjects* to set security levels. This initial model is therefore called the *Mandatory Access Control (MAC)* where individual users have no capabilities to set up access levels on any of the objects. Later on, this model was changed so that users who are defined as owners of an object could define the access control permissions for that object. This model of *Discretionary Access Control (DAC)* was first used in Multics in the form of *Access Control Lists (ACLs)* and later on borrowed over to Unix and still used to this day.

Few decades later, David F. Ferraiolo and D. Richard Kuhn identified the following shortcoming in the Bell-LaPadula model [46]:

> *"In many organizations, the end users do not "own" the information for which they are allowed access. For these organizations, the corporation or agency is the actual "owner" of system objects as well as the programs that process it. Control is often based on employee functions rather than data ownership."*

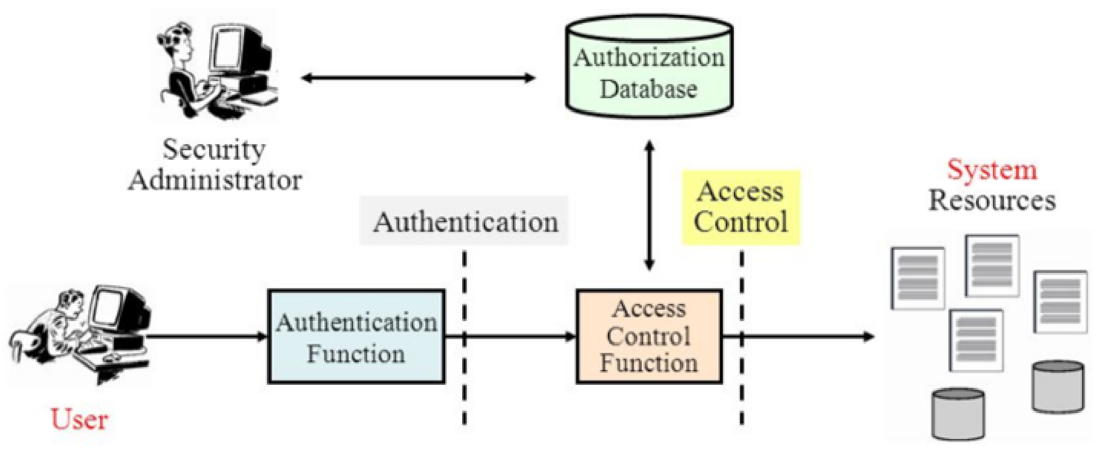
They proposed the *Role-Based Access Control (RBAC)* [46] model where subjects could have many roles and access control permissions are assigned based on these. To decide whether to allow or deny access to an object, the system looks at the role of the subject.

Almost at the same time RBAC was proposed, work on developing a more general access control model which could not only decide access based on the subject attribute (e.g., role) but also based on the object attributes and even contextual conditions. This led to *Attribute-Based Access Control (ABAC)* [61] which is also referred to as policy-based access control.

In ABAC, policies are decoupled from software or users. The policy model used in thesis is modelled on ABAC because of its flexibility.

Based on ABAC, one can define more detailed policies, or Fine-Grained Access Control (FGAC) policies [35]. Compared to more coarse-level policies, FGAC policies support the definition of diverse and detailed conditions. For instance, in the IoT domain, FGAC policies would enable the definition of different contextual parameters in addition the parameters related to the specific and heterogeneous sensor data.

2.5 Access Control in DBMS



"Access control: principle and practice." Sandhu RS et.al., IEEE Communications Magazine 1994

Figure 2.4: Access Control mechanism.

Today, database management systems (DBMSs) implement the above access control models, and in particular FGAC which is used in this thesis, by one of two mechanisms [18]: 1) *Policy as schema* and 2) *Policy as data.*

In *Policy as schema*, access control policies are expressed as authorization views [89]. The policies specified at table, view, column level are used to construct the views that are used in query answering. Authorized views reduced the need for large number of views by introducing parameters in view definition which are instantiated at query time [89]. In our scenarios with large number of complex dynamic policies, even with authorized views, a large number of views with complex view definition will have to created and maintained. Oracle Virtual Private Database (VPD) [73] is a database implementation of this approach in which authorized views are implemented by attaching policy functions to views or tables which suffers from the same problem.

On the other hand in *Policy as data*, the DBMS rewrites the query and executes it against the database [93]. Policies are stored in tables, just like data. The DBMS rewrites queries to include the policy predicates prior to execution [8, 21, 28, 30]. This mechanism allows users to express more fine-grained policies compared to views. Hippocratic [8] database pioneered representing fine-grained policies as data in the form of policy tables and rewriting queries to implement policy based data access but this mechanism can be easily supported by today's DBMSs. An excellent survey of access control mechanisms for databases can be found in [18].

Chapter 3

A Policy-based Privacy-by-Design Framework for IoT Smart Spaces

"Not unnaturally, many elevators imbued with intelligence and precognition became terribly frustrated with the mindless business of going up and down, up and down, experimented briefly with the notion of going sideways, as a sort of existential protest, demanded participation in the decision-making process and finally took to squatting in basements sulking."
Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

From smart cars to smart buildings and from activity bracelets to smart fridges, every object in our environment is increasingly being endowed with sensing, computing, communication, and actuation functionalities. This rapid transformation of the world we live in is opening the door to many potential benefits. One such domain where the IoT is opening the door to potential benefits is that of *smart buildings*. Here, traditional HVAC (heating, ventilating, and air conditioning) systems are being enhanced with functionalities that ties to beacons, presence sensors, cameras, and personal devices such as smartphones carried by the building's inhabitants. The reliance on the collection of data in smart buildings contradicts the expectations of privacy. Especially since in IoT environments, such as smart buildings, users are less likely to be aware of the technologies with which they might be interacting.

This chapter describes a framework for smart buildings which includes three main components. First, *IoT Resource Registries* (IRRs) which broadcast data collection policies and sharing practices of the IoT technologies with which users interact. Second, *IoT Assistants* which selectively notify users about the policies advertised by IRRs and configure any available privacy settings. Third, *privacy-aware smart buildings*, which publish building policies (e.g., through IRRs), receive the privacy settings of users (e.g., from IoTAs) and enforce them when collecting user data or sharing it with services[1].

To make this possible, it is important to have a language for expressing and communicating the space/building's policies as well as the privacy preferences of its inhabitants. While the existing privacy policy languages are expressive [98], they do not completely support capturing the policies of the building and preferences of its inhabitants regarding their data. Therefore, this chapter also presents an overview of a language which can be used for informing users about policies on what data is collected, how it will be safeguarded, what it will be used for, and the choices a user has with respect to these policies.

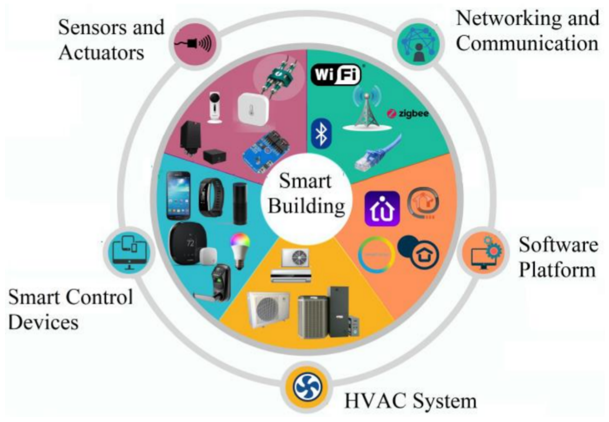
3.1 Overview of Smart Buildings

Building Management Systems (BMS) are cyber-physical systems that are used to manage buildings by monitoring different utility services. In general, a smart building includes a BMS along with the sensors/actuators, networking and communication, and smart services and devices (see Figure 3.1b[2]). As an example, Donald Bren Hall (DBH) is a 90 000+ square

[1]Note that, in order to differentiate the capture policies of the building against user-defined policies, this chapter refers to the former as building policies and to the latter as user preferences. The rest of the thesis focuses on user-defined preferences and hence the term policies will be used to refer to those.

[2]Figure 3.1b extracted from [84].

(a)



(b)

Figure 3.1: Main elements of a smart building and a sample smart building (Donald Bren Hall) at UC Irvine.

feet 6-story building at University of California, Irvine (UCI) equipped with a BMS (see Figure 3.1a). DBH is equipped with more than 40 surveillance cameras covering all the corridors and doors (for security purposes), 60 WiFi Access Points (AP) (for Internet connectivity), 200 Bluetooth beacons (for broadcasting information of interest to inhabitants), and 100 Power outlet meters (for monitoring energy usage).

3.1.1 Privacy Threats in Current Smart Building Scenarios

BMS capture a digital representation of a dynamically evolving building at any point in time for purposes such as comfort and security. But this representation might contain distinct patterns which can reveal the absence or presence of people and their activities, potentially resulting in the disclosure of data that people might not feel comfortable disclosing (e.g., where they go, what they do, when and with whom they spend time, whether they are healthy and more) [15].

For example, when a user connects to a WiFi AP in DBH, this event is logged for security purposes (the information logged includes the MAC address of the device and AP, and a timestamp) as part of the *building policy*. Using background knowledge (e.g., the location of the AP) it is possible to infer the real-time location of a user. Also, using simple heuristics (e.g., non-faculty staff arrive at 7 am and leave before 5 pm, graduate students generally leave the building late, and undergrads spend most of the time in classrooms), it is possible to infer whether a given user is a member of the staff or a student. Furthermore, by integrating this with publicly available information (e.g., schedules of professors and the courses they teach or event calendars), it would be possible to identify individuals. Some people may not object to such data collection, while others might. One challenge associated with privacy is that often not all users feel comfortable about the same data practices. Therefore, it is important to understand *user preferences* and *expectations* with respect to the information collected and used by a system like BMS [72, 86].

3.1.2 Privacy-Aware Smart Buildings

Adapting current building management systems to handle policies and user preferences is a complex task. The main components required involve a privacy-aware smart building management system and a mean for the user to define their privacy preferences. The former would need to, among others, capture and enforce privacy preferences expressed by the building's inhabitants. The latter would need to, among others, to explain the policies of the building wrt data capture, interact with the user to understand their privacy preferences, and communicate those to the privacy-aware smart building management system.

Chapter 6 will explain in more detail an implementation of both elements: 1) A privacy-aware smart building testbed (*TIPPERS* [75, 6]) which captures raw data from the different sensors in the building, processes higher-level semantic information from such data, and empowers development of different building services; 2) User *IoT Assistants* [4] which configure available privacy settings - whether automatically or via interactions with the user. In the rest of the chapter, the terms TIPPERS and IoTA will be used interchangeably with privacy-aware building management system and personal assistants, respectively.

3.1.3 User Interactions in Privacy-Aware Smart Buildings

Figure 3.2 outlines how a user (who will be referred to as *Mary* from now onward for ease of explanation) interacts with this infrastructure. In particular, consider a scenario where a building, DBH, is managed by a privacy-aware smart building management system (in this case TIPPERS) and users have personal assistants which handle their privacy preferences (in this case IoTA).

First, the building admin of DBH uses the smart building management system (such as TIPPERS) to define policies regarding the collection and management of data within the building (step (1) in Figure 3.2). Based on these policies, the different sensors in the building are actuated and data from them, some of which might be related to its inhabitants (step (2)), is captured and stored (step (3)). These policies are made publicly available through one or more IoT Resource Registries (step (4)). As Mary walks into the building carrying her smartphone with IoTA installed on it, the IoTA discovers available registries that pertain to resources in her vicinity and obtains machine-readable privacy policies detailing the practices of resources close to her location (step (5)). The IoTA displays summaries of relevant elements of these policies to the user (step (6)) by focusing on the elements of a policy that are important respect to the users privacy preferences. This is done using a model of Mary's privacy preferences learned over time. This might include information about those data collection and use of practices she cares to be informed about (step (7)). If a policy identifies the presence of settings, the IoTA can also use knowledge of Mary's privacy preferences to help configure these settings by communicating with TIPPERS (e.g., submitting requests to change settings) (step (8)). If a service later requests TIPPERS about Mary's location (step (9)), the request will be processed according to the settings communicated by Mary's

IoTA to TIPPERS (e.g., the request might be rejected, if Mary's IoTA requested to opt-out of location sharing; step (10)).

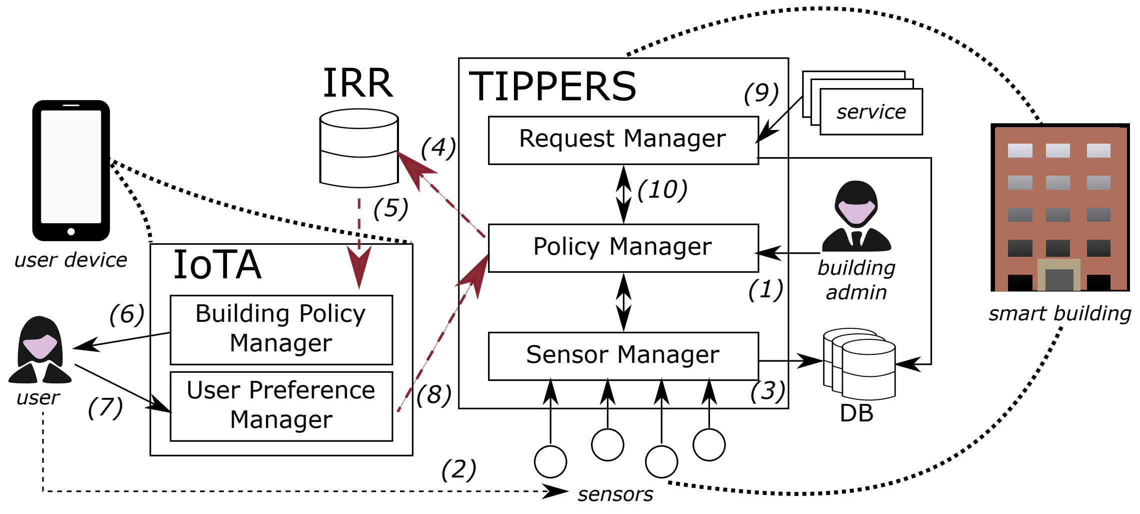


Figure 3.2: Interaction between privacy-aware smart building management system (TIPPERS), IoT Resource Registries (IRR) and IoT Assistants (IoTA).

To implement this interaction, a machine-readable policy language, as a mechanism to capture and communicate building policies of smart buildings to its inhabitants, is required. The policy language is used to convey users' preferences and settings to the smart building system by the personal assistant. In the interaction described above different elements could use the language to advertise building policies (step (4)), match them with the user preferences (step (5)), and communicate the matched user preferences to the building system (step (8)).

3.2 Facets of a Privacy-Aware Smart Building Infrastructure

Building policies and user preferences are important to ensure that a smart building system meets the privacy needs of its inhabitants. In the following, both building policies and user preferences are explained with examples.

3.2.1 Building Policies

A *building policy* states requirements for data collection and management set by the temporary or permanent owner. Building policies can be related to the infrastructure of the building, specific sensors deployed in the building or even events taking place inside the building. These policies (in most cases) have to be met completely by the other actors in the pervasive space. Here are some examples of building policies that can be entered into TIPPERS and advertised by the IRR.

- *Policy 1*: A facility manager sets the thermostat temperature of occupied rooms to 70°F to match the average comfort level of users.
- *Policy 2*: The building management system stores your location to locate you in case of emergency situations.
- *Policy 3*: A building administrator defines that either an ID card or fingerprint verification is needed to access meeting rooms.
- *Policy 4*: An event coordinator requires that details regarding an event are disclosed to registered participants only when they are nearby.

To implement these policies, they have to be translated into settings that change the state of sensors. For example to execute *Policy 1* it is necessary to *i*) make a request to motion sensors in each room to determine whether the room is occupied or not, *ii*) pull information from temperature sensors to determine whether the HVAC system has to be activated, and *iii*) change the settings of the HVAC system to increase or decrease the fan speed to adjust the temperature.

3.2.2 User Preferences

Building policies support building management but at the same time put user's privacy at risk. For example, using the data collected based on *Policy 1* it is possible to discover whether someone's office is occupied or not which in turn can be used to learn the occupant's working pattern. Therefore, in smart buildings, users should be able to express their privacy preferences regarding the data collected by the building.

A *user preference* is a representation of the user's expectation of how data pertaining to her should be managed by the pervasive space. These preferences might be partially or completely met depending on other policies and user preferences existing in the same space. Some examples of user preferences are:

- *Preference 1*: Do not share the occupancy status of my office in after-hours.
- *Preference 2*: Do not share my location with anyone.

Smart buildings such as DBH also provide services, built on top of the collected sensor data, to the inhabitants of the building. Two examples of such services operating at DBH are 1) *Smart Concierge* service, which helps users locate rooms, inhabitants and events in the building, 2) *Smart Meeting* service, which can help organize meetings effectively. These services take information from the user captured by the building (e.g., their current location) and return interesting information (e.g., nearest coffee machine). In addition to services provided by the building, there could be other third-party services running on top of the smart building management system. For example, a food delivery company can automatically locate and deliver food to building inhabitants during lunch time.

While using a service inside the building, a user can also specify their policies in the form of permissions allowed for the service. This is similar to how the permissions are managed in mobile apps. This allows a user to directly review what information the service requests and for what purpose. For the previously described services, possible user permissions could be:

- *Preference 3*: Allow *Concierge* access to my fine grained location for directions.
- *Preference 4*: Allow *Smart Meeting* access to the details of the meeting and its participants.

It is possible that user preferences conflict with the existing building policies (e.g., *Policy 2* and *Preference 2*). These conflicts should be detected by the smart building management system (e.g., with the help of a policy reasoner) which is in charge of enforcing the policies by resolving these conflicts while informing users about it through the personal privacy assistant.

3.3 Communicating Policies and Preferences

Building policies and user preferences have context specific requirements that need to be captured and communicated in a flexible manner. In this section, we first describe the various elements of our machine-readable policy. Second, we describe a high-level language schema that can be used to capture such policy.

For expressing a building policy, a semi-structured language would be the best fit as the user is cognizant of the IoT space itself. In the case of a user preference, the goal is to reduce privacy fatigue as much as possible and therefore a natural language interface or a privacy assistant like IoTA mentioned earlier would be more suitable.

3.3.1 Building Specific Policy Elements

There are different elements in a building that have to be represented in policies such as space, users, sensors and services. For the elements described below, we use existing ontologies if available.

1) Spatial Model/Environment includes information about infrastructure, such as buildings, floors, rooms, corridors, and is inherently hierarchical. The spatial model also supports operators such as “contained”, “neighboring”, and “overlap”. It is an approximation of the different properties of spatial entities of interest.

2) User Profile models the concept of people in the environment. Profiles can be based on groups (students, faculty, staff etc.) and share common properties (e.g., access permissions). A user can have multiple profiles which includes information such as department, affiliation, and office assignment in our sample scenario.

3) Sensor describes the entity which captures information about its environment. Each sensor has a sensor type and can produce a reading based on its type. Sensors of the same type can be organized into sensor subsystems. Examples of such subsystems are camera subsystem, beacon subsystem, and HVAC subsystem (modelled using the haystack[3] ontology and Semantic Sensor Network ontology [33]).

4) Settings of a sensor is a set of valid parameters associated with the sensor which determines its behavior (e.g., for a camera it could be the capture frequency or the resolution of the image). A sensor is actuated based on the parameters specified in its current settings. A sensor can have multiple settings dictated by its type.

5) Observation models the type of data captured by a sensor based on the type and settings associated with it. Each observation has a timestamp and a location (determined based on

[3]http://project-haystack.org

whether the sensor is mobile or fixed) associated with it.

6) Service Model describes the services that run on top of smart building systems and provide interesting information to the users. The service model captures meta-data about the service such as the developer (e.g., building owner or third party), permissions to sensors, and observations. This model also describes details about the service itself such as the information returned or functionality provided.

3.3.2 Privacy Specific Policy Elements

While building and sensor specific models can capture information about different entities, there is a need for describing the data collection practices in a building from the perspective of a user. Peppet [82] analyzed privacy policies of companies that manufacture IoT devices and concluded that through these policies, users not only want to be informed about what data is collected by which devices and for what purposes, but also about the granularity of data collection (whether or not it is aggregated or anonymized) and with whom the data is shared. Based on above requirements, we introduce the following policy elements to model a user's privacy settings.

1) Context describes meta information about the building and the BMS that point users to general information (e.g., who is responsible for data collection in a building, where are sensors located, and whom to contact when it comes to questions regarding the policy). This meta information can also contain a general description of data security and ownership of information which are relevant to the user.

2) Data collected and inferred. While the observation model captures information about the data collected, a user might be more interested in knowing what can be inferred from the collected data. Therefore, it is important to specify the abstract information that can be inferred from an observation captured by a sensor. For example, to model the occupancy of a room, it would be better to describe it as “if a room is occupied by anyone” compared to an observation model which might only have information such as “images from camera”, “logs from WiFi APs”, etc. Data collection description also contains information about the granularity of the data collected as granularity can directly impact the capability of inference.

3) Purpose models the requirement of data collection which is closely related to a service that uses this data. In a BMS, some data collections such as temperature monitoring serves a straightforward purpose for setting the thermostat, but for other data collections such as the information of connecting to WiFi APs can be used for different purposes (e.g., for logging as well as to track the location of a particular MAC address). We are currently working on a taxonomy to model purpose which includes information about whether or not the data is shared (e.g., with law enforcement officers for security purposes) and for how long it will be stored (i.e., retention).

3.3.3 Overview of the Language Schema

Based on the aforementioned elements, we are designing a language schema that is capable of capturing both building policies and user preferences. In the following we give an overview of the language by representing some of the examples from Section 3.2. We use a JSON-Schema v4[4] for the representation. We choose JSON over other formats mainly because of the rapid adoption of JSON-based REST APIs.

Figure 3.3 shows how *Policy 2* (“Location tracking for emergency response”) can be expressed using the language. The first part of the language expresses the general information about the location and sensor type (in this case location is DBH at UCI with WiFi APs being the sensors) whereas the second part expresses the data collection purpose (emergency response),

[4]http://json-schema.org

data type, and retention period of the data itself.

```
{"resources": [{
  "info": { "name": "Location tracking in DBH" },
  "context": {
    "location": {
      "spatial": {
        "name": "Donald Bren Hall",
        "type": "Building"
        },
        "location_owner": {
          "name": "UCI",
          "human_description": {
            "more_info": "http://ics.uci.edu"
          }}},
    "sensor": {
      "type": "WiFi Access Point",
      "description": "Installed inside the building and covers rooms and corridors"
  }},
  "purpose": {
    "emergency response": {
      "description": "Location is stored continuously"
  }}
  "observations": [{
    "name": "MAC address of the device",
    "description": "If your device is connected to a WiFi Access Point in DBH, its MAC
        address is stored"
   }],
  "retention": {
    "duration": "P6M"}}]}
```

Figure 3.3: Policy related to data collection inside DBH.

In case of the policies related to services such as the *Smart Concierge* can be expressed as shown in Figure 3.4. The first part describes the information required by the service and the second part shows the purpose of collecting this information.

```
{"observations": [{
  "name": "wifi_access_point",
  "description": "Whenever one of your devices connects to the DBH WiFi its MAC address is
      stored"
  }, {
  "name": "bluetooth_beacon",
  "description": "When you have Concierge installed and your bluetooth senses a beacon, the
      room you are in is stored"}],
"purpose": {
  "providing_service": {
    "description": "Your location data is used to give you directions around the Bren Hall."
        },
  "service_id": "Concierge"}}
```

Figure 3.4: Policy related to a service in the building.

Concerning user's preference settings, the language can express choices related to policies and services. In the context of *Smart Concierge* service, Figure 3.5 shows options for the different granularities at which location data can be collected. Thus, if a user is comfortable with sharing fine-grained location data with the Concierge service for directions then our language can capture such *Preference*.

```
{"settings": [
  {"select": [
    {"description": "fine grained location sensing",
     "on": "http://tippers/user/concierge?beacon=opt-in&wifi=opt-in"},
    {"description": "coarse grained location sensing",
     "on": "http://tippers/user/concierge?beacon=opt-out&wifi=opt-in"},
    {"description": "No location sensing",
     "on": "http://tippers/user/concierge?beacon=opt-out&wifi=opt-out"}]}
```

Figure 3.5: Privacy settings available.

3.4 Conclusions and Challenges

We presented a template for future smart buildings which includes privacy-aware building management systems and IOT assistants and can give users better control over the information that buildings collect about them. We described the requirements and elements of a machine-readable language required for this collaboration, which can represent building policies and user preferences. However, to make this vision of a building that takes user privacy into account a reality, many challenges have to be tackled.

First, challenges associated with the design of IoT Assistants, which are out of the scope of this thesis. While an IoT Assistant can help users in understanding the policies broadcast by the smart building, identifying which privacy practices are most relevant to users is important [54, 86]. This requires a unified way to discover IoT technologies through IRRs and we envision that the setup of IRRs can be automated (e.g. by leveraging Manufacturer Usage Descriptions [42]). An IoTA could make recommendations to users following an approach similar to the work done by Liu et al. [72] for mobile applications. For such a mechanism to work correctly, the assistant requires labeled data over a period of time to decipher the patterns in a user's behavior and represent them as preferences for the user. Therefore, the challenges include when and how to notify a user and how to obtain user feedback without inducing user fatigue.

Second, challenges associated with the development of privacy-aware smart buildings/spaces, which is the focus of this thesis. The high-level policies and preferences have to mapped into appropriate entities in the building space before their enforcement. This mapping determines the *where* (at devices or BMS), *when* (during capture, storage, processing, or sharing) and *how* (accept/deny data access or add noise) these policies and preferences should be enforced on the user data. The possibilities for customization in this mapping, and thus expressibility of policies and preferences, are decided by the capabilities of privacy-aware buildings. With large number of users, services, sensors, policies, and preferences the cost of enforcement can be large enough to be prohibitive in any real setting. Additionally, it is complex to determine the relation between policies expressed on higher-level, more semantically meaningful, concepts that people understand and the raw sensor data captured by the building.

3.5 Discussion

In this chapter, we developed a framework for IoT spaces with the right mechanisms to empower users with control over capture and sharing of their data. Given a sensor and a sensor event, we identify a user and their policy and either ensure that sensor data is collected or not collected or shared/not shared based on the policy.

We implicitly assume that each sensor event corresponds to a user or is an information about a user. Under this assumption, the framework allows user to opt-in/opt-out of data collection. However, associating a user with their data may not be straight forward for different sensors and their data. We are developing a *Data Subject Association Manager* to do this as part of our work in PE-IoT (explained in Chapter 6). This mechanism creates the association between data records in the sensor stream and a data subject. The implementation of a Data Subject Association Manager depends on the specific sensor stream it is handling and therefore there will be as many of these managers as there are different sensor data types. For example, in WiFi association data stream, the MAC address captured is mapped to a corresponding data subject by looking up the device registry.

Furthermore, depending upon the use case of IoT applications, there may be need for a fine-grained policies than described in this chapter. In such scenarios, users do not just control the capture of data but also how the data is shared with services and other users after it is captured. In Chapter 4, we explain such a scenario – Classroom attendance – in detail and the fine grained policy model required for specification.

The remainder of this thesis focuses on dealing with the challenges associated with the supporting fine-grained access control policies in data management systems.

Chapter 4

Scalable Enforcement of Fine-Grained Access Control Policies

"Query optimization is not rocket science. When you flunk out of query optimization, we make you go build rockets."
David DeWitt, *PASS Summit 2010*

In IoT domains, with large number of sensors and data collection at unprecedented rates, the number of of user-defined fine-grained policies is going to be in the order of tens of thousands. When it comes to enforcement of these large number of policies, today's data management systems are not able to efficiently handle the large number of checks required at the time of answering queries. Supporting such fine grained policies raises several significant challenges that are beginning to attract research attention. This chapter addresses one such challenge: scaling enforcement of access control policies in the context of database query processing when the set of policies becomes a dominant factor/bottleneck in the computation due to their large number. This has been highlighted as one of the open challenges for Big Data management systems in recent surveys such as [32].

In modern data management systems, data is dynamically captured from sensors and shared with people via queries based on user-specified access control policies. We describe a motivating use case of a smart campus in Section 4.1 which shows that data involved in processing a simple analytical query might require checking against hundreds to thousands of access control policies. Enforcing that many access control policies in real-time during query execution is well beyond database systems today. While our example and motivation is derived from the smart space and IoT setting, the need for such query processing with a large number of policies also applies to many other domains. This applicability will only increase as emerging legislation such as GDPR empowers users to control their data.

As we described in Section 2.4, existing DBMS support Fine-Grained Access Control (FGAC) mechanisms by performing a query rewrite [93]. This is done by appending policies as predicates to the `WHERE` clause of the original query. However, they are limited in the complexity of applications they can support due to the increased cost of query execution when the rewriting includes a large number of policies. Thus, scalable access control-driven query execution presents a novel challenge. We evaluated the existing approach of query rewrite on top of a relational DBMS (MySQL) with two different queries from a IoT Benchmark called SmartBench [57]. The results are shown in Figure 4.1. The first query (on the left) is a real time query from Smart Bench which retrieves the data belonging to an individual user. We perform this experiment for different users and DBMS has to evaluate between 100 to 350 policies depending on the user. In the second query (on the right), it is an analytical query where we progressively decrease the selectivity of the query and thus increasing the number of policies to be evaluated. In both queries, policy evaluation overhead increases linearly with number of policies.

In this chapter we propose Sieve, a general purpose middleware to support access control in DBMSs that enables them to scale query processing with large number of access control policies. It exploits a variety of features (index support, UDFs, hints) supported by modern

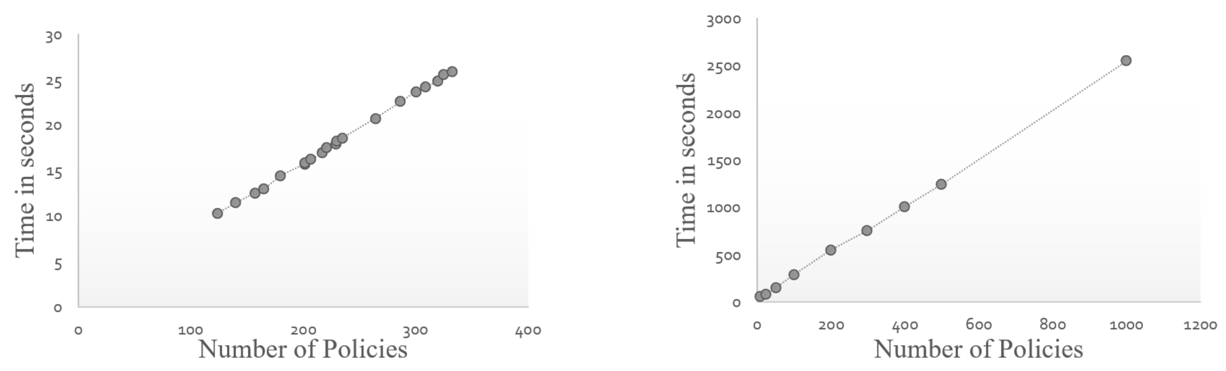


Figure 4.1: Policy Evaluation overhead vs. Number of Policies.

DBMSs to scale to a large number of policies. A middleware implementation, layered on top of an existing DBMS, allows us to test Sieve independent of the specific DBMS used. This is particularly useful in our case (motivated by IoT) since different systems offer different trade-offs in IoT settings as highlighted in [57]. The comparative simplicity of implementing the technique in middleware enables us to explore the efficacy of different ideas instead of being constrained by the design choice of a specific system, as shown in previous work such as [29]. Sieve intercepts the query by a user and rewrites it with relevant policies which is then executed by the DBMS which then returns the policy enforced query results to the user (see Figure 4.2).

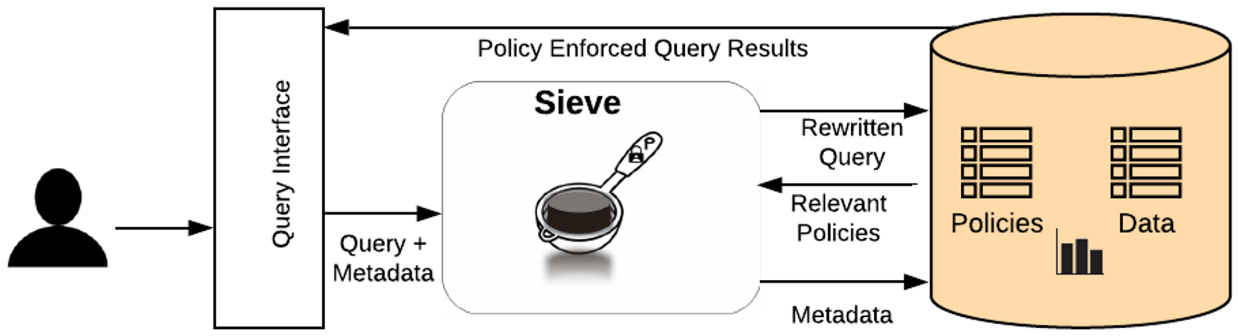


Figure 4.2: Overview of Sieve.

Sieve incorporates two distinct strategies to reduce overhead: reducing the number of tuples that have to be checked against complex policy expressions and reducing the number of policies that need to be checked against each tuple. First, given a set of policies, it uses them to generate a set of *guarded expressions* that are chosen carefully to exploit the best existing database indexes, thus reducing the number of tuples against which the complete and complex policy expression must be checked. This strategy is inspired by the technique for predicate simplification to exploit indices developed in [24]. Second, Sieve reduces the overhead of dynamically checking policies during query processing by filtering policies that must be checked for a given tuple by exploiting the context present in the tuple (e.g., user/owner associated with the tuple) and the query metadata (e.g., the person posing the query –i.e., querier– or their purpose). We define a policy evaluation operator Δ for this task and present an implementation as a User Defined Function (UDF).

Sieve combines the above two strategies in a single framework to reduce the overhead of policy checking during query execution. Thus, Sieve adaptively chooses the best strategy possible given the specific query and policies defined for that querier based on a cost model estimation. We evaluate the performance of Sieve using a real WiFi connectivity dataset captured in our building at UC Irvine, including connectivity patterns of over 40K unique devices/individuals. On this real dataset, we generate a synthetic set of policies that such individuals could have defined to control access to their data by others. We also test the performance of our system on a synthetic dataset based on a smart mall where connectivity data of devices are logged inside shops in the mall. Our results highlight the benefit of Sieve-generated query rewrite when compared to the traditional query rewrite approach for access control when processing different queries. Additionally, we perform these experiments on two different DBMSs, MySQL and PostgreSQL, showcasing Sieve's abilities as a middleware.

The rest of the chapter is organized as follows. Section 4.1 presents a case study of a real IoT deployment, with a large set of access control policies defined. Section 4.2 reviews related work followed by Section 4.3 formalizes the query and policy model, and the access control semantics used by Sieve. We also describe an overview of the approach used by Sieve with an outline of two different strategies. Section 4.5 presents an algorithmic solution for the first strategy i.e., to generate appropriate *guarded expressions*. Section 4.6 describes the details of the Sieve generated query rewrite along with various optimization techniques used. Section 4.7 describes how Sieve deals with dynamic scenarios in which policies are continuously inserted into the database. Section 4.8 presents the experimental evaluation using two different datasets and two different DBMSs. Finally, Section 4.9 presents a discussion.

4.1 Case Study

We present a case study based on a smart campus setting where there are a large number of FGAC policies specified by users for their collected data. We consider a motivating application wherein an academic campus supports variety of smart data services such as real-time queue size monitoring in different food courts, occupancy analysis to understand building usage (e.g., room occupancy as a function of time and events, determining how space organization impacts interactions amongst occupants, etc.), or automating class attendance and understanding correlations between attendance and grades [60]. While such solutions present interesting benefits, such as improving student performance [60] and better space utilization, there are privacy challenges [80] in the management of such data. This case study is based on our own experience building a smart campus with variety of applications ranging from real-time services to offline analysis over the past 4 years. The deployed system, entitled TIPPERS [75], is in daily use in several buildings in our UC Irvine campus[1]. TIPPERS at our campus captures connectivity events (i.e., logs of the connection of devices to WiFi APs) that can be used, among other purposes, to analyze the location of individuals to provide them with services.

We use the UC Irvine campus, with the various entities and relationships presented in Figure 4.3 (along with the expected number of members in brackets), as a use case. Consider

[1]More information about the system and the applications supported can be found at [6]

a professor in the campus posing the following analytical query to evaluate the correlation between regular attendance in her class vs. student performance at the end of the semester:

```
StudentPerf(WifiDataset, Enrollment, Grades)=
(SELECT student, grade, sum(attended)
 FROM (
  SELECT W.owner AS student, W.ts-date AS date,
       count(*)/count(*) AS attended
  FROM WiFiDataset AS W, Enrollment AS E
  WHERE E.class="CS101" AND E.student=W.owner AND W.ts-time
     between "9am" AND "10am" AND W.ts-date between "9/25/19"
     AND "12/12/19" AND W.wifiAP="1200"
  GROUP BY W.owner, W.ts-date) AS T, Grades AS G
 WHERE T.student=G.student
 GROUP BY T.student)
```

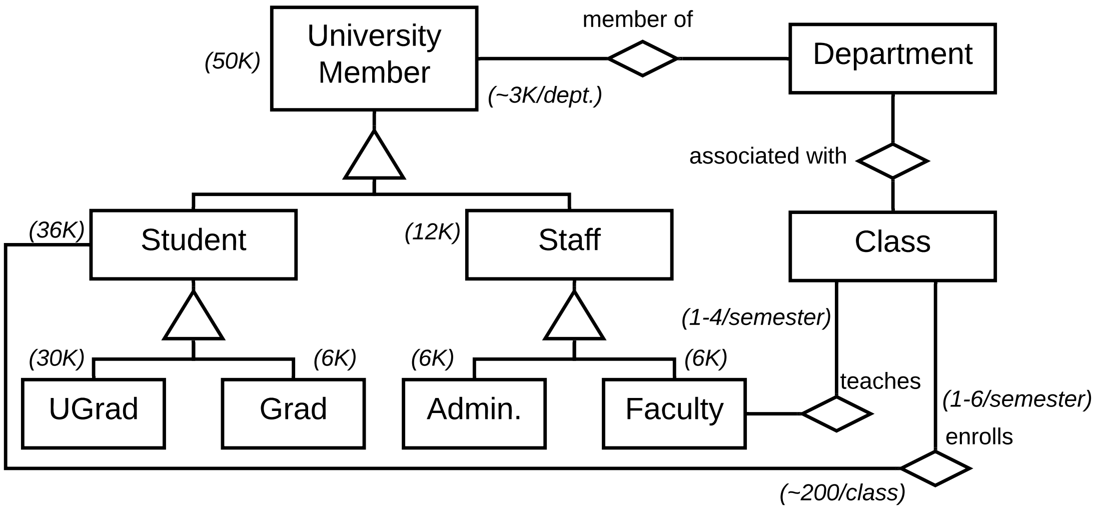


Figure 4.3: Entities and relationships in a Smart Campus Scenario.

