Abstract of Practical Privacy via New Systems and Abstractions, by Kinan Dak Albab, Ph.D., Brown University, May 2025.

Data privacy has become a focal point of public discourse. In response, data protection and privacy regulations have been enacted throughout the world, including GDPR and CCPA, and companies make various promises to end users in their privacy policies. However, high-profile privacy violations remain commonplace, in part because complying with regulations and policies is challenging for applications and developers.

This dissertation demonstrates how we can help application developers achieve privacy compliance by designing new privacy-conscious systems and abstractions. The dissertation discusses two systems. The first, K9db, is a privacy-compliant database that supports GDPR-style subject access requests by construction. The second, Sesame, is a system for end-to-end enforcement of privacy policies in web applications. Sesame provides practical guarantees by combining a new static analysis for data leakage with advances in memory-safe languages, lightweight sandboxing, and engineering practices around code review. This dissertation demonstrates that creating privacy abstractions at the system level simplifies compliance and guarantees privacy requirements by design.

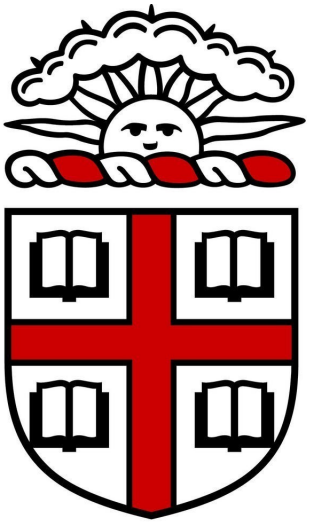
Practical Privacy via New Systems and Abstractions

by

Kinan Dak Albab

B.S., American University of Beirut, 2015

M.S., Boston University, 2020



A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2025

This dissertation by Kinan Dak Albab is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Malte Schwarzkopf, Advisor

Recommended to the Graduate Council

Date _____ _____

Shriram Krishnamurthi, Reader

Date _____ _____

Ugur Cetintemel, Reader

Approved by the Graduate Council

Date _____ _____

Thomas A. Lewis
Dean of the Graduate School

Dedication

To Kirby and Toshiro. Roam happy and play free. Forever.

To those who sought Life, Freedom, and the Pursuit of Happiness and made the ultimate sacrifice along the way, be it under fiery skies, solitary exiles, or torturous dungeons. To my fellow Syrians who stood against tyranny and terrorism: Bassel Shehadeh, May Skaf, Raed Al Fares, Ghiath Matar, Michel Kilo, Samira Khalil, Razan Zaitouneh, Wael Hamada, Nazem Hammadi, Fadwa Suleiman, Paolo Dall'Oglio, Hussein Harmoush, and many, many others I only know by their courage but not by name or face. You are forever my north star and my impenetrable shield against nihilism, cynicism, and despair. You had something worth dying for, so there must be something worth living for, I just have to find it.

Acknowledgments

I owe everything in this dissertation and all that I am and will ever be to the many friends I found along the way. I hope the blind Dahr[1] is kind enough to preserve, for eternity, this woefully inadequate account of your role in my research and journey through life.[2] I am not good enough with words to truly express my overwhelming gratitude and love, but I will give it a go. I am deeply thankful:

To Malte Schwarzkopf for being the greatest gift that can happen to a researcher. I am proud to call myself your student. I hope to follow in your footsteps and be as you have shown me. If I am ever lost, I will always look up to you to remind myself of how to do good science and how to teach, mentor, and empower students.

To Artem Agvanian[3] for a combination of intelligence and gentleness rarely found in men and for good taste in Coffee and Sazeracs. To Corinn Tiffany[3] for teaching me how to footshake dance and helping me present professionally at conferences. I stopped having things I can teach you a long time ago, but I still learn so much from the both of you every day. This dissertation and its author owes you boundless credit. You made wanting to be a professor a no-brainer, but I am afraid neither my future students nor my future friends will live up to your example.

To Sophie Byrne for teaching me true kindness and showing me nothing but bangers, always. You made this small city and this run-of-the-mill man feel so big and extraordinary. Whatever achievements I may have in the future, being a good friend to you will always remain my favorite. To Nell for love, trust, routine, purpose, and irresistible charm. Without the two of you, there would be no Sesame, with or without the bun.