Let us consider that students define polices to allow/deny access to their connectivity data to others (e.g., to faculty) in certain situations (e.g., given geospatial context, or at certain times, for certain purposes). Given the different contextual attributes of the scenario, such policies might be complex. For instance, a student might be fine with sharing her data with a professor during the time of class (i.e., 9am to 10am on Mondays, 11am-12pm on Thursdays) if she is connected to the AP in the classroom (i.e., WiFiAP 1200 on Mondays, WiFiAP 4011 on Thursdays) and if this information is going to be used only for attendance. Similarly, a professor could allow students of her class to access her connectivity events during tutoring hours if they have attended at least 75% of the classes so far.

Given the number of users in the scenario (ranging from 200 per class to 50K at the campus level), the pieces of data captured by TIPPERS in the smart campus (e.g., connectivity data, video, audio, etc.), and the different contextual control options available, one can expect a considerable number of policies being defined. Continuing with the classroom example, let us assume that within the students there exist different privacy profiles (as studied in the mobile world by Lin et al. [69]). Using the distribution of users by profile from [69][2], consider that 20% of the students might have a common default policy ("unconcerned" group [69]), 18% may want to define their own precise policies ("advance users"), and the rest will depend on the situation (for which we consider, conservatively, 2/3 to be "unconcerned" and 1/3 "advance"). Let us focus on a single data type captured in this analysis (i.e., connectivity data), time and location as control options (as explained before), and policies defined by a given user at the group-level (and not at the individual-level, which will even further increase the number of policies). Such profiles in the example correspond to students, faculty, administrative staff, etc. as shown in the figure.

Consider that there exists a default policy (see Figure 4.4) for each person p_i belonging to a specific group (e.g., student) that restricts/allows access to other individuals, p_j belonging to the same or different groups. Such default policies control access to the data of "unconcerned users". Advanced users, on the other hand, would create policies of their own based

[2]Note that the percentages of different profiles could be different in our setting compared to that studied in [69] given different context. The numbers above are solely for the purpose of motivating the need for databases to support large number of policies.

on context such as space and time (let us assume that each of them create one policy in addition to the default policy per group (see Figure 4.5). In the classroom setting, with 200 students, 120 "unconcerned" students will have associated 2 default policies, one per group (i.e., students, faculty), Likewise, 80 advanced users will have 4 refined policies, two per group. Thus, even under these conservative assumptions and limited number of profiles, connectivity data from WiFi APs will be subject to adherence of 560 policies. Being less conservative we can assume that advance users define two additional policy per group which will increase the number of policies to 880, or 1.2K (with three additional policies per group).

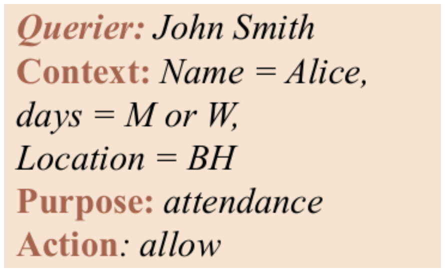

Querier: *John Smith*
Context: *Name = Alice,*
days = M or W,
Location = BH
Purpose: *attendance*
Action: *allow*

Figure 4.4: Default policy with smaller number of object and querier conditions

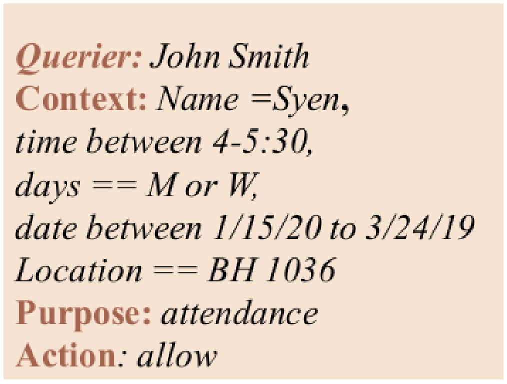

Querier: *John Smith*
Context: *Name =Syen,*
time between 4-5:30,
days == M or W,
date between 1/15/20 to 3/24/19
Location == BH 1036
Purpose: *attendance*
Action: *allow*

Figure 4.5: Advanced policy

Given the above policies for a single class, if students take 1-6 classes and faculty teach 1-4 classes per semester, a query to analyze students attendance listed above with performance over classes a professor taught over the year would be 3.3K (560 policies/class X 2 classes/quarter X 3 quarters/year) to 7.2K (considering our 1.2K policies/class estimation). It would be even worse if we consider now that the Chair of the department wishes to run a similar query for all the classes taught at the department since now the number of policies, and hence number of predicates in the `WHERE` clause will increase further. This query, given the list of over 100 courses offered by the department, would involve from 56K to 120K policies only taking into account the policies defined by the students (faculty members could had their own policies too).

The case study above motivates the requirements for emerging domains, such as smart spaces and IoT, on scalable access control mechanisms for large policy sets that the DBMS must support. While our example and motivation is derived from the smart space and IoT setting, the need for such query processing with a large number of policies applies to many other domains. Especially, as argued in the introduction, for emerging legislatures such as GDPR that empower users to control their data. Additionally, a recent survey on future trends for access control and Big Data systems made a similar observation about the open challenge to scale policy enforcement to a large number of policies [32].

4.2 Related Work

Using the context from case study, we review access control strategies in the literature and show that they fall short when enforcing large policy sets. As discussed in the comprehensive survey of access control in databases in [18], techniques to support FGAC can be broadly classified as based on views (e.g.,authorization views [89] and Oracle Virtual Private Database [73]) or based on storing policies in the form of data (e.g., Hippocratic databases [8] and the follow up work [67, 7]). In either of these approaches, input queries are rewritten to filter out tuples for which the querier does not have access permission. The view-based approach would be infeasible given the potentially large number of queriers/purposes which would result in creating and maintaining materialized views for each of them. In the policy-as-data based approach, the enforcement results on computationally expensive query processing. This is because the rewrite is done by adding conditions to the query's `WHERE` clause as ⟨`query predicate`⟩ `AND (`P_1 `OR ... OR` P_n`)` (where each P_i above refers to the set of predicates in each policy) or by using case-statement and outer join. In a situation like the one in our use case study, it results in appending hundreds of policy conditions to the query in a disjunctive normal form which adds significant overheads. Both strategies currently do not scale to scenarios with large number of policies.

Other approaches, such as [22], have proposed augmenting tuples with the purpose for which they can be accessed. This reduces the overheads at query time and as policy checking could be performed at data ingestion. Such pre-processing based approaches have significant limitations in the context where there are large number of fine-grained polices such as in the context that motivates our work. Determining permissions for individuals and encoding them as columns or multiple rows can result in exorbitant overhead during ingestion, specially when data rates are high (e.g., hundreds of sensor observations per second). Additionally, pre-processing efforts might be wasted for those tuples that are not queried frequently or at all. Other limitations include: 1) Impossibility of pre-processing policy predicates that depend on query context or information that is not known at that time of insertion; and 2) Difficulty to deal with dynamic policies which can be updated/revoked/inserted at any time (thus requiring processing tuples already inserted when policies change). Recent work [27, 29], that performs some pre-processing for access control enforcement, limits pre-processing to policies explicitly defined to restrict user's access to certain type of queries or to certain tables. The checking/enforcement of FGAC at tuple level is deferred to query-time and enforced through query rewriting as is the case in our paper.

Several research efforts have focused on implementing access control in the context of the IoT and smart spaces. In [77], the authors propose an approach for policy evaluation on streaming sensor data punctuated with access control policies. Their approach does not handle analytical queries with policies on the arriving data. Additionally the implementation of their approach requires significant modification to existing DBMS to make different operators *security-aware* for a large number of policies. In [31], the authors proposed a new architecture based on MQTT for IoT ecosystems. However, like [77] the focus of this work is not on managing large number of policies at run time and hence, they would experience the same issues highlighted for traditional query rewrite strategies.

4.3 Modelling Access Control Policies

We describe our modeling of the fundamental entities in policy-driven data processing: data, query, and policies. Using these, we describe the access control semantics used in this paper. We finish the section with a sketch of the approach followed by Sieve to speed up policy enforcement. We have summarized frequently used notations in Table 4.1 for perusal.

Table 4.1: Frequently used notations.

Notation	*Definition*
$\mathcal{D}$	Database
$i_i \in \mathcal{I}$	Index and set of indexes in $\mathcal{D}$
$r_i \in \mathcal{R}$	Relation and set of relations in $\mathcal{D}$
$u_k \in \mathcal{U}$	User and set of users in $\mathcal{D}$
$t_j \in \mathcal{T}$; $\mathcal{T}_{r_i}$; $\mathcal{T}_{Q_i}$; $\mathcal{T}_{p_l}$	Tuple and set of tuples: in $\mathcal{D}$; required to compute Q_i; controlled by p_l
$group(u_k)$	Groups u_k is part of
Q_i; $\mathtt{QM}^i$	Query; Metadata of Q_i
$p_l \in \mathcal{P}$; $\mathcal{P}_{\mathtt{QM}^i}$	Access control policy and set of policies in $\mathcal{D}$; set of policies related to a query given its metadata
$\mathtt{oc}^l_i \in \mathtt{OC}^l$;$\mathtt{qc}^l_i \in \mathtt{QC}^l$;$\mathtt{AC}^l$	Object conditions; querier conditions; action of p_l
$\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{\vert\mathcal{P}\vert}$	*Policy expression* of $\mathcal{P}$
$\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$	*Guarded policy expression* of $\mathcal{P}$ (DNF of guarded expressions)
$G_i = \mathtt{oc}^i_g \wedge \mathcal{P}_{G_i}$	*Guarded expression* consisting of *guard* ($\mathtt{oc}^i_g$) and its *policy partition* ($\mathcal{P}_{G_i}$)
$\mathcal{CG}$	*Candidate guards* for $\mathcal{E}(\mathcal{P})$
$eval(exp, t_t)$	Function which evaluates a tuple t_t against a expression *exp*
$\Delta(G_i, \mathtt{QM}^i, t_t)$	Ppolicy evaluation operator
$\rho(pred)$	Cardinality of a predicate
c_e	Cost of evaluating a tuple against the set policies
c_r	Cost of reading a tuple using an index

4.3.1 Data Model

Let us consider a database $\mathcal{D}$ consisting of a set of relations $\mathcal{R}$, a set of data tuples $\mathcal{T}$, a set of indexes $\mathcal{I}$, and set of users $\mathcal{U}$. $\mathcal{T}_{r_i}$ represents the set of tuples in the relation $r_i \in \mathcal{R}$. Users are organized in collections or *groups*, which are hierarchical (i.e., a group can be subsumed by another). For example, the group of undergraduate students is subsumed by the group of students. Each user might belong to multiple groups and we define the method *group(u_k)* which returns the set of groups u_k is member of. Each data tuple $t_j \in \mathcal{T}$ belongs to a $u_k \in \mathcal{U}$ or a group whose access control policies restrict/grant access over that tuple to other users. We assume that for each data tuple $t_j \in \mathcal{T}$ there exists an owner $u_k \in \mathcal{U}$ who owns it, whose access control policies restrict/grant access over that tuple to other users (the ownership can be also shared by users within a group).

This ownership is explicitly stated in the tuple by using the attribute $r_i.owner$ that exists for all $r_i \in \mathcal{R}$ and that we assume is indexed (i.e., $\forall\ r_i \in \mathcal{R}\ \exists\ i_j \in \mathcal{I} \mid i_j$ is an index over the attribute $r_i.owner$). $\mathcal{T}_{u_k}$ represents the set of tuples owned by user u_k [3].

4.3.2 Query Model

The SELECT-FROM-WHERE query posed by a user u_k is denoted by Q_i and tuples in the relations in the FROM statement(s) of the query are denoted by $\mathcal{T}_{Q_i} = \bigcup_{i=1}^{n} \mathcal{T}_{r_i}$. In our model, we consider that queries have associated metadata $\texttt{QM}^i$ which consists of information about the querier and the context of the query. This way, we assume that for any given query Q_i, $\texttt{QM}^i$ contains the identity of the querier (i.e., $\texttt{QM}^i_{querier}$) as well as the purpose of the query (i.e., $\texttt{QM}^i_{purpose}$). In the example query in Section 4.1, $\texttt{QM}^i_{querier}$=“Prof.Smith” and $\texttt{QM}^i_{purpose}$=“Analytics”.

[3]By owner of the tuple above, we refer to the entity who can define policies on the tuple. In our example use case, where tuple corresponds to WiFi connectivity data, determining ownership is straightforward - data owner is the owner of the device being detected by the sensor. Ownership, in general, can be difficult to determine, specially when sensors capture data which is not directly linked to the identity of the user as mentioned in Chapter 3

4.3.3 Access Control Policy Model

A user specifies an access control policy (in the rest of the chapter we will refer to it simply as policy) to allow or to restrict access to certain data she owns, to certain users/groups under certain conditions. Let $\mathcal{P}$ be the set of policies defined over $\mathcal{D}$ such that $p_l \in \mathcal{P}$ is defined by a user u_k to control access to a set of data tuples in r_i. Let that set of tuples be $\mathcal{T}_{p_l}$ such that $\mathcal{T}_{p_l} \subseteq \mathcal{T}_{u_k} \cap \mathcal{T}_{r_i}$. We model such policy as $p_l = \langle \texttt{OC}^l, \texttt{QC}^l, \texttt{AC}^l \rangle$, where each element represents:

- **Object Conditions** ($\texttt{OC}^l$) are defined using a conjunctive boolean expression $\texttt{oc}_1^l \wedge \texttt{oc}_2^l \wedge ... \wedge \texttt{oc}_n^l$ which determines the access controlled data tuple(s). Each *object condition* ($\texttt{oc}_c^l$) is a boolean expression $\langle attr, op, val \rangle$ where *attr* is an attribute (or column) of r_i, *op* is a comparison operator (i.e., $=$, $!=$, $<$, $>$, $\geq$, $\leq$, `IN`, `NOT IN`, `ANY`, `ALL`), and *val* can be either: (1) A constant or a range of constants or (2) A derived value(s) defined in terms of the expensive operator (e.g., a user defined function to perform face recognition) or query on $\mathcal{D}$ that will obtain such values when evaluated. In this paper, we focus on the object conditions with values as constants. To represent boolean expressions involving a range defined by two comparison operators (e.g., $4 \leq a < 20$) we use the notation $\langle attr, op1, val1, op2, val2 \rangle$ (e.g., $\langle a, \geq, 4, <, 20 \rangle$). As an example, $\texttt{oc}_{owner}^l$ is an $\texttt{oc}_c^l \in \texttt{OC}^l$ such that $\texttt{oc}_c^l = \langle r_i.owner, =, u_k \rangle$ or $\texttt{oc}_c^l = \langle r_i.owner, =, group(u_k) \rangle$[4]..

- **Querier Conditions** ($\texttt{QC}^l$) identify the metadata attributes of the query to which the access control policy applies. $\texttt{QC}^l$ is a conjunctive boolean expression $\texttt{qc}_1^l \wedge \texttt{qc}_2^l \wedge \cdots \wedge \texttt{qc}_m^l$. Our model is inspired by the well studied Purpose Based Access Control (Pur-BAC) model [21] to define the querier conditions. Thus, we assume that each policy contains has at least two querier conditions such as $\texttt{qc}_{querier}^l = \langle \texttt{QM}_{querier}^i, =,$

[4]To simplify the notation we will represent the pair of object conditions of a policy used to represent a range (e.g., $\texttt{oc}_{s1}^1 = \langle a, >, 10 \rangle$ and $\texttt{oc}_{s1}^1 = \langle a, <, 65 \rangle$) as a single object condition $\texttt{oc}_s^1 = \langle a, [>,<], [10,65] \rangle$.

$u_k\rangle$ or $\texttt{qc}^l_{querier} = \langle \texttt{QM}^i_{querier}, =, \mathit{group(u_k)}\rangle$ (that defines either a user or group), and a $\texttt{qc}^l_{purpose} = \langle \texttt{QM}^i_{purpose}, =, \mathit{purpose}\rangle$ which models the intent/purpose of the querier (e.g., safety, commercial, social, convenience, specific applications on the scenario, or any [65]). Other pieces of querier context (such as the IP of the machine from where the querier posed the query, or the time of the day) can easily be added as querier conditions although in the rest of the paper we focus on the above mentioned querier conditions.

- **Policy Action** ($\texttt{AC}^l$) defines the enforcement operation, or *action*, which must be applied on any tuple $t_j \in \mathcal{T}_{p_l}$. We consider the default action, in the absence of an explicit policy allowing access to data, to be *deny*. Such a model is standard in systems that collect/manage user data. Hence, explicit access control actions associated with policies in our context are limited to *allow*.

Based on this policy model, we show two sample policies in the context of the motivating scenario explained before. First, we describe a policy with object conditions containing a constant value. This policy is defined by John to regulate access to his connectivity data to Prof. Smith only if he is located in the classroom and for the purpose of class attendance as follows:

```
⟨[W.owner = John ∧ W.ts-time ≥ 09:00 ∧ W.ts-time ≤ 10:00 ∧ W.wifiAP
 = 1200], [Prof. Smith ∧ Attendance Control], allow⟩
```

Second, we describe the same policy with an object condition derived from a query to express that John wants to allow access to his location data only when he is with Prof. Smith. The object condition is updated as:

```
[W.owner = John ∧ W.wifiAP = (SELECT W2.wifiAP FROM WifiDataset
AS W2 WHERE W2.ts-time = W.ts-time AND W2.owner = "Prof.Smith")]
```

As policy-based access control implementations typically deny access in absence of a policy explicitly allowing access to data [5]. Moreover, if a user expresses a policy with a deny action (e.g., to limit the scope/coverage of an allow policy), we can translate it into the explicitly listed allow policies. For instance, given an allow policy, “allow John access to my location” and an overlapping deny policy from the same user “deny everyone access to my location when in my office”, we express both by replacing the original allow policy by “allow John access to my location when I am in locations other than my office”. We therefore restrict our discussions to allow policies.

4.3.4 Access Control Semantics

We define access control as the task of deriving $\mathcal{T}'_{Q_i} \subseteq \mathcal{T}_{Q_i}$ which is the projection of $\mathcal{D}$ on which Q_i can be executed with respect to access control policies defined for its querier. Thus, $\forall\ t_t \in \mathcal{T}_{Q_i},\ t_t \in \mathcal{T}'_{Q_i} \Leftrightarrow eval(\mathcal{E}(\mathcal{P}), t_t) = True$. The function $eval(\mathcal{E}(\mathcal{P}), t_t)$ evaluates a tuple t_t against the policy expression $\mathcal{E}(\mathcal{P})$ that applies to Q_i as follows:

$$eval(\mathcal{E}(\mathcal{P}), t_t) := \begin{cases} True, & if\ \exists\ p_l \in \mathcal{P} \mid eval(\mathtt{OC}^l, t_t) = \ True \\ False, & otherwise \end{cases}$$

where $eval(\mathtt{OC}^l, t_t)$ evaluates the tuple against the object conditions of p_l as follows:

$$eval(\mathtt{OC}^l, t_t) := \begin{cases} True\ , & if\ \forall\ \mathtt{oc}^l_c \in \mathtt{OC}^l \mid t_t.attr = \mathtt{oc}^l_c.attr \implies \\ & \qquad eval(\mathtt{oc}^l_c.op, \mathtt{oc}^l_c.val, t_t.val) = True \\ False, & otherwise \end{cases}$$

[5]A default of deny is also standard is systems that collect/user data. For instance, apps running on mobile device need explicit permission to access and use user data.

where $eval(\texttt{oc}^l_c.op, \texttt{oc}^l_c.val, t_t.val)$ compares the object condition value ($\texttt{oc}^l_c.val$) to the tuple value ($t_t.val$) that matches the attribute of the object condition, using the object condition operator. If the latter is a derived value, the expensive operator/query is evaluated to obtain the value.

This access control semantics satisfies the sound and secure properties of the correctness criterion defined by [97]. If no policies are defined on t_t then the tuple is not included in $\mathcal{T}'_{Q_i}$ as our access control semantics is opt-out by default. Depending upon the query operations, evaluating policies after them is not guaranteed to produce correct results. This is trivially true in the case for aggregation or projection operations that remove certain attributes from a tuple. In queries with non-monotonic operations such as set difference, performing query operations before policy evaluation will result in inconsistent answers. Let $\mathcal{P}$ be the set of policies defined on r_k that control access to Q_i (a query with a set difference). $\mathcal{E}(\mathcal{P})$ is the Disjunctive Normal Form (DNF) expression of $\mathcal{P}$ such that $\mathcal{E}(\mathcal{P}) = \texttt{OC}^1 \vee \cdots \vee \texttt{OC}^{|\mathcal{P}|}$ where $\texttt{OC}^l$ is conjunctive expression of object conditions from $p_l \in \mathcal{P}$. After appending $\mathcal{E}(\mathcal{P})$ to Q_i we obtain: `SELECT * FROM` r_j `MINUS SELECT * FROM` r_k `WHERE` $\mathcal{E}(\mathcal{P})$. Consider a tuple $t_k \in \mathcal{T}_{r_k}$ which has policy $p_l \in \mathcal{P}$ that denies Q_i access to t_k. If there exists a tuple $t_j \in \mathcal{T}_{r_j}$ such that $t_j = t_k$, then performing set difference operations before checking policies on r_k will result in a tuple set that does not include t_j. On the other hand, if policies for r_k are checked first, then $t_k \notin \mathcal{T}_{Q_i}$ and therefore t_j will be in the query result.

4.4 Overview of the Sieve Approach

For a given query Q_i, the two main factors that affect the time taken to evaluate the set of policies for the set of tuples $\mathcal{T}_{Q_i}$ required to compute Q_i (i.e., $eval(\mathcal{E}(\mathcal{P}), t_t)\ \forall\ t_t \in \mathcal{T}_{Q_i}$) are the large number of complex policies and the number of tuples in $\mathcal{T}_{Q_i}$. The overhead of policy evaluation can thus be reduced by first eliminating tuples using low cost filters before checking the relevant ones against complex policies and second by minimizing the length of policy expression a tuple t_t needs to be checked against before deciding whether it can be included in the result of Q_i or not. These two fundamental building blocks form the basis for Sieve.

- **Reducing Number of Policies.** Not all policies in $\mathcal{P}$ are relevant to a specific query Q_i. We can first easily filter out those policies that are defined for different queriers/purposes given the query metadata $\mathtt{QM}^i$. For instance, when Prof. Smith poses a query for grading, only the policies defined for him and the faculty group for grading purpose are relevant out of all policies defined on campus. Thus, given our policy model (that controls access based on querier's identity and purpose), the set of policies relevant to the query can be filtered using $\mathtt{QM}^i$. We denote the subset of policies which are relevant given the query metadata $\mathtt{QM}^i$ by $\mathcal{P}_{\mathtt{QM}^i} \subseteq \mathcal{P}$ where $p_l \in \mathcal{P}_{\mathtt{QM}^i}$ iff $\mathtt{QM}^i_{purpose} = \mathtt{qc}^l_{purpose} \wedge (\mathtt{QM}^i_{querier} = \mathtt{qc}^l_{querier} \vee \mathtt{qc}^l_{querier} \in group(\mathtt{QM}^i_{querier}))$. In addition, for a given tuple $t_t \in \mathcal{T}_{Q_i}$ we can further filter policies in $\mathcal{P}_{\mathtt{QM}^i}$ that we must check based on the values of attributes in t_t. For instance, the owner of the tuple (i.e., $t_t.owner$) can be used to filter out policies which do not apply to the tuple (i.e., are not part of $\mathcal{P}_{t_t} \subseteq \mathcal{P}_{\mathtt{QM}^i}$ where $\mathcal{P}_{t_t}$ is such that $p_l \in \mathcal{P}_{t_t}$ iff $t_t.owner = \mathtt{oc}^l_{owner}$ (i.e., the owner of the tuple is the same than the owner/creator of the policy).

- **Reducing Number of Tuples.** Even if the number of policies to check are minimized, the resulting expression $\mathcal{E}(\mathcal{P})$ might still be computationally complex. To speed up processing of $\mathcal{E}(\mathcal{P})$ further, we derive low cost filters (object conditions) from it which can filter out tuples by exploiting existing indexes $\mathcal{I}$ over attributes in the database. We therefore rewrite the policy expression $\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{|\mathcal{P}|}$ as a *guarded policy expression* $\mathcal{G}(\mathcal{P})$ which is a disjunction of *guarded expressions* $\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$. Each G_i consists of a *guard* $\mathtt{oc}^i_g$ and a *policy partition* $\mathcal{P}_{G_i}$ where $\mathcal{P}_{G_i} \subseteq \mathcal{P}$. Note that $\mathcal{P}_{G_i}$ partitions the set of policies, i.e., $\mathcal{P}_{G_i} \cap \mathcal{P}_{G_j} = \emptyset \ \forall \ G_i, G_j \in \mathcal{G}(\mathcal{P})$. Also, all policies in $\mathcal{P}$ are covered by one of the guarded expressions, i.e., $\forall\ p_i \in \mathcal{P}\ (\exists\ G_i \in G$ such that $p_i \in \mathcal{P}_{G_i})$. We will represent the guarded expression $G_i = \texttt{oc}_g^i \wedge \mathcal{P}_{G_i}$ where $\mathcal{P}_{G_i}$ is the set of policies but for simplicity of expression we will use it as an expression where there is a disjunction between policies.

The *guard* term $\texttt{oc}_g^i$ is an object condition that can support efficient filtering by exploiting an index. In particular, it satisfies the following properties:

– $\texttt{oc}_g^i$ is a simple predicate over an attribute (e.g., $ts - time > 9am$) and the attribute in $\texttt{oc}_g^i$ has an index defined on it (i.e., $\texttt{oc}_g^i.attr \in \mathcal{I}$).

– The guard $\texttt{oc}_g^i$ is a part of all the policies in the partition and can serve as a filter for them $\mathcal{P}_{G_i}$ (i.e., $\forall\, p_l \in \mathcal{P}_{G_i}\ \exists\ \texttt{oc}_j^l \in \texttt{OC}^l \mid \texttt{oc}_j^l \implies \texttt{oc}_g^i$).

As an example, consider the policy expression of all the policies defined by students to grant the professor access to their data in different situations. Let us consider that many of such policies grant access when the student is connected to the WiFi AP of the classroom. For instance, in addition to John's policy defined before, let us consider that Mary defines the policy $\langle$`[W.owner = Mary` $\wedge$ $\wedge$ `W.wifiAP = 1200], [Prof. Smith` $\wedge$ `Attendance Control], allow`$\rangle$. This way, such predicate (i.e., wifiAP=1200) could be used as a guard that will group those policies, along with others that share that predicate, to create the following expression: `wifiAP=1200 AND ((owner=John AND ts-time between 9 AND 10am OR (owner=Mary) OR ...).`

Sieve adaptively selects a query execution strategy when a query is posed leveraging the above ideas. First, given Q_i, Sieve filters out policies based on $\texttt{QM}^i$. Then, using the resulting set of policies, it replaces any relation $r_j \in Q_i$ by a projection that satisfies policies in $\mathcal{P}_{\texttt{QM}^i}$ that are defined over r_j. It does so by using the guarded expression $\mathcal{G}(\mathcal{P}_{r_j})$ constructed as a query `SELECT * FROM` r_j `WHERE` $\mathcal{G}(\mathcal{P}_{r_j})$.

By using $\mathcal{G}(\mathcal{P}_{r_j})$ and its guards $\texttt{oc}_g^i$, we can efficiently filter out a high number of tuples and only evaluate the relevant tuples against the more complex policy partitions $\mathcal{P}_{G_i}$. The generation of $\mathcal{G}(\mathcal{P}_{r_j})$ might take place offline if the policy dataset is deemed to undergo small number of changes over time. Otherwise, the generation can be done either when a change is made in the policy table or at query time for more dynamic scenarios (our algorithm is efficient enough for dynamic scenarios as we show in Section 4.8).

A tuple that satisfies the guard $\texttt{oc}_g^i$ is then checked against $\mathcal{E}(\mathcal{P}_{G_i}) = \texttt{OC}^1 \vee \cdots \vee \texttt{OC}^{|\mathcal{P}_{G_i}|}$. This evaluation could be expensive depending upon the number of policies in $\mathcal{P}_{G_i}$. As it is a DNF expression, in the worst case (a tuple that does not satisfy any policy) will have to be evaluated against each $\texttt{OC}^j \in \mathcal{P}_{G_i}$. We introduce a policy evaluation operator $(\Delta(G_i, \texttt{QM}^i, t_t))$ which takes a guarded expression G_i, query metadata $\texttt{QM}^i$, and each tuple t_t that satisfied $\texttt{oc}_g^i$ and retrieves a subset of $\mathcal{P}_{G_i}$ (filtered using $\texttt{QM}^i$ and t_t). Then, policy evaluation on the tuples that satisfy the guard is only performed on this subset of policies instead of $\mathcal{P}_{G_i}$. Sieve situationally selects based on each $G_i \in G$ whether to use the policy evaluation operator for evaluating $\mathcal{P}_{G_i}$ to minimize the execution cost. We explain the details of implementation of this operator and the selection strategy in Section 4.6.

Hence, the main challenges are: 1) Selecting appropriate guards and creating the guarded expression; 2) Dynamically rewriting query by evaluating different strategies and constructing a query that can be executed in an existing DBMS. We explain our algorithm to generate guarded expressions for a set of policies in Section 4.5. This generation might take place offline if the policy dataset is deemed to undergo small number of changes over time. Otherwise, the generation can be done either when a change is made in the policy table or at query time for more dynamic scenarios (our algorithm is efficient enough for dynamic scenarios as we show in Section 4.8). We later explain how Sieve can be implemented in existing DBMSs and how it selects an appropriate strategy depending on the query and the set of policies that apply to the query.

4.5 Creating Guarded Expressions

Our goal is to translate a policy expression $\mathcal{E}(\mathcal{P}) = \texttt{OC}^1 \vee \cdots \vee \texttt{OC}^{|\mathcal{P}|}$ into a guarded policy expression $\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$ such that the cost of evaluating $\mathcal{G}(\mathcal{P})$ given database $\mathcal{D}$ and set of indices $\mathcal{I}$ is minimized

$$\min cost(\mathcal{G}(\mathcal{P})) = \min \sum_{G_i \in G} cost(G_i) \tag{4.1}$$

where G is the set of all the guarded expressions in $\mathcal{G}(\mathcal{P})$. A guarded expression G_i corresponds to $G_i = \texttt{oc}_g^i \wedge \mathcal{P}_{G_i}$ where $\texttt{oc}_g^i$ is a guard and $\mathcal{P}_{G_i}$ is a policy partition. The cost of evaluating a tuple against a set of policies is defined by

$$cost(eval(\mathcal{E}(\mathcal{P}_{G_i}))) = \alpha.|\mathcal{P}_{G_i}|.c_e \tag{4.2}$$

where α represents the average number of policies in $\mathcal{P}_{G_i}$ that the tuple t_t is checked against from the disjunctive expression in $\mathcal{E}(\mathcal{P}_{G_i})$ (we assume that the DBMS stops the execution of such a disjunctive expression with the first policy that the tuple satisfies and skips the rest), and c_e represents the average cost of evaluating t_t against the set of object conditions for a policy $p_l \in \mathcal{P}_{G_i}$ (i.e., $\texttt{OC}^l$). We model $cost(G_i)$ as

$$cost(G_i) = \rho(\texttt{oc}_g^i).(c_r + cost(eval(\mathcal{E}(\mathcal{P}_{G_i})))) \tag{4.3}$$

where $\rho(\texttt{oc}_g^i)$ denotes the estimated cardinality[6] of the guard $\texttt{oc}_g^i$ and c_r represents the cost of reading a tuple using an index. The values of c_r, c_e, and α are determined experimentally using a set of sample policies and tuples.

[6]Estimated using histograms maintained by the DBMS.

The percentage of policies (α) that have to be checked before one returns true, and c_e, the cost of evaluating a policy against a single tuple, are obtained experimentally. We compute α by executing a query which counts the number of policy checks done over $\mathcal{P}_{G_i}$ before a tuple either satisfies one of the policies or is discarded (does not satisfy any policy) and averaging the number of policy checks across all tuples. We estimate c_e by computing the difference of the read cost per tuple without policies (estimated by dividing the time it takes to perform a table scan by the total number of tuples) and the average cost per tuple with policies. The former is estimated by executing a table scan with different number of policies with different selectivities (number of tuples) and averaging the cost per tuple per policy.

The first step in determining $\mathcal{G}(\mathcal{P})$ is to generate all the *candidate guards* ($\mathcal{CG}$), given the object conditions from $\mathcal{P}$, which satisfy the properties of guards as explained in Section 4.3. Different choices of guarded expressions may exist for the same policy given $\mathcal{I}$ and therefore the second step is to select a set of guards from $\mathcal{CG}$ with the goal of minimizing the evaluation cost of $\mathcal{G}(\mathcal{P})$.

4.5.1 Generating Candidate Guards

Any object condition oc_c^l in a policy p_l is added to the candidate guard set $\mathcal{CG}$ if it satisfies the properties of a guard i.e., $\mathsf{oc}_c^l.val$ is a constant and $\mathsf{oc}_c^l, attr \in \mathcal{I}$. Each policy $p_l \in \mathcal{P}_{Q_j}$ is guaranteed to have at least one object condition that satisfies these properties (e.g., oc_{owner}^l or $\mathsf{oc}_{profile}^l$). Guards group together policies and act as a filter reducing the tuples to be evaluated against policies. If only the identical object conditions were to be used as guards, they might group only a small number of policies in their corresponding policy partitions $\mathcal{P}_{G_i}$. This would result in a larger number of guarded expressions in $\mathcal{G}(\mathcal{P})$ of a querier and thus increase the cost of evaluation according to Equation 4.1.

To improve the grouping capability of a guard, we present an approach which generates additional candidate guards from the already existing candidate guards which have $\texttt{oc}.val$ as a range of values ($[val_1, val_2]$.). This is done by *merging* together these candidate guards on the same attribute that belong to different policies. For example, consider two policies with the following object conditions on attribute a: $3 \leq a \leq 10$ ($\texttt{oc}_c^x$) and $4 \leq a \leq 15$ ($\texttt{oc}_c^y$). Depending on whether it is beneficial to do so, they could be merged to create a new candidate guard $3 \leq a \leq 15$ ($\texttt{oc}_c^{x \oplus y}$). After merging, this new object condition could be used as a guard for the two policies. The following theorem states the requirement for this merging of object conditions to be beneficial based on their overlap.

Idea of Guarded expression These guarded expressions serve a role similar to that of blocking in entity resolution [11] in that if a tuple does not satisfy the guard, then it will not satisfy the policy. As a result, it generates a projection of the database in which the query can be executed. We present an algorithmic solution to generate the most appropriate guarded expression for a complex policy expression involving a large number of access control policies. Additionally, we present also an approach to renewing *guarded expressions* in dynamic settings when new policies may arrive (and/or old policies might be updated) so as to optimize both query time as well as system time. The second strategy is inspired by pub-sub approaches such as [90, 56, 101, 63].

Theorem 4.1. *Given two object conditions* $\texttt{oc}_c^x = (attr_c,\ op_1^x,\ val_1^x,\ op_2^x,\ val_2^x) \in p_x$, $\texttt{oc}_c^y = (attr_c, op_1^y, val_1^y, op_2^y, val_2^y) \in p_y$ *and* $attr_c \in \mathcal{I}$*, the object condition generated by merging them i.e.,* $\texttt{oc}_c^{x \oplus y} = (attr_c,\ op_1,\ val_1^{x \oplus y}, op_2, val_2^{x \oplus y})$ *with* $val_1^{x \oplus y} = min(val_1^x, val_1^y)$ *and* $val_2^{x \oplus y} = max(val_2^x, val_2^y)$ *is only beneficial if* $[val_1^x, val_2^x] \cap [val_1^y, val_2^y] \neq \phi$.