To Shriram Krishnamurthi and Ugur Çetintemel for patience and guidance from my comps proposal

[1] An arabic word referring to the unpredictable and cruel passage of time and fate.

[2] After all, fate memorialized far less glorious things https://en.wikipedia.org/wiki/Complaint_tablet_to_Ea-n%C4%81%E1%B9%A3ir.

[3] A failure of language is that it seeks to impose an order even when none fits. The order here is merely alphabetical.

to this dissertation and the many paper submissions in between. You took valuable time above and beyond to offer sharp observations and helpful feedback. Your efforts did not only improve my papers and dissertation, but also helped me mature as a researcher and academic.

To Bassel Mabsout for being the ultimate sous chef and hype man. You were my guardian angel in my darkest hour. I hope you remain as lucky as ever. To Nicolas Alhaddad for forgiving my deep faults and many imperfections. We had over a decade of unrepeatable memories. Here's to many more.

To Justus Adam and his "two cousins" for gentle accountability, wit, and insight in research and beyond. There is no one else I would trust to take the reign of my domain.

To Mayank Varia for teaching me all I know about cryptography and taking a genuine interest in my growth and career, even when none was expected. I owe much of this to you. To Azer Bestavros and Assaf Kfoury for believing in me when I was even more of a nobody and time and again since. I look forward to being your colleague.

To Livia Zhu for unrelenting dedication and grit hidden behind an untroubled friendly smile. You are my hero. To Howie Chen for tantalizing elegance and fashion. To Timothée Zerbib for being a great sport with infuriating taste buds. To Hannah Gross whom I strive to mimic in intelligence, flair, and attitude.

To Alex Portland, Sarah Ridley, Allen Aby, Benjamin (and Banji) Kilimnik, Wyatt Howe, Ishan Sharma, Alexandre Doukhan, Aaron Jeyaraj, and Raj Paul for their indispensable time, sweat, and insight.

To Muna Khalil for unlimited support unaffected by geography or time. When my life was a cycle of work and sleep, I lived through your adventures and stories. To Dany, Karam, Omar, Yasser, Mehdi, Constantine, André, and Hadi for bonds that survive time, distance, and trauma. On your collective rock I have built my church.

To Merlin Cunniff for telling me about poetry, teaching me to mine and fish, and for overall wizardry and linguistic mischief. To Jesse Smith and Patrick Cull for perspectives otherwise beyond my reach. To Bill, Ivan, Johnny, Mark, Martin, Matt, Ryan, Susan, and all the staff and regulars at GCB. To Bobby, Brian, Ron, and all the recurring and one-off guests at the Wednesday evening drinking society (weds).

To Rawane Issa for setting me on this long PhD journey and seeing me through foolish youth until this more foolish maturity. Even in absence, you are the voice that my head uses to put me on the path of reason and good judgment.

To Joseph, the prankster Amtrak conductor, for brightening a dull and long commute. To my friend the Chef at Oppa Sushi, the sandwich makers at Cutty's, and the shuckers at Gift Horse. To David Lynch and Agent Cooper for my many dreams and nightmares that kept me and my students entertained.

To Maryam Abuissa for an echo of a far away home and a sense of belonging I had long lost. To Franklin Harding, Eileen Nolan, and Phuoc Pham Van Long (aka the funny man) for humoring me and my tin foil hats. To Gavin Gray, Alexander Lee, Jasmine Liu, and the rest of the first years for bringing with them a dose of much needed sunshine.

To Peter (Peyer) Flockhart for under-credited contributions, ribs, and good times that always pick up where we left off. To Lucy Qin whom fate seems to always have us chasing each other.

To Carolyn Zech, Franco Solleza, Aijah Garcia, Megan Frisella, Vic Li, Shirley Loayza Sanchez, Sreshtaa Rajesh, Leonhard Spiegelberg, Lily Tsai, Julia Netter, the up-and-coming Maxi, and the entirety of ETOS, past, present, and future: I could not ask for a better family.

To Vincenzo Prosperi, Olivia Tiedemann, Anders Erickson, and Leandro DiMonriva for beloved recipes and crowd-favorite cocktails that drew smiles on my students' faces on countless occasions.