Proof: Let p_x and p_y be two policies with candidate guards $\texttt{oc}_c^x$ and $\texttt{oc}_c^y$, respectively. Based on Equation 4.3, the cost of evaluating a single policy p_x with $\texttt{oc}_c^x$ as the guard is

$$cost(p_x) = \rho(\texttt{oc}_c^x).(c_r + c_e) \tag{4.4}$$

To simplify the notation in this proof, we use oc_c^x to denote the values in the range $[val_1^x, val_2^x]$ (similarly for oc_c^y). W.l.o.g. let $min(val_1^x, val_1^y) = val_1^x$ and $max(val_2^x, val_2^y) = val_2^y$. If $\mathsf{oc}_c^x \cap \mathsf{oc}_c^y = \emptyset$ the cost of evaluating the merged policy is given by

$$\begin{aligned} cost(p_x \oplus p_y) = \rho(\mathsf{oc}_c^{x \oplus y}).(c_r + 2.\alpha.c_e) = \\ (\rho(\mathsf{oc}_c^x) + \rho(\mathsf{oc}_c^y)).(c_r + 2.c_e) + \rho(\mathsf{oc}_c^e).(c_r + 2.\alpha.c_e) \end{aligned} \tag{4.5}$$

where $\mathsf{oc}_c^e = (attr_c, op_1, val_2^x, op_2, val_1^y)$ (denotes the extra values that are not covered by either oc_c^x or oc_c^y). Since $\rho(\mathsf{oc}_c^e) \geq 0$, $cost(p_x \oplus p_y) >= cost(p_x) + cost(p_y)$. Thus, when candidate guards do not overlap, merging them is not beneficial. $\square$

We now check when it is beneficial to merge candidate guards if they overlap i.e., $\mathsf{oc}_c^x \cap \mathsf{oc}_c^y \neq \emptyset$. If these candidate guards were to be merged, the values covered by the merged object condition would be the union of the two ranges, represented by $\mathsf{oc}_c^x \cup \mathsf{oc}_c^y$. The cost of evaluation is given by

$$cost(p_x \oplus p_y) = \rho(\mathsf{oc}_c^x \cup \mathsf{oc}_c^y).(c_r + 2.\alpha.c_e) \tag{4.6}$$

Applying the inclusion-exclusion principle[7], we have

$$cost(p_x \oplus p_y) = (\rho(\mathsf{oc}_c^x) + \rho(\mathsf{oc}_c^y) - \rho(\mathsf{oc}_c^x \cap \mathsf{oc}_c^y)).(c_r + 2.\alpha.c_e) \tag{4.7}$$

Given that merging will be beneficial if $cost(p_x \oplus p_y) < cost(p_x) + cost(p_y)$, using Equations 4.4 and 4.7 we have the following inequality

$$(\rho(\mathsf{oc}_c^x) + \rho(\mathsf{oc}_c^y) - \rho(\mathsf{oc}_c^x \cap \mathsf{oc}_c^y)).(c_r + 2.\alpha.c_e) < \rho(\mathsf{oc}_c^x).(c_r + c_e) + \rho(\mathsf{oc}_c^y).(c_r + c_e) \tag{4.8}$$

[7] A ∪ B = A + B - A ∩ B

Simplifying using again the inclusion exclusion principle

$$\frac{\rho(\mathsf{oc}_c^x \cap \mathsf{oc}_c^y)}{\rho(\mathsf{oc}_c^x \cup \mathsf{oc}_c^y)} > \frac{c_e}{c_r + \alpha.c_e} \tag{4.9}$$

as the right side are all constant values, we can replace it with C. We denote the ratio on the left by $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y)$. Thus, the merging is beneficial if $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) > C$. If this condition is satisfied, we add $\mathsf{oc}_c^{x \oplus y}$ to $\mathcal{P}_x$, $\mathcal{P}_y$, and $\mathcal{CG}$.

As merging is only beneficial, if $|\mathsf{oc}_c^x \cap \mathsf{oc}_c^y| \neq \phi$, we first order the candidate guards by their left range values in the ascending order. Considering *transitive* merges, the number of pair-wise checks to be done between candidate guards could be linear. For instance, consider an additional policy added to the previous example with the following object condition on attribute a, $12 \leq a \leq 18$ (oc_c^z). It is possible that $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) < C$ while $\theta(\mathsf{oc}_c^y, \mathsf{oc}_c^z) > C$ and therefore $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^{y \oplus z}) > C$ (i.e., the merged object condition $3 \leq a \leq 18$ is beneficial). The following theorem characterizes when the transitive merges will not be beneficial for candidate guards with certain properties in a $\mathcal{CG}$ sorted in the ascending order of their left range values.

Theorem 4.2. *Given three candidate guards oc_c^x, oc_c^y, and oc_c^z sorted in the ascending order of their left range values with the following properties: $\mathsf{oc}_c^x \cap \mathsf{oc}_c^y \neq \phi$, $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) < C$, and $\theta(\mathsf{oc}_c^y, \mathsf{oc}_c^z) > C$. The transitive merge between oc_c^x and $\mathsf{oc}_c^{y \oplus z}$ will not be beneficial (i.e., $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^{y \oplus z}) < C$) if $\mathsf{oc}_c^x \cap \mathsf{oc}_c^z = \phi$.*

We prove this theorem by contradiction. As $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) < C$, using Equation 4.9 we have

$$\frac{\rho(\mathsf{oc}_c^x \cap \mathsf{oc}_c^y)}{\rho(\mathsf{oc}_c^x \cup \mathsf{oc}_c^y)} < C \tag{4.10}$$

Note that since $\mathsf{oc}_c^x \cap \mathsf{oc}_c^z = \phi$, $\mathsf{oc}_c^x \cap \mathsf{oc}_c^{y \oplus z} = \mathsf{oc}_c^x \cap \mathsf{oc}_c^y$. Thus, for the transitive merge between oc_c^x and $\mathsf{oc}_c^{y\oplus z}$ to be beneficial, we should have $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^{y\oplus z}) > C$.

$$\frac{\rho(\mathsf{oc}_c^x \cap \mathsf{oc}_c^y)}{\rho(\mathsf{oc}_c^x \cup \mathsf{oc}_c^y \cup \mathsf{oc}_c^z)} > C \tag{4.11}$$

This is not possible as the numerator is same in both Equation 4.10 and Equation 4.11, while the denominator is larger in Equation 4.11. □

Given Theorems 4.1 and 4.2, the steps for generating $\mathcal{CG}$ from a set of policies $\mathcal{P}$ defined on a relation r_i are: 1) For every $attr_j$ that is part of r_i and has an index defined on it (i.e., $attr_j \in \mathcal{I}$); 2) S_j is the set of all object conditions for all $p^l \in \mathcal{P}$ such that, $\mathsf{oc}_c^l.val$ is a constant and $\mathsf{oc}_c^l.attr = attr_j$; 3) For each S_j containing object conditions with range of values, sort the object conditions by their left values to create a sorted list; 4) For the first candidate guard (oc_c^x) in this sorted list, verify whether the next candidate guard (oc_c^y) is such that $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) > C$. If true, then merge both the candidate guards to generate $\mathsf{oc}_c^{x\oplus y}$ which is added to S_j along with p_x and p_y. Else if $\theta(\mathsf{oc}_c^x, \mathsf{oc}_c^y) < C$, then we check if it is beneficial to transitively merge oc_c^x with the following candidate guard (oc_c^z) using Theorem 4.2. 5) When transitive merge is no longer beneficial, we move on to the next candidate guard (oc_c^y). The final $\mathcal{CG}$ is constructed by combining all the S_j corresponding to each $attr_j$ that is part of r_i.

4.5.2 Selecting Guards To Minimize Cost

We next select the set of guards $G \in \mathcal{CG}$ that minimizes the cost of policy evaluation according to Equation 4.1. The goal of guard selection is to select G from $\mathcal{CG}$ such that every policy in $\mathcal{P}$ is covered exactly once and the cost of evaluation is minimized. We show that this problem is NP-hard, by reducing the well-known weighted Set-Cover problem to it. In the weighted Set-Cover problem, we have a set of elements $E = e_1, \cdots, e_n$ and a set of subsets over E denoted by $S = S_1, \cdots, S_m$ with each set $S_i \in S$ having a weight w_i associated with it. The goal of set cover problem is to select $\min_{\hat{S} \subseteq S} \sum S_i.w_i \mid S_i \in \hat{S}$ and $E = \bigcup_{S_i \in \hat{S}} S_i$. We map E to $\mathcal{P}$, S to $\mathcal{CG}$, and $\hat{S}$ to G. We assign the element e_i to $S_i \in \hat{S}$ when the corresponding policy p_i is assigned to $G_i \in G$. The weight function w_i is set to the read cost of G_i based on using the guard oc_c^i to read the tuples i.e., $w_i = \mathit{read\_cost}(G_i) = \rho(\mathsf{oc}_c^i).c_r$. If a polynomial time algorithm existed to solve this problem, then it would solve set-cover problem too.

For the purpose of selecting guards that minimize cost of evaluation, we define a utility heuristic which ranks the candidate guards by their benefit per unit read cost (similar to the one used by [59] for optimizing queries with expensive predicates). Each guard oc_c^i, based on its selectivity, reduces the number of tuples that have to be checked against $\mathcal{P}_{G_i}$. The benefit of a guarded expression captures this reduction in evaluation cost for a relation r_i as defined by $\mathit{benefit(G_i)} = c_e.|\mathcal{P}_{G_i}|.(|r_i| - \rho(\mathsf{oc}_c^i))$. Using this benefit method, and the read cost defined earlier, we define the utility of G_i as $\mathit{utility(G_i)} = \frac{\mathit{benefit(G_i)}}{\mathit{read\_cost(G_i)}}$.

Algorithm 1 uses this heuristic to select the best possible guards to minimize the cost of policy evaluation. First, it iterates over $\mathcal{CG}$ and stores each guarded expression $G_i \in \mathcal{CG}$ in a priority queue in the descending order of their utility ($PriorityInsert(Q, G_i, U[i])$. Second, the priority queue is polled for the G_i with the highest utility ($Extract - Maximum(Q)$). If $\mathcal{P}_{G_i}$ intersects with another $\mathcal{P}_{G_j} \in \mathcal{CG}$, $\mathcal{P}_{G_j}$ is updated to remove the intersection of policies ($\mathcal{P}_{G_i} \cap \mathcal{P}_{G_j}$). After removal, *utility(G_j)* is recomputed and the updated G_j is reinserted into priority queue in the order of its utility. The result is thus the subset of candidate guards (G) that covers all the policies in $\mathcal{P}$ and minimizes $\mathit{cost}(\mathcal{G}(\mathcal{P}))$ as in Equation 4.1.

Algorithm 1: Selection of guards.

```
Function GuardSelection($\mathcal{CG}$):
    for $i$ in $1 \cdots |\mathcal{CG}|$ do
        C[i] = COST($G_i$); U[i] = UTILITY($G_i$)
    end
    $Q \leftarrow \phi$ for $i$ in $1 \cdots |\mathcal{CG}|$ do
        PRIORITYINSERT($Q$,$G_i$,$U[i]$)
    end
    while $Q$ is not empty do
        $G_{max}$ = EXTRACT-MAXIMUM($Q$); $G \leftarrow G_{max}$ foreach $G_i$ in $Q$ do
            if $\mathcal{P}_{G_i} \cap \mathcal{P}_{G_{max}} \neq \phi$ then
                $\mathcal{P}_{G_i} = \mathcal{P}_{G_i} \setminus \mathcal{P}_{G_{max}}$; REMOVE($Q$, $G_i$) if $\mathcal{P}_{G_i} \neq \phi$ then
                    B = BENEFIT($G_i$); U[i] = $\frac{B}{C[i]}$ PRIORITYINSERT($Q$,$G_i$,$U[i]$)
                end
            end
        end
    end
    return $G$
```

4.5.3 Discussion

We briefly discuss when the approach of generating guards is not an effective strategy for query processing with policy enforcement. Guards and guarded expressions are generated by factorization of the complex policy expressions which are in disjunctive normal form. If the policy expression were any arbitrary boolean expressions, generating effective guards would be extremely challenging. This is because the search space for guards becomes much larger and developing efficient algorithms that guarantee the quality of the guards becomes harder[24].

If none of the policies contain common object conditions, the only possible guard that can be generated will have a policy partition of length 1. In this situation the number of guards will be high and this increases the cost of performing sorting and union of the final results from each guard expression. On the other hand if all the policies contained the same number and type of object conditions it would be much more efficient to perform a temporary table join as discussed in Section 4.9.

4.6 Implementing Sieve

Sieve[8] is a general-purpose middleware that intercepts queries posed to a DBMS, optimally rewrites them, and submits the queries back to the underlying DBMS for efficient execution that is compliant with the access control policies. In this section, we first present the rewrite approach with *guarded expressions* in DBMSs. Then, we present two optimization techniques to improve this rewrite by utilizing policy evaluation operator and query predicates. Finally, we illustrate a sample rewritten query in Sieve.

4.6.1 Persistence of Policies and Guards

To store policies associated with all the relations in the database, Sieve uses two additional relations, the policy table (referred to as r_P), which stores the set of policies, and the object conditions table (referred to as r_{OC}), which stores conditions associated with the policies. The structure of r_P corresponds to $\langle$`id, owner, querier, associated-table, purpose, action, ts-inserted-at`$\rangle$, where `associated-table` is the relation r_i for which the policy is defined and `ts-inserted-at` is the timestamp at policy insertion. The schema of r_{OC} corresponds to $\langle$`policy-id, attr, op, val`$\rangle$ where `policy-id` is a foreign key to r_P and the rest of attributes represent the condition oc_c^l=$\langle$*attr*, *op*, *val*$\rangle$. We emphasize that the value `val` in r_{OC} might correspond to a complex SQL condition in case of nested policies. For instance, the two sample policies defined in Section 4.3 regulate access to student connectivity data for Prof. Smith; they are persisted as tuples $\langle$`1, John, Prof.Smith, WiFiDataset, Attendance Control, Allow, 2020-01-01 00:00:01`$\rangle$ and $\langle$`2, John, Prof.Smith, WiFiDataset, Attendance Control, Allow, 2020-01-01 00:00:01`$\rangle$ in r_P and with the tuples $\langle$`1, 1, wifiAP, =, 1200`$\rangle$, $\langle$`2, 1, ts-time, ≥, 09:00`$\rangle$, $\langle$`3, 1, ts-`

[8]The implementation of Sieve (with connectors for both MySQL and PostgreSQL) is available at https://github.com/primalpop/sieve.

`time, `$\leq$`, 10:00`$\rangle$, $\langle$`4, 2, wifiAP, =, SELECT W2.wifiAP FROM WiFiDataset AS W2 WHERE W2.owner = "Prof.Smith" and W2.ts-time = W.ts-time`$\rangle$ in r_{OC}.

A guarded policy expression $\mathcal{G}(\mathcal{P})$ generated, per user and purpose, is stored in r_{GE} with the schema $\langle$`id, querier, associated-table, purpose, action, outdated, ts-inserted-at`$\rangle$. Guarded policy expressions are not continuously updated based on incoming policies as this would be unnecessary if their specific queriers do not pose any query. We use the `outdated` attribute, which is a boolean flag, to describe whether the guarded expression includes all the policies belonging to the querier. If at query time, the `outdated` attribute associated to the guarded policy expression for the specific querier/purpose (as specified in the query metadata $\texttt{QM}^i_{querier}, \texttt{QM}^i_{purpose}$) is found to be true, then that guarded policy expression is regenerated. After the guarded expression is regenerated for a querier, it is stored in the table with `outdated` set to false. Guard regeneration comes with an overhead. However, in our experience, the corresponding overhead is much less than the execution cost of queries. As a result, we generate guards during query execution using triggers in case the current guards are outdated. Guarded expressions G_i associated with a guarded policy expression $\mathcal{G}()$ are stored in two relations: r_{GG}=$\langle$`id, guard-expression-id, attr, op, val`$\rangle$ to store the guard (i.e., $\texttt{oc}^i_g$=$\langle$*attr*, *op*, *val*$\rangle$) and r_{GP}=$\langle$`guard-id, policy-id`$\rangle$ to store the policy partition (i.e., $\mathcal{P}_{G_i}$).

4.6.2 Implementing Operator Δ

We implement the policy evaluation operation Δ (see Section 4.4) using User Defined Functions (UDFs) supported by DBMSs. Consider a set of policies $\mathcal{P}$ and the query metadata $\texttt{QM}^i$ and a tuple t_t belonging to relation r_j. $\Delta(\mathcal{P}, \texttt{QM}^i, t_t)$ is implemented as the following UDF

`CREATE FUNCTION delta(`[*policy*]`, `*querier*`, `*purpose*`, `[*attrs*]`)`

```
{BEGIN
 Cursor c =
  SELECT $r_{OC}.attr$ as attr, $r_{OC}.op$ as op, $r_{OC}.val$ as val
  FROM $r_P, r_{OC}$
  WHERE $r_P.querier = querier$ AND $r_P.purpose = purpose$ AND $r_P.id$ IN $[policy]$
     AND $r_P.owner = [attrs].owner$ AND $r_P.id = r_{OC}.policy-id$
  LET satisfied_flag = true
  READ UNTIL c.isNext() = false:
     FETCH c INTO p_attr, p_op, p_val
      FOR each t_attr in $[attrs]$
        IF t_attr = p_attr THEN
          satisfied_flag = satisfied_flag AND /*Check whether
             t_val satisfies p_op p_val*/
  return satisfied_flag
END}
```

The UDF above performs two operations: 1) It takes a set of policies and retrieves a subset $\hat{\mathcal{P}}$ which contains the relevant policies to be evaluated based on the query metadata $\texttt{QM}^i$ and the tuple t_t; 2) It evaluates each policy $p_i \in \hat{\mathcal{P}}$ on t_t.

4.6.3 Query Rewrite with Guarded Expressions

Our goal is to evaluate policies for query Q_i by replacing any relation $r_j \in Q_i$ by a projection of r_j that satisfies the guarded policy expression $\mathcal{G}(\mathcal{P}_{r_j})$ where $\mathcal{P}_{r_j}$ is the set of policies defined for the specific querier, purpose, and relation. To this end, we first use the `WITH` clause for each relation $r_j \in Q_i$ that selects tuples in r_j satisfying the guarded policy expression. Using the `WITH` clause, the policy check is performed only once even if the same relation appears multiple times in the query. This rewritten query replaces every occurrence of r_j with the corresponding $\hat{r}_j$.

```
WITH $\hat{r}_j$ AS (SELECT * FROM $r_j$ WHERE $G_1$ OR $G_2$ OR ··· OR $G_n$)
```

Optimizers might choose sub-optimal plans when executing the complex Sieve rewritten queries. Sieve utilizes DBMS extensibility features (e.g., index usage hints[9], optimizer explain[10], UDFs) offered by DBMSs that allows it to suggest index plans to the underlying optimizer. Since such features vary across DBMSs, guiding optimizers requires a platform dependent connector that can rewrite the query appropriately. In systems such as MySQL, Oracle, DB2, and SQL Server that support index usage hints, Sieve can rewrite the query to explicitly force indexes on guards. For example, in MySQL using `FORCE INDEX` hints, which tell the optimizer that a table scan is expensive and should only be used if the DBMS cannot use the suggested index to find rows in the table, the rewritten query will be as follows:

```
WITH $\hat{r}_j$ AS (
  SELECT * FROM $r_j$ [FORCE INDEX (oc$_g^1$)] WHERE $G_1$ UNION
  SELECT * FROM $r_j$ [FORCE INDEX (oc$_g^2$)] WHERE $G_2$ UNION···
  SELECT * FROM $r_j$ [FORCE INDEX (oc$_g^n$)] WHERE $G_n$)
```

Some systems, like PostgreSQL, do not support index hints explicitly. In such cases, Sieve still does the above rewrite (without index usage hints) but depends upon the underlying optimizer to select appropriate indexes.

[9]https://dev.mysql.com/doc/refman/8.0/en/index-hints.html
[10]https://www.postgresql.org/docs/13/sql-explain.html

4.6.4 Policy Evaluation Operator

For a given guarded expression G_i, a tuple that satisfies its guard $\texttt{oc}_g^i$ is checked against its policy partition $\mathcal{P}_{G_i}$. We define this strategy of evaluating policies inline with a guard as *Guard&Inlining*. This evaluation strategy could be expensive depending upon the number of policies in $\mathcal{P}_{G_i}$. We introduce an alternative strategy which uses the policy evaluation operator $\Delta(G_i, \texttt{QM}^i, t_t)$ as an alternative to evaluating policies inline with a guard. This operator retrieves only the policies that are applicable to a tuple t_t (that satisfied $\texttt{oc}_g^i$ of G_i) based on G_i, the query metadata $\texttt{QM}^i$, and tuple context of t_t. We call this strategy of using the policy evaluation operator in conjunction with guards as *Guard&*Δ. Sieve adaptively chooses between these two strategies depending on the number of policies in the guard partition (i.e., $|\mathcal{P}_{G_i}|$).

The policy evaluation operator $\Delta(G_i, \texttt{QM}^i, t_t)$ (which is part of *Guard&*Δ) is implemented using User Defined Functions (UDFs) supported by DBMSs. This implementation is done per $r_j \in \mathcal{R}$ as the tuple context of t_t used to retrieve policies varies per relation. The policy evaluation operator performs two operations. First, it retrieves a subset of $\mathcal{P}_{G_i}$ which only includes the relevant policies based on the query metadata $\texttt{QM}^i$ and the tuple context of t_t. The tuple context is determined by its values for different attributes (e.g., $r_i.owner$) and $\texttt{QM}^i$ is information associated with the query Q_i such as $\texttt{QM}^i_{querier}$ and $\texttt{QM}^i_{purpose}$. Second, given such a subset, $\bar{\mathcal{P}}_{G_i}$, the operator evaluates each policy in it, $p_i \in \bar{\mathcal{P}}_{G_i}$, on t_t using the access control semantics defined in Section 4.3. An example invocation of the $\Delta(G_i, \texttt{QM}^i, t_t)$ is as follows: *delta(32, "Prof.Smith", "Analysis", "owner", "ts-date", "ts -time", "wifiAP")*. The first parameter, *32*, denotes the id of the persisted guarded expression G_i in the database. The second set of parameters {*"Prof.Smith", "Analysis"*} belong to the metadata of the query $\texttt{QM}^i$. The final set of parameters (*"owner", "ts-date", "ts -time", "wifiAP"*) denote the attributes of the tuple and thus defines its context.

Sieve uses a cost model to compare between the two different strategies ($Guard\&\Delta$ and $Guard\&Inlining$) and chooses the best one for performing the rewrite. This comparison is performed for each guarded expression G_i in a guarded policy expression $\mathcal{G}(\mathcal{P})$ (along with query metadata). We model the cost of each strategy by computing the cost of evaluating policies per tuple since the number of tuples to check are the same in both cases. The cost of the $Guard\&\Delta$ strategy is estimated by the invocation and execution cost of the UDF implementation of policy evaluation operator. Thus, $cost(Guard\&\Delta) = UDF_{inv} + UDF_{exec}$ where the UDF_{inv} and UDF_{exec} represent the cost of invocation and execution of the UDF, respectively. We compute this cost by executing the UDF with different guards (with different number of policies in their partitions) and averaging across them. As both the terms involved are constants, $cost(Guard\&\Delta)$ does not vary across guarded expressions.

The cost of $Guard\&Inlining$ is determined by the number of policies in the policy partition of the guard i.e., $|\mathcal{P}_{G_i}|$. Thus, $cost(Guard\&Inlining)=\alpha.|\mathcal{P}_{G_i}|.c_e$ (based on Equation 4.2). Unlike $cost(Guard\&\Delta)$, this cost is not a constant and varies depending upon the guarded expression. Therefore, after generating guarded expressions Sieve computes $cost(Guard\&Inlining)$ and compares it against the pre-computed $cost(Guard\&\Delta)$ to determine the appropriate rewriting for each guarded expression. Our experiments (see Section 4.8) indicate that the usage of the $Guard\&\Delta$ strategy is beneficial if $|\mathcal{P}_{G_i}| > 120$.

4.6.5 Exploiting Selective Query Predicates

In the query rewrite strategy with guarded expressions presented in Section 4.6.3, we used the guards to read the relevant tuples using an index. This is followed by evaluating against policies using inlining or policy evaluation operator. We now consider the situation where the selection predicates that appear in Q_i are highly selective and could be exploited to read the tuples using an index on them instead of on the guards. Sieve considers the following two strategies for reading tuples using the index. 1) Using guards followed by evaluation of the policy partitions associated with them (referred to as $IndexGuards$); 2) Using the query predicate $Q_i.pred$ in Q_i followed by the evaluation of the guarded policy expression (referred to as $IndexQuery$). Each of these strategies use guarded expressions to evaluate the policies and generate $\hat{r_j}$ from r_j on which Q_i is evaluated.

Sieve uses a cost model to compare between these two strategies ($IndexGuards$ and $IndexQuery$) and chooses the best one for performing the final rewrite. This comparison is done per query Q_i. The cost of $IndexQuery$ is determined by using the query explain feature of DBMSs for Q_i, which returns the query predicate $Q_i.pred$ in Q_i used for reading tuples using the index (if any) and its estimated selectivity. We use this to compute *cost(IndexQuery)* $=$ $\rho(Q_i.pred).c_r$. If index is not used for access, we set *cost(IndexQuery)* $= \infty$. For $IndexGuard$, Sieve estimates *cost(IndexGuards)* $= \sum_{G_i \in G} \rho(\texttt{oc}^i_g).c_r$ where $\texttt{oc}^i_g$ is the guard used in the guarded expression G_i. Note that this is an upper bound of the cost for reading tuples using index as it does not consider any optimizations such as index merge. Sieve chooses the best strategy, at query execution time, by comparing their costs (*cost(IndexGuards)* vs. *cost(IndexQuery)*). If the $IndexGuards$ strategy is chosen, we use the rewrite illustrated in Section 4.6.3. Otherwise, if $IndexQuery$ is selected, we use index usage hints with $Q_i.pred$ (instead of $\texttt{oc}^i_g$).

4.6.6 Sieve generated Query Rewrite

Using the different strategies presented, we now revisit the query in Section 4.1 to study the tradeoff between student performance and attendance to classes. A possible rewrite by Sieve to evaluate the policies defined on $WifiDataset$ and generate $WiFiDatasetPol$ is as follows:

```
WITH WiFiDatasetPol AS (
  SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^1) WHERE (oc_g^1 AND
     (G_1))
  UNION
  SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^2) WHERE (oc_g^1 AND
     (G_2))
  ....
  UNION
  SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^n) WHERE (oc_g^n AND
     delta(32,"Prof.Smith", "Analysis","owner","ts-date", "ts-
     time", "wifiAP")=true)
) StudentPerf(WifiDatasetPol, Enrollment, Grades)
```

The `WiFiDatasetPol` replaces `WiFiDataset` in the original query. This rewrite includes the set of guards generated for the querier (Prof. Smith) and his purpose (Analysis) given the policies defined for him. As Sieve selected the *IndexGuards* strategy, we use the index usage hints on guards (through the `FORCE INDEX` command since this rewrite is for MySQL) as explained in Section 4.6.5. Finally, for one specific guarded expression (G_n) Sieve selected the *Guard*&Δ strategy (see Section 4.6.4). Hence, its policy partition was replaced by the call to the UDF that implements the Δ operator.

Discussion: When queries contain large number of disjunctions, often DBMSs execute them by converting the disjunctive conditions into a temporary table and performing a join of this temporary table with the data table. This approach can be used when the policies are uniform (same number and type of predicates in each policy). However, with non-uniform policies this approach doesn't scale well with large number of policies. The reason for this is that when policies are non-uniform, DBMS is unable to exploit index for performing join with the temporary table and has to resort to using nested loop join which is much slower. We tested out this approach with a querier to whom 193 policies applied and observed that this approach took more than 4 minutes to execute for a *SELECT ** query.

4.7 Managing dynamic Scenarios

As mentioned before, the generation of guarded expressions for a set of users can be performed offline. However, in general, the dataset of access control policies defined for a database can change along time (i.e., users add new policies or update existing ones). Hence, Sieve would need to regenerate guarded expressions to reflect the changes in the policy dataset. The cost associated with guard generation is a function of the number of policies and thus, in situations with very large policy datasets, this cost might not be trivial. Regenerating everytime that a change is made in the policy dataset might not be thus optimal if no queries are executed in between changes. Selecting the frequency of guard regeneration carefully can reduce the total system time. In this section, we first extend the cost model presented earlier to include the query evaluation time. Then, we derive the optimal number of policy insertions before guard regeneration as a function of policy and query rates.

4.7.1 Query Evaluation with Guarded Expression

The cost of evaluating G associated with a u_j is given by

$$cost(G) = \sum_{G_i \in G} cost(G_i) \tag{4.12}$$

Given Equation 4.3, and the simplifying assumption that $\rho(\mathsf{oc}_g^i)$ is the same for all the guards in G and can be represented by $\rho(\texttt{oc}_g)$, we can express the previous cost as

$$\begin{aligned}
&\sum_{G_i \in G} cost(G_i) \\
&\qquad = \sum_{G_i \in G} \rho(\texttt{oc}_g^i).(c_r + c_e.\alpha.|\mathcal{P}_{G_i}|) \\
&\qquad = \rho(\texttt{oc}_g).(c_r + c_e.\alpha(|\mathcal{P}_{G_1} + \mathcal{P}_{G_2} + \cdots + \mathcal{P}_{G_m}|) \\
&\qquad = \rho(\texttt{oc}_g).(c_r + c_e.\alpha.|\mathcal{P}_n|) \qquad (4.13)
\end{aligned}$$

where $|\mathcal{P}_{G_1}| + |\mathcal{P}_{G_2}| + \cdots + |\mathcal{P}_{G_m}| = |\mathcal{P}_n|$ as every policy is exactly covered by one guard. We now define the cost of query evaluation for Q_j (posed by u_j) along with G (using the *IndexGuards* approach presented in Section 4.6.3) as

$$cost(G, Q_j) = \sum_{i=1}^{|G|} cost(G_i) + \rho(G).eval(\mathcal{E}(Q_j), t_t) \qquad (4.14)$$

where $\rho(G)$ is the cardinality of the guarded expression for u_j (i.e., the number of tuples that satisfy G and are then checked against the Q_j posed by u_j). We expand this cost using Equation 4.13 and substitute $\rho(G)$ with $\rho(\texttt{oc}_g)$ which gives an upper bound of the cost as $\rho(\texttt{oc}_g) > \rho(G)$.

$$cost(G, Q_j) = \rho(\texttt{oc}_g).(c_r + c_e.\alpha.(|\mathcal{P}_n| + |Q_j|) \qquad (4.15)$$

4.7.2 Computing Optimal Regeneration Rate

Sieve will be able to cut the total cost for a querier which includes query evaluation and guard generation following the optimal regeneration rate. The cost of generating the guarded expression is proportional to the number of policies for the querier ($\mathcal{P}_n$). Assuming k policies belonging to the querier are newly added since the guard (G) was last generated, we denote cost of guard generation by $C_G(\mathcal{P}_n + \mathcal{P}_k)$. Given $\mathcal{D}$ and u_j with N policy insertions and Q queries posed by u_j, the optimal number of policy insertions ($\widetilde{k}$) before regenerating the guarded expression for u_j is given by

$$\widetilde{k} = \underset{k \leq N}{\operatorname{argmin}} \sum_{i=1}^{\frac{N}{k}} (\mathit{cost(}G, Q_{f(k)}, \mathcal{P}_k\mathit{)} + C_G(\mathcal{P}_n + \mathcal{P}_k)) \tag{4.16}$$

We divide that the total number of policies (N) into equal intervals of size k. To simplify the derivation, we assume that queries are uniform and the number of queries posed by the querier during that interval is given by $f(k)$. We define $f(k)$ based on r_p which is the rate at which new policies are added (number of policies per unit time) and r_q which is the rate at which queries are posed by u_j to $\mathcal{D}$. We combine both to define r_{pq} as the number of queries posed per policy insertion $(\frac{r_q}{r_p})$ [11]. The number of queries during each interval of $\frac{N}{k}$ is given by $f(k) = (j \mid 1 \leq j \leq k * r_{pq}$. Finally, we simplify the guard generation cost as a constant (C_G) as it is dominated by the much larger $\mathcal{P}_n$. Putting all these together we have:

$$\widetilde{k} = \underset{k \leq N}{\operatorname{argmin}} \sum_{i=1}^{\frac{N}{k}} \Big(\sum_{j=1}^{k*r_{qp}} \mathit{cost(}G, Q_j, \mathcal{P}_k\mathit{)} + C_G \Big) \tag{4.17}$$

Expanding the first cost term with the cost of query evaluation from Equation 4.15 for insertion of k policies with the assumption that all queries are uniform and

[11]We assume that $\mathcal{D}$ remains static which is only true for OLAP queries. Monitoring data insertion rate for each user will incur a significant overhead that will invalidate the usefulness of this approach.

$\rho(\mathsf{oc}_{\mathcal{P}_1} \cup \mathsf{oc}_{\mathcal{P}_2} \cdots \cup \mathsf{oc}_{\mathcal{P}_k}) \subseteq \rho(\mathsf{oc}_G)$

$$\begin{aligned}
&\mathit{cost}(G, Q_j, \mathcal{P}_k) \\
&\qquad = r_{pq}.\rho(\mathsf{oc}_G).(c_r + \alpha.c_e.(|\mathcal{P}_n| + |Q|)) + r_{pq}.\rho(\mathsf{oc}_G).(c_r + \alpha.c_e.(|\mathcal{P}_n| + 1 + |Q|)) \\
&\qquad\qquad + \cdots + r_{pq}.\rho(\mathsf{oc}_G).(c_r + \alpha.c_e.(|\mathcal{P}_n| + k + |Q|)) \\
&\qquad = k.r_{pq}.\rho(\mathsf{oc}_G).c_r + r_{pq}.\rho(\mathsf{oc}_G).c_e.\alpha.(k.|Q| + k.|\mathcal{P}_n| + \frac{k.(k-1)}{2})
\end{aligned}$$

Using this equation in the previous minimization and replacing the summations with uniformity assumptions, the $\widetilde{k}$ is given by

$$\widetilde{k} = \operatorname*{argmin}_{k \leq N} \frac{N}{k}.\left(k.r_{pq}.\rho(\mathsf{oc}_G).(c_r + c_e.\alpha.(|Q| + |\mathcal{P}_n| + \frac{(k-1)}{2}))\right)$$

As our goal is to find the minimal k, we take the derivative of the above with respect to k and set it equal to 0.

$$\frac{\rho(\mathsf{oc}_G).\alpha.c_e.r_{pq}}{2} - \frac{2.C_G}{k^2} = 0$$

Simplifying it for k, we have

$$k = \sqrt{\frac{4.C_G}{\rho(\mathsf{oc}_G).\alpha.c_e.r_{pq}}} \tag{4.18}$$

The second derivative with respect to k is a positive value and therefore the k value derived by Equation 4.18 minimizes the cost of query evaluation and guard generation for u_j. Based on the simplifying assumptions used in this derivation, $\widetilde{k}$ is an upper bound on the number of policy insertions before the guarded expression is updated. We now prove when it is most beneficial to regenerate the guarded expression after the insertion of $k^t h$ policy.

Theorem 4.3. *If the optimal rate of guard regeneration is set to k policies as in Equation 4.18, then it is best to regenerate immediately after the k^{th} policy has arrived.*

We prove this by contradiction. Assuming the guard regeneration rate is set to k policies for a querier and we regenerate G at k + δ. If $\delta > \frac{1}{r_p}$, then regeneration rate is set at k + 1 and not k which is a contradiction. If $\delta > \frac{1}{r_q}$, then the new query will be evaluated using G and the set of k policies which is higher compared to using the regenerated guarded expression as shown in the derivation above. Therefore $\delta < \frac{1}{r_p}$ and $\delta < \frac{1}{r_q}$ and thus regenerating immediately after k^{th} policy will minimize the cost. □

4.8 Experimental Evaluation

This section presents details of the experimental setup (dataset, queries, policies, and the DBMS setup) followed by the experimental results illustrating the performance of Sieve.

4.8.1 Experimental Setup

Datasets. We used the *TIPPERS* dataset [75] consisting of connectivity logs generated by the 64 WiFi Access Points (APs) at the Computer Science building at UC Irvine for a period of three months. These logs are generated when a WiFi enabled device (e.g., a smartphone or tablet) connects to one of the WiFi APs and contain the hashed identification of the device's MAC, the AP's MAC, and a timestamp. The dataset comprises 3.9M events corresponding to 36K different devices (the signal of some of the WiFi APs bleeds to outside the building and passerby devices/people are also observed). This information can be used to derive the occupancy levels in different parts of the building and to provide diverse location-based services (see Section 4.1) since device MACs can be used to identify individuals. Since location information is privacy-sensitive, it is essential to limit access to this data based on individuals' preferences. Table 4.2 shows the schema of the different tables in the *TIPPERS* dataset. *WiFi_Dataset* stores the logs generated at each *WiFi_AP* when the devices of a *User* connects to them. *User_Group* and *User_Group_Membership* keeps track of the groups and their members respectively.

Table 4.2: TIPPERS data schema.

Table	Columns	Data type
Users	id	int
	device	varchar
	office	int
User_Groups	id	int
	name	varchar
	owner	varchar
User_Group_Membership	user_group_id	int
	user_id	int
Location	id	int
	name	varchar
	type	varchar
WiFi_Dataset	id	int
	wifiAP	int
	owner	int
	ts-time	time
	ts-date	date

We also used a synthetic dataset containing WiFi connectivity events in a shopping mall for scalability experiments with even larger number of policies. We refer to this dataset as *Mall*. We generated the *Mall* dataset using the IoT data generation tool in [57] to generate synthetic trajectories of people in a space (we used the floorplan of a mall extracted from the Web) and sensor data based on those. The dataset contains 1.7M events from 2,651 different devices representing customers. Table 4.3 shows the schema of the tables in the *Mall* dataset.

Queries. We used a set of query templates based on the recent IoT SmartBench benchmark [57] which include a mix of analytical and real-time tasks and target queries about (group of) individuals. Specifically, query templates Q_1 - Retrieve the devices connected for

Table 4.3: Mall data schema.