To everyone who has ever taken or will ever take CS2390. If I ever come to regret my career choices, I will blame thee.

To Ben Getchell and all the members of Leather Lung. Whenever I celebrate a submission at one of your shows, it ends up being accepted. Keep playing (my career depends on it).

To the entire New England Metal and Prog communities, all the strangers I befriended at shows, and all the staff at the Palladium, Middle East, Paradise Rock Club, Fete, and Sinclair. To the old guy at every prog rock concert colloquially known as "Bilbo Baggins".

To Elijah Rivera, Siddhartha Prasad, Nikos Vasilakis, Akshay Naranyan, Deepti Raghavan, Will Crichton, and the systems, programming languages, and cryptography communities at Brown for being amazing colleagues and a fantastic support system. You have set the bar so high.

To Tom Brady for encouraging me to resist the passage of time, and to Zinedine Zidane and all the Real Madrid legends for helping me accept it.

To Paul Attie for that first spark of curiosity and inspiration. I am forever grateful. To Mohamad Jaber,

George Turkiyyah, Wadi' Jureidini, Louay Bazzi, Nader El-Bizri, Eric Goodfield and my mentors at AUB. To George, Adel, Bayan, Baydon, Mohammad El-Hajj, and the Bliss 207 crew where this all started.

To Andre Toriz and Zu Jaber for teaching me etiquette and story telling. To Zakaria, Assem, and all those who passed by the Captain's Cabin. To Beirut where I felt human for the very first time. *"Remember me whenever the night goes dark."*

To my friends, mentors, and teachers from a previous life: Sarah Scheffler, Dina Bashkirova, Palak Jain, Soham Sinha, Craig Einstein, Sasan Golchin, Yara Awad, Ran Canetti, Gabe Kaptchuk, Adam Smith, Leonid Reyzin, Leonid Levin, and the entire BUSec group. To Frederick Jansen, San Tran, and all of the SAIL team and interns.

To Neal, Jack, Belew, and Fripp: absent lovers, absent lovers. To Meshuggah, Calle Thomér, Buster Odeholm, Calder Hannan and everyone who dances to any discordant system. To Frank Klepacki and Stuart Chatwood for the soundtrack of the many 70s movies training montages I had in my journey. To Ozzy, you are no ordinary man.

To Naruto and Jiraya for the power of friendship and ramen and for getting me through high school and a pandemic. To Spades Slick and the Midnight Crew, John Egbert, Jade Harley, and the 12 trolls for "cool" jazz and data structure humor. To Gary Cooper and the *"strong, silent type"*.

To the camels in the desert, olives in the mountains, and branzini in the sea. To Abu Nuwas, Sadiq Jalal al-Azm, and all the heretics between. To Shiner Bock and anyone who brews it and Arak Al Rayan and anyone who drinks it.

To the long line of Dak Albabs that came before for the solid ground under my feet. *"Everyone should have a monument named after their family."*

To everyone who faced me in a game of Dota 2 and lost. You are all fresh meats.

"To absent friends, in memory still bright"

Contents

List of Figures

CHAPTER 1

Introduction

We live in a privacy dystopia where online services, both large and small, frequently misuse data, either intentionally or unintentionally. This is likely to get worse as services collect more and more data on users, *e.g.,* through wearables and other smart devices [Cho25], and as this data is used to train larger and larger LLMs and other machine learning models, which may themselves leak individual data from their training sets to other users [Gho24]. This is evidenced by recent calls by the FTC reminding AI companies to uphold their privacy and confidentiality commitments [Sta24], and by the increasingly negative consumer sentiment towards the scale of data collection [ARA+19] and how much control internet companies have over users' personal data [Bru21].

Recent privacy and data protection laws attempt to alleviate these privacy concerns by posing various requirements on how applications may collect and process data. We discuss these laws and their requirements in greater detail later in this dissertation, including the EU's GDPR [GDPR16] and California's CCPA [CCPA18], among others [Bra19; Ind19; Tho19]. Complying with these laws poses technical challenges for organizations and their developers, even when they are well-meaning and have meaningful incentives to comply.

A large portion of these challenges stem from an ad hoc approach to compliance, where application developers must manually implement features required by laws, keep a mental model of all the policies that govern how data is collected and used, and add the required explicit checks or implicit logic flows throughout their application code to enforce these policies. This manual approach imposes significant burdens on application developers. It is also prone to human error and omission, especially as the application logic and features continue to evolve over time.

As a result, hefty fines due to privacy violations have become commonplace, which indicates systematic problems in current approaches to privacy and compliance, further decreases user trust, and costs companies millions in fines. For example, Instagram was fined €405M in 2022 for an application bug that causes children's contact information to be publicly revealed in limited cases (*e.g.,* when they have business accounts), although Instagram had disallowed this in its policies and correctly enforced it in most cases [Eur22].

My research presented in this dissertation implements new practical privacy abstractions in familiar systems that are compatible with the current web architecture and applications. Furthermore, it shows how these systems and abstractions can help well-intentioned application developers meet their privacy requirements with reasonable engineering and performance overhead. This dissertation focuses on two of my systems, K9db [DSA+23] and Sesame [DAA+24a], which target users' consent and control over their data in web applications, specifically along the dimensions specified in privacy laws such as the GDPR. Some of my other work outside the scope of this thesis demonstrates that baking privacy abstractions into other kinds of systems also simplifies realizing other notions of privacy that extend beyond privacy laws [DIL+19; DIV+22].

Thesis Statement. The results presented in this dissertation seek to provide evidence in support of the following thesis statement:

> It is possible to design *familiar* infrastructure systems *compatible* with the modern web that provide new privacy abstractions to help *well-intentioned* developers meet their desired privacy properties, including the requirements of privacy laws around user consent and control.
>
> Tracking data ownership and privacy policies at the database and web framework levels provides automatic compliance guarantees for the storage and processing of data, respectively.
>
> These new privacy systems and abstractions help with compliance while requiring reasonable engineering effort and with low performance overhead.

This thesis statement refers to three key terms and assumptions that we define below.

Familiar Systems. Earlier work demonstrates that systems can provide strong privacy guarantees against

powerful adversaries by using complex cryptographic techniques or niche programming languages. However, application developers face steep learning curves when attempting to use such systems. In order to achieve our goal of simplifying compliance for developers in practice, we aim to provide systems that are familiar to application developers and thus easier for them to learn and use.

Indeed, K9db is an SQL database that transparently implements the MySQL protocol, a popular paradigm familiar to many web developers. Sesame relies on the Rust programming language, which as of this writing is the 8th most popular programming language according to the PYPL index [Car23], and continues to rapidly increase in popularity [Sza22]. We discuss ideas and challenges for adapting the techniques in K9db and Sesame to other popular database systems and programming languages in §6.1.

Compatible Systems. We aim to build systems that are compatible with the modern web ecosystem and its application architecture. Currently, applications collect data from users and other services, store it in back-end databases, and process it for a variety of purposes. Furthermore, applications may invoke third-party services and APIs to perform part of their processing *e.g.,* online payment processors. Applications frequently offer free or discounted services to users by relying on targeted advertising to generate revenue.

Privacy laws aim to improve end-user privacy without radically disrupting or redesigning this ecosystem. Similarly, our goal is to build practical systems that are compatible with the modern web application ecosystem. For example, K9db simplifies the handling of user data access and deletion requests without disrupting the way applications store and query data. Contrast this with alternative Web3 systems [MSH+16; Nos23] that ensure users have control of their data but require a radical redesign of web applications to operate in a decentralized manner, or with cryptographic and distributed systems [BKV+21; HZX+16] that are incompatible with the advertising-driven revenue model of the web.

Well-Intentioned Developers. We aim to help application developers within well-meaning organizations comply with their desired privacy policies. An organization's privacy policies may encode end-user consent and control requirements from privacy laws as well as other self-imposed policies *e.g.,* for authentication, security, or liability.

In this model, we assume that organizations are motivated to define and comply with such policies by a combination of reputation, legal, and financial incentives. We aim to help organizations meet these policies even when they have complex applications built and maintained by large teams of developers, where

manual ad hoc approaches to compliance are error-prone. Instead, we envision that such organizations mandate their developers to use systems and tools like Sesame and K9db, and create internal processes around them, such as code review or the use of various associated linter and static analysis tools.