Table	Columns	Data type
Users	id	int
	device	varchar
	interest	varchar
Shop	id	int
	name	varchar
	type	varchar
WiFi_Connectivity	id	int
	shop_id	int
	owner	int
	obs_time	time
	obs_date	date

a list of locations during a time period (e.g., for location surveillance); Q_2 - Retrieve devices connected for a list of given MAC addresses during a time period (e.g., for device surveillance); Q_3 - Number of devices from a group or profile of users in a given location (e.g., for analytic purposes). Based on these templates, we generated queries at three different selectivities (low, medium, high) by modifying configuration parameters (locations, users, time periods). Below, when referring to a particular query type (i.e., Q_1, Q_2, or Q_3), we mean the set of queries generated for such type.

The SQL version of the queries is thus:

```
Q1=(SELECT * FROM WiFi_Dataset AS W
    WHERE W.wifiAP IN ([ap]) W.ts-time BETWEEN t1 AND t2 AND W.ts-
      date BETWEEN d1 AND d2)
Q2=(SELECT * FROM WiFi_Dataset AS W
    WHERE W.owner in ([devices]) AND W.ts-time BETWEEN t1 AND t2
      AND W.ts-date BETWEEN d1 AND d2)
Q3=(SELECT * FROM WiFi_Dataset AS W, User_Group_Membership AS
  UG
    WHERE UG.user_group_id = group-id
    AND UG.user_id = W.owner AND W.ts-time BETWEEN t1 AND t2 AND
```

```
        W.ts-date BETWEEN d1 AND d2)
```

Policy Generation. The TIPPERS dataset, collected for a limited duration with special permission from UC Irvine for the purpose of research, does not include user-defined policies. We therefore generated a set of synthetic policies. As part of the TIPPERS project, we conducted several town hall meetings and online surveys to understand the privacy preferences of users about sharing their WiFi-based location data. The surveys, as well as prior research [65, 69], indicate that users express their privacy preferences based on different user profiles (e.g., students for faculty) or groups (e.g., my coworkers, classmates, friends, etc.). Thus, we used a profile-based approach to generate policies specifying which events belonging to individual can be accessed by a given querier (based on their profile) for a specific purpose in a given context (e.g., location, time).

We classified devices in the TIPPERS dataset as belonging to users with different profiles (denoted by *profile(u_k)* for user u_k) based on the total time spent in the building and connectivity patterns. Devices which rarely connect to APs in the building (i.e., less than 5% of the days) are classified as *visitors*. The non-visitor devices are then classified based on the type of rooms they spent most time in: *staff* (staff offices), *undergraduate students* (classrooms), *graduate students* (labs), and *faculty* (faculty offices). As a result, we classified 31,796 visitors, 1,029 staff, 388 faculty, 1,795 undergraduate, and 1,428 graduate from a total of 36,436 unique devices in the dataset. Our classification is consistent with the expected numbers for the population of the monitored building. We also grouped users into groups based on the affinity of their devices to rooms in the building which is defined in terms of time spent in each region per day. Thus, each device is assigned to a group with maximum affinity. In total, we generated 56 groups with an average of 108 devices per group.

We define two kinds of policies based on whether they are an unconcerned user or an advanced user as described in Section 4.1. Unconcerned users subscribe to the default policies set by administrator which allows access to their data based on user-groups and profiles. Given the schema in Table 4.2 and the unconcerned user u_k we generate the following default policies:

- Data associated with u_k collected during working hours can be accessed by members of *group*(u_k).
- Data associated with u_k collected at any time can be accessed by overlapping members of *group*(u_k) and *profile*(u_k).

Advanced users define on average 40 policies, given the large number of control options (such as device, time, groups, profiles, and locations) in our setting. In total, the policy dataset generated contains 869,470 policies with each individual defining 472 policies on average and appearing as querier in 188 policies defined by others on average. The above policies are defined to allow access to data in different situations. Any other access that is not captured by the previous policies will be denied (based on the default opt-out semantics defined in Section 4.3).

Table 4.4 shows the schema and several sample policies generated for three different queriers. The *inserted_at* and *action* columns are skipped for brevity. Table 4.5 shows the corresponding object conditions which are part of two policies.

Table 4.4: Policy Table

id	table	querier	purpose
1	WiFi_Dataset	Prof.John Smith	Attendance
2	WiFi_Dataset	Bob Belcher	Lunch Group
3	WiFi_Dataset	Prof.John Smith	Attendance
4	WiFi_Dataset	Liz Lemon	Project Group
5	WiFi_Dataset	Prof.John Smith	Attendance

For the *Mall* dataset, the shops were categorized into six types based on the services they provide, such as arcade or movies. We also classified customers into regular and irregular, based on their shop visits. For each customer, we then defined two types of policies depending

Table 4.5: Policy Object Conditions Table

id	policy_id	attr_type	attr	op	val
1	1	int	owner	$=$	120
2	1	time	ts-time	$\geq$	09:00:00
3	1	time	ts-time	$\leq$	10:00:00
4	1	int	wifiAP	$=$	1200
5	2	int	owner	$=$	145
6	2	int	wifiAP	$=$	2300

on whether they were regular or irregular. Regular customers allowed shops they visit the most to have access to their location during open hours. Irregular customers shared their data only with specific shop types depending on whether there were sales or discounts. Finally, if a customer expressed an interest in a particular shop category, we also generated policies which allowed access of their data to the shops in the category for a short period of time (e.g., lightning sales). In total, this policy dataset generated on top of *Mall* dataset contains 19,364 policies defined for 35 shops (queriers) in the mall with 551 policies on average per shop.

Database System. We ran the experiments on an individual machine (CentOS 7.6, Intel(R) Xeon(R) CPU E5-4640, 2799.902 Mhz, 20480 KB cache size) in a cluster with a shared total memory of 132 GB. We performed experiments on MySQL 8.0.3 with InnoDB as it is an open source DBMS which supports index usage hints. We configured the *buffer_pool_size* to 4 GB. We also performed experiments on PostgreSQL 13.0 with *shared_buffers* configured to 4 GB.

4.8.2 Experimental Results

We first study the performance of the guarded expression generation algorithm (Experiment 1). Then, we validate the design choices in Sieve (Experiment 2) and compare the performance of Sieve against the baselines (Experiment 3). The previous experiments are performed on the MySQL system. Next, we study the performance of Sieve on PostgreSQL which, in contrast to MySQL, does not support index usage hints (Experiment 4). In the final experiment, we stress test our approach with a very large number of policies (Experiment 5). The first four experiments use the *TIPPERS* dataset and the final experiment the *Mall* dataset.

Experiment 1: Cost for generating Guarded Expressions and Effectiveness. The goal of this experiment is to study the cost of generating guarded expressions for a querier, as factor of the number of policies, and the quality of generated guards. To analyze the cost of guarded expression generation, we generate guarded expressions for all the users using the algorithm described in Section 4.5 and collect the generation times in a set. We sort these costs (in milliseconds) and average their value in groups of 50 users showing the result in Figure 4.6.

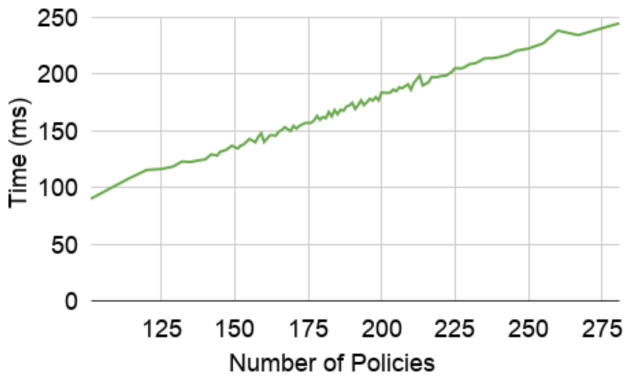


Figure 4.6: Guard generation cost.

The cost of guard generation increases linearly with number of policies. As guarded expression generation is also dependent on the selectivity of policies, number of candidate guards generated, which is also a factor of overlap between predicates, we sometimes observe a slight decrease in the time taken with increasing policies. The overhead of the cost of generating guarded expression is minimal, for instance, the cost of generating a guard for a querier with 160 policies associated (e.g., the student trying to locate classmates explained in Section 4.1) is around 150ms.

Table 4.6: Analysis of policies and generated guards.

	min	avg	max	SD
$\lvert\mathcal{P}_{u_k}\rvert$	31	187	359	38
$\lvert G\rvert$	2	31	60	10
$\lvert\mathcal{P}_{G_i}\rvert$	4	7	60	5
$\rho(G_i)$	0.01%	3%	24%	2%
Savings	0.99	0.99	1	7e−4

Table 4.7: Analysis of number of guards and total cardinality.

$\rho(G)$	$\lvert G\rvert$	
	low	high
low	227.2	537.0
high	469.0	1,406.7

We present the results of the analysis of the policies and generated guarded expressions in Table 4.6. Each user is affected, on average, by 187 policies ($\lvert\mathcal{P}_{u_k}\rvert$). This number depends on their profiles (e.g., student) and group memberships. Sieve creates an average of 31 guards per user with the mean partition cardinality (i.e., $\lvert\mathcal{P}_{G_i}\rvert$) as 7. The total cardinality of guards in the guarded expression is low (i.e., $\rho(G_i)$) which helps in filtering out tuples before performing policy evaluation. In cases with high cardinality guards (e.g., maximum of 24%), Sieve will not use force an index scan in that particular guard as explained in Section 4.6. Savings is computed as ratio of the difference between total number of policy evaluations without and with using the guard and the number of policy evaluations. This was computed on a smaller sample of the entire dataset and the results show that guards help in eliminating around 99% of the policy checks compared to policy evaluation.

Experiment 2.1: Inline vs. Operator Δ. SIEVE uses a cost model to determine for each guard whether to inline the policies or to evaluate the policies using the Δ operator. The Δ operator has an associated overhead of UDF invocation but it can utilize the tuple context to reduce the number of policies that need to be checked per tuple. For the purpose of studying this tradeoff in both inlining and using the Δ operator, we gradually increased the number of policies that are part of the partition of a guard and observed the cost of policy evaluation. As expected, we observed that when the number of policies are about

120, the cost of UDF invocations is amortized by the savings from filtering policies by the Δ operator (see Figure 4.7).

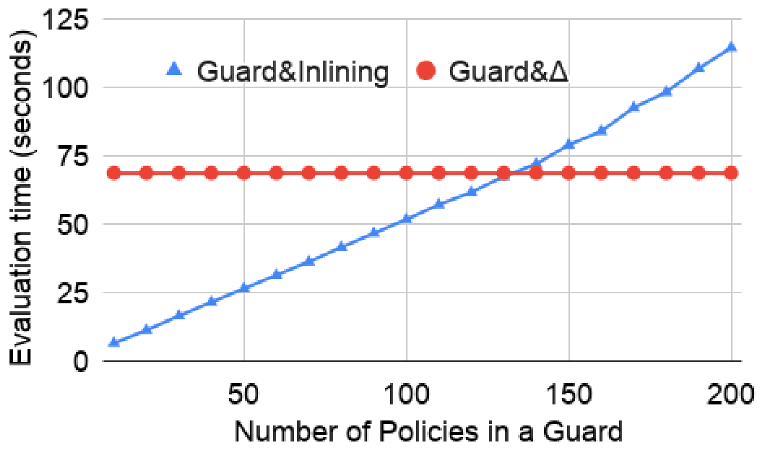


Figure 4.7: *Inlining* vs. Δ.

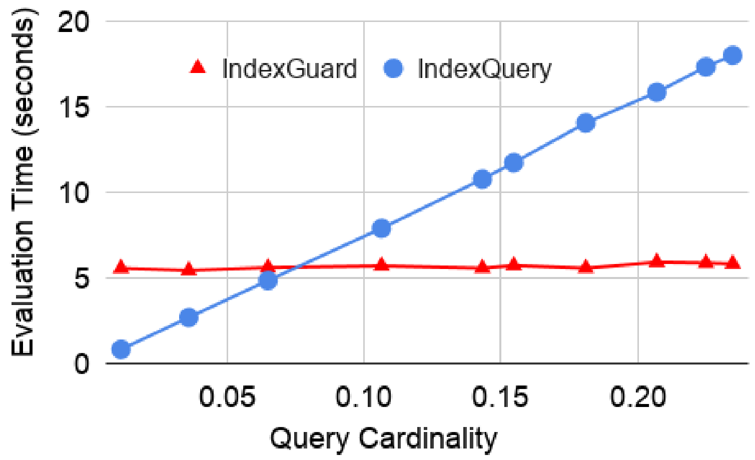


Figure 4.8: Index choice.

Experiment 2.2: Query Index vs. Guard Index. In Sieve, we use a cost model to choose between using the *IndexQuery* and *IndexGuards* as explained in Section 4.6. We evaluated this cost model by analyzing the cost of evaluation against increasing query cardinality for three different guard cardinalities (low, medium, high). Figure 4.8 shows the results averaged across these three guard cardinalities. As expected, at low query cardinality it is better to utilize *IndexQuery*, while at medium and high query cardinalities (> 0.07), *IndexGuards* are the better choice. Note that in both these options, guarded expressions are used as filter on top of the results from Index Scan.

Experiment 3: Query Evaluation Performance. We compare the performance of Sieve (implemented as detailed in Section 4.6) against three different baselines. In the first baseline, $Baseline_P$, we append the policies that apply to the querier to the `WHERE` condition of the query. Second, $Baseline_I$, performs an index scan per policy (forced using index usage hints) and combines the results using the `UNION` operator. Third, $Baseline_U$ is similar to $Baseline_P$ but instead of using the policy expression, it uses a UDF defined on the relation to evaluate the policies. The UDF takes as input all the attributes of the tuple. $Baseline_U$ significantly reduces the number of policies to be evaluated per tuple and is therefore an interesting optimization strategy for low cardinality queries. UDF invocations are expensive, so it might be preferable to execute the UDF as late as possible from the optimization perspective [59]. To preserve correctness of policy enforcement as defined in Section 4.3, UDF operations have to be performed before any non-monotonic query operations.

For each of the query types ($Q1$, $Q2$, Q_3), we generate a workload of queries with three different selectivity classes posed by five different queriers of belonging to four different profiles. The values chosen for these three selectivity classes (low, medium, high) differed depending upon the query type. We execute each query along with the access control mechanism 5 times and average the execution times. The experimental results below give the average warm performance per query. The time out was set at 30 seconds. If a strategy timed out for all queries of that group we show the value TO. If a strategy timed out for some of the queries in a group but not all, the table shows the average performance only for those queries that were executed to completion; those time values are denoted as t^+.

Table 4.8 shows the average performance for the three query types. The performance of $Baseline_P$ and $Baseline_U$ degrades with increasing cardinality of the associated query as they rely on the query predicate for reading the tuples. The relative reduction in overhead for Q3 for $Baseline_P$ at high cardinalities is because the optimizer is able to use the low cardinality join condition to perform a nested index loop join. The performance of Sieve

Table 4.8: Overall performance for $Q1$, $Q2$, and $Q3$ (in ms).

	$\rho(Q)$	$Baseline_P$	$Baseline_I$	$Baseline_U$	Sieve
	low	1,668	906	9,122	418
$Q1$	mid	15,356	910	23,575$^+$	453
	high	TO	937	TO	523
	low	860	916	7,787	407
$Q2$	mid	7,191	922	22,617$^+$	454
	high	29,765$^+$	962	TO	475
	low	883	881	14,379	477
$Q3$	mid	2,217	2,209	TO	476
	high	3,502	3,543	TO	521

and $Baseline_I$ remains the same across query cardinalities as they utilize the policy and guard predicates for reading the tuples and hence are not affected by the query cardinality. The increase in the speedup between these two sets of approaches clearly demonstrate that exploiting indices paid off. For $Baseline_P$, the optimizer is not able to exploit indices at high cardinalities and resorts to performing linear scan. In $Baseline_U$, the cost of UDF invocation per tuple far outweighed any benefits from filtering of policies. $Baseline_I$, generated by careful rewriting with an index scan per policy, performs significantly better than the previous two baselines. The performance degrade of $Baseline_I$ for Q3 is due to the optimizer preferring to perform the nested loop join first instead of the index scans. In comparison to all these baselines, Sieve is significantly faster at all different query cardinalities.

The extended results for $Q1$, $Q2$, and $Q3$ by querier profile are shown in Table 4.9, Table 4.10, and Table 4.11, respectively.

Experiment 4: Sieve on PostgreSQL. In the previous experiments we used MySQL, which supports hints for index usage, thus enabling SIEVE to explicitly force the optimizer to choose guard indexes. However, other DBMSs, such as PostgreSQL, do not support index usage hints explicitly (as discussed in Section 4.6.3). To study Sieve's performance in such systems, we implemented a Sieve connector to PostgreSQL using the same rewrite strategy but without index usage hints. To have a cumulative set of policies (i.e., the larger set of

Table 4.9: Comparison of performance for $Q1$ (in ms).

Profile	$\rho(Q)$	$Baseline_P$	$Baseline_I$	$Baseline_U$	Sieve
Faculty	l	1,560	972	9,398	357
	m	14,533	949	23,362$^+$	352
	h	TO	962	TO	413
Graduate	l	1,794	998	9,573	426
	m	16,737	994	23,735$^+$	495
	h	TO	990	TO	565
Undergrad	l	1,618	681	9,661	362
	m	15,432	751	23,692$^+$	394
	h	TO	720	TO	422
Staff	l	1,701	975	7,854	526
	m	14,722	946	23,511$^+$	571
	h	TO	1,077	TO	691

Table 4.10: Comparison of performance (in ms) for $Q2$.

Profile	$\rho(Q)$	$Baseline_P$	$Baseline_I$	$Baseline_U$	Sieve
Faculty	l	822	961	7,655	354
	m	6,929	975	22,502$^+$	354
	h	26,397	991	TO	362
Graduate	l	947	1,009	8,084	404
	m	7,806	1,028	22,676$^+$	506
	h	TO	1,080	TO	537
Undergrad	l	848	739	8,336	380
	m	7,156	725	22,863$^+$	368
	h	TO	769	TO	399
Staff	l	822	954	7,073	489
	m	6,874	960	22,425$^+$	589
	h	28,347	1,007	TO	603

policies contain the smaller set of policies) for evaluation, we chose 5 queriers to whom at least 300 policies apply. For each querier, we divided their policies into 10 different sets of increasing number of policies starting with smallest set of 75 policies. The order and the specific policies in these sets were varied 3 times by random sampling. The results in Figure 4.9 shows the average performance of different strategies for each set size averaged across queriers and the samples for `SELECT ALL` queries.

The four strategies tested in this experiment are: the best performing baseline for MySQL ($Baseline_I$(M)), the baseline in PostgreSQL ($Baseline_P$(P)), and Sieve in both MySQL and

Table 4.11: Comparison of performance (in ms) for $Q3$.

Profile	$\rho(Q)$	$Baseline_P$	$Baseline_I$	$Baseline_U$	Sieve
	l	892	871	14,279	372
Faculty	m	2,302	2,248	TO	379
	h	3,595	3,662	TO	405
	l	893	886	13,287	524
Graduate	m	2,183	2,200	TO	518
	h	3,486	3,487	TO	568
	l	881	884	10,601	619
Undergrad	m	2,174	2,200	TO	613
	h	3,512	3,446	TO	668
	l	865	885	11,947	319
Staff	m	2,211	2,188	TO	392
	h	3,512	3,576	TO	444

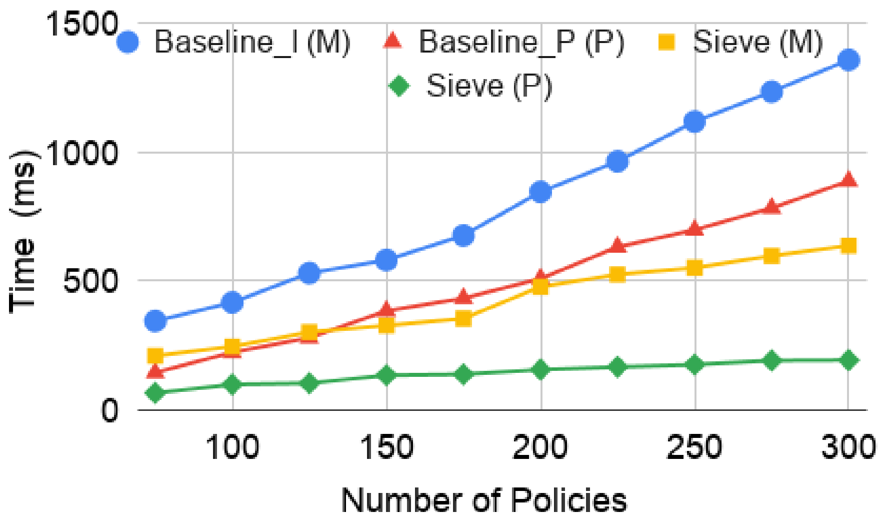


Figure 4.9: Sieve on MySQL and PostgreSQL.

PostgreSQL (Sieve (M) and Sieve (P)). The results show that not only Sieve outperforms the baseline in PostgreSQL but also the speedup factor w.r.t. the baseline is even higher than in MySQL. Additionally, the speedup factor in PostgreSQL is the highest at largest number of policies. Based on our analysis of the query plan chosen by PostgreSQL, it correctly chooses the guards for performing index scans (as intended by Sieve) even without the index usage hints. In addition, PostgreSQL supports combining multiple index scans by preparing a bitmap in memory. It used these bitmaps to *OR* the results from the guards whenever it was possible, and the only resultant table rows are visited and obtained from the disk. With a larger number of guards (for larger number of policies), PostgreSQL was also able to more efficiently filter out tuples compared to using the policies. Thus, Sieve benefits from reduced number of disk reads (due to bitmap) as well as a smaller number of evaluations against the partition of the guarded expression.

Experiment 5: Scalability. The previous experiment shows that the speedup of Sieve w.r.t. the baselines increases with an increasing number of policies, especially for PostgreSQL. We explore this aspect further on PostgreSQL using the *Mall* dataset where the generation of very large number of policies per querier (in this case the querier is a shop) is more feasible as we can simulate more customers. We used the same process than in Experiment 4 to generate cumulative set of policies by choosing 5 queriers/shops with at least 1,200 policies defined for them. Figure 4.10 reaffirms how the speedup of Sieve compared against the baseline increases linearly starting from a factor of 1.6 for 100 policies to a factor of 5.6 for 1,200 policies. We analyzed the query plan selected by the optimizer for the Sieve rewritten queries. We observed that with larger number of guards, PostgreSQL is able to utilize the bitmaps in memory to gain additional speedups from guarded expressions (as explained in Experiment 4). Also, this experiment shows that Sieve outperforms the baseline for a different dataset which shows the generality of our approach.

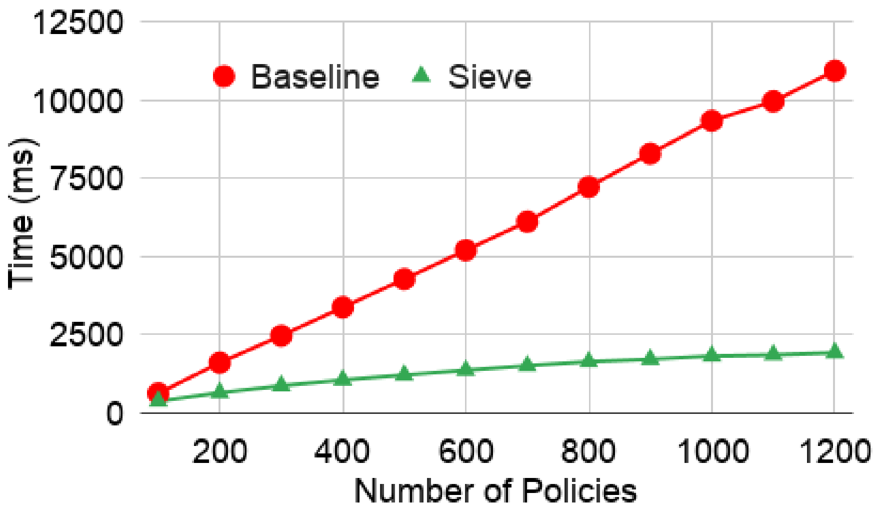


Figure 4.10: Scalability comparison.

4.9 Discussion

This chapter presented Sieve, a layered approach to enforcing large number of fine-grained policies during query execution. Sieve combines two optimizations: reducing the number of policies that need to be checked against each tuple, and reducing the number of tuples that need to be checked against complex policy expressions. Sieve is designed as a general purpose middleware approach and we have layered it on two different DBMSs. The experimental evaluation, using a real dataset and a synthetic one, highlights that Sieve enables existing DBMSs to perform efficient access control. Sieve significantly outperforms existing strategies for implementing policies based on query rewrite. The speedup factor increases with increasing number of policies and Sieve's query processing time remains low even for thousands of policies per query.

We believe that Sieve has opened up a fertile research area of co-optimizing policy enforcement in DBMSs. In the list below, we discuss some of the interesting extensions possible for Sieve in the increasing order of complexity.

- The guards generated in Sieve are based on single-attribute but could be easily extended to multi-attributes if the underlying DBMSs supported multi-attribute indexes (many do) and maintained histograms for joint attributes (rarely done).
- The cost model discussed in this chapter could be improved if better estimates are available from the query optimizer. This will improve the selection of guards in Sieve and thus improve the performance.
- Supporting complex policy expressions involving derived object conditions. An example of a derived object condition is *John wants to allow access to his location data only when he is with Prof. Smith.*

```
[W.owner = John AND W.wifiAP = (SELECT W2.wifiAP
    FROM WifiDataset AS W2
    WHERE W2.ts-time = W.ts-time AND W2.owner = ‘‘Prof.Smith’’)]
```

In this example W.wifiAP is a derived object condition and generating guards based on such conditions is an open research problem.

- In the existing policy model, only *Allow* or *Opt-In* is used as a policy action. While *Deny* policies can be expressed as *Allow* policies, for non-trivial deny policies this can lead to an explosion in the number of policies. On the other hand, *NOT* queries (required for deny policies) are not supported by indices in most modern DBMSs. A straightforward solution is execute the *Allow* and *Deny* guards independently on the DBMS and compute their set difference. But this approach has also many drawbacks as can be easily observed. Supporting both types of actions in policies efficiently remains an open problem.

DBMS	Index hints supported	Shortened Reference URL
MySQL	Yes	dev.mysql.com
PostgreSQL	No direct support	N.A.
ORACLE	Yes	docs.oracle.com
DB2	No direct support	N.A.
MongoDB	Yes	docs.mongodb.com

Table 4.12: Support for Index hints in DBMS

We chose MySQL to test our approach as it supports index hints which enabled Sieve to force the optimizer to utilize the index plan suited for evaluating policy expressions based on guards. In Table 4.12, we describe the support for index hints in few of the popular DBMS. This table is by no means exhaustive and it is quite possible that this information becomes outdated in the near future as they add/remove support for index hints from these DBMSs. Nevertheless this table serves as starting point for anyone who might be interested in developing the layer for Sieve on other DBMSs.

Chapter 5

Preventing leakages through data dependencies on access control protected data

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"
Arthur Conan Doyle, *The Sign of the Four*

Access Control mechanisms enforce policies by either allowing or denying access to a sensitive object. This way, they might not be sufficient for protecting sensitive objects since an adversary with background knowledge might infer information about the sensitive data from non-sensitive data. The problem of learning about sensitive data from non-sensitive data combined with metadata is known as the *inference problem* [45].

This chapter studies the inference problem in databases with discretionary access control with two classes of data dependencies. The first one consists of commonly used types of data constraints (such as functional dependencies, conditional functional dependencies, etc.) which are expressed in the form of denial constraints [26]. In modern Database Management Systems, raw data is transformed into derived data through various user defined functions [91]. Depending upon the property of the enrichment function, it might be also possible to reconstruct the raw data when only the derived data is shared. The second type of data constraint called *provenance-based dependencies (PBD)* captures these different forms of relationship between raw data and enriched data. These dependencies are publicly known and constitute the inference channels available to the adversary. They can use dependencies along with the disclosed non-sensitive data to limit the set of possible values that the sensitive data can (or cannot) take or in some cases be able to completely reconstruct the sensitive data.

The rest of the chapter is structured as follows. Section 5.1 describes the various concepts, defines the notations used in the paper, presents a high level problem definition, and explains the related work. Section 5.2 presents in detail the dependencies considered in our work along with the security model. Section 5.3 describes how the leakage of a sensitive data occurs through dependencies and explains how to detect leakage. Section 5.4 shows the different steps in our approach and presents the different algorithms. Section 5.5 evaluates our approach by varying different parameters of interest and finally Section 5.7 concludes the work by summarizing the work, limitations and possible future extensions.

5.1 Preliminaries

Let us consider the following example, with a simple conditional functional dependency (CFD), to illustrate the the inference problem.

Example 5.1. *Consider the Employees table, shown in Table 5.1, and the Wages table, shown in Table 5.2. Every tuple from the employee table specifies an employee in a department with their employee id (Eid), employee full name (EName), Zip code (Zip), state residence (State), role in the department (Role), number of hours they are allowed to work* *every week (WorkHrs), and the salary they earn per hour (SalPerHr). The Wages table stores the weekly salary of each employee in the Employee table, with attributes that could specify an employee by their employee id (Eid), name of the department they are part of (DeptName), and the total salary they earn per week (Salary) (derived from the WorkHrs attribute and the SalPerHr attributes). Consider an access control policy specified by a user to hide their WorkHrs. If there exists a conditional functional dependency on the Employee table, "[Role='Staff'] $\rightarrow$ [WorkHrs='30']" the adversary can learn about their weekly work hours by querying roles in the department and checking if it is equal to 'Staff'.*

Table 5.1: Employee details table.

Emp	Eid	EName	Zip	State	Role	WorkHrs	SalPerHr
e_1	34	A. Land	45678	AZ	Student	20	40
e_2	56	B. Hill	54231	CA	Faculty	40	200
e_3	78	C. Wood	53567	CA	Faculty	40	200
e_4	12	D. Boi	54231	CA	Staff	30	70

Table 5.2: Wages table.

Wages	Eid	DeptName	Salary
w_1	34	CS	800
w_2	56	EE	8000
w_3	78	CS	8000
w_4	12	BIO	2100

It could be trivially observed that the inference attack in Example 5.1 can be defended by simply hiding the corresponding *Role* when *WorkHrs* data is sensitive. In a more realistic setting, there could exist a Functional Dependency such as *SalPerHr* $\rightarrow$ *Role* or a more complex Denial Constraint such as $\forall t_i, t_j \in Emp\ \neg(t_i[State] = t_j[State]\ \wedge t_i[Role] = t_j[Role] \wedge t_i[SalPerHr] > t_j[SalPerHr])$. Both of these dependencies give more knowledge about the sensitive cell to the adversary. In such situations, identifying and preventing against potential inferences on sensitive data becomes challenging because the leakage can propagate through different dependencies. Furthermore, these dependencies can span over a number of tuples in the database and can include conditions on multiple attributes. This makes it difficult to determine the non-sensitive data that should be hidden to prevent inferences on sensitive data.

5.1.1 Background

Table 5.3: Notation for the chapter.

Notation	*Definition*
$\mathcal{D}$	A database instance
r	A database relation in $\mathcal{D}$
A	A attribute in R
c	A cell in R
$c.val$	The value of c
$Dom((c)$	The domain of the c
$\mathbb{C}$	Set of cells
δ	A schema level data dependency
$\tilde{\delta}$	An instantiated data dependency
Δ	Set of data dependencies
$Preds(\tilde{\delta})$	The set of predicates associated with a DC
$State(c)$	Set of possible values for a cell
sf	State function
$fn(\tilde{\delta})$	The function associated with a PBD
$\mathcal{U}$ and u	Set of Users and an individual user
Q	A Query and its metadata
p	An access control policy

Table 5.3 summarizes the commonly used notations in this chapter. Consider a database $\mathcal{D}$ as a **database instance** from a **database model** $\mathcal{D}$, that consists of a set of **relations** $\mathcal{R}$ and each relation $r \in \mathcal{R} = \{A_1, A_2, \ldots, A_n\}$ where A_j is an attribute in the relation. The notation $Dom(A_j)$ is used to denote the **domain** of the attribute and $|Dom(A_j)|$ denotes the number of unique values in the domain (i.e. **the domain size**)[1]. A relation contains a number of **tuples**, i.e., $R = \{\ldots, t_i, \ldots\}$. The notation t_i is to represent **a particular tuple** and **a set of tuples** is denoted by $\mathcal{T}$. The combination (or intersection) of a tuple and an attribute of a table is called a **cell**, which can be denoted by $t_i[A_j]$, in this notation system.

[1]We say the domain size in the context of an attribute with discrete domain values and for continuous attributes we discretized their domain values into a number of non-overlapping buckets.

For example in Figure 5.1, Employee is a relation in $\mathcal{D}$ and $t_1[EName]$ is a cell whose value is "B. Hill". In our access control setting, each data tuple t_i belongs to a user $u \in \mathcal{U}$ whose fine-grained access control policies can restrict the access over some specified cells in those tuple to designated users.

To simplify notational overhead, we use a cell representation instead of relation-tuple-attribute representation. We introduce the notation of cells for reason of flexibility and simplicity in discussing the fine-grained access control policies and the complex compositions among data dependencies. In this representation, a database can be regarded as a set of **cells**, $\mathcal{D} = \{\ldots, c_k, \ldots\}$, where we use the notation c_k to denote the **cell** with ID k where the value of each c_k is given by $t_i[A_j]$ from the previous representation. A cell can be **assigned** with a value and we use $c_k.val$ to denote the **cell value** assigned to the cell c_k. The **domain** of c_k is denoted by $Dom(c_k)$ which is the domain of the attribute A_j. The **size of the domain** is correspondingly denoted by $|Dom(c_k)|$, which is equivalent to the size of domain of the attribute that the cell is associated with. The cell notation system is equivalent to or interchangeable with the relation-tuple-attribute notation system, if considering a function (i.e. one-to-one mapping) *flatten* and its inverse function *flatten*$^{-1}$ that could map $t_i[A_j]$ to a cell in the cell representation and vice versa. Therefore, from now on, we will simply use the notation for a set of cells $\mathbb{C} = \{\ldots, c_k, \ldots\}$ to represent a row or a set of rows, a column or a set of columns, or a table in the database, if the context is clear.

Query model: The SELECT-FROM-WHERE query posed by a user u is denoted by Q. In our model, we consider that queries have associated metadata which consists of information about the querier and the context of the query. This way, we assume that for any given query Q, it contains the metadata such as identity of the querier (i.e., $Q^{querier}$) as well as the purpose of the query (i.e., $Q^{purpose}$). For example, $Q^{querier}$="Mr.Smith" and $Q_i^{purpose}$="Analytics".

5.1.2 Access Control Policies

Access control policies, or simply policies, are specified by the owner of the data tuple (a single user or a group of users) and marks one or more cells in the tuple as sensitive. When another user queries the database, the returned data has to be policy compliant (i.e., policies relevant to the user are applied to the database to hide sensitive cells). A policy P is expressed as $<OC,\ SC,\ AC>$. We explain each of the attributes of the policy in detail below.

• *OC* denotes the object conditions that identify the cells to which the policy applies. It consists of three parts: $\{R,\ \sigma,\ \Phi\}$. R is the table to which P applies, σ is the set of selection conditions that select the set of tuples $\mathcal{T}$ in R to which P applies, and finally Φ is the projection conditions that identifies the set of columns of t to which the policy is applied. We denote the set of cells identified by OC as $\mathbb{C}_{OC}$.

• *SC* denotes the subject conditions that identify the user for whom the policy applies based on attributes such as querier and purpose. This is modelled after the Purpose-based access control model [22].

• *Action* defines the enforcement operation that is either allow or deny. We assume that the default is allow in the absence of a policy controlling the access of a cell value in the database and therefor the *Action* to deny in the policies used in this paper. When a cell is *denied* by policy, we hide the value of the cell by setting it to *NULL*.

Example 5.2. *An example policy from the running policy is* $<\{Employee,\ Eid = \text{"C. Wood"},\ SalPerHr\},\ \{B.\ Hill,\ Analytics\},\ \{Opt\text{-}out\}>$. *In the first part (Object Conditions), the policy specifies the sensitive cell, which is in the Employee table and corresponding to the tuples belonging to 'Eid = "C. Wood"' with the attribute SalPerHr. The second part (Querier Conditions) of this policy indicates that it applies to queries from the user Querier = B. Hill and when the purpose of query is Analytics. The final component (Action) "deny" mandates* *that the cells identified through object conditions will be hidden from the querier's query results.*

Definition 5.1. ***(Sensitive Cell.)*** *A cell c is sensitive to a user u if there exists a policy P such that $c \in \mathbb{C}_{OC}$ where $OC \in P.OC$, $u \in P.SC$, and $P.Action = deny$. The set of cells sensitive to the user u is denoted by $\mathbb{C}_U^S$ (or simply $\mathbb{C}^S$ when the context is clear). The sensitive set cannot appear in the result of queries by U, which is restricted by the access control policies. Conversely, the set of non-sensitive cells are denoted by $\mathbb{C}^{NS}$ where $\mathbb{C}^{NS} = \mathcal{D} - \mathbb{C}^S$.*

From here on in the chapter, we assume that the set of sensitive cells identified by policies applicable to a user are known. We explain in Section 5.4 how the policy enforcement could be done at compile time through pre-processing.

5.1.3 Data Dependencies

Data dependencies restrict the possible set of values for a cell based on another set of values in the database instance. Thus, through existing dependencies, knowledge about sensitive cells restricted by access control policies can leak to queriers. We look at two forms of data dependencies[2] in this work: 1) Denial constraints (DCs) and 2) Provenance-based dependencies (PBDs). Consider the two tables from Example 5.1. In addition to the CFD mentioned earlier, we consider the following types of schema level data dependencies (specified at the level of attributes) in these two relations. We will explain the specification and semantics of these dependencies in the following subsection.