We observe that many (but not all) cases of GDPR violations fall under this model, where organizations did not have any obvious advantage, financial or otherwise, from their incompliant behavior. Instead, their violations were simply the result of application bugs or human error due to the manual nature of current approaches to compliance. We believe that this includes the Instagram violation discussed earlier that accidentally revealed children's data when they created business accounts, and many of the violations we discuss in §2.1. Techniques for guaranteeing privacy properties when organizations and applications are actively malicious exist, but they come with several performance and compatibility tradeoffs and are outside the scope of this dissertation.

Below, we provide a brief overview of privacy laws and their requirements, other conceptual privacy frameworks, and existing systems and abstractions that support them (§1.1). we state the contributions described in this dissertation (§1.2) and outline its overall structure (§1.3). Finally, we list prior publications that relate to or inform this dissertation (§1.4).

1.1 Background

Privacy is a fuzzy concept that often means different things to different people in different contexts. This dissertation focuses on privacy from the lens of ensuring that applications respect user consent and privacy preferences and provide them with reasonable control over their data, notably the ability to access or delete that data on request. This is a challenging task that application developers often get wrong, in large part because they lack automatic systems and tooling support, and must instead rely on ad hoc and error-prone manual reasoning approaches. This dissertation explores how to assist application developers in ensuring that their applications meet these requirements by integrating new privacy abstractions into off-the-shelf infrastructure systems, such as databases and web frameworks.

Other complementary notions of privacy include secure computation of functions over hidden, secret data, and statistical privacy techniques for providing individuals who may have contributed their data to statics and aggregates with plausible deniability. The widespread adoption of these techniques also faces challenges in practice, originating similarly from a lack of practical systems and tools. In some of my other work outside the scope of this thesis, I similarly explore how to make these techniques easier to

use in the real world by integrating them into easy-to-use familiar systems. Crucially, these techniques provide *orthogonal* privacy and security guarantees to the consent and control requirements that are the focus of this dissertation, and they target a *different threat model* than the research presented here. We provide a brief discussion of some future work ideas on how the work presented in this dissertation may complement these techniques in §6.3.

Finally, other work looks at privacy with respect to powerful, malicious adversaries, such as nation-state mass surveillance or protections against data breaches and hacking groups. Privacy is also often intertwined with adjacent notions, such as content moderation, algorithm fairness, and censorship resistance, among others. Although important and interesting, these concepts are orthogonal and go beyond the scope of this dissertation.

1.1.1 Conceptual Privacy Frameworks

User Consent and Control. The first notion of privacy focuses on ensuring that applications respect user consent and that users maintain sufficient control over data about them stored by applications and services. This approach is at the heart of privacy laws, such as the EU's GDPR [GDPR16] and the CCPA in California [CCPA18], which require companies to acquire and respect informed user consent when collecting and processing user data, and allow users certain control rights over that data, including the rights to access and delete it. These laws rely on active enforcement from government and data protection agencies, and violations often result in large fines. See §2.1 for more details.

These fines along with the increasing public interest mean that privacy scandals have significant reputation and financial consequences for companies. These consequences create strong incentives for companies to comply with privacy laws. As a result, companies can no longer treat privacy as an afterthought, and some are even using privacy in their branding, *e.g.,* Apple's "Privacy. That's iPhone" ad campaign [OFl22]. However, compliance remains technically challenging and can pose a significant burden on smaller organizations without significant technical expertise. In large part because the status quo relies on manual and ad hoc approaches that are error-prone and pose a significant burden on application developers. Some existing research attempts to create tools and systems to simplify compliance, but faces significant performance, expressivity, and usability challenges (see §2.2.1). The research presented in this dissertation is an important step towards overcoming these challenges.

Secure Computation and Privacy-Enhancing Technologies. Privacy-enhancing technologies (PETs) aim to enable computation over sensitive data while minimizing data collection and maximizing data security. Cryptographic techniques, such as secure multiparty computation [BGW88; Sha79; Yao86] and fully homomorphic encryption [Gen09], as well as recent advances in trusted execution environments and enclaves, allow computations over sensitive data in "encrypted" (or otherwise hidden) form. As a result, the parties performing the computation are unable to observe or learn information about the inputs or other intermediary values other than the final output of the computation. These techniques have seen some use in the real world *e.g.,* for wage data analysis [LJD+18] and private auctions [BCD+09], but their adoption remains limited to a handful of sensitive scenarios due to a variety of performance, engineering, and usability challenges [HHN+19; QLJ+19].