1. Key Constraint: $Key\{Eid\}$
2. Functional dependency (FD): $Zip \rightarrow State$.

[2]Other data dependencies such as Join dependencies (JD) and Multivalued dependencies are not common in a clean, normalized database and therefore not interesting to our problem setting.

3. Denial Constraint (DC): $\forall t_i, t_j \in Emp\ \neg(t_i[State] = t_j[State]\ \wedge t_i[Role] = t_j[Role] \wedge t_i[SalPerHr] > t_j[SalPerHr])$
4. Provenance Based Dependency (PBD): $Salary = fn(WorkHrs,\ SalPerHr)$

Data Constraints are traditional types of integrity constraints such as keys, foreign keys, functional dependencies (FDs), conditional functional dependencies (CFDs), and check constraints. We use the general model of denial constraints as our constraint defintition language to represent all forms of constraints, including aforementioned data constraints Thus, from now on, we use the term *data constraint* and *denial constraint* interchangeably in this paper. Denial constraints have been applied many fields, such as data cleaning and data synthesis [26, 62], to state and preserve the structure of the database. We chose DCs to express the data dependencies as it is capable of modelling common kind of dependencies (such as FDs, CFDs) and also flexible enough to model more complex dependencies among cells. There are also algorithms which have been developed for discovering DCs in a database [26]. The first-order formula form of DC makes it possible to evaluate the DCs using similar techniques as Access Control Policies as the object conditions in them are expressed in first order form too [81]. In this work, we extend the power of DCs and show the usage of DCs in access control policies. We use the general notation of denial constraints (as in [26]) to represent data dependencies at the schema level. Under this representation, a DC (δ) is a first-order formula of the form $\forall\, t_i, t_j, \ldots \in \mathcal{D}, \delta : \neg(Pred_1 \wedge Pred_2 \wedge \ldots \wedge Pred_N)$ where $Pred_i$ is the ith predicate in the form of $t_x[A_j]\theta t_y[A_k]$ or $t_x[A_j]\theta const$ with $x, y \in \{i, j, \ldots\}$, $A_j, A_k \in R$, *const* is a constant, and $\theta \in \{=, >, <, \neq, \geq, \leq\}$. We skip the universal quantifiers for DC if it is clear from the context. A DC is satisfied if at least one of the predicates evaluates to *False* which results in DC evaluating to *True*. We express the three different types of data constraints from Example 5.1 in DC as follows.

Example 5.3. *The previously mentioned data dependencies can be represented in DC format as follows.*

- δ_1*:* $\forall t_i, t_j \in R, \neg(t_i[Eid] = t_j[Eid])$

- δ_2*:* $\forall t_i, t_j \in R, \neg(t_i[Zip] = t_j[Zip] \wedge t_i[State] \neq t_j[State])$

- δ_3*:* $\forall t_i \in R, \neg(t_i[Role] =$ *"Faculty"* $\wedge t_i[WorkHrs] \neq$ *"40")*

The first DC corresponds to the key constraint (i.e. Key{Eid}) in Example 5.1. It formally states there does not exist two tuples with the same Eid's in the database. The second example is a functional dependency (i.e. Zip → State) written in the form of DC. The third DC example is the conditional functional dependency (i.e. [Role=Faculty] → [WorkHrs = 40]) in our running example, which is a unary DC applied to every tuple in the database with the "Role" assigned as a faculty.

Provenance Based dependencies: We present a model for dependencies used to capture the relationships between derived data and its inputs. From the running example, the Salary in the Wages table (see Table 5.2) is a attribute derived by executing the following query on the Employee table (see Table 5.1).

```
Wages(Salary) = (SELECT Salary
                 FROM (SELECT E.Eid, E.WorkHrs*E.SalPerHr AS Salary
                       FROM Employee as E, Wages as W
                       WHERE E.Eid = W.Eid))
```

Thus, for each employee tuple Salary is a function over WorkHrs and SalPerHr such that *Salary := fn(WorkHrs, SalPerHr) = WorkHrs × SalPerHr.* A *Provenance Based Dependency (PBD)* captures this relationship between the derived value and input values based on the function. The above function definition (*Salary := fn(WorkHrs, SalPerHr*) is the schema level representation of a Provenance Based Dependency. In general, given a function fn with $r_1, r_2, \ldots, r_n$ as the input cell and s_i as the derived or output cell, the PBD is represented by $fn(r_1, r_2, \ldots, r_n) = s_i$. If δ is a PBD, then $fn(\tilde{\delta})$ returns the corresponding function associated with it. Function definitions (or schema level PBD) are published as part of the schema (just like FDs, DCs) and is considered as background knowledge. We now define the property of invertibility of a function expressed in a PBD.

Definition 5.2. ***(Invertibility.)*** *Invertibility is a property of the function fn such that a given function $fn(r_1, r_2, \ldots, r_n) = s_i$ if it is invertible, given the output (s_i), it is possible to infer knowledge about the inputs ($r_1, r_2, \ldots, r_n$). Conversely if the fn is non-invertible, given s_i it does not lead to any inferences about the inputs.*

From the above example, the function fn to compute Salary is invertible as it is possible to learn about the inputs SalPerHr and WorkHrs given Salary and the background knowledge (such as domains). Cross product (Cartesian product) is another example of an invertible function[3]. On the other hand, complex user-defined functions (UDFs) (e.g., sentiment analysis code which outputs the sentiment of a person in a picture), oblivious functions, secret sharing, and many aggregation functions are non-invertible.

5.1.4 Problem definition

Given a database instance $\mathcal{D}$ which is represented in the cell notation presented earlier i.e., $\mathcal{D} = \{\ldots, c_k, \ldots\}$. The set of dependencies on the database instance are given by Δ. These are defined by an expert or automatically discovered by running data profiling tools such as Metanome [79]. The set of schema level dependencies is considered as background knowledge. Each of the data tuple is owned by a $u \in \mathcal{U}$ who defines the fine-grained access control policies that control sharing of the cells in the tuple. When answering queries involving accessing the c, its corresponding tuple policy should be enforced. Each query Q has policies applicable to

[3]The described model of invertibility does not make distinctions about different types of invertibility. Please see Chapter 5.6 for a more advanced model of invertibility called (m,n)-invertibility which captures this notion.

it depending on the query metadata such as querier, purpose and also the cells identified by the query conditions. For any given querier, based on the policies applicable to them a set of cells are sensitive ($\mathbb{C}^S \in \mathcal{D}$) and therefore should be hidden while answering any queries by the user. The goal is to ensure that given the result of the query Q, the adversary is not able to learn more about knowledge about any of the cells in $\mathbb{C}^S$ which are set of cells sensitive to them.

Fine-grained policies defined earlier are used to control whether certain cells are available or not when answering queries by User u. For example, in the Employees table, consider that there are two policies p_1, p_2 applicable to the user u. p_1 protects *WorkHrs* value of tuple t_2 against queries by User U_i and similarly, p_2 protects *SalPerHr* value of tuple t_3. These two cells are marked in a red box. Suppose now u_i asks the following two queries on the database instance.

```
Query 1: SELECT Eid, Role, WorkHrs, SalPerHr FROM Emp
```

```
Query 2: SELECT Eid, Salary FROM Wages
```

The query answer for Query 1 would not include $t_2[WorkHrs]$ and $t_3[SalPerHr]$ as these are protected by policy. However, as mentioned in Example 5.1, u_i can accurately guess it using the conditional functional dependency *[Role=Professor]* $\rightarrow$ *[WorkHrs=40]* and thus, inferring that the hidden cell is $t_2[WorkHrs] = 40$. Similarly, adversary can learn more about $t_3[SalPerHr]$ through the provenance based dependency *Salary* $= fn($*WorkHrs, SalPerHr*$)$. Therefore, it is not sufficient to hide the sensitive cells when adversary has knowledge of the dependencies. We describe the adversary model in detail below stating our assumptions w.r.t the data, dependencies, and policies.

Adversary model: We extend the meaning of policy compliance to include preventing inferences through existing data dependencies in the database. The user identified in the subject conditions of a policy (querier) is the adversary in our model. Therefore, we use adversary and querier interchangeably in this work. We assume that all the tuples as well as cells in a tuple are Independently and Identically Distributed (I.I.D), except for explicitly specified dependencies. All the dependencies are given (generated automatically or by an expert) and there exist no dependency violations in the database. An adversary has knowledge of all the data dependencies and can instantiate the dependencies using the cells that are available to them based on the policies. They are also aware that there are no dependency violations in the database. They can run different queries on the database and get results based on the non-sensitive cells.

We assume that the adversary has no knowledge about the access control policies and therefore do not know which cells are marked as sensitive[4]. Thus, based on the query results and data dependencies, the goal of the adversary is to determine what values a $c_i \in \mathbb{C}^S$ can (or cannot) take from its domain $Dom(c_i)$, given query result based on $\mathbb{C}^{NS}$ ($\mathcal{D} - \mathbb{C}^S$) and Δ which they could not infer only given Δ (i.e., all their queries are denied). Finally, we assume that there are no collusions among the queriers with different access control policies.

5.1.5 Related Work

The inference problem in databases occurs when knowledge about sensitive data can be learned from non-sensitive data as well as the background knowledge that is available to the adversary [45]. This problem has been studied extensively in the areas of Mandatory Access Control (MAC) and Multi-level relational databases. Qian *et al.*[83] developed a tool for analyzing multilevel relational databases to identify explicit inferences through foreign key relationships as inference channels. They recommend upgrading the foreign key relationships to prevent leakages when a user with low level clearance is able to learn data with higher level

[4]We leave the extension where policies are public and the adversary has knowledge about which cells are sensitive as future work.

sensitivity. Delugachi *et al.*[38], authors characterized different kind of data associations (e.g., part-of, is-a) and used a conceptual graph based analysis approach to identify the inference of classified information from unclassified information. While both these works look at the inference problem, their approach was limited to foreign keys and associations respectively.

The inference problem has been addressed through a query control approach where only consistent answers are allowed. Denning *et al.* [39] used authorization views as filters in front of the database which suppresses entire tuples instead of cells. This approach restricts more data than necessary to protect the sensitive cells. Thuraisingham [96] presented an approach where access to data is allowed by security constraints (similar to access control policies). In this work, the DBMS is augmented with an inference engine which looks at the security constraints and integrity constraints specified on the data. At query time, it determines if for a given query any of the security constraints might be violated directly or indirectly through the integrity constraints. Both of these works are based on the non-Truman model [89] which is not widely adopted compared to the Truman model of answering queries (where all possible answers to a query are returned which improves the utility of query answering).

The work done by Brodsky [20] is most similar to our work. They developed DiMon which can identify direct security violations (access control) and indirect security violations (through data constraints). Their model is based on MAC as well where each query-answer is associated with a security clearance. The model for constraints are based on as Horn clauses which can used to specify some of the integrity constraints but cannot express more complex constraints such as possible by Denial Constraints and Provenance Based Dependencies in our work.

Approaches for preventing unwanted disclosures from sensitive data have been also studied in secure data outsourcing. Vimercati *et al.*[37] identified the problem of improper leakage due to data dependencies in data fragmentation. They mark attributes as sensitive (using confidentiality constraints) and block the information flow from non-sensitive attribute to sensitive attributes through dependencies. Haddad *et al.* [58] studied the problem of identifying (at design time) modifications to access control policies specified on a database when inferences are possible on sensitive data items using dependencies. Albertini *et al.* [9] studied the orthogonal problem of increasing utility by exploiting data dependencies by extending access control authorizations using non-harmful data dependencies.

A different category of work is where inference channels are not explicitly specified. These works can be either classified into data-dependent privacy preserving or data-independent privacy preserving. In the first category, inference channels are learned directly from database instances and their distributions [102, 103, 76]. In the second category, the privacy preserving method is independent of data and only relies on the algorithm to achieve the required privacy while answering queries (e.g., Differential Privacy [41]).

Denial constraints have been heavily used in data cleaning for the purpose of expressing data dependencies and detecting their violations [34, 49, 62, 87]. For example, in Holoclean ([99] a hypergraph is constructed based on violations of data dependencies is used to drive the holistic cleaning of the database.

Finally, the well known chase procedure[44] is related to our work as well. *Chase* is used in data exchange scenarios [78] and checks if the source database can be transformed into a target database while satisfying all of the dependencies. The state function used in our work is inspired by the certain answer semantics used in chase algorithm. Even though chase procedure has been used in numerous scenarios [78], to the best of our knowledge it hasn't been used for checking data leakages through dependencies.

5.2 Our Approach

When enforcing policies to database tables, we consider specific database instances. Therefore, we first need to instantiate the data dependencies to discuss the information leakage through data dependencies to sensitive cells restricted by the policies. As mentioned in Section 5.1, we use δ to express a schema level constraint.

Definition 5.3. *(**Instantiated Dependency.**) The different instantiations of dependency δ with the different set of cells from the database instance are $\{\ldots, \tilde{\delta}_i, \ldots\}$. A particular instantiation $\tilde{\delta}_i$ with a set of cells $\mathbb{C}$ is denoted by $\tilde{\delta}_i(c_1, \ldots, c_n)$. We simplify this notation to $\tilde{\delta}_i(\mathbb{C})$ or simply $\tilde{\delta}_i$ when $\mathbb{C}$ associated with it is clear from the context.*

An example of an instantiated DC is given by $\tilde{\delta}_2(c_{10}, c_{11}, c_{17}, c_{18})$ where $c_{10}, c_{11}, c_{17}, c_{18}$ corresponds to $e_2[Zip], e_2[State], e_3[Zip]$ and $e_3[State]$, respectively. In the case of PBDs, we assume that the DBMS maintains provenance of the derived attribute in a database (see Table 5.4). We also maintain various metadata associated with the functions such as its definition, whether it is invertible or non-invertible, etc. Using this provenance database, function metadata, we generate instantiated PBDs: $\tilde{\delta}_5(c_7, c_8, c_{31})$ where c_7, c_8, c_{31} corresponds to $e_1[WorkHrs], e_1[SalPerHr]$ and $w_1[Salary]$, respectively. In a Provenance Based Dependency, we use $\mathbb{C}_{in}$ to denote the cells which are the input to the function f of PBD and similarly $\mathbb{C}_{out}$ to denote the output cell from the function. In this example, we have $\mathbb{C}_{in} = \tilde{\delta}_5(c_7, c_8)$ and $\mathbb{C}_{out} = c_{31}$.

Table 5.4: Provenance example.

Function	Derived Value	Input Values
fn_S	$w_1[Salary]$	$e_1[WorkHrs], e_1[SalPerHr]$
fn_S	$w_2[Salary]$	$e_2[WorkHrs], e_2[SalPerHr]$
fn_S	$w_3[Salary]$	$e_3[WorkHrs], e_3[SalPerHr]$
fn_s	$w_4[Salary]$	$e_4[WorkHrs], e_4[SalPerHr]$

Note that this instantiation of dependency does not have any assignment to the cells and therefore the dependency cannot be checked to see if it is satisfied or not until $c_k \in \mathbb{C}$ are assigned values. We first define the state of a cell which characterizes the knowledge available about the cell to the adversary and use that to define the assignment.

Definition 5.4. ***(State of a cell.)*** *A state of c_i denoted by $State(c)$ is the set of possible values that can be assigned to it from its domain ($Dom(c)$).*

The state of a cell c changes based on the knowledge of the adversary. When the cell is disclosed, the $State(c) = x$ where $x \in Dom(c)$. On the other hand, when a cell is sensitive and therefore hidden, $State(c) \subseteq Dom(c)$ based on various instantiated dependencies the cell is part of and adversary's background knowledge.

Definition 5.5. ***(Assignment and World)*** *An assignment to a cell c is a value, x, assigned to it from its state such that $x \in State(c)$. The assigned world, or simply world, is a set of assigned values, $\{x_1, x_2, \ldots, x_n\}$, to cells in an instantiated dependency given by $\tilde{\delta}(c_1 = x_1, c_2 = x_2 \ldots, c_n = x_n)$ where each $x_n \in State(c_n)$. The world is denoted by $W(\mathbb{C}) \in State(\mathbb{C})$ where $State(\mathbb{C})$ is the set of all possible worlds based on states of all cells in $\mathbb{C}$.*

In a Denial Constraint, $Preds(\tilde{\delta})$ returns the predicates from an instantiated dependency. For a given cell $c_i \in \mathbb{C}$ and a dependency $\tilde{\delta}$, $Preds(c_i, \tilde{\delta}(\mathbb{C}))$ returns the predicate(s) $Pred \in Preds(\tilde{\delta})$ such that $Pred = c_i \; \theta \; c_j$ or $Pred = c_i \; \theta \; const$ where $c_i, c_k \in \mathbb{C}$, and $const$ is a constant. The function $Preds(c_i, \tilde{\delta})$ returns ϕ if $\tilde{\delta}$ doesn't contain such a predicate.

Let $\mathbb{C}_{Pred}$ denote the cells associated with the $Pred$ such that $\forall c_i \in \mathbb{C}_{Pred}$, $Pred \in Preds(c_i)$. We define the evaluation function for a predicate as: $eval(Pred, W(\mathbb{C}_{Pred})) = True$ if the predicate evaluates to True based on the assignment of values to $\mathbb{C}_{Pred}$ from $W(\mathbb{C}_{Pred})$. Similarly, $eval(Pred, W(\mathbb{C}_{Pred})) = False$ if the predicate evaluates to False based on the assignment of values to $\mathbb{C}_{Pred}$ from $W(\mathbb{C}_{Pred})$. We also define the evaluation function based on $State(\mathbb{C}_{Pred})$ as

$$eval(Pred, State(\mathbb{C}_{Pred})) = \begin{cases} \textbf{True} \; if \; \forall \; W \in State(\mathbb{C}_{Pred}), \; eval(Pred, W(\mathbb{C}_{Pred})) = True \\ \textbf{False} \; if \; \forall \; W \in State(\mathbb{C}_{Pred}), \; eval(Pred, W(\mathbb{C}_{Pred})) = False \\ \textbf{Unknown} \; if \; \exists \; W_1, W_2 \in State(\mathbb{C}_{Pred}) \; such \; that \\ eval(Pred, W_1(\mathbb{C}_{Pred})) = True \; and \; eval(Pred, W_2(\mathbb{C}_{Pred})) = False \end{cases}$$

Similarly, the eval function for the instantiated dependency $\tilde{\delta}$ and an assignment $W \in State(\mathbb{C})$ returns true (i.e., $eval(\tilde{\delta}, W) = True$). We call this W *a valid world* for a dependency if it does not lead to dependency violations (i.e., assignments that do not violate the dependency). As our dependencies are expressed in Denial Constraints, W is a valid assignment to the $\tilde{\delta}$ if $\exists \; Pred \in Preds(\tilde{\delta})$ such that $eval(Pred, W(\mathbb{C}_{Pred})) = False$. An invalid assignment occurs when $W \in State(\mathbb{C})$ and $eval(\tilde{\delta}, W) = False$.

We now define a *State Function (sf)* which computes the $State(c^*)$ based on an instantiated dependency and the state of its cell set ($State(\mathbb{C})$ and $c^* \in \mathbb{C}$).

$$sf(c^* \mid \tilde{\delta}, State(\mathbb{C})) \coloneqq \{x \in State(c^*) \; \exists \; W \; \in State(\mathbb{C}), eval(\tilde{\delta}, W) = True\} \tag{5.1}$$

Considering all the dependencies ($\tilde{\delta}_j \in \Delta$) and their assignments from $\mathbb{C}_k \in \mathcal{D}$, the possible values for c^* is given by

$$sf(c^* \mid \Delta, State(\mathcal{D})) \coloneqq \bigcap sf(c^* \mid \tilde{\delta}_j, State(\mathbb{C}_k)) \; \forall \; \tilde{\delta}_j \in \Delta, \mathbb{C}_k \in \mathcal{D} \tag{5.2}$$

The state function can be generalized to compute the state of a set of cells $\mathbb{C}^*$ as follows

$$sf(\mathbb{C}^* \mid \tilde{\delta}, State(\mathbb{C})) \coloneqq \{\{\ldots, x_i, \ldots\} \mid x_i \in State(c_j)\{\ldots, c_j = x_i, \ldots\} \mid \\ c_j \in \mathbb{C}^* \; \exists W \; \in State(\mathbb{C}) \;\; eval(\tilde{\delta}, W) = \;\; True\}$$

Security model

We first define $State^{max}(\mathbb{C})$ as the set of possible values for all cells in $\mathbb{C}$ when the adversary has no knowledge about any of the cells (other than the previously mentioned background knowledge). We can compute the set of possible values for $c^* \in \mathbb{C}$ based on an instantiated dependency $\tilde{\delta}$ and $State^{max}(\mathbb{C})$ using the previously defined State Function.

$$sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) \coloneqq \{x \in State(c^*) \; \exists \; W \;\; \in State^{max}(\mathbb{C}) \;\; eval(\tilde{\delta}, W) = \;\; True\}$$

This returns the set of possible values for a c^* from the valid worlds for $\tilde{\delta}$ and its maximum achievable deniability value.

Upon sharing some of the cells in $\mathbb{C} \in \mathbb{C}^{NS}$, partially or completely, the adversary learns more about the state of $\mathbb{C}$. We denote this updated state of $\mathbb{C}$ as $State'(\mathbb{C})$. The new set of possible values for c^* based on adversary's knowledge is given by:

$$sf(c^* \mid \tilde{\delta}, State'(\mathbb{C})) \coloneqq \{x \in State(c^*) \; \exists \; W \;\; \in State'(\mathbb{C}) \;\; eval(\tilde{\delta}, W) = \;\; True\}$$

Equation 5.2 can be used to compute the full state based on $State^{max}(\mathcal{D})$, $State'(\mathcal{D})$, and the set of all dependencies Δ. We now define two different security models for our problem setting: 1) *Full Deniability* and 2) *k-value Deniability.* Full deniability occurs when the adversary cannot distinguish the actual value of the cell from any of the possible values in $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C}))$ (i.e., they learn no new knowledge from the dependencies and disclosure of non-sensitive cells).

Definition 5.6. ***(Full Deniability.)*** *For every sensitive cell $c^* \in \mathbb{C}^S$, based on all the dependencies $\tilde{\delta} \in \Delta$, we achieve full deniability if*

$$sf(c^* \mid \Delta, State^{max}(\mathcal{D})) = sf(c^* \mid \Delta, State'(\mathcal{D}))$$

Full deniability is sometimes hard to achieve and in the worst case might require denying almost all the cells in the database when a relatively large number of cells are marked as *sensitive* by the access control policies. Therefore, inspired by the well-known *k-anonymity* privacy, we relax our security definition to a novel definition called *k-value deniability.*

Definition 5.7. ***(k-value Deniability.)*** *For every sensitive cell $c^* \in \mathbb{C}^S$, based on all the dependencies $\tilde{\delta} \in \Delta$ and $\mathcal{D}$, we achieve k-value deniability when*

$$sf(c^* \mid \Delta, State^{max}(\mathcal{D})) - sf(c^* \mid \Delta, State'(\mathcal{D})) \leq k \cdot Dom(c^*)$$

Note: The authors in [97] identify three criteria for correctly enforcing fine-grained access control policies in relational databases. They are sound, secure, and maximal. Our approach achieves the soundness property as $\mathcal{D} - \mathbb{C}^S$ does not contain more information than $\mathcal{D}$. We extend the security property to prevent leakages through data dependencies with the above security definitions. Finally, we achieve the maximal property through *k-value Deniability* which maximizes utility.

5.3 Analysis of Leakage

This section discusses when the leakage of a sensitive cell happens due to the two types of dependencies considered. We present an efficient algorithm to compute the leakage based on the state function and discuss briefly about the composition of state functions based on this algorithm.

5.3.1 Leakage of a Sensitive Cell

When c^* is marked as sensitive to u by an access control policy, it is hidden by setting it to *NULL*. This removes the sensitive cell from the query results for that user u. However, if there exist an instantiated dependency $\tilde{\delta}$ which contains the sensitive cell, it is possible for the adversary to learn about the hidden sensitive cell. We explain the conditions under which it is possible for the adversary to learn about the sensitive cell through denial constraints and provenance based dependencies.

Denial Constraints: Suppose $\tilde{\delta}$ is a denial constraint and the predicate corresponding to the sensitive cell is given by $\mathit{Pred}(c^*)$. We first like to note the semantics of a denial constraint that at least one predicate in a DC has to be False in a valid and clean database (only valid worlds from $W \in \mathit{State}(\mathbb{C})$ such that $\mathit{eval}(\tilde{\delta}, W) = \mathit{True}$ are considered). When the denial constraint is not trivial and contains at least two predicates ($\mid \mathit{Preds}(\tilde{\delta}) \mid \geq 2$), after hiding the sensitive cell c^*, we have $\mathit{State}(c^*) \subseteq \mathit{Dom}(c^*)$ and thus eval($\mathit{Pred}(c^*)$, $\mathbb{C}$) $= \mathit{Unknown}$. However, when all the other predicates in the dependency evaluate to True (i.e., $\forall \mathit{Pred}_i \in \mathit{Preds}(\tilde{\delta}) \setminus \mathit{Pred}(c^*)\ \mathit{eval}(\mathit{Pred}_i, \mathit{State}(\mathbb{C})) = \mathit{True}$) the adversary can infer that $\mathit{eval}(\mathit{Pred}(c^*), \mathbb{C}) = \mathit{False}$ even if they do not know the exact value of c^*. We now formally state the theorem that states when leakage of a sensitive cell, c^*, through an instantiated dependency, $\tilde{\delta}$, happens based on sharing of non-sensitive values along with background knowledge (e.g., schema, data dependencies) of the adversary.

Theorem 5.1. *For a sensitive cell c^*, a data dependency $\tilde{\delta}$, $sf(c^* \mid \tilde{\delta}(State^{max}(\mathbb{C}))) \neq sf(c^* \mid \tilde{\delta}, State'(\mathbb{C})) \leftrightarrow \forall$ Pred $\in \{Preds(\tilde{\delta}) \setminus Pred(c^*)\ \}$, eval(Pred, State$(\mathbb{C})) = True$.*

Proof. When no information (other than background knowledge) is available to adversary, $State(c^*) = sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C}))$ which is the maximum number of possible values for c^* considering the valid worlds for the dependency $\tilde{\delta}$. However, when all the non-sensitive cells in $\mathbb{C}$ are shared and if all the other *Pred* in the $\tilde{\delta}$ evaluate to true (except for $Pred(c^*)$), as this is a denial constraint, the remaining predicate $(Preds(c^*, \tilde{\delta}_k))$ should evaluate to False. Thus, the modified state of the sensitive cell given by $State'(c^*) = sf(c^* \mid \tilde{\delta}, State'(\mathbb{C}))$ is limited by the possible values when $eval(Pred(c^*, \mathbb{C}) = False$. If the predicate $Pred(c^*)$ is non-trivial, the $Dom(State'(c^*)) < Dom(State(c^*))$ and therefore we have $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) \neq sf(c^* \mid \tilde{\delta}, State'(\mathbb{C}))$. Thus knowledge about the sensitive value is leaked.

On the other hand, when $\exists$ $Pred \in Preds(\tilde{\delta}) \mid eval(Pred, \mathbb{C}) = False$, it is not possible to infer the true value of $Pred(c^*)$ when the sensitive cell is hidden. Thus, the modified state, $State'(c^*)$, when all non-sensitive cells in $\mathbb{C}$ are shared, is the same as $State(c^*)$ based on $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = sf(c^* \mid \tilde{\delta}, State'(\mathbb{C}))$. Thus, no further knowledge about the sensitive cell is leaked. □

Provenance based Dependencies: Suppose δ is a provenance based dependency given by $fn(r_1, r_2, ..., r_n) = s_i$. In the instantiation of the PBD denoted by $\tilde{\delta}$, if the sensitive cell c^* is the output of the function then disclosing any of the input values leaks knowledge about c^*. Furthermore, if any of the input values is sensitive (i.e., c^*) disclosing the output value leaks knowledge about it only if the fn is invertible. If fn is non-invertible, disclosing output value does not leak any knowledge about c^*.

In both of these situations, we have to hide other cells in the cell set of instantiated dependency (c in $\mathbb{C}$) so as to prevent the leakage of the sensitive cell through this dependency. In DCs, our goal is to have one other predicate in the dependency that evaluates to unknown and thus making it impossible for the adversary to infer the truth value of $\mathit{Pred}(c^*)$. We achieve this by hiding a cell $c_j \in \mathbb{C}$ such that $\mathit{Pred}(c_j) \in \mathit{Preds}(\tilde{\delta}) \backslash \mathit{Pred}(c^*)$. This results in $\mathit{eval}(\mathit{Pred}(c_j), \mathit{State}(\mathbb{C})) = \mathit{Unknown}$. As now two predicates in $\tilde{\delta}$: $\mathit{Pred}(c_j)$, $\mathit{Pred}(c^*)$ evaluate to unknown (and either could evaluate to *False* to satisfy the DC semantics), it impossible for adversary to learn anything about the sensitive cell c^* through this $\tilde{\delta}$.

Note that we did not choose the non-sensitive cell in $\mathit{Pred}(c^*)$ as a candidate for hiding. If we chose to hide c_k when $\mathit{Pred}(c^*) = c^* \theta c_k$ in $\tilde{\delta}$ because of leakage on c^* (based on Theorem 5.1), the adversary can still infer that $\mathit{Pred}(c^*) = \mathit{False}$. This leaks knowledge about the combined state of the two hidden cells (c^* and c_k) and thus $sf(\{c^*, c_k\} \mid \tilde{\delta}, \mathit{State}^{max}(\mathbb{C})) \neq sf(\{c^*, c_k\} \mid \tilde{\delta}, \mathit{State}'(\mathbb{C}))$. Thus, to prevent any possible leakages on the sensitive cell c^* and its corresponding predicate $\mathit{Pred}(c^*)$, we choose the cell to hide from other predicates. We now define the cueset of c^* from $\tilde{\delta}$ as follows.

Definition 5.8. ***Cueset:*** *When the condition (Theorem 5.1) for leakage is met, the set of non-sensitive cells, from the cell set of an instantiated dependency $\tilde{\delta}$ that can leak knowledge about the sensitive cell, is called a cueset*[5]. *Conversely, denying at least one of the cells in the cueset will make c^* secure w.r.t that dependency.*

For each such data dependency $\tilde{\delta}_k$ of type denial constraint, the *cueset* for a sensitive value c^* based on $\tilde{\delta}$ and $\mathbb{C}$ is given by

$$\mathit{cueset}(c^*, \tilde{\delta}, \mathbb{C}) = \{c_i \in \mathbb{C} \ \exists \ \mathit{Pred}(c_i) \in \mathit{Preds}(\tilde{\delta}) \backslash \mathit{Pred}(c^*)\} \tag{5.3}$$

As previously mentioned, the cue-set for a sensitive cell does not contain itself and the other non-sensitive cell from $\mathit{Pred}(c^*)$. The complete set of cueset for the sensitive cell based on

[5]As this set of values gives a *cue* about the sensitive value to the adversary.

the set of all dependencies Δ and $\mathcal{D}$ is given by

$$cueset(c^*, \Delta, \mathcal{D}) = \{\ cueset(c^*, \tilde{\delta}, \mathbb{C})\ \forall\ \mathbb{C} \in \mathcal{D}, \forall\ \tilde{\delta} \in \Delta\} \tag{5.4}$$

If the dependency $\tilde{\delta}$ only contains a single predicate then the cueset will be empty based on the above definition. We handle such dependencies by hiding the non-sensitive cell in $Pred(c_k)$. In such instantiated dependencies, it is not possible to prevent the leakage of the combined state space of c^* and c_k.

In the case of PBDs, the cueset is determined by whether $c^* \in \mathbb{C}_{out}$ or $c^* \in \mathbb{C}_{in}$ as well as invertibility of the function $fn(\tilde{\delta})$. Suppose we have $c^* \in c_{out}$, $\tilde{\delta}$ is the instantiated PBD, and $\mathbb{C}$ is the set of all cells in $\tilde{\delta}$, then $cueset(c^*, \tilde{\delta}, \mathbb{C}) = \{c_i \in \mathbb{C}_{in}\ \}$. Now if $c^* \in \mathbb{C}_{in}$, if the $fn(\tilde{\delta})$ is non-invertible $cueset(c^*,\ \tilde{\delta},\ \mathbb{C}) = \phi$, otherwise $cueset(c^*, \tilde{\delta}, \mathbb{C}) = \{c_i \in \mathbb{C}_{out}\}$ when $fn(\tilde{\delta})$ is invertible.

5.3.2 Computing Leakage

We compute the leakage on a sensitive cell due to an instantiated dependency using the state function. Leakage is defined as the difference in the state of sensitive cell due to sharing of the cueset of an instantiated dependency and represents adversary's inferred knowledge about the cell. The state of a cell is represented by: $State(c) = [low,\ high,\ minus\_set]$. The first element, *low*, and the second element, *high*, denote the starting value and ending value of the range of values in the state of c, respectively. The last element, *minus_set*, represents the set of values in the domain which are not valid assignments for c. For example, when $State(c) = [1, 5, \{3, 4\}]$ then it contains the set of values $\{1,\ 2,\ 5\}$. For larger states of c, this representation helps in compressing significantly the number of values to be maintained.

Denial Constraints: As defined in Equation 5.1, to compute the state of c^* based on the instantiated dependency $\tilde{\delta}$ we must check for each value in $State(c^*)$ whether it is possible to find a valid world W from $State(\mathbb{C}) = State(c_1) \times State(c_2) \times \ldots \times State(c_n)$ such that for the dependency $\tilde{\delta}$, $eval(\tilde{\delta}, W) = True$. When the adversary has no knowledge about any of the cells $State(c_i) = State_{max}(c_i)$ which is the maximum set of possible values for that cell (based on the valid worlds). In the case of denial constraints for each of the predicates $Pred_i \in Preds(\tilde{\delta})$ (including $Pred(c^*)$), $eval(Pred_i, \mathbb{C}) = unknown$. Thus, $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = Dom(c^*)$. The updated state of a cell c_i is computed based on set of all its cuesets. In Algorithm 2, we initialize the *minus_set* as empty and *low* and *high* values as minimum and maximum values of $Dom(c_i)$ respectively. For each cueset, we retrieve the dependency corresponding to it (*cueset.dep*) and the predicate corresponding to the c_i for which we wish to compute the state. We represent the non-sensitive cell in $Pred(c_i)$ as c_k. We set the three elements of $State(c_i)$ based on the θ as shown in the algorithm.

Provenance Based Dependencies: We use a simple model of leakage for PBDs only distinguishing between whether sensitive cell c^* is in $\mathbb{C}_{out}$ or $\mathbb{C}_{in}$ and whether the function is invertible is invertible or not. Thus, with a PBD when $c^* \in \mathbb{C}_{out}$ of a $\tilde{\delta}$, we have $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = Dom(c^*)$ (when no cells in $\mathbb{C}_{in}$ are shared). The updated state when all the input cells are shared is given by $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = \{c^*.val\}$ (complete leakage). When $c^* \in \mathbb{C}_{out}$ of a $\tilde{\delta}$ and $fn(\tilde{\delta})$ is invertible, we have similarly $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = Dom(c^*)$ (when no cells in $\mathbb{C}_{out}$ are shared). The updated state when all the output cells are shared is given by $sf(c^* \mid \tilde{\delta}, State^{max}(\mathbb{C})) = \{c^*.val\}$ (complete leakage).

5.3.3 Composing Leakages

In Figure 5.1, we have a set of sensitive cells denoted by $c_1^*, \ldots, c_n^*$ which is involved in various instantiated dependencies. For example, consider c_1^* which is part of two instantiated dependencies $\tilde{\delta}_1$ which is a Functional Dependency and $\tilde{\delta}_2$ which is a Provenance Based De-

Algorithm 2: Compute state based on instantiated dependencies (denial constraints)

```
Function ComputeState($c_i$, $cuesets$):
    minus_set = { }
    low, high = $min(Dom(c_i))$, $max(Dom(c_i))$
    for $cueset \in cuesets$ do
        $\tilde{\delta}$ = cueset.dep
        $c_i$, $\theta$ , $c_k$ = $Pred(c_i) \in Preds(\tilde{\delta})$
        switch $\theta$ do
            case "$\leq$" do
                if $high > c_k.val$ then
                    high = $c_k$.val
            case "$\geq$" do
                if $low < c_k.val$ then
                    low = $c_k$.val
            case "$\neq$" do
                minus_set=minus_set $\cup$ $c_k$.val
            case "$=$" do
                low = $c_k$.val, high = $c_k$.val
                minus_set= { }
                break
        end
    end
    state = [low, high, minus_set]
    return $state$
```

pendency. The corresponding cuesets are: $cueset(c^*, \tilde{\delta}_1, \mathbb{C}) = \{c_1, c_2, c_3\}$ and $cueset(c^*, \tilde{\delta}_2, \mathbb{C}) = \{c_1, c_5, c_6\}$ which we will denote as $\mathbb{C}_1$ and $\mathbb{C}_2$, respectively.

The composed state of c^* based on these two instantiated dependencies is calculated using *horizontal composition* which is the intersection of the states derived based on the individual instantiated dependencies. It is given by $sf(c^* \mid \{\tilde{\delta}_1, \tilde{\delta}_2\}, \{\mathbb{C}_1, \mathbb{C}_2\}) = sf(c^* \mid \tilde{\delta}_1, \mathbb{C}_1) \cap sf(c^* \mid \tilde{\delta}_2, \mathbb{C}_2)$.