Without additional mechanisms, these technologies are insufficient to ensure that applications respect end-user consent, *e.g.,* that they use end-user data to compute functions for purposes allowed by that user, or that users maintain control over their data throughout the computation, *e.g.,* by deleting their data.

Statistical Privacy. This approach focuses on ensuring that the aggregates computed on some input data set do not leak information about individuals in that data set. This includes differential privacy (DP) [DMN+06], which provides protections by adding noise sampled from a carefully selected distribution to the aggregates prior to their release. DP is seeing increased adoption in practice, including in the US Census [Abo18], contact tracking, typing auto-complete on smart phones, and telemetry data analysis, among others [Des24]. However, DP remains difficult to configure correctly *e.g.,* in the presence of floating point errors [Mir12], and its guarantees are difficult to interpret in practice. Furthermore, applying DP to real-world analytics is complex, especially for novices, due to the lack of easy-to-use DP tools and systems [NSN+24].

1.1.2 Systems for Privacy

Each of the above three privacy frameworks faces unique technical challenges to adoption in practice. However, at a high level, these challenges share a common thread. Namely, it is technically challenging for developers to apply them to their applications in a correct, performant, and non-intrusive way *i.e.,* without significant application redesign or engineering effort. This reflects the lack of easy-to-use privacy abstractions in familiar systems that developers rely on when building their applications, *e.g.,* in the underlying database management systems, compilers, language runtimes, web frameworks, and so on.

In fact, existing survey work identifies the lack of tools and abstractions for privacy at the system level as a particular pain point in each of the above three frameworks [HHN+19; NSN+24; SBW+20]. Academic research systems that aim to alleviate some of this pain suffer significant drawbacks that hinder their adoption in practice. Some come with large runtime overheads [SBW+19; YWZ+09], often because they aim to protect against powerful adversaries [HZX+16]. Others require significant effort from application developer *e.g.,* by mandating that they use unfamiliar programming languages with complex type systems [LKB+21], propagate and reason about complex security labels [ZBK+06], or transitively port or re-engineer all of their application's libraries and dependencies [LTB+24]. Many are incompatible with the existing web ecosystem and its practices *e.g.,* they rely on decentralization [MSH+16] or separate user data into isolated universes [WKM19] or dedicated virtual machines [KSB+19]. We discuss such related work in greater depth in §2.2.1.

This presents us with new opportunities and challenges. Application developers have new incentives to meet privacy requirements that stem from privacy laws and increased public attention. But they need the necessary toolkit to achieve this. They need a common vocabulary to describe their desired requirements, design principles and practical tooling to reason about whether their code meets these requirements, and privacy support from the underlying systems they commonly use. Thus, this dissertation argues that we can help developers correctly meet their privacy requirements and consequently improve privacy for end-users by building and baking such privacy abstractions into familiar systems that are compatible with the current web ecosystem and that require low development and runtime costs.

1.2 Contributions

This dissertation describes three primary contributions:

1. The first contribution is the design and evaluation of K9db, a new privacy-compliant database that supports GDPR-style subject access requests and other *storage* requirements. K9db reorganizes storage around data ownership as a first-order principle and provides new compliance-related abstractions that mirror traditional concepts in SQL databases familiar to most developers, *e.g.,* ownership annotations and compliance transactions that correspond to SQL foreign keys and SQL transactions, respectively. We demonstrate that K9db's organization and abstractions allow it to correctly handle user access and deletion requests, while ensuring that regular application queries exhibit performance and ACID guarantees comparable to traditional SQL databases.

2. The second contribution is Sesame, a new system for end-to-end enforcement of privacy policies in web applications. Sesame focuses on policies that govern the application logic and its *processing* of user data. Sesame relies on recent advances in memory-safe programming languages to get automatic, low-friction guarantees for the vast majority of application code, and to distill enforcement of privacy policies down to small, isolated, and infrequent privacy regions. Sesame reasons about these regions by combining new a static analysis for data leakage with recent advances in lightweight sandboxing, and with engineering best practices around code review.