After selecting to hide one of the cells in the cueset, we have to recursively generate the newly hidden cell's cueset. For example, consider that c_2 is chosen to be hidden to protect c_1^* and it is marked as sensitive. As c_2 is part of the instantiated dependency $\tilde{\delta}_3$ (which is a

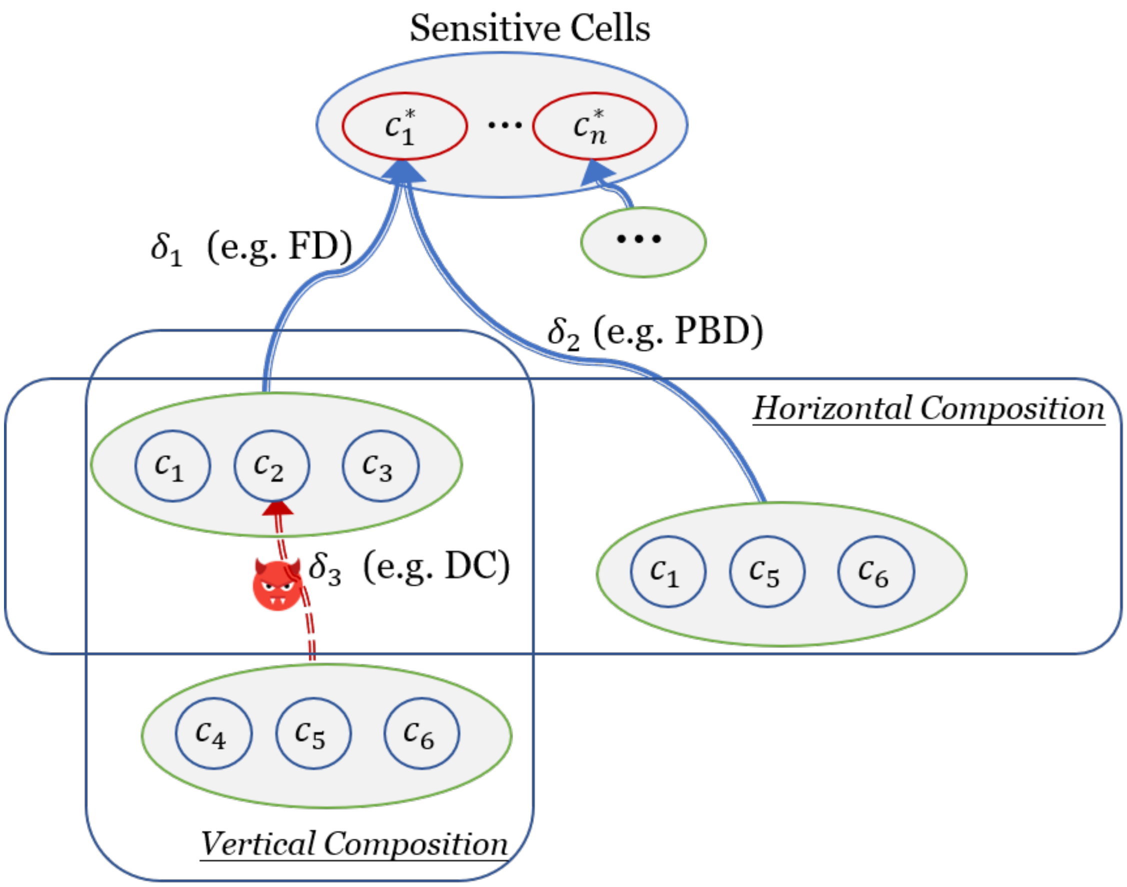


Figure 5.1: Leakage of a sensitive cell due to different instantiated dependencies.

Denial Constraint), we generate the corresponding cueset as: $cueset(c^2, \tilde{\delta}_3, \mathbb{C}) = \{c_4, c_5, c_6\}$ (denoted as $\mathbb{C}_3$). We now need to verify whether this new cueset leaks substantial knowledge on c_2 which further results in leakage of c_1^* through *vertical composition*. For performing vertical composition, we first update $State(c_2) = sf(c_2 \mid \tilde{\delta}_3, \mathbb{C}_3)$ and update $State'(\mathbb{C})$ with the modified state of c_2 which we then use to compute state of c^* as follows.

$$sf(c^* \mid \tilde{\delta}_1, \mathbb{C}_1) = \{x \in State(c^*) \mid \exists W \in State'(\mathbb{C}) \ eval(\tilde{\delta}_1), W) = \ True\}$$

5.4 Preventing data leakages

Given a database $\mathcal{D}$ which is a collection of cells and a set of data dependencies Δ, and a set of fine-grained access control policies P_u that identify the cells ($\mathbb{C}^S$) that should be denied while answering the queries by u. Our goal is to generate $\mathcal{D}'$, on which the queries by u are answered, and which protects the sensitive cells $c_i \in \mathbb{C}^S$ while maximizing utility. Utility is defined as the number of cells that are shared from $\mathcal{D} - \mathbb{C}^S$ while meeting the deniability requirement.

We perform the following steps to prevent data leakages of sensitive cells. As shown in Figure 5.2, the first step is policy enforcement, which takes as input the policies applicable to a user and produces a set of cells which are marked as sensitive. In the next step, for each of the sensitive cell, we instantiate their relevant dependencies and detect cuesets for them. Finally, we select cells to hide from the cueset until the specified deniability requirement is met. These steps are done by pre-processing at compile time as all the necessary information to do so is available prior to query time.

5.4.1 Policy Enforcement

The goal of policy enforcement is to identify the access permissions of each cells for a given user. For each tuple and its attribute values, metadata is added after checking the policies [22]. As an example, consider the following policy, “Bobby denies access of his *Work Hours* (from Employee Table) to Danny”. For each tuple in Employee table, we will retrieve all such policies that are applicable to them and then for each tuple, we identify the cell to be denied for specific queriers. We encode the appropriate permissions for cell and querier combination. This metadata can be stored as additional columns or as repeated rows. If the total number of unique queriers are small because the policies are specified for groups of

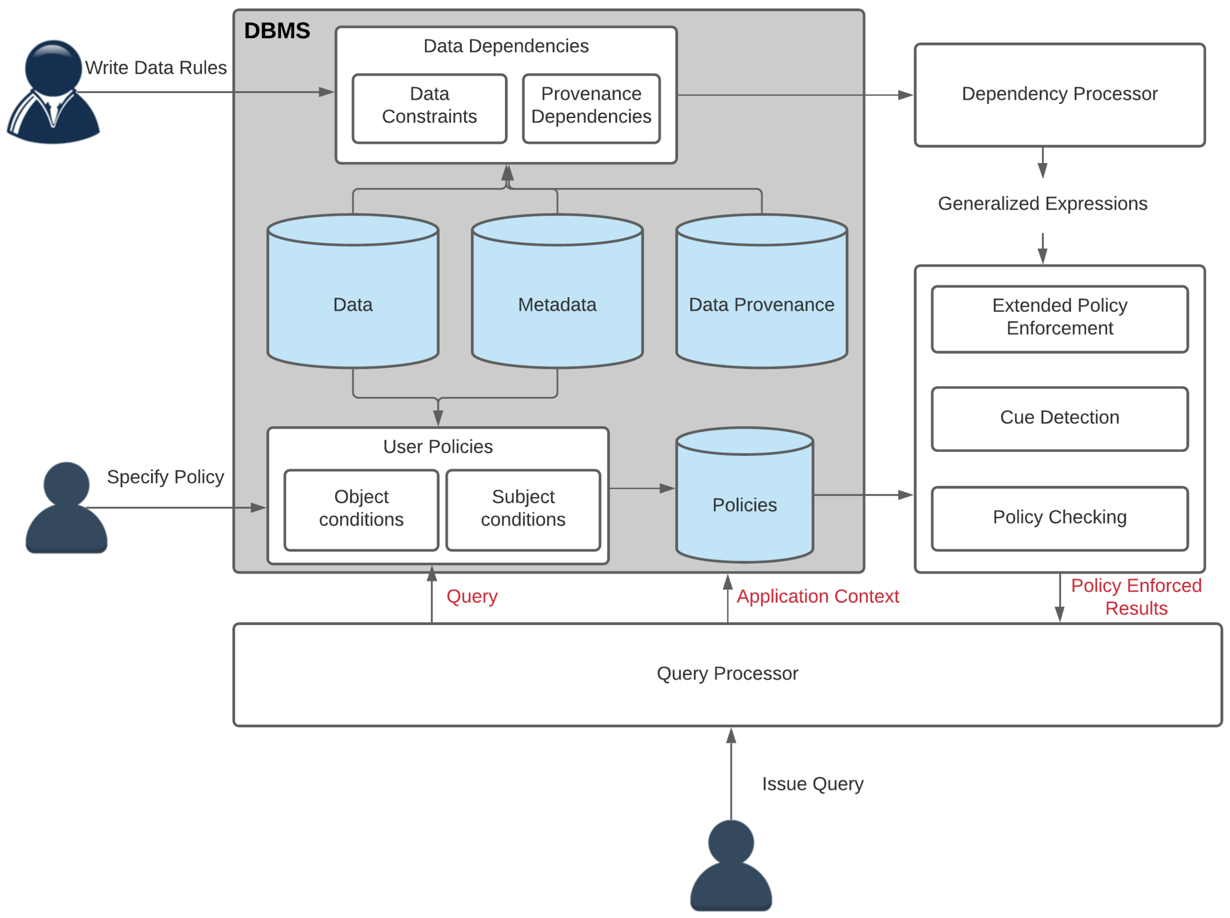


Figure 5.2: System architecture.

queriers (e.g., role, profile), we can also store the policy decisions in a bitmap (1 for allow and 0 for deny) per attribute value. The starting point of our algorithm is a policy enforced database where a set of cells ($\mathbb{C}^S$) are marked as sensitive based on the access control policies.

5.4.2 Cueset Detection

In this section, we present an algorithm to detect cuesets for a given sensitive cell, the set of data dependencies, and a database instance.

The first step in Algorithm 3 is to take each schema level dependency δ and instantiate it with the sensitive cell c^* and the appropriate selection of the set of other cells $\mathbb{C} \in \mathcal{D}$.

The instantiation of the dependency is hinged on the type of dependency. For a unary dependency, instantiation is only based on the tuple the sensitive cell is part of. For a binary dependency (most DCs), instantiation is based on the pairwise comparison of tuple containing the sensitive cell as well as other tuples in the database. Similarly for a N-ary dependency, the tuple consisting of the sensitive cell will be compared against the set of other N tuples. The number of comparisons required for instantiation depends upon the number of predicates in the dependency. In most common case of binary dependencies, for a given database instance $\mathcal{D}$ of size $\mid \mathcal{D} \mid$, in the worst case for each sensitive cell there would be $\mid \mathcal{D} \mid -1$ instantiations. In order to reduce the number of instantiations, when the predicate of c^* is of the following form $Pred(c^*) = c^* \theta c_k$, we derive the condition when it evaluates to False. For example, suppose we have $c^* = 5$ and $\theta =$ "$>$". We only instantiate the tuples with corresponding attribute (attribute of c^*) value satisfying the condition (> 5 from our example). This optimization is not possible when the sensitive predicate is of the form $Pred(c^*) = c^* \theta const$. Thus for each schema level dependency, we have a set of dependency instantiations given by $S_{\tilde{\delta}}$ and set of instantiations for all dependencies is given by S_{Δ}.

In the next step, we check for each of the instantiated dependency whether we need to generate a cueset based on Theorem 5.1. We verify the condition by assigning values to each $Pred_j$ (except for $Pred(c^*)$) for its corresponding $\mathbb{C}_{Pred_j}$ and verifying if $eval(Pred_j, State(\mathbb{C}_{Pred_j})) = True$. If one of the predicates evaluates to False, we generate an empty cueset and move onto the next instantiation. On the other hand, if the condition is met, we generate the cueset by iterating through all the predicates $Pred_j$ (except for $Pred(c^*)$) and adding the cells to the cueset corresponding to that dependency instantiation. The exception to this rule is when the instantiated dependency contains only a single predicate. We generate a cueset consisting of the non-sensitive cell (c_k) in $Pred(c^*)$. After iterating through all the dependency instantiations, we return the *cuesets* which is a set of cuesets.

Algorithm 3: Cueset detection

```
1  Function CuesetDetect($c^*$, $\Delta$, $\mathcal{D}$):
2      $S_{\Delta} = \{\ \}$
3      for $\delta \in \Delta$ do
4          $S_{\tilde{\delta}} \leftarrow$ instantiate($\delta$, $c^*$, $\mathcal{D}$)        $\triangleright$ Set of $\tilde{\delta}$ for $\delta_i$ $S_{\Delta}$.add($S_{\tilde{\delta}_i}$)
5      end
6      cuesets = { }
7      for $\tilde{\delta}_i \in S_{\Delta}$ do
8          cueset.dep = $\tilde{\delta}_i$
9          if $|\ Preds(\tilde{\delta}_i)\ | == 1$ then
10             cueset.add($\{c_k\}$)        $\triangleright$ $Pred(c^*) = c^* \theta c_k$
11         else if $\exists Pred_j \in Preds(\tilde{\delta}_i) \setminus Pred(c^*), eval(Pred_j, State(\mathbb{C}_{Pred}) = False$ then
12             cueset = $\phi$
13         else
14             cueset = $\{\ c_i \in \tilde{\delta}.\mathbb{C}\ \exists\ Pred(c_i) \in Preds(\tilde{\delta}) \setminus Pred(c^*)\ \}$
15         end
16         cuesets.add(cueset)
17     end
18     return $cuesets$
```

5.4.3 Selecting Non-Sensitive Cells to Hide

After generating the cuesets, we have to select the cells to hide from the cueset to protect the corresponding sensitive cell. We first present a greedy algorithm for *Full Deniability* in Algorithm 4. For a given sensitive cell c^*, the algorithm identifies the cuesets based on the instantiated dependencies by calling Function 3 (*Step 2*). Then it iterates through the set of cuesets and for each cueset, the algorithm selects a cell to hide (*Step 3-6*). Then it identifies the cuesets of the hidden cell and these new cuesets are added to list of cuesets (*Step 7*). The previous cueset is removed after this step or if it already contains a hidden cell (*Step 8*). We repeat this process until list of cuesets is empty which means every cueset has at least one hidden cell and thus we achieve Full deniability for the sensitive cell.

Holistic Full Deniability: In Algorithm 4, we hide a cell in the cueset when it does not already contain a hidden cell. Therefore, it is possible to minimize the number of extra

```
Algorithm 4: Full deniability
    Input: Sensitive cell $c^*$, Data dependencies $\Delta$
    Output: $\mathbb{C}^S$
    Data: $\mathcal{D}$
1   $\mathbb{C}^S = \{ c_* \}$
2   cuesets ← Cuesetdetect($c^*, \Delta, \mathcal{D}$)
3   while $cuesets \neq \phi$ do
4       cs ← cuesets.get()                                  ▷ any cueset
5       if $cs \cap \mathbb{C}^S = \phi$ then
6           $c_i$ ← cs.get()                            ▷ any cell in the cueset
7           $\mathbb{C}^S = \mathbb{C}^S \cup c_i$ cuesets.add(Cuesetdetect($c_i, \Delta, \mathcal{D}$))
8       cuesets.remove(cs)
9   end
10  return $\mathbb{C}^S$
```

hidden cells if we consider a holistic approach with all the sensitive cells. In the holistic version of Algorithm 4 we pass the set of sensitive cells $\mathbb{C}^S$ (instead of a single cell $c^* \in \mathbb{C}^S$). After identifying the cuesets for all the sensitive cells, we execute a *Minimum-Subset-Cover* of the cuesets to obtain good candidates (for hiding) that covers the maximum number of cuesets. We hide the cells in the Minimum-Subset-Cover and identifies the cuesets for them. The rest of the algorithm works similarly to the previous.

5.4.4 k-value Deniability

We now present a greedy algorithm for achieving *k-value deniability* where k is the minimum required deniability factor for a sensitive cell c^*, which means that there always exist a minimum of k possible worlds for the sensitive cell and the adversary cannot infer anything beyond that. In this algorithm, we start similar to *Full deniability* algorithm and identify the cuesets for a given sensitive cell c^*. We compute state of c^* w.r.t each cueset (cs.parent_state) when all the cells in it are disclosed by calling the function in Algorithm 2 and also the combined state (c^*.state) w.r.t all the cuesets. If c^*.state $\geq$ k, then we terminate. Otherwise, we sort the cuesets in the ascending order of cs.parent_state. We select the first cueset (the one that has highest leakage on the sensitive cell) and pass it to the function in Algorithm 6 along with set of already hidden cells (*toHide*).

```
Algorithm 5: k-value deniability
   Input: Sensitive cell $c^*$, Data dependencies $\Delta$, Deniability parameter $k$
   Output: $\mathbb{C}^S$
   Data: $\mathcal{D}$
1  toHide = { $c^*$ }
2  cuesets ← Cuesetdetect($c^*, \Delta, \mathcal{D}$)
3  for cs ∈ cuesets do
4      cs.parent_state = ComputeState($c^*$, cs)
5  end
6  $c^*$.state ← ComputeState($c^*$, cuesets)
7  if $c^*$.state >= $k$ then
8      return
9  Sort cuesets in the ascending order
10 while $c^*$.state < $k$ do
11     lcs ← cuesets.getFirst()
12     toHide ← hideR(toHide, lcs)
13     cuesets.remove(lcs)
14     $c^*$.state = ComputeState($c^*$, cs)
15 end
16 return toHide
```

In Algorithm 6, we first check if the cueset contains an already hidden cell and return to the algorithm if that is the case. If the intersection of cueset with the set of already hidden cells is empty, we retrieve a cell (*hideCell*) from cueset, add it to *toHide*, and identify *hideCell*'s cuesets. We compute the combined state of *hideCell* based on all the cuesets and, if there is no leakage (i.e., *hideCell*.state = *Dom*(*hideCell*)), we return the updated toHide. Otherwise, for each cueset of *hideCell*, we compute *hideCell.parent_state* and recursively call Algorithm 6 if there is leakage.

Holistic Version: We have also implemented a holistic version of the *k-value deniability* algorithm which, just like the holistic version of *Full deniability*, use Minimum-Subset-Cover to decide good candidates for hiding first in the cuesets. The rest of the implementation is the same as presented in the non-holisitic version.

Algorithm 6: Hiding of cuesets based on leakage

```
Function hideR(toHide, cueset):
    if $cueset \cap toHide \neq \phi$ then
        return toHide
    hideCell ← cueset.get()                                  ▷ any cell in the cueset
    toHide = toHide ∪ { hideCell }
    cuesets ← CuesetDetect(hideCell, $\Delta$, $\mathcal{D}$)
    hideCell.state = ComputeState(hideCell, cuesets)
    if $cuesets = \phi$ OR $hideCell.state == Dom(hideCell)$ then
        return toHide
    for $hcs \in cuesets$ do
        hcs.parent_state = ComputeState(hideCell, hcs) if size(hcs.parent_state) ==
          1 then
            toHide = hideR(toHide, hcs)                      ▷ full leakage
    end
    return toHide
```

5.5 Experimental Evaluation

5.5.1 Experimental setup

Dataset. We use the synthetic Tax dataset from [26]. Each record represents an individual's address and tax information (see Table 5.5). Every tuple (*T_ID*) from the tax table specifies tax information of an individual with their first name (*FName*), last name (*LName*), gender (*gender*), area code for phone number (*AC*), phone number (*phone*), state of residence (*state*), zip (*zip*), marital status (*marital*), Has Children (*HC*), salary earned (*salary*), tax rate (*rate*), Single Exemption rate (*SE*), Married Exemption rate (*ME*), and Child Exemption rate (*CE*).

The address information is populated using real semantic relationship. Furthermore, salary is synthetic, while tax rates and tax exemptions (based on salary, state, marital status and number of children) correspond to real life scenarios. The dataset comprises of 1000 entries.

Data Dependencies. We identified a large number of hard and soft denial constraints on

Table 5.5: Schema of the Tax dataset.

T_ID	FName	LName	gender	AC	phone	city	state	zip	marital	HC	salary	rate	SE	ME	CE	**tax**
1	Xiaolin	Bannelier	M	916	1000	A	CA	93383	S	N	52000	9.3	87	0	0	4827.91
2	Inderpal	Siler	F	330	1000	B	OH	45506	M	Y	29000	1.508	0	2600	1300	378.67
3	Bijan	Rehak	M	605	1000	C	SD	57741	M	Y	97500	0	0	0	0	0
4	Dhananjai	Jaakkola	F	603	1000	D	NH	3466	M	Y	47500	0	0	0	0	0
5	Serafim	Strivastav	F	267	1000	E	PA	15943	M	Y	57500	3.07	0	0	0	1765.25
6	Rengathan	Bollman	M	231	1000	F	MI	49879	S	Y	41000	3.9	3100	0	3100	1357.2

Table 5.6: Dependency List for Tax Dataset

ID	Type	Dependency
δ_1^t	FD	$\neg(t_1[\text{zip}]=t_2[\text{zip}] \wedge t_1[\text{city}]<>t_2[\text{city}])$
δ_2^t	FD	$\neg(t_1[\text{areaCode}]=t_2[\text{areaCode}] \wedge t_1[\text{state}]<>t_2[\text{state}]\ )$
δ_3^t	FD	$\neg(t_1[\text{zip}]=t_2[\text{zip}] \wedge t_1[\text{state}]<>t_2[\text{state}])$
δ_4^t	DC	$\neg(t_1[\text{state}]<>t_2[\text{state}] \wedge t_1[\text{hasChild}]=t_2[\text{hasChild}] \wedge t_1[\text{childExemp}]<>t_2[\text{childExemp}]))$
δ_5^t	DC	$\neg(t_1[\text{state}]<>t_2[\text{state}] \wedge t_1[\text{marital}]=t_2[\text{marital}] \wedge t_1[\text{singleExemp}]<>t_2[\text{singleExemp}])$
δ_6^t	DC	$\neg(t_1[\text{state}]<>t_2[\text{state}] \wedge t_1[\text{salary}]>t_2[\text{salary}] \wedge t_1[\text{rate}]<t_2[\text{rate}])$
δ_7^t	DC	$\neg(t_1[\text{areaCode}]<>t_2[\text{areaCode}] \wedge t_1[\text{zip}]=t_2[\text{zip}] \wedge t_1[\text{hasChild}]=t_2[\text{hasChild}] \wedge t_1[\text{salary}]\text{¿}t_2[\text{salary}] \wedge t_1[\text{rate}]\text{¡}t_2[\text{rate}] \wedge t_1[\text{singleExemp}]<>t_2[\text{singleExemp}])$
δ_8^t	DC	$\neg(t_1[\text{marital}]<>t_2[\text{marital}] \wedge t_1[\text{salary}]<>t_2[\text{salary}] \wedge t_1[\text{rate}]=t_2[\text{rate}] \wedge t_1[\text{singleExemp}]=t_2[\text{singleExemp}] \wedge t_1[\text{childExemp}]<>t_2[\text{childExemp}])$
δ_9^t	DC	$\neg(t_1[\text{state}]<>t_2[\text{state}] \wedge t_1[\text{marital}]<>t_2[\text{marital}] \wedge t_1[\text{rate}]=t_2[\text{rate}] \wedge t_1[\text{singleExemp}]=t_2[\text{singleExemp}] \wedge t_1[\text{childExemp}]<>t_2[\text{childExemp}])$
δ_{10}^t	DC	$\neg(t_1[\text{state}]=t_2[\text{state}] \wedge t_1[\text{salary}]=t_2[\text{salary}] \wedge t_1[\text{rate}]<>t_2[\text{rate}])$
δ_{11}^t	DC	$\neg(t_1[\text{state}]=t_2[\text{state}] \wedge t_1[\text{salary}]>t_2[\text{salary}] \wedge t_1[\text{rate}]<t_2[\text{rate}])$
δ_{12}^t	PBD	”tax” = fn(”salary”, ”rate”, ”singleExemp”, ”marriedExemp”, ”childExemp”)

the tax dataset by using a DC discover algorithm implemented by the data profiling tool Metanome [79]. We manually analyzed these DCs and selected 11 interesting DCs from them. If any of them were soft DCs, we updated/deleted the violating tuples to turn them into hard DCs. Finally, we also added a provenance based dependency based on the function to compute tax based on attributes salary, rate, singleExemp, marriedExemp, and childExemp. The final set of dependencies used in the experiments can be seen in Table 5.6.

Database System. We ran the experiments on an individual machine (CentOS 7.6, Intel(R) Xeon(R) CPU E5-4640, 2799.902 Mhz, 20480 KB cache size) in a cluster with a shared total memory of 132 GB. We performed experiments on MySQL 8.0.3 with InnoDB as it is an open source DBMS. We configured the *buffer_pool_size* to 4 GB.

Sensitive cells. Sensitive cells were only chosen from the attributes that participated in at least one dependency. This ensured that there were non-zero number of inference channels possible for each of them. Based on the number of relevant schema level dependencies, a cell participates in, we categorized them into low (< 2 dependencies), medium, and high (> 6 dependencies).

5.5.2 Evaluation

We perform the following experiments to test various aspects of our approach. Sensitive cells were chosen from low, medium, and high categories for each such selection we ran the following experiments. During algorithm execution, the cell to be hidden from a cueset was selected randomly and we ran the experiment 10 times and dropped the 2 highest and lowest runs and took the median value from the remaining. These are a preliminary set of experiments to analyze the amount of leakage through dependencies for various sensitive cells. In all the following experiments, we analyzed the number of cells hidden with respect to changing other control parameters

Experiment 1: Sensitive cells versus number of cells hidden

We increased the number of sensitive cells from 10 to 64 and measured the number of additional cells (percentage of the database) that needs to be hidden to prevent inferences. In the beginning, with increasing number of sensitive cells, as expected the number of hidden cells increases exponentially (Figures 5.3, 5.4, 5.5). This increase depends on upon the number of dependency instantiations and cuesets corresponding to the sensitive cells. In later stages, the number of hidden cells plateaus as a greater number of cells are hidden which already covers the newly added cuesets.

Experiment 2: k versus number of cells hidden

We increased the k value from 0 (Full deniability) to 0.9 of the domain size of the sensitive cell

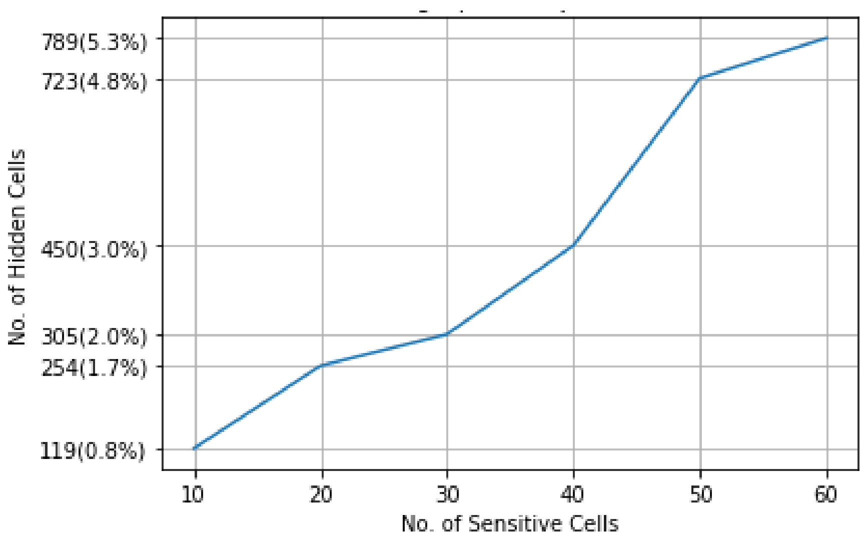


Figure 5.3: Experiment 1.1: Sensitive cells with low number of relevant dependencies

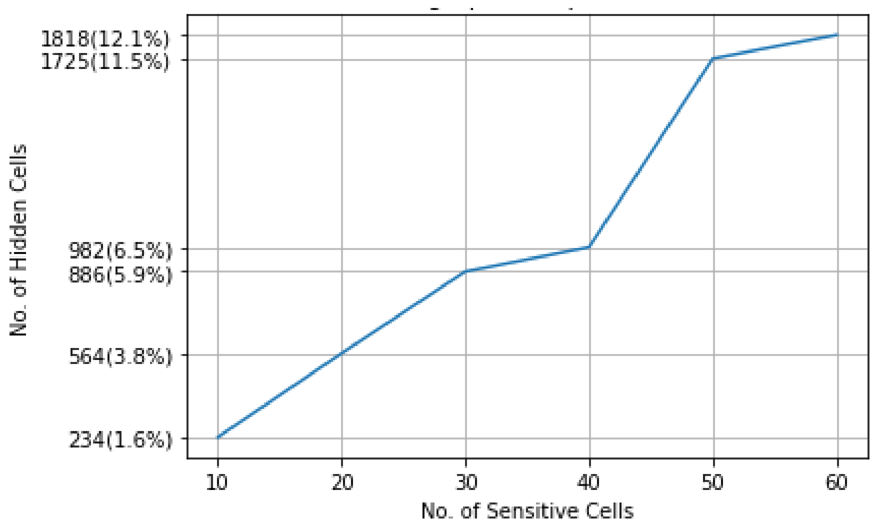


Figure 5.4: Experiment 1.2: Sensitive cells with medium number of relevant dependencies

and measured the number of additional cells that needs to be hidden to prevent inferences.

We selected a smaller number of cells from an attribute in each category (low, medium, high) and performed the experiment (Figures 5.6, 5.7, 5.8). We increased the k-percentile from 0.1 to 0.9 and the number of hidden Cells increases with increasing value of k. The impact of k on utility is minimal and more experiments are needed on a bigger dataset to study the full impact.

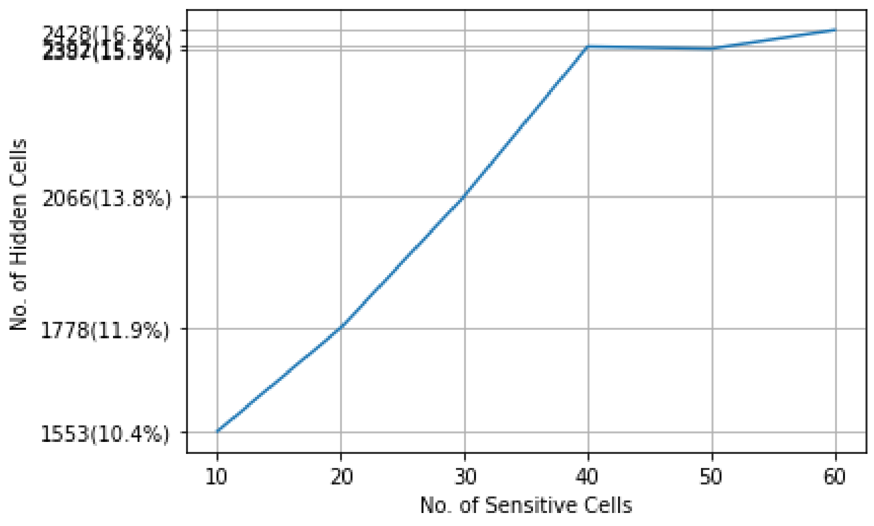


Figure 5.5: Experiment 1.3: Sensitive cells with high number of relevant dependencies

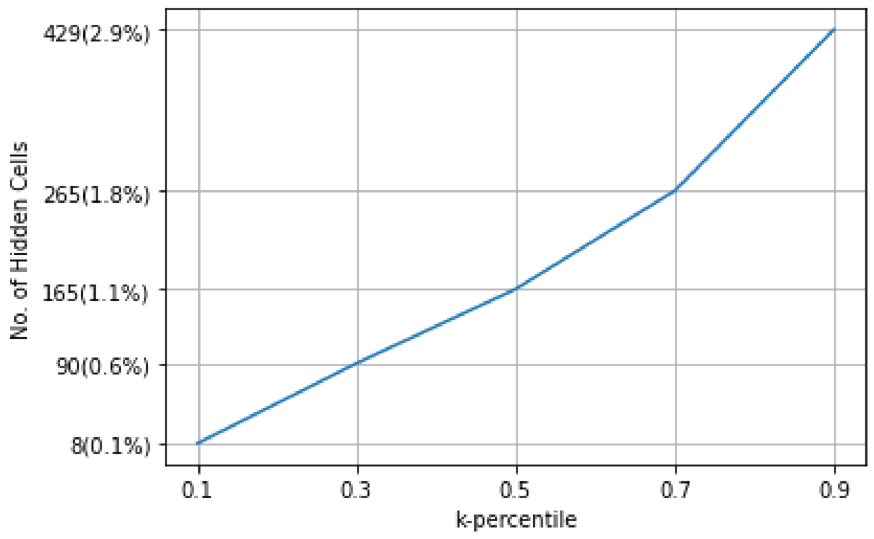


Figure 5.6: Experiment 2.1: Sensitive cells with low number of relevant dependencies

5.6 Extended model of Provenance based dependencies

In this section we describe a general model of provenance based dependencies based on different types of invertibility relationships. Table 5.7 defines different types of invertibility relationships possible between the input value and derived value depending upon the function. We also have *Fully non-invertible* $\rightarrow$ *non-invertible* and *Fully Invertible* $\rightarrow$ *Partially*

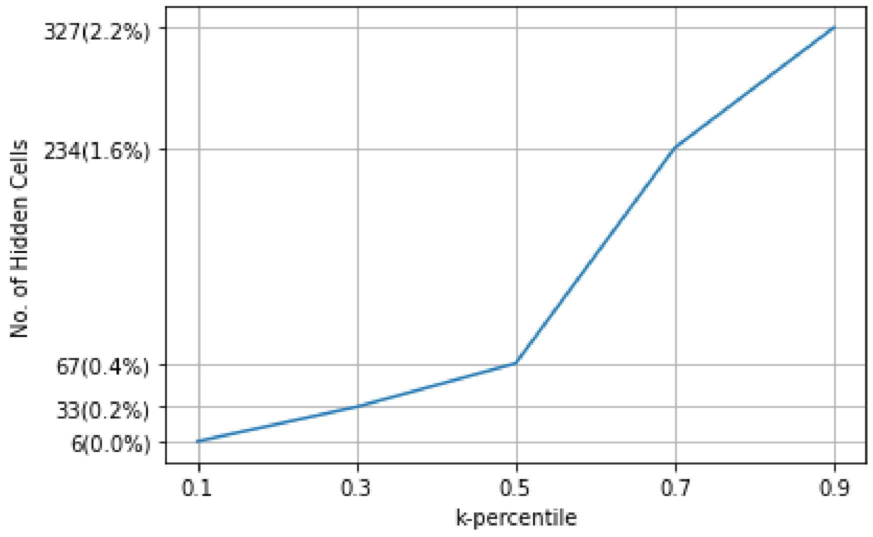


Figure 5.7: Experiment 2.2: Sensitive cells with medium number of relevant dependencies

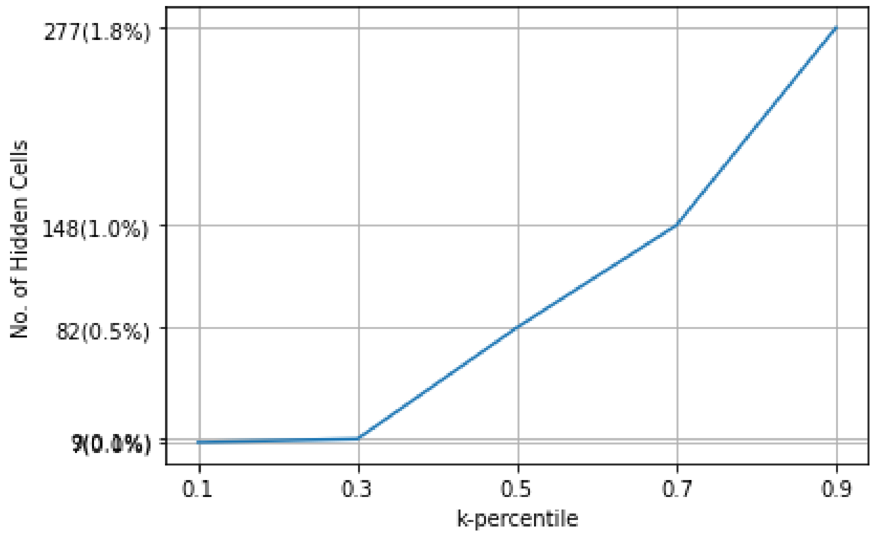


Figure 5.8: Experiment 2.3: Sensitive cells with high number of relevant dependencies

invertible.

Table 5.7: Invertibility types.

Invertibility type	Given	Discloses
Fully invertible	s_i	$r_1, r_2, \ldots, r_n$
Partially invertible	$s_i; r_1, r_2, \ldots, r_{n-1}$	r_n
Non-invertible	s_i	not any of $r_1, r_2, \ldots, r_n$
Fully non-invertible	$s_i; r_1, r_2, \ldots, r_i$	not $r_j \ldots r_n$

We will now define the property of (m, n)-Invertibility for a a function $fn(r_1, r_2, \ldots, r_n) =$

s_i where $r_1, r_2, \ldots, r_n$ are the general representation for input values (e.g., WorkHrs) to a function fn and s_i is the general representation of the derived value or the output of the function (e.g., Salary).

Definition 5.9. *((m, n)-**Invertibility.**) For a function $fn(r_1, r_2, \ldots,\ r_p) = s_i$, given its output s_i and any $m-1$ out of p inputs, if we could find another function $fn'(r_t, r_{t+1}, \ldots, r_{t+m-2}; s_i) = \{r_k, r_{k+1}, \ldots,\ r_{k+n-1}\}$ that disclose n of the rest input values, we say this function fn is $(m, n)-$invertible; otherwise, we say this function is $(m, n)-$non-invertible.*

The previously mentioned Salary function is (2, 1)-invertible as given any two of the three variables, the rest one could be disclosed.

Definition 5.10. *(**Fully invertible.**) If a function f is (1, n)-invertible, we say this function is fully invertible.*

Cross product (Cartesian product) is an example of full invertibility, since all the input values can be inferred if given the result of cross product. That is to say, cross product is $(1, n)$-invertible. Other examples of commonly used functions are user-defined functions (UDFs) (e.g. oblivious functions, secret sharing), and aggregation functions.