3. The final contribution is a case study that describes how application developers use K9db and Sesame in practice to obtain compliance assurances. The case study is based on WebSubmit, an in-house homework submission application that we use in various courses at Brown. This case study compares the experience of application developers using K9db and Sesame to the status quo, where developers must manually implement any functionality required for compliance and explicitly enforce the needed privacy policies throughout their application code.

I have led and contributed to the design and implementation of all the systems, abstractions, and algorithms in this dissertation. However, colleagues and mentees in the ETOS research group have at times assisted me in some of the implementation and evaluation, or contributed specific components.

In K9db, Ishan Sharma implemented parts of the dataflow engine (§3.5.2) under my supervision for his Masters thesis, specifically for achieving read-your-writes consistency. Benjamin Kilimnik and Aaron Jeyaraj investigated the database schema and data ownership semantics for Shuup (§3.7.3). Justus Adam investigated variable ownership in ownCloud and implemented its experiment harness (§3.7.1) for his Fall 2021 CS2390 course project. Finally, a much earlier initial prototype of K9db that I built on top of SQLite was inspired by a high-level design proposed by Malte Schwarzkopf et al. [SKK+19].

In Sesame, Artem Agvanian and I sketched the high-level algorithmic design for Sesame's static analysis tool, SCRUTINIZER (§4.5.1), with Artem contributing its implementation in its entirety. Allen Aby and I implemented the FFI tooling required to sandbox Rust functions with RLBox (§4.5.2), which Alexander Portland and I later optimized by implementing "pointer swizzling" and enabling safe reuse of sandbox instances across invocations. Corinn Tiffany implemented the code signature and verification mechanism for Sesame's critical regions (§4.5.3). Finally, I worked with Alex Portland and Sarah Ridley on porting Portfolio and Voltron to Sesame, respectively (§4.7).

1.3 Dissertation Outline

§2 provides background information on privacy laws and discusses their main requirements, including the technical challenges that application developers face when trying to comply with them using current practices. It also discusses related systems that aim to simplify compliance or provide related privacy guarantees for web applications.

§3 describes the design of K9db, a new database that complies with GDPR subject access requests. This section describes K9db's schema annotations, which developers use to specify their application's data ownership semantics, the data ownership graph (DOG), and K9db's data ownership-centered storage organization. It also evaluates the performance and memory overhead of K9db and the schema annotation effort that it requires using several real-world applications.

§4 describes the design of Sesame, a new system for end-to-end enforcement of privacy policies in web applications. This section describes Sesame's policy API, policy containers, and privacy regions, which are based on static analysis, sandboxing, and code review and signing. It discusses the process and effort required from application developers to port or implement their applications with Sesame and evaluates its performance by looking at loads from two web applications: WebSubmit and Portfolio.

§5 describes the end-to-end experience of ensuring that an application is GDPR-compliant using K9db and Sesame from the perspective of the developers of that application. This section revolves around a case study based on our experience in porting and deploying WebSubmit with K9db and Sesame. It also compares this experience with current practice, in which developers must manually implement data access and deletion functionality and need to manually ensure that their application logic satisfies their various privacy policies *e.g.,* through explicit checks in their code.

§6 discusses how the design of K9db and Sesame can be adapted to other familiar systems, including non-SQL databases and other web frameworks and programming languages. It also outlines three ongoing extensions of K9db and Sesame, including SesameBun, which aims to simplify how applications can jointly use K9db and Sesame at the same time, and Tahini, which aims to provide end-to-end enforcement guarantees in the presence of remote- and microservices.

§7 highlights our overall results and concludes this dissertation.

1.4 Related Publications

Parts of the work described in this dissertation was covered in two of my peer-reviewed publications:

[DAA+24a] *Kinan Dak Albab*, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2024)*. Austin, Texas, USA, November 2024.

[DSA+23] *Kinan Dak Albab*, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, Malte Schwarzkopf. "K9db: Privacy-Compliant Storage For Web Applications By Construction". In: *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*. Boston, MA, USA, July 2023.