Theorem 5.2. *Any (m, n)-invertible function is $(m - 1, n)$-non-invertible.*

Instantiation of PBDs: Function definitions are published as part of the schema (just like FDs, DCs) and known to the adversary. We assume that the DBMS maintains provenance of the derived attribute in a database (Table 5.4). In our system, we also maintain various metadata associated with the functions such as its definition, its invertibility type, etc. Using this provenance database, function metadata, our system derives the following instantiated provenance based dependencies for Example 1. As the function used to derive Salary is $(2, 1)$-invertible, we have the following expressions for some of the derived values and their corresponding input values.

1. $w_1[Salary] \wedge e_1[WorkHrs] \rightarrow e_1[SalPerHr]$
2. $w_1[Salary] \wedge e_1[SalPerHer] \rightarrow e_1[WorkHrs]$
3. $w_2[Salary] \wedge e_2[WorkHrs] \rightarrow e_2[SalPerHr]$
4. $w_2[Salary] \wedge e_2[SalPerHer] \rightarrow e_2[WorkHrs]$
5. ...

5.6.1 Computing leakage for PBDs

As for a (m, n)-invertible function, denoted by $fn'(r_1, r_2, \ldots r_n) = \{s_1, s_2, \ldots, s_m\}$, it can be apparently observed that, given the m inputs $\{g_1, g_2, \ldots g_m\}$, the function will lead to the leakage towards n values. Take, for example, the *Salary*, *WorkHrs* and the *SalPerHr*. Since as analyzed, the function to calculate the salary is $(2, 1)$-invertible, it means that if taking any two values of the three attributes, the adversary can fully convert and leak the exact value of the remaining attribute. We call this case *full leakage* from provenance based dependencies.

Note. The information leakage caused by provenance-based dependencies (PBD) can be composed with other leakages (from other data dependencies). For example, given *Salary* and *WorkHrs*, the adversary could convert the value of *SalPerHr* based on the invertibility. Then, due to the data dependency that the *Role* determines *SalPerHr*, the adversary can possibly get the information about what role the queried employee takes in consequence.

However, a subset of the m input values of an (m, n)-invertible function could also leak some information of some disclosable values based on some domain knowledge. As an example, suppose the adversary knows that the *salary* of a employee is 8,000 but they do not know the exact *WorkHrs* and *SalPerHr*. Even though, with some background knowledge, for e.g., the information that no one could work more than 40 hours per week by law, the adversary could reduce the domain of possibilities the *SalPerHr* value could take. We call this *partial* *leakage* from provenance based dependencies.

Leakage Oracle Model.

To characterize the partial leakage from PBD, we consider the leakage oracle model that can be conceived as an interactive party to determine if some values cannot be taken from the attribute domain according to the background knowledge on the inputs to the oracle machine. Stated more formally, the leakage oracle is a function ensemble, denoted by $\mathcal{O}^{Leak} = \{\mathcal{O}_i^S : S \to l(o_i) \mid S \subset \{g_1, g_2, \ldots g_m\}, i \in [n]\}$. On taking in the invertible function and some inputs, the oracle outputs the quantified partial leakage on each disclosable values. We use the notation $l(o_i)$ to denote the reduction to the domain of the attribute o_i due to the partial leakage from S. In particular, $l(o_i)$ is presented in fractions $l(o_i) = \frac{1}{Dom(o_i) - Red(o_i)}$ where $Red(o_i)$ is the number of values that the attribute o_i cannot take from its domain due to the leakage; in the case of fully leakage, the leakage oracle will output $l(o_i) = 1$. Take the previous running example. Supposing the domain of *SalPerHr* is [0, 600], discrete, the oracle query $\mathcal{O} \models \mathcal{O}^{Leak} : \mathcal{O}(Salary = 8000)$ will output $l(SalPerHr) = \frac{1}{600-400} = \frac{1}{200}$, since the value of the *WorkHrs* attribute is known to be less than 40 with some background knowledge.

5.7 Discussion

This chapter presented a new form of inference attacks on access control protected data through denial constraints and provenance based dependencies. The chapter introduced a new security model based on ensuring deniability for the sensitive cell. A set of algorithms were developed to ensure that sensitive cells met the necessary deniability requirement. These algorithms use as input a database instance, a set of access control policies, and a set of data dependencies.

This work used a state function to perform analysis as this was enough to achieve deniability guarantees. A possible extension is to perform probabilistic analysis of the sensitive cells w.r.t non-sensitive cells and instantiated dependencies to quantify the security loss due to disclosing non-sensitive cells. The provenance based dependency model presented in this work could be extended to model partial invertibility as well and computing leakages based on a oracle model. Instead of performing leakage analysis at compile time, an alternative approach will be to do this at query time. While this might add the overhead to real time query processing, it could result in avoiding large number of unnecessary computations and potentially improving utility for query workloads. Determining the appropriate k to set for different sensitive cells is an open challenge.

An interesting direction to explore would be whether data cleaning algorithms (such as Holoclean [87] which rely on data dependencies to detect and clean databases) are able to recover the hidden cells in the output of the algorithm. Finally, adapting our approach to handle a dynamic setting where policies and/or dependencies are updated would be an interesting extension. In such settings, a hybrid model that smartly partitions the work between compile time and query time would make most sense. If logs are maintained by the Policy Enforcement system for the purpose for auditing, these can be utilized to detect when changing policies and dependencies resulted in leakages with respect to prior queries.

Chapter 6

Incorporating Policies to IoT Systems Deployed in the Real World

"This, then, is the true reward for excellence: privacy. And choice."

N K Jemisin, *Fifth Season*

This chapter presents the details of implementing *policy-based privacy-by-design approaches* in two real world IoT settings. First, the chapter discusses a realization of the approach towards privacy-aware smart buildings presented in Chapter 3. A policy engine and interfaces to capture user-defined policies and enforce them (using a similar query rewriting approach to the one described in Chapter 4) have been developed and deployed. This policy engine has been integrated into the TIPPERS IoT testbed deployed in several US campuses (with the main deployment at the UC Irvine campus). Second, the chapter discusses the design and integration of a policy engine into PE-IoT, privacy-preserving middle-ware in which sensor data streams (e.g., WiFi connectivity data) are evaluated against policies before applying PETs on them.

6.1 Incorporating Policies in TIPPERS

TIPPERS [75, 6] is a novel IoT testbed for smart spaces that incorporates a variety of smart space applications. A key design feature of the TIPPERS architecture is that it is space, sensor, and task agnostic, allowing it to be used as plug-and-play technology to create smart spaces. In addition, TIPPERS embodies a privacy-by-design architecture, which enables the integration of different Privacy Enhancing Technologies (PETs). A variety of PETs have been already integrated into TIPPERS including secure computing and differential privacy.

As depicted in Figure 6.1, the TIPPERS architecture includes several decisions to support the goal of privacy by design. Firstly, TIPPERS provides an abstraction of the underlying sensor infrastructure by translating between the IoT devices' world (i.e., sensors, actuators, raw observations, etc.) and the people's world (i.e., interactions of people, spaces, phenomena, etc.). The system is based on a domain model that represents both worlds and enables users/developers to interact with high-level semantically meaningful concepts. It also includes ontology-based translation algorithms to convert user requests at the high level (e.g., "decrease the temperature of rooms where the occupancy is greater than 75% of their capacity") into actions on the specific underlying device infrastructure [100, 10]. The main advantage is that it simplifies the development of smart applications and facilitates their portability in between spaces as they are built on high-level concepts instead of on IoT devices. Secondly, it simplifies the definition of privacy policies as users can focus on what they want to protect (e.g., "do not capture my location when I am with John in a private space during working hours"). TIPPERS uses such privacy policies to guide its data collection, storage, and sharing practices.

As a mechanism to implement the translation of raw data into higher-level semantically meaningful interpretations, TIPPERS supports virtual sensors wherein streams of sensor data can be used to create streams of such inferences. For instance, a virtual sensor can

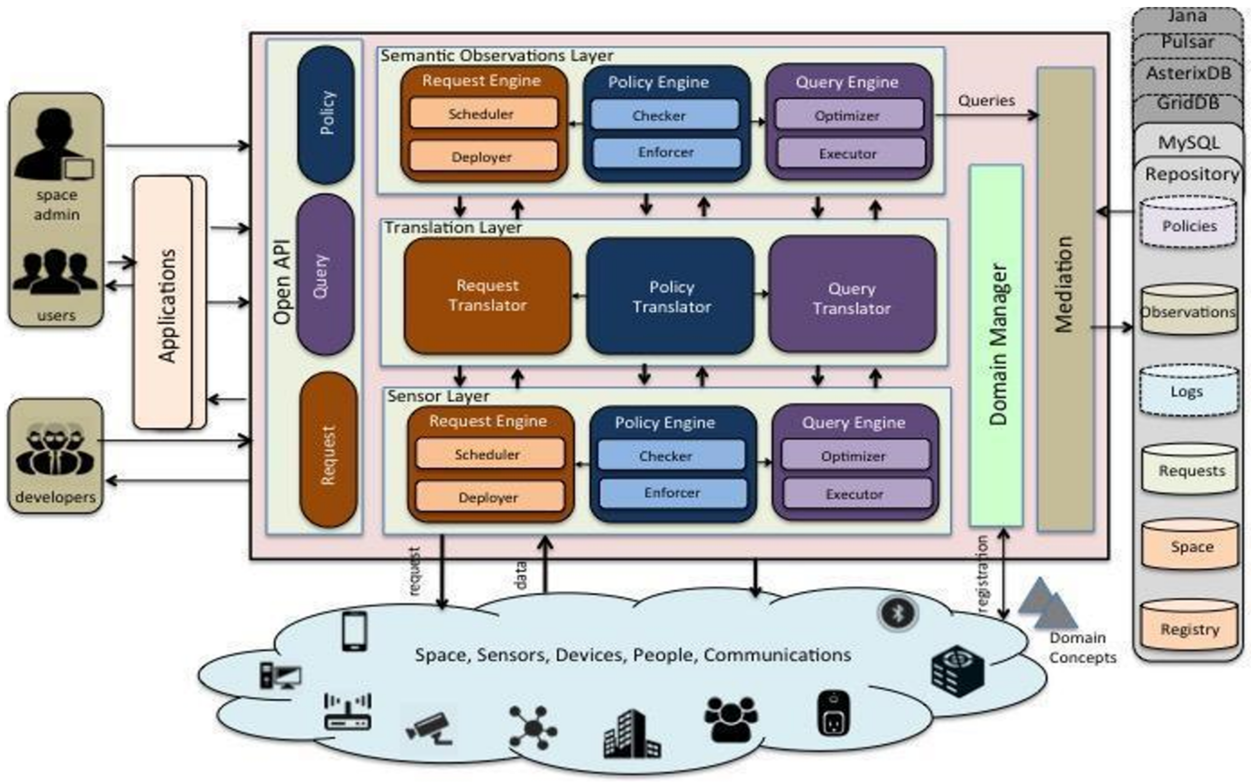


Figure 6.1: High-level architecture of the TIPPERS system.

translate connectivity data (e.g., logs from WiFi APs containing information about which devices are connected to them) into occupancy of different spaces along time. This enables TIPPERS to incorporate further PETs. For example, a stream of sensor data can be scrubbed of personally identifiable information (PII) when passed to operators.

6.1.1 TIPPERS Policy Engine

An important component of TIPPERS is its *Policy Engine* (see Figure 6.1 where the policy engine plays a role in the different layers of the system). In TIPPERS, policies are used to guide the collection, storage, processing, and sharing of data. These policies are either defined by the administrator of the space, and therefore apply to any device in it (e.g., due to security reasons), or defined by users to express how their data should be managed (e.g., to restrict access to pieces of information about them). In the current TIPPERS implementation, the primary focus is on user policies. This is driven by the stringent privacy requirements of the spaces in which it is deployed (university campuses). A sample policy in TIPPERS is as follows: allow a specific user (e.g., John) to access location data of an individual while using a specific application and as long as the data has been captured in a public space and during working hours (similar to policies presented in Chapter 4).

The system denies access to an individual's data by default when an application tries to access it. Hence, user-defined policies are meant to allow access to parts of user data under certain circumstances. Internally, these policies are managed by the TIPPERS policy engine that enforces them at query time using the approach in Chapter 4. In particular, TIPPERS offers the possibility to define such policies in high-level terms (e.g., restricting access to location data instead of to specific sensor data) and translates it into access control to low-level sensor data. This way, if a user tries to access, for instance, connectivity records of a specific device, this will be denied if the user defined a policy to restrict access to their location as this low-level data can be used to infer such information.

Access control APIs in TIPPERS

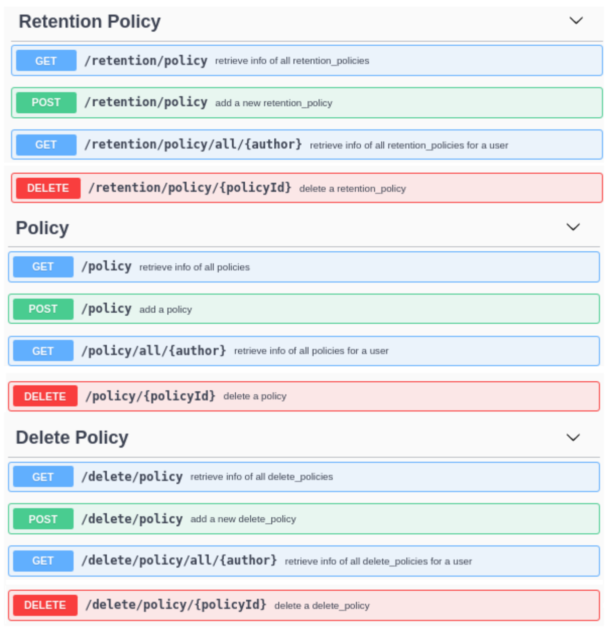
The TIPPERS API includes a set of endpoints (see Figure 6.2 for a screenshot of the Swagger[1] specification of the endpoints)[2]. These enable the insertion/deletion/update of three types of policies:

- Data Sharing, which controls with whom individual's data can be shared an under which circumstances.
- Data Retention, which controls for how long data of an individual can be stored.
- Data Deletion, which trigger a deletion of specific individual's data items.

[1]https://swagger.io

[2]We would like to acknowledge Vikram Miryala's help in implementing these endpoints.

The enforcement of the policies defined through the API by the policy engine is done based on the approach presented in Chapter 4 for sharing policies. For a deletion policy a query is run to delete all data that fulfills the conditions in the policy. For retention policies, at insertion time the policy gets associated with a specific deletion date. Then, the engine schedules a daily check of policies and for those which came in effect that day, it triggers a deletion action like in the previous type of policy.



Retention Policy

GET /retention/policy retrieve info of all retention_policies
POST /retention/policy add a new retention_policy
GET /retention/policy/all/{author} retrieve info of all retention_policies for a user
DELETE /retention/policy/{policyId} delete a retention_policy

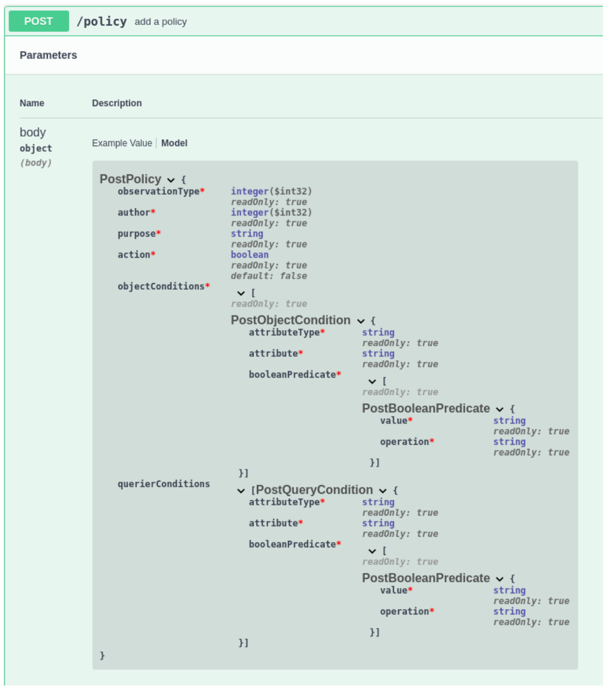
Policy

GET /policy retrieve info of all policies
POST /policy add a policy
GET /policy/all/{author} retrieve info of all policies for a user
DELETE /policy/{policyId} delete a policy

Delete Policy

GET /delete/policy retrieve info of all delete_policies
POST /delete/policy add a new delete_policy
GET /delete/policy/all/{author} retrieve info of all delete_policies for a user
DELETE /delete/policy/{policyId} delete a delete_policy

(a)



POST /policy add a policy

Parameters

Name | Description
body object (body) | Example Value | Model

```
PostPolicy ⌄ {
   observationType*     integer($int32)
                        readOnly: true
   author*              integer($int32)
                        readOnly: true
   purpose*             string
                        readOnly: true
   action*              boolean
                        readOnly: true
                        default: false
   objectConditions*
                        ⌄ [
                        readOnly: true
                        PostObjectCondition ⌄ {
                           attributeType*     string
                                              readOnly: true
                           attribute*         string
                                              readOnly: true
                           booleanPredicate*
                                              ⌄ [
                                              readOnly: true
                                              PostBooleanPredicate ⌄ {
                                                 value*          string
                                                                 readOnly: true
                                                 operation*      string
                                                                 readOnly: true
                                              }]
                        }]
   querierConditions
                        ⌄ [PostQueryCondition ⌄ {
                           attributeType*     string
                                              readOnly: true
                           attribute*         string
                                              readOnly: true
                           booleanPredicate*
                                              ⌄ [
                                              readOnly: true
                                              PostBooleanPredicate ⌄ {
                                                 value*          string
                                                                 readOnly: true
                                                 operation*      string
                                                                 readOnly: true
                                              }]
                        }]
}
```

(b)

Figure 6.2: Swagger specification of the policy API.

As an example of the usage of the APIs, Figure 6.3 shows a policy expressed in JSON that can be passed as a parameter of the insert sharing policy endpoint. This policy allows TIPPERS to share the location data of an individual with another one (i.e., John Doe) when the latter uses a specific application (i.e., Occupancy app) and when the location data fulfills certain restrictions (i.e., it shows the user located in room 2065 and was captured between 9am-10am on a given date).

```json
{
    "policyId": 1,
    "author": 1,
    "observationType": 1,
    "observationName": "location",
    "purpose": "Occupancy Tool",
    "action": true,
    "objectConditions": [
        {
            "attribute": "Duration",
            "attributeType": "timestamp",
            "booleanPredicate": [
                {
                    "value": "2021-01-13 09:00:00",
                    "operation": "<="
                },
                {
                    "value": "2021-01-13 10:00:00",
                    "operation": ">="
                }
            ]
        },
        {
            "attribute": "Location",
            "attributeType": "int",
            "booleanPredicate": [
                {
                    "value": "2065",
                    "operation": "="
                }
            ]
        }
    ],
    "queryConditions": [
        {
            "attributeType": "int",
            "attribute": "querier",
            "booleanPredicate": [
                {
                    "value": "1",
                    "operation": "="
                }
            ]
        }
    ]
}
```

Figure 6.3: Example policy inserted into TIPPERS.

6.1.2 TIPPERS Policy Definition

Since defining policies as Json objects is prone to errors, TIPPERS includes two mechanisms to simplify the process. The first one is the IoTA personal assistant developed by researchers at CMU[36]. The second is the TIPPERS Portal policy definition GUI.

Defining Policies Through the IoTA

The IoT Privacy Infrastructure developed by CMU [4] interacts with TIPPERS to realize the vision of a privacy-aware smart building described in Chapter 3. In particular, the CMU infrastructure consists of:

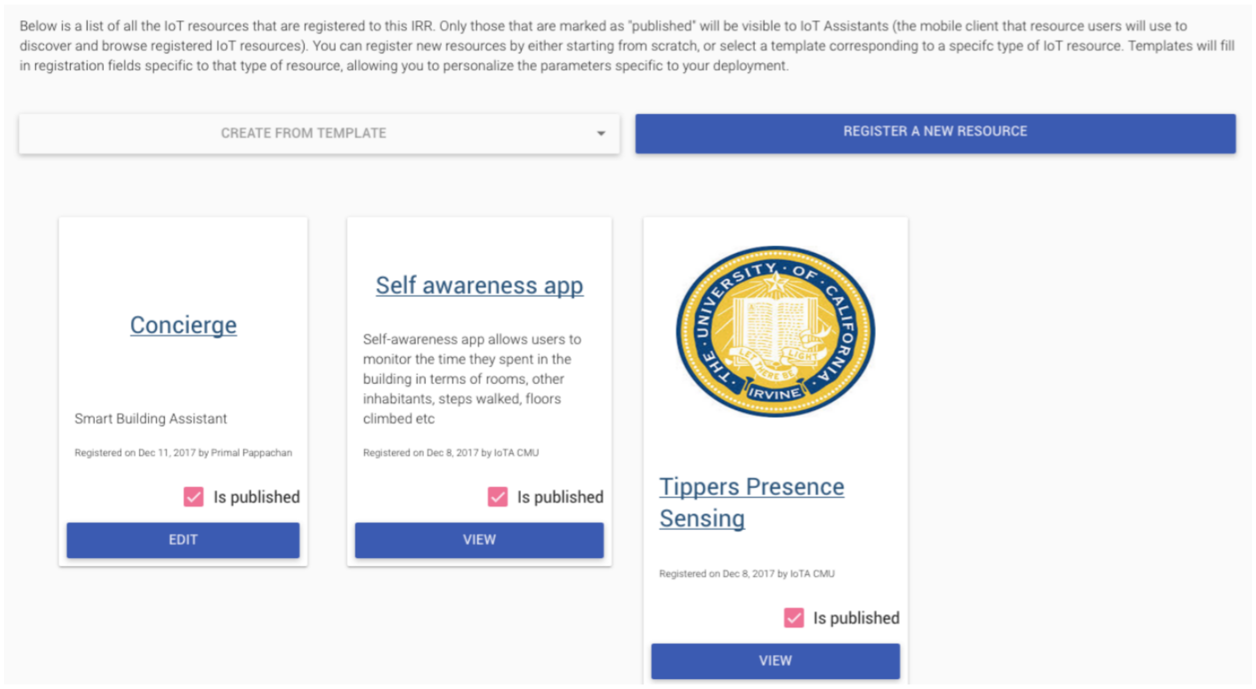
- A collection of IoT Resource Registries ("IRRs") that enables organizations or people to publicize the presence of IoT systems and the data they collect, including any privacy options made available by these IoT systems.
- An IoT Assistant ("IoTA") app that people can download on their smartphones (both iOS and Android[3] smartphones) to discover the presence of IoT systems around them along with the data they collect and any privacy options they have available.

As part of the integration of such infrastructure with TIPPERS, the first step was to define different resources in the IRR for UCI (see Figure 6.4). The IRR deployed at the DBH building contains information about services offered by the TIPPERS system (e.g., applications such as Concierge, Self-Awareness, Building Analytics, Noodle) and the sensor subsystems managed by TIPPERS (e.g., Localization through WiFi connectivity, video cameras).

The information defined in the IRR for sensor subsystems include an option to opt-out from sharing of data captured by the subsystem by TIPPERS. This option internally connects to one of the APIs described before through which the user can specify whether their data can be shared with others or not.

A user of IoTA can discover these resources on their device when they are in proximity to DBH. Figure 6.5 shows the information the user will receive. On the left, the figure shows the different resources defined in the IRR. On the right, the figure shows the details when the user selects the location sensing subsystem of TIPPERS which uses data captured from WiFi APs and bluetooth beacons. In addition to the description of the resource, the

[3]https://play.google.com/store/apps/details?id=io.iotprivacy.iotassistant&hl=en_US&gl=US


Below is a list of all the IoT resources that are registered to this IRR. Only those that are marked as "published" will be visible to IoT Assistants (the mobile client that resource users will use to discover and browse registered IoT resources). You can register new resources by either starting from scratch, or select a template corresponding to a specifc type of IoT resource. Templates will fill in registration fields specific to that type of resource, allowing you to personalize the parameters specific to your deployment.

Concierge

Smart Building Assistant

Registered on Dec 11, 2017 by Primal Pappachan

Self awareness app

Self-awareness app allows users to monitor the time they spent in the building in terms of rooms, other inhabitants, steps walked, floors climbed etc

Registered on Dec 8, 2017 by IoTA CMU

Tippers Presence Sensing

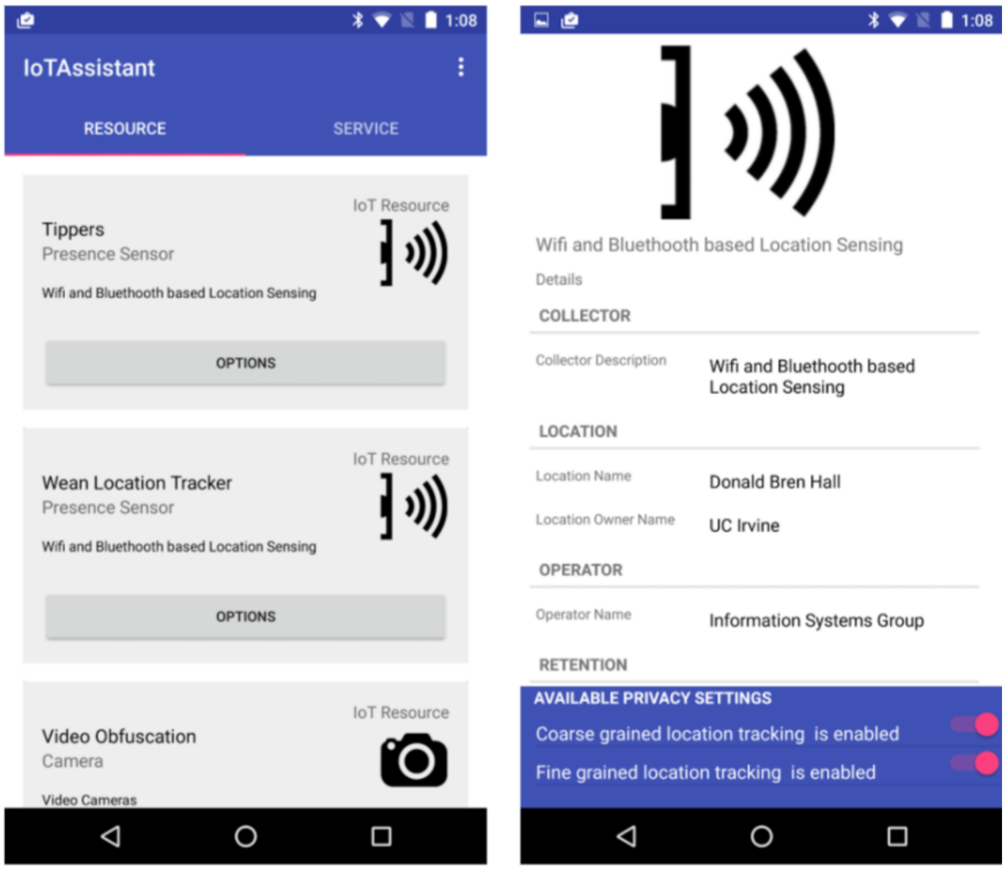
Registered on Dec 8, 2017 by IoTA CMU

Figure 6.4: Resources defined in the IRR.

user has two options on the bottom to specify their privacy preferences with respect to the sharing of their location data. If the user opts-in fine-grained location tracking, the IoTA generates an API call to submit a policy that allows TIPPERS to share the user location data (generated using a virtual sensor that combines WiFi AP and bluetooth beacon data) with others through the different applications available. If the user opts-in coarse-grained location tracking, a similar policy is generated to allow TIPPERS to share the user location data generated using a virtual sensor that only uses WiFi AP data.

Defining Policies Through the TIPPERS Portal

An alternative mechanism to define user policies to guide data management by TIPPERS is the TIPPERS Portal. This is the GUI interface to the TIPPERS system for users which enables them to configure their information (e.g., their name, office, owned devices, etc.), access services (e.g., Concierge and Self-Awareness), and define policies.



IoT Resource
Tippers
Presence Sensor
Wifi and Bluethooth based Location Sensing

IoT Resource
Wean Location Tracker
Presence Sensor
Wifi and Bluethooth based Location Sensing

IoT Resource
Video Obfuscation
Camera
Video Cameras

Wifi and Bluethooth based Location Sensing
Details

COLLECTOR

Collector Description	Wifi and Bluethooth based Location Sensing

LOCATION

Location Name	Donald Bren Hall
Location Owner Name	UC Irvine

OPERATOR

Operator Name	Information Systems Group

RETENTION

AVAILABLE PRIVACY SETTINGS
Coarse grained location tracking is enabled
Fine grained location tracking is enabled

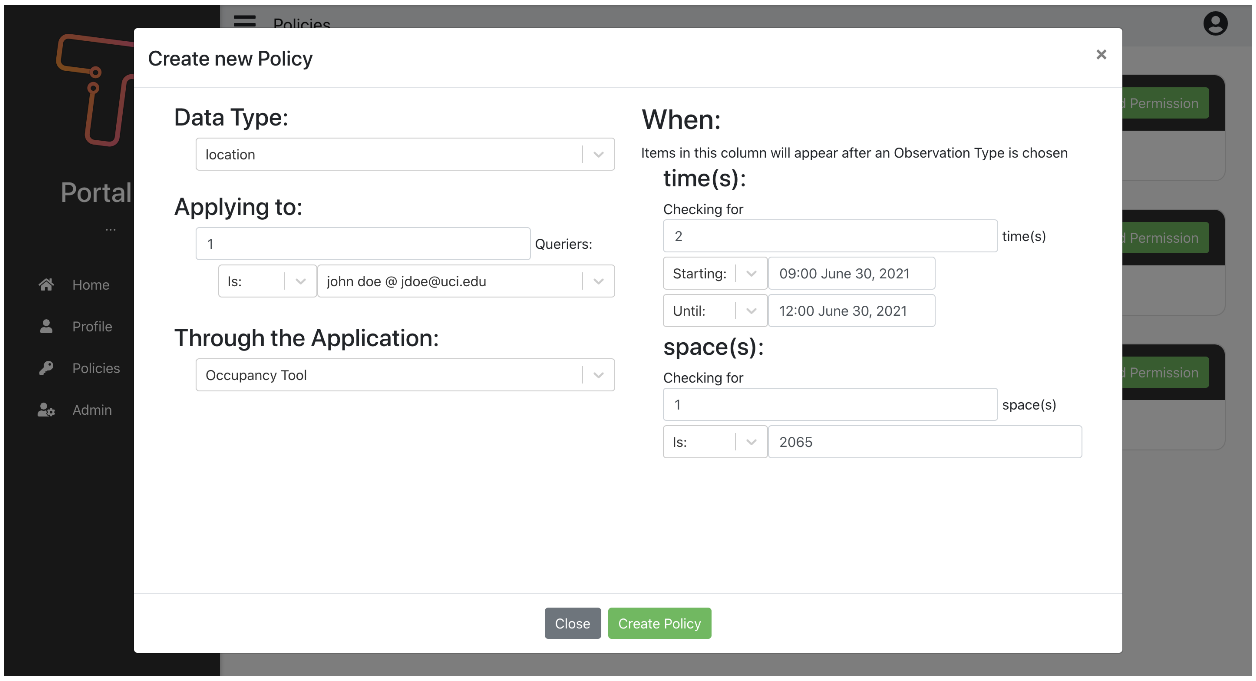
Figure 6.5: Definition of policies through the IoTA.

The policy definition GUI[4] enables users to define the three types of policies supported by the TIPPERS API (i.e., data sharing, retention, and deletion). After being defined, the policies are communicated to TIPPERS through the APIs for their enforcement by the policy engine.

As an example, Figure 6.6 and Figure 6.7 show the Portal interface to define two data sharing policies. Figure 6.6 shows the definition of a policy to enable TIPPERS to share location data of the user defining the policy with John Doe using the Occupancy Tool when the location data shows that the user was in location 2065 between two specific time periods. Figure 6.6 shows the definition of a policy to enable TIPPERS to share vital signal data (e.g., heart rate and temperature) with John Doe using the Self-Awareness App when the user had normal signs (in this example a heart rate between 60 and 100bmp and a temperature

[4]We would like to acknowledge Aliyah Byon's help in implementing this GUI.

between 97 and 99F) during a time interval.



Create new Policy

Data Type: location

Applying to: 1 Queriers:
Is: john doe @ jdoe@uci.edu

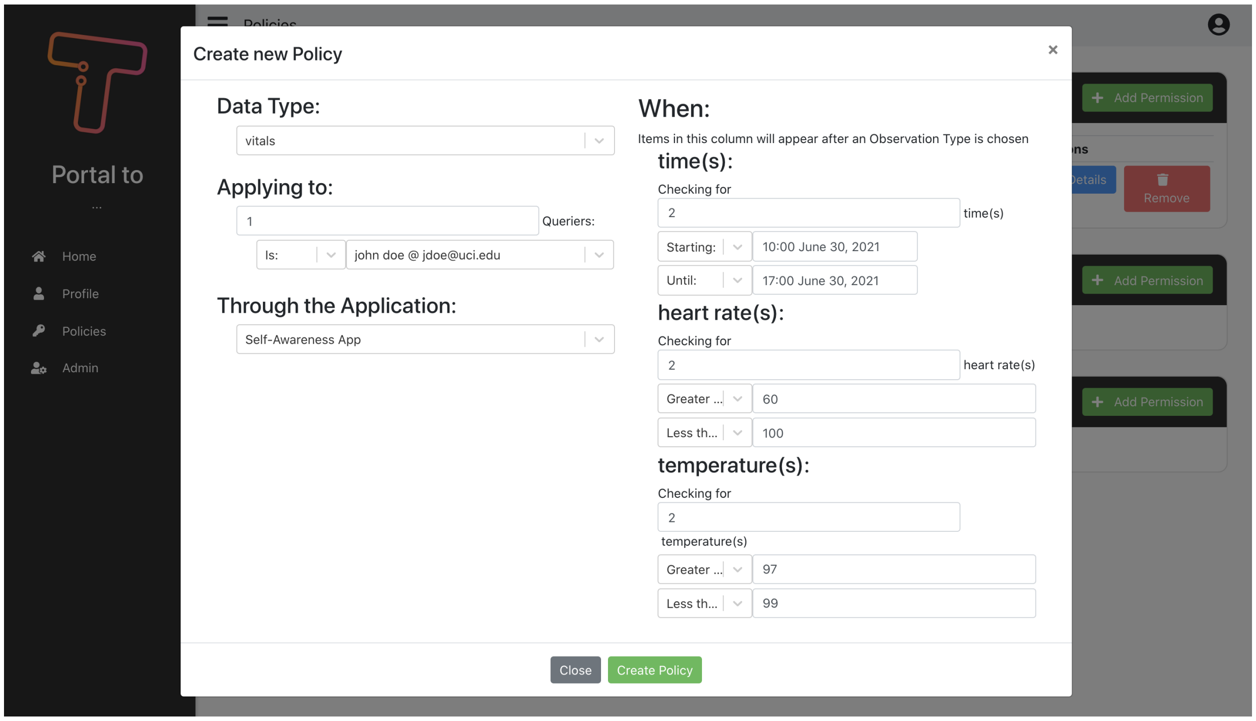
Through the Application: Occupancy Tool

When:
Items in this column will appear after an Observation Type is chosen

time(s):
Checking for 2 time(s)
Starting: 09:00 June 30, 2021
Until: 12:00 June 30, 2021

space(s):
Checking for 1 space(s)
Is: 2065

Figure 6.6: Definition of a policy to handle sharing of location data in TIPPERS.



Create new Policy

Data Type: vitals

Applying to: 1 Queriers:
Is: john doe @ jdoe@uci.edu

Through the Application: Self-Awareness App

When:
Items in this column will appear after an Observation Type is chosen

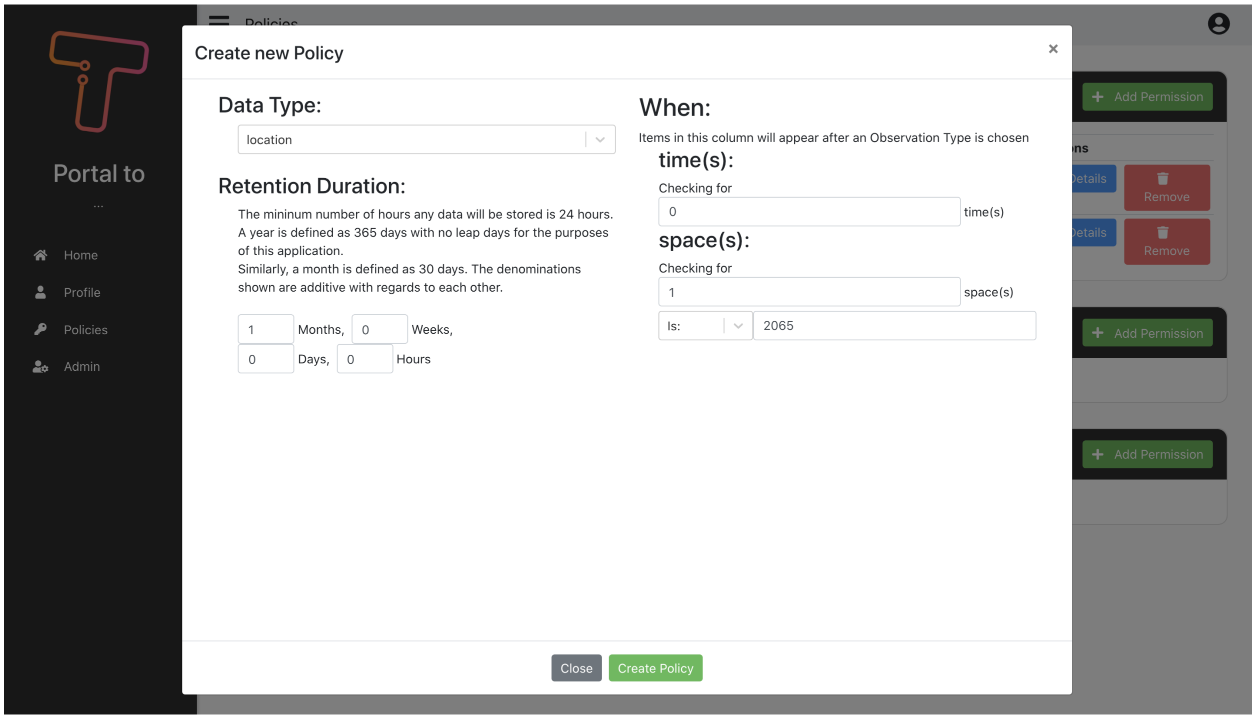
time(s):
Checking for 2 time(s)
Starting: 10:00 June 30, 2021
Until: 17:00 June 30, 2021

heart rate(s):
Checking for 2 heart rate(s)
Greater ... 60
Less th... 100

temperature(s):
Checking for 2 temperature(s)
Greater ... 97
Less th... 99

Figure 6.7: Definition of a policy to handle sharing of vital signs data in TIPPERS.

Figure 6.8 shows the definition of a data retention policy that specifies that location data of this particular user when they are located in a specific space (room 2065) must be deleted after 30 days.


Create new Policy

Data Type: location

Retention Duration:

The mininum number of hours any data will be stored is 24 hours. A year is defined as 365 days with no leap days for the purposes of this application. Similarly, a month is defined as 30 days. The denominations shown are additive with regards to each other.

1 Months, 0 Weeks, 0 Days, 0 Hours

When:

Items in this column will appear after an Observation Type is chosen

time(s):

Checking for 0 time(s)

space(s):

Checking for 1 space(s)

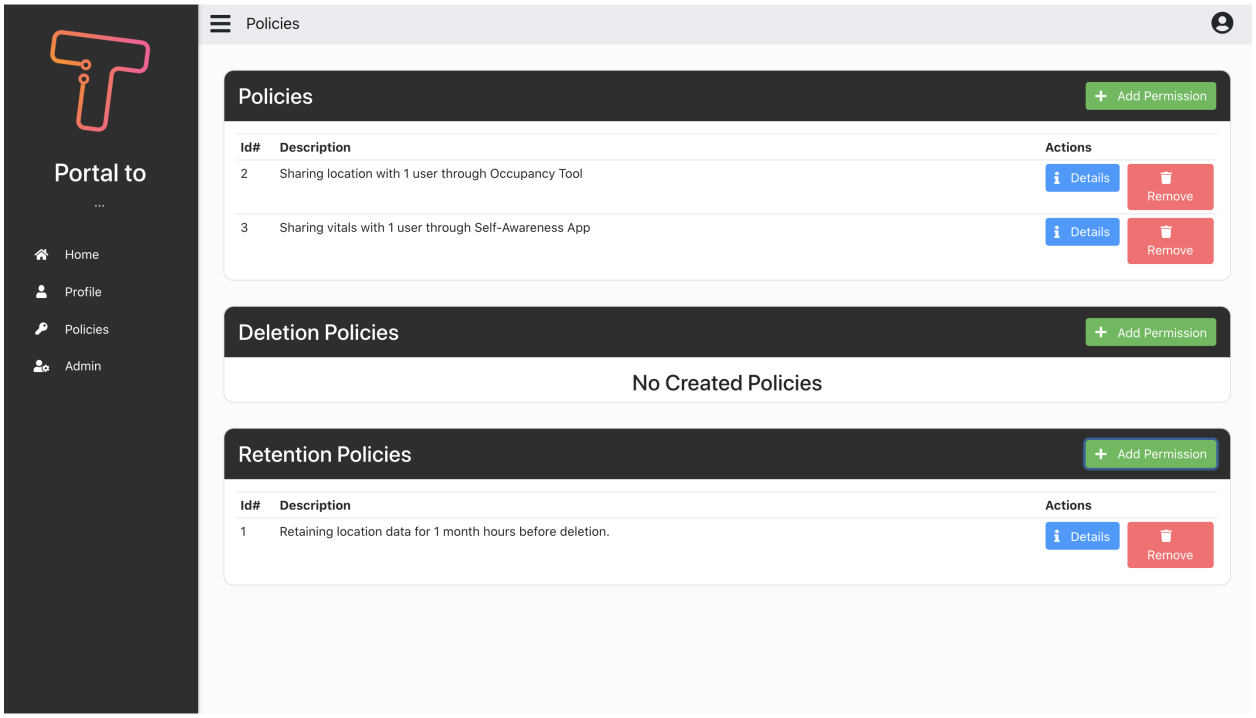
Is: 2065

Figure 6.8: Definition of a policy to handle retention of location data in TIPPERS.

Finally, Figure 6.9 shows the previously defined policies for the user and the different options to obtain more details about them (and edit those) or remove the policies.

6.2 Incorporating Policies to PE-IoT

PE-IoT (Privacy Enhanced-Internet of Things) is a privacy-compliant sensor data sharing system [53]. PE-IoT is designed as a middleware that intercepts information flow in existing IoT data processing systems to add privacy-compliance (see Figure 6.10 during data ingest and sharing). Specifically, PE-IoT gets raw sensor data streams from an *ingestion and sharing system* and produces corresponding privacy-enhanced data streams by enforcing data subject policies and applying suitable Privacy Enhancing Technologies (PETs).



Policies

Id#	Description	Actions
2	Sharing location with 1 user through Occupancy Tool	Details Remove
3	Sharing vitals with 1 user through Self-Awareness App	Details Remove

Deletion Policies

No Created Policies

Retention Policies

Id#	Description	Actions
1	Retaining location data for 1 month hours before deletion.	Details Remove

Figure 6.9: List of policies defined in TIPPERS.

PE-IoT introduces the concept of a *Data product*, an abstraction shared by various actors involved in the PE-IoT dataflow. Operationally, PE-IoT is deployed by the data controller (see Figure 6.10 that takes in incoming raw sensor data-streams and transforms them into (privacy-enhanced) data products by enforcing organizational and individual policies that implement privacy regulations. The data products, thus generated, are shared with service providers based on their needs through *Access Control*. PE-IoT is also capable of storing the sensor stream data from the data controller using a *Storage Manager*. This stored data can be accessed by service providers through PE-IoT which then generates Data Products based on the past data. PE-IoT also provides timely deletion, logging based auditing, and encryption at rest to meet other GDPR design requirements outlined earlier.

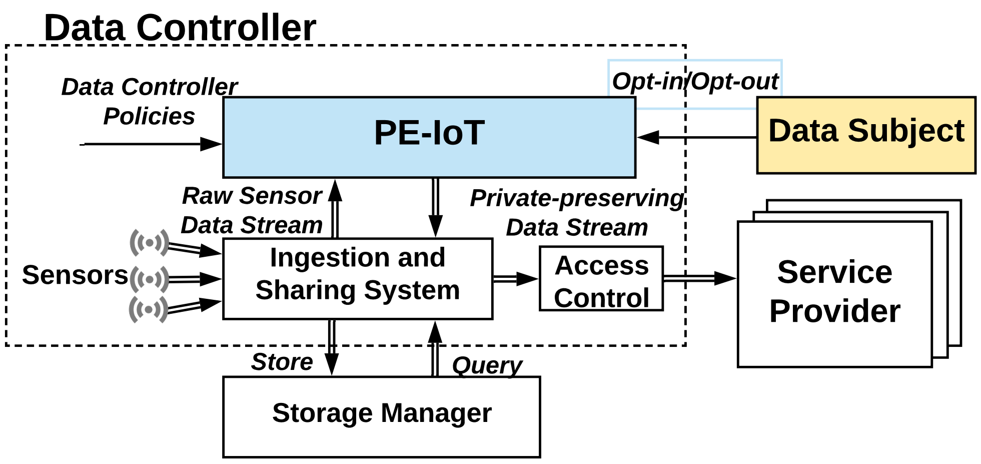


Figure 6.10: High-level architecture of the PE-IoT middleware.

6.2.1 Policy model in PE-IoT

PE-IoT receives sensor data streams from multiple sensors where each tuple of a sensor stream is associated with a data subject if it captures personal information. For example, for Wi-Fi AP association data, the owner of the device that connects to the infrastructure is the data subject. The PE-IoT data model is based on the design principles derived through privacy laws such as GDPR (presented in Chapter 2). This model is also developed to minimize the overhead for data subjects and controllers in order to fulfill their rights and responsibilities.

The core of PE-IoT data model is *Data Product* which is an abstraction for privacy enhanced data that data controllers can share with service providers. Data products, created using incoming sensor streams, are the unit used by data controllers to share data with service providers. More formally, a data product is defined as: θ=$(Filter, Policy, PET)$ where *Filter* is a set of selection conditions on sensor data that produce a subset of the sensor data that constitutes a data product. For example, for a WiFi association data set, a possible filter might be based on a particular building. *Policy* in a data product consists of two components: 1) A data controller's sharing policies, which specify conditions under which a service provider can access the data product, and 2) The data subject's choices, which determine the inclusion of a data subject's data in a data product. *PET* defines the information about the Privacy Enhancing Technology (PET) that a data controller uses when sharing the data product with service providers. We now explain the policy model in PE-IoT and refer the reader to [53] for a full description of the other two components of a data product.

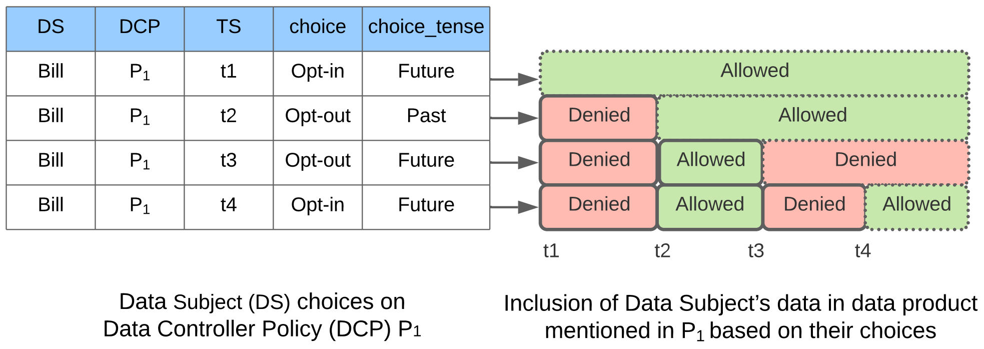
6.2.2 Data Controller Policy and Data Subject Choices

In PE-IoT, policy captures the purposes for which the data product is shared by the data controller and provides a mechanism for data subjects to opt-in/opt-out of the data product. This satisfies regulation requirements in which user data is only collected and processed for specific purposes and sharing of data is explicitly allowed by the users. The data controller policy for sharing a data product (θ_j) is specified as $DCP_i = < \theta_j, \{pa_1 ... pa_n\} >$. Each policy attribute pa_i consists of a set of tags and represents the metadata associated with a data product. Examples of policy attributes are purpose and the category of service providers who can gain access to the data product. A possible set of policy attributes for a data controller policy on a data product (θ_1) may be pa_1, pa_2, where pa_1 = [*"COVID-19-tracking", "UCI-health"*] (indicating that the health officials at UCI can access the data for tracking COVID-19), and pa_2 = [*"Occupancy", "UCI-facilities"*] (indicating that θ_1 is accessible to facility entities from UCI campus for determining building occupancy)[5]. Each service provider is associated with the set of attributes that characterize the service provider[6]. A service provider advertises itself using its set of attributes when requesting access to a data product. This set of attributes is matched against the policy attributes associated with data product (through data controller policy) to determine if that data product can be shared with the service provider.

[5]The implementation of this policy model is based on a simplified subset of the model presented in Chapter 4.

[6]Validation of service provider attributes is done by a trusted third party.

A data subject can choose to opt-in or opt-out of the participation in a specific data product. A data subject choice is modelled as $< DS, DCP, choice, TS, choice\_tense >$ where DCP is the data controller policy data subject DS is opting in or opting out ($choice$) at timestamp TS. The $choice\_tense$ is used to denote whether the action (of opting-in/out) applies to data subject's *future* data or *past* data). The choice_tense is used by a data subject to retroactively to opt-in/opt-out from inclusion of their data in a data product. Figure 6.11 illustrates how this retroactive policy semantics works. Bill has opted in for the data product sharing policy P_1 at timestamp $t1$ with choice_tense as $future$ after which his data is allowed to be included in the data product corresponding to P_1. Later on, at $t2$ Bill opt-outs with choice_tense as past and therefore his data is denied from historical queries to that data product. At $t3$, when Bill opts-out with choice_tense as future, his data is denied from the data product but his past data between $t3$ and $t4$ is allowed to be included in that data product. Finally, at $t4$ when he opt-outs with choice_tense as future, his data is denied from being included in the data product again.



DS	DCP	TS	choice	choice_tense
Bill	P_1	t1	Opt-in	Future
Bill	P_1	t2	Opt-out	Past
Bill	P_1	t3	Opt-out	Future
Bill	P_1	t4	Opt-in	Future

Allowed
Denied
Allowed
Denied
Allowed
Denied
Denied
Allowed
Denied
Allowed
t1
t2
t3
t4
Data Subject (DS) choices on Data Controller Policy (DCP) P1
Inclusion of Data Subject's data in data product mentioned in P1 based on their choices

Figure 6.11: Retroactive Policy Semantics.

If the data subject choice for a data product is opt-in, then the user is opting in to sharing their data with any service provider that satisfies one of the policy attributes in the data controller's policy. Likewise, if it is opt-out, then they are opting out of sharing with any service providers who have access to that data product. Each data product is associated with the default choice (opt-in or opt-out) and choice_tense (future or past) for data subjects

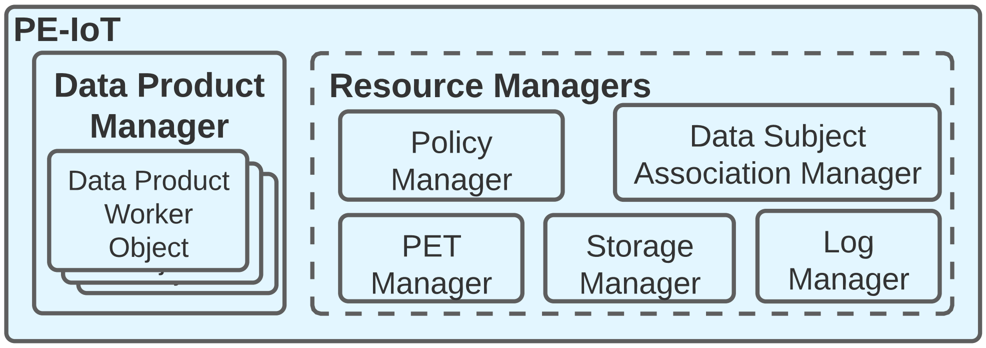


Figure 6.12: The prototype PE-IoT system's components

which are set by the data controller. The data controller policies and data subject choices are stored in the database.

6.3 Policy Manager in PE-IoT

We discuss a prototype system design of PE-IoT (See Figure 6.12) that realizes the data model and policy model previously presented. This prototype consists of a *Data Product manager* which conducts the flow of intercepted sensor stream through PE-IoT and coordinates with different resource managers to produce the units of privacy-preserving data stream defined by Data Products. The different aspects of PE-IoT data model are implemented as independent resource managers which can be executed separately. The rationale for such a system design are two-fold: 1) decoupling functions allows each resource manager to perform its tasks independently, 2) some of the resources (e.g., data controller policy, PET) might be stored remotely and independent resource managers allow us to move them closer to the resource. Each resource manager also include interfaces to interact with the Data Product Manager, and other resource managers. In this section, we briefly describe the *Policy Manager* and refer the reader to [53] for description of the other managers.

Policy Manager meets the requirement of ensuring that data sharing with service providers is explicitly allowed by users. It handles creating, modifying, and updating of data controller policies. It also stores the data subject choices associated with data controller policies. After associating data subject with the tuples, the Data Product Manager sends the the sensor stream to the Policy Manager. The Policy Manager evaluates the data subject choices for this stream and decides whether if a data subject's data can be included in a data product. Policy Manager also stores all the policies and provide the interface for both data controller and data subjects to create, read, update, and delete their policies.

6.4 Discussion

This chapter describes the integration of the mechanisms described in the thesis into TIPPERS and PE-IoT. This attests to the feasibility of the mechanisms presented. The framework to make smart spaces privacy-aware described in Chapter 3 was implemented using TIPPERS as the *privacy-aware smart space* and two different methods for user interaction. Creating such interfaces was not a goal of the thesis, and in part it was addressed by collaborators at CMU. However, developing such interfaces highlighted the challenge of assisting users in policy specification. Enforcement of user policies during query processing was done using the mechanisms presented in Chapter 4. Finally, integration with PE-IoT highlighted the potential benefits of policies and other PETs interacting and thus improving the security and privacy for user data.

Chapter 7

Conclusions and Future Work

"It is good to have an end to journey toward; but it is the journey that matters, in the end."

Ursula K. Le Guin, *The Left Hand of Darkness*

This chapter summarizes the conclusions of the work presented in this thesis on addressing challenges that arise when supporting fine grained access control policies in data management systems for IoT applications. Then, it discusses some future research directions in the context of the work presented in the thesis.

7.1 Conclusions

With advances in technology and arrival of new domains, the data collected and managed by data management systems increasingly contain newer forms of personally identifying information (PII). It is important that such PII is protected from unwanted access and inferences. This requirement of privacy has been made much more urgent by the recent introduction of stringent privacy laws such as General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA).

As a result of this changing landscape of technologies and privacy laws, today's data management systems face a variety of challenges in 1) meeting the privacy requirements mandated by regulations 2) enforcing privacy mechanisms efficiently in real time. Therefore, it is important to redesign these systems taking privacy into consideration. This is challenging as it introduces additional overheads to the different phases of data management. This thesis has focused on making fine grained policies in data management practical and scalable. In particular, the main contributions of this thesis have been:

Policy-based Privacy-by-Design Framework for IoT Smart Spaces: With the advent of Internet of Things (IoT), almost every device around us is smart, always on, and collecting data about us. The need for privacy has therefore become more crucial in IoT settings. An example of IoT is today's smart buildings where facility managers require a mechanism to notify residents and visitors of data collection practices. Similarly, building residents might want to specify their privacy preferences regarding these data collection practices. This thesis (Chapter 3) presented an approach to enable smart buildings and their inhabitants to communicate the data capture policies of the former and the privacy policies/preferences of the latter.

Scalable Fine Grained Access Control Policy Management for DBMS: Domains such as Big Data and the IoT can involve a potential large number of user-defined fine-grained policies. When it comes to enforcement of these large number of policies, today's data management systems are not able to efficiently handle the large number of checks required at the time of answering queries. The traditional approaches used in DBMSs are specification of authorization views and query writing. These methods have high overheads which makes them unsuitable for real-time query processing under large policy loads. There is a need for scalable Fine Grained access Control (FGAC) for Database Management Systems

(DBMS). This thesis (Chapter 4) presented Sieve, a general purpose middleware for DBMSs that scales access control for real-time query processing. Sieve does this by reducing the number of checks to be performed by filtering irrelevant records and impertinent policies. Given a large corpus of policies, Sieve uses them to generate a set of guarded expressions that are chosen carefully to exploit the best existing indexes, thus filtering the tuples against which policies are checked. Sieve also includes a policy evaluation operator which utilizes the context of a record (e.g., user/owner associated of record) and the query metadata (e.g., the person posing the query) to filter away policies which are not of interest to the tuple under consideration. By adaptively combining these two strategies based on a cost model, Sieve is able to significantly reduce overhead of policy checking. The experimental evaluation of the system in different IoT settings shows that Sieve significantly reduces the overhead (2X-10X) of access control at query time when compared with the baselines. This work is the first to identify and propose a solution for scalable access control and has opened up an interesting research area.

Protecting access controlled data from leakages: Although restricted by access control policies, an adversary with background knowledge can infer information about the sensitive data from non-sensitive data. This thesis (Chapter 5) presented a new form of inference attacks on access control protected data through data dependencies. In particular, this work focused on denial constraints and provenance based dependencies which represent integrity constraints and function based constraints respectively. A new security model was developed for ensuring privacy for the sensitive cells under the presence of aforementioned inference channels. The prototype system built includes holistic algorithms which take as input a database instance, set of access control policies, and a set of data dependencies and made sure all the sensitive cells met the necessary security requirement.

Integration into IoT systems Finally, this thesis (Chapter 6) also describes the experiences of integrating the various techniques presented into two IoT systems. The approach of privacy-aware smart spaces along with the policy model included, has been incorporated into the TIPPERS IoT data management system which is deployed at UC Irvine . The access control interfaces developed for TIPPERS, assisted users in specifying privacy preferences on the data sharing practices advertised by the building. The policy engine developed as part of this thesis was also integrated with PE-IoT which is privacy-preserving middle-ware for managing IoT sensor data flows.

7.2 Future Work

This thesis explored supporting fine-grained access control policies in Data Management Systems. Thus the focus has mainly been on the aspect of data sharing. However, supporting policies in other phases of data management such as – 1) Capture, 2) Store 3) Process – is an exciting challenge. Building an end to end policy framework which performs policy enforcement on the complete life cycle of data in a Data Management system remains an open problem. In addition, this thesis has opened up doors to future research in the following directions.

Policy based data management systems: This thesis explored some aspects of co-optimizing query processing and policy enforcement but it also opened up a fertile research area. There is still room for significant improvement if data management systems consider policies as first class citizens. DBMSs can then use this knowledge to build policy-based indices to intertwine query optimization and policy enforcement more closely. Further on, the DBMS can also determine unreachable parts of the database based on policies and move these away from caches. Similar to the saying of *one size does not fit all* in the database community, *one policy or access control model does not fit for every scenario.* Thus, there is a need to develop policy models supporting different data models (e.g., non-relational, array based), different database technologies (e.g., polystores) and new domains (e.g., intelligent transportation systems, smart water management). In many of these scenarios, multiple entities might be participating in data exchanges across different forms of networks such as cloud or fog based. Therefore, it would be important to support multiparty distributed access control across such systems where data is manipulated and policies need to be enforced at different places.

Translation of the regulatory requirements into system-level design choices: As stricter regulations are coming up in different parts of the world, this translation remains an open challenge. For example, the Right to be Forgotten requirement mandates that users should be able to ask data controllers to delete their data and provide proof of such deletion. However, due to data distribution and redundancy in Big Data systems, ensuring that data is completely destroyed is extremely challenging. As data moves from one place to the other, compliance of retention policies and their verification has to be repeated again and again. Secure deletion schemes which utilize different cryptographic techniques have been proposed but none of them have been integrated into data management systems. This becomes even more challenging with complex data processing pipelines which utilize Machine Learning algorithms. In this setup, the contribution of each individual pieces of data becomes fuzzy in the deep layers of processing. Similarly, for verifiable compliance and Data Protection Impact Assessments (GDPR Article 35), current models of policy enforcement, require a trusted centralized entity. This becomes challenging in the aforementioned newer domains and therefore a verification method which relies on tamper proof logs is required in distributed settings with no trusted entities. In cases of potentially high-risk processing activities, data controllers can use this to study the impact of privacy policies on individual's data.

Combining privacy policy and privacy mechanism: The work presented in Chapter 5, focuses on a specific challenge which arises when dealing with the combination of these traditionally disparate fields. Further exploration of this issue is required. For example, in

Differential Privacy, which provides bounds on privacy leakage with an unknown adversary, an open problem is how to appropriately set the noise factor. The policies specified by users could be potentially used to compute with the appropriate value. There are many challenges to be addressed to do such a combination meaningfully and efficiently. Translation of user specified policies into the parameters of an enforcement mechanism is an exciting avenue to explore. Similarly, building bridges between policy requirements and guarantees of the mechanism is a compelling problem to solve. Combining these two separate areas of research can also spur improvements in implementation of both. Understanding the impact on the privacy guarantees of mechanisms when policies are dynamically updated. Privacy and security systems are only as strong as its weakest component and in today's systems more often than not these are the users who are the least informed on privacy. Through combination of policy and mechanism it becomes possible to build **Explainable Security and Privacy** where users can understand, appropriately trust, and effectively manage the privacy by design systems.

Bibliography

[1] California consumer privacy act CCPA. https://oag.ca.gov/privacy/ccpa. [Online; accessed 1-June-2020].

[2] California online privacy protection act CalOPPA. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=BPC&sectionNum=22575. [Online; accessed 1-June-2020].

[3] General data protection regulation GPDR. https://gdpr.eu/. [Online; accessed 1-June-2020].

[4] Privacy assistant. https://privacyassistant.org. [Online; accessed 1-June-2020].

[5] Privacy-preserving contact tracing. https://www.apple.com/covid19/contacttracing. [Online; accessed 1-June-2020].

[6] Tippers. https://tippers.ics.uci.edu. [Online; accessed 1-June-2020].

[7] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st International Conference on Data Engineering (ICDE)*, 2005.

[8] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the VLDB Endowment*, 2002.

[9] D. A. Albertini, B. Carminati, and E. Ferrari. An extended access control mechanism exploiting data dependencies. *International Journal of Information Security*, 16(1), 2017.

[10] S. Almanee, G. Bouloukakis, D. Jiang, S. Ghayyur, D. Ghosh, P. Gupta, Y. Lin, S. Mehrotra, P. Pappachan, E. Shin, N. Venkatasubramanian, G. Wang, and R. Yus. Semiotic: Bridging the semantic gap in iot spaces. In *6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys)*, 2019.

[11] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[12] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster. Keeping the smart home private with smart(er) iot traffic shaping. *PoPETs*, 2019(3), 2019.

[13] D. Banisar. National comprehensive data protection/privacy laws and bills 2020. *Privacy Laws and Bills*, 2020.

[14] D. Bell. Looking back at the Bell-La Padula model. In *21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[15] M. Benisch, P. G. Kelley, N. Sadeh, and L. F. Cranor. Capturing location-privacy preferences: quantifying accuracy and user-burden tradeoffs. *Personal and Ubiquitous Computing*, 2011.

[16] M. Berenguer, M. Giordani, F. Giraud-By, and N. Noury. Automatic detection of activities of daily living from detecting and classifying electrical events on the residential power line. In *10th International Conference on e-health Networking, Applications and Services (HealthCom)*, 2008.

[17] E. Bertino. Data security and privacy in the IoT. In *19th International Conference on Extending Database Technology (EDBT)*, 2016.

[18] E. Bertino, G. Ghinita, A. Kamra, et al. Access control for databases: concepts and systems. *Foundations and Trends in Databases*, 3(1–2), 2011.

[19] L. Brandeis and S. Warren. The right to privacy. *Harvard law review*, 4(5), 1890.

[20] A. Brodsky, C. Farkas, and S. Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering*, 12(6), 2000.

[21] J.-W. Byun, E. Bertino, and N. Li. Purpose based access control of complex data for privacy protection. In *10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2005.

[22] J.-W. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal*, 17(4), 2008.

[23] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *ACM Transactions on Privacy and Security*, 14(3), 2011.

[24] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2003.

[25] Y. Chen, A. Machanavajjhala, M. Hay, and G. Miklau. PeGaSus: Data-adaptive differentially private stream processing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[26] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13), 2013.

[27] P. Colombo and E. Ferrari. Enforcement of purpose based access control within relational database management systems. *IEEE Transactions on Knowledge and Data Engineering*, 26(11), 2014.

[28] P. Colombo and E. Ferrari. Efficient enforcement of action-aware purpose-based access control within relational database management systems. *IEEE Transactions on Knowledge and Data Engineering*, 27(8), 2015.

[29] P. Colombo and E. Ferrari. Fine-grained access control within NoSQL document-oriented datastores. *Data Science and Engineering*, 1(3), 2016.

[30] P. Colombo and E. Ferrari. Towards a unifying attribute based access control approach for NoSQL datastores. In *33rd International Conference on Data Engineering (ICDE)*, 2017.

[31] P. Colombo and E. Ferrari. Access control enforcement within MQTT-based Internet of Things ecosystems. In *23nd ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2018.

[32] P. Colombo and E. Ferrari. Access control technologies for big data management systems: literature review and future trends. *Cybersecurity*, 2(1), 2019.

[33] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, 17, 2012.

[34] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: A commodity data cleaning system. In *ACM SIGMOD International Conference on Management of Data*, 2013.

[35] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for xml documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2), 2002.

[36] A. Das, M. Degeling, D. Smullen, and N. Sadeh. Personalized privacy assistants for the internet of things: Providing users with notice and choice. *IEEE Pervasive Computing*, 17(3):35–46, 2018.

[37] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. Fragmentation in presence of data dependencies. *IEEE Transactions on Dependable and Secure Computing*, 11(6):510–523, 2014.

[38] H. Delugach and T. Hinke. Wizard: a database inference analysis and detection system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 1996.

[39] D. E. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *IEEE Symposium on Security and Privacy*, pages 134–134, 1985.

[40] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3&#8211;4), 2014.

[41] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4), 2014.

[42] E. Lear, R. Droms, and D. Romascanu. Manufacturer Usage Description Specification. Internet-Draft, IETF Network Working Group, 2017.

[43] N. Eagle and A. S. Pentland. Reality mining: sensing complex social systems. *Personal and ubiquitous computing*, 10(4), 2006.

[44] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[45] C. Farkas and S. Jajodia. The inference problem: A survey. *ACM SIGKDD Explorations Newsletter*, 4(2), 2002.

[46] D. F. Ferraiolo and D. R. Kuhn. Role—based access controls. In *15th NIST——NSA National Computer Security Conference*, 1992.

[47] B. C. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys*, 42(4), 2010.

[48] C. Ge, X. He, I. F. Ilyas, and A. Machanavajjhala. APEx: Accuracy-aware differentially private data exploration. In *International Conference on Management of Data (SIGMOD)*, 2019.

[49] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The llunatic data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9), 2013.

[50] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing, (STOC)*, 2009.

[51] S. Ghayyur, Y. Chen, R. Yus, A. Machanavajjhala, M. Hay, G. Miklau, and S. Mehrotra. IoT-Detective: Analyzing IoT data under differential privacy. In *International Conference on Management of Data (SIGMOD)*, 2018.

[52] S. Ghayyur, D. Ghosh, X. He, and S. Mehrotra. Towards accuracy aware minimally invasive monitoring (MiM). In *International Workshop on Theory and Practice of Differential Privacy (TPDP@CCS)*, 2019.

[53] S. Ghayyur, P. Pappachan, G. Wang, S. Mehrotra, and N. Venkatasubramanian. Designing privacy preserving data sharing middleware for Internet of Things. In *3rd Workshop on Data: Acquisition To Analysis (DATA@SenSys)*, 2020.

[54] J. Gluck, F. Schaub, A. Friedman, H. Habib, N. Sadeh, L. F. Cranor, and Y. Agarwal. How Short Is Too Short? Implications of Length and Framing on the Effectiveness of Privacy Notices. In *12th Symposium on Usable Privacy and Security (SOUPS)*, 2016.

[55] O. Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78, 1998.

[56] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In H. Jacobsen, editor, *ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2004.

[57] P. Gupta, M. J. Carey, S. Mehrotra, and R. Yus. SmartBench: A benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment*, 13(11), 2020.

[58] M. Haddad, J. Stevovic, A. Chiasera, Y. Velegrakis, and M.-S. Hacid. Access control for data integration in presence of data dependencies. In *International Conference on Database Systems for Advanced Applications*, 2014.

[59] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS)*, 23(2), 1998.

[60] J. Heo, H. Lim, S. B. Yun, S. Ju, S. Park, and R. Lee. Descriptive and predictive modeling of student achievement, satisfaction, and mental health for data-driven smart connected campus life service. In *9th International Conference on Learning Analytics & Knowledge (LAK)*, 2019.

[61] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas. Attribute-based access control. *Computer*, 48(2), 2015.

[62] I. F. Ilyas and X. Chu. *Data cleaning*. ACM, 2019.

[63] H. Jafarpour, S. Mehrotra, N. Venkatasubramanian, and M. Montanari. MICS: an efficient content space representation model for publish/subscribe systems. In A. S. Gokhale and D. C. Schmidt, editors, *3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2009.

[64] M. Langheinrich. Privacy by design—principles of privacy-aware ubiquitous systems. In *3rd International Conference on Ubiquitous Computing (UbiComp)*, 2001.

[65] H. Lee and A. Kobsa. Privacy preference modeling and prediction in a simulated campuswide IoT environment. In *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2017.

[66] P. Lee, E. Shin, V. Guralnik, S. Mehrotra, N. Venkatasubramanian, and K. T. Smith. Exploring privacy breaches and mitigation strategies of occupancy sensors in smart buildings. In *1st ACM International Workshop on Technology Enablers and Innovative Applications for Smart Cities and Communities (TESCA)*, 2019.

[67] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the VLDB Endowment*, 2004.

[68] F. Li, J. Sun, S. Papadimitriou, G. A. Mihaila, and I. Stanoi. Hiding in the crowd: Privacy preservation on evolving streams through correlation tracking. In *IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007.

[69] J. Lin, B. Liu, N. Sadeh, and J. I. Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *10th USENIX Conference on Usable Privacy and Security (SOUPS)*, 2014.

[70] Y. Lin, D. Jiang, R. Yus, G. Bouloukakis, A. Chio, S. Mehrotra, and N. Venkatasubramanian. LOCATER: cleaning wifi connectivity datasets for semantic localization. *Proc. VLDB Endow.*, 14(3), 2020.

[71] M. A. Lisovich, D. K. Mulligan, and S. B. Wicker. Inferring personal information from demand-response systems. *IEEE Security & Privacy*, 8(1), 2010.

[72] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti. Follow My Recommendations: A Personalized Privacy Assistant for Mobile App Permissions. In *12th Symposium on Usable Privacy and Security (SOUPS)*, 2016.

[73] K. Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, Inc., 2008.

[74] W. Medhat, A. Hassan, and H. Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal*, 5(4):1093–1113, 2014.

[75] S. Mehrotra, A. Kobsa, N. Venkatasubramanian, and S. R. Rajagopalan. TIPPERS: A privacy cognizant IoT environment. In *IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, 2016.

[76] G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. In *ACM SIGMOD international conference on Management of data*, 2004.

[77] R. V. Nehme, H.-S. Lim, and E. Bertino. Fence: Continuous access control enforcement in dynamic data stream environments. In *3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[78] A. Onet. The chase procedure and its applications in data exchange. In *Dagstuhl Follow-Ups*, volume 5. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[79] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. *Proc. VLDB Endow.*, 8(12), 2015.

[80] P. Pappachan, M. Degeling, R. Yus, A. Das, S. Bhagavatula, W. Melicher, P. E. Naeini, S. Zhang, L. Bauer, A. Kobsa, et al. Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.

[81] P. Pappachan, R. Yus, S. Mehrotra, and J. Freytag. Sieve: A middleware approach to scalable access control for database management systems. *Proc. VLDB Endow.*, 13(11), 2020.

[82] S. R. Peppet. Regulating the Internet of Things: First Steps toward Managing Discrimination, Privacy, Security and Consent. *Texas Law Review*, 93, 2014.

[83] X. Qian, M. Stickel, P. Karp, T. Lunt, and T. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

[84] B. Qolomany, A. I. Al-Fuqaha, A. Gupta, D. Benhaddou, S. Alwajidi, J. Qadir, and A. C. M. Fong. Leveraging machine learning and big data for smart buildings: A comprehensive survey. *IEEE Access*, 7, 2019.

[85] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-preserving stream analytics. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[86] A. Rao, F. Schaub, N. Sadeh, A. Acquisti, and R. Kang. Expecting the Unexpected: Understanding Mismatched Privacy Expectations Online. In *12th Symposium on Usable Privacy and Security (SOUPS)*, 2016.

[87] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820*, 2017.

[88] J. L. Riccardi. The german federal data protection act of 1977: Protecting the right to privacy. *BC Int'l & Comp. L. Rev.*, 6:243, 1983.

[89] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2004.

[90] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.

[91] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat. Provsql: Provenance and probability management in postgresql. *Proceedings of the VLDB Endowment*, 11(12), 2018.

[92] S. Shastri, M. Wasserman, and V. Chidambaram. The seven sins of personal-data processing systems under GDPR. In *11th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[93] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *1974 Annual Conference*, 1974.

[94] G. Sun, V. Chang, M. Ramachandran, Z. Sun, G. Li, H. Yu, and D. Liao. Efficient location privacy algorithm for Internet of Things (IoT) services and applications. *Journal of Network and Computer Applications*, 89, 2017.

[95] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[96] M. Thuraisingham. Security checking in relational database management systems augmented with inference engines. *Computers & Security*, 6(6), 1987.

[97] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the VLDB Endowment*, 2007.

[98] Y. Wang and A. Kobsa. Privacy-enhancing technologies. *Social and Organizational Liabilities in Information Security*, 2008.

[99] Xu Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *29th International Conference on Data Engineering (ICDE)*, 2013.

[100] R. Yus, G. Bouloukakis, S. Mehrotra, and N. Venkatasubramanian. Abstracting interactions with iot devices towards a semantic vision of smart spaces. In *6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys)*, 2019.

[101] D. Zhang, C.-Y. Chan, and K.-L. Tan. An efficient publish/subscribe index for e-commerce databases. *Proceedings of the VLDB Endowment*, 7(8):613–624, 2014.

[102] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. Privbayes: Private data release via bayesian networks. *ACM Transactions on Database Systems (TODS)*, 42(4), 2017.

[103] J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *International Conference on Management of Data*, 2016.