In addition, I also authored or coauthored other peer-reviewed publications, some of which are related to the themes discussed here, but did not directly contribute to this dissertation's contents:

[DDH+22] *Kinan Dak Albab*, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, Minlan Yu. "SwitchV: automated SDN switch validation with P4 models". In: *Proceedings of the 36th ACM Special Interest Group on Data Communication Conference (SIGCOMM 2022)*. Amsterdam, Netherlands, August 2022.

[DIV+22] *Kinan Dak Albab*, Rawane Issa, Mayank Varia, Kalman Graffi. "Batched differentially private information retrieval". In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security 2022)*. Boston, MA, USA, August 2022.

[LDI+19] Andrei Lapets, *Kinan Dak Albab*, Rawane Issa, Lucy Qin, Mayank Varia, Azer Bestavros, Frederick Jansen. "Role-based ecosystem for the design, development, and deployment of secure multi-party data analytics applications". In: *Proceedings of the 4th IEEE Secure Development Conference (SecDev 2019)*. McLean, VA, USA, September 2019.

[DIL+19] *Kinan Dak Albab*, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, Ira Globus-Harris. "Tutorial: Deploying secure multi-party computation on the web using JIFF". In: *Proceedings of the 4th IEEE Secure Development Conference (SecDev 2019)*. McLean, VA, USA, September 2019.

[QLJ+19] Lucy Qin, Andrei Lapets, Frederick Jansen, Peter Flockhart, *Kinan Dak Albab*, Ira Globus-Harrirs, Shannon Roberts, Mayank Varia. "From usability to secure computing and back again". In: *Proceedings of the 15th USENIX Symposium on Usable Privacy and Security (SOUPS 2019)*. Santa Clara, CA, USA, August 2019.

[LJD+18] Andrei Lapets, Frederick Jansen, *Kinan Dak Albab*, Rawane Issa, Lucy Qin, Mayank Varia, Azer Bestavros. "Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities". In: *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies (COMPASS 2018)*. San Francisco, CA, USA, February 2018.

[JFD+18] Mohamad Jaber, Yliès Falcone, *Kinan Dak Albab*, John Abou-Jaoudeh, Mostafa El-Katerji. "A high-level modeling language for the efficient design, implementation, and testing of Android applications". In: *The International Journal on Software Tools for Technology Transfer 20.1 (February 2018)*.

[ADS17] Paul Attie, *Kinan Dak Albab*, Mouhammad Sakr. "Model and Program Repair via SAT Solving". In: *ACM Transactions on Embedded Computing Systems 17.2 (December 2017)*.

[JDL+17] Frederick Jansen, *Kinan Dak Albab*, Andrei Lapets, Mayank Varia. "Brief Announcement: Federated Code Auditing and Delivery for MPC". In: *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2017)*. Boston, MA, USA, November 2017.

[DIL+17] *Kinan Dak Albab*, Rawane Issa, Andrei Lapets, Azer Bestavros, Nikolaj Volgushev. "Scalable secure multi-party network vulnerability analysis via symbolic optimization". In: *Proceedings of the 38th IEEE Symposium on Security and Privacy Workshops (SPW 2017)*. San Jose, CA, USA, May 2017.

Although these publications did not directly contribute to this dissertation, some have influenced some of its themes indirectly. JIFF [DIL+19] and its data types that encapsulate secret shares and their oblivious operations inspired policy containers in Sesame. Furthermore, my interest in systems for consent and control (the focus of this dissertation) began as an attempt to augment end-users' loss of control over their data in outsourced secure computation. For example, when my earlier work altered the computation circuit after users submitted their data to perform error correction [LJD+18; QLJ+19].

CHAPTER 2

Background

2.1 Privacy Regulations

Web services must comply with new privacy and data protection laws, including the GDPR [GDPR16] in the EU, California's CCPA [CCPA18], and similar laws in Brazil [Bra19], India [Ind19], and other countries [Gaz19; Tho19]. These laws share many similarities, and while the discussion below focuses on GDPR for exposition, it is similarly applicable to other laws. Furthermore, many of these laws have a comprehensive scope: *e.g.,* the EU's GDPR applies to anyone who offers services to users physically in the EU and touches many aspects of web services [SWC19]. Thus, they broadly apply to applications, even those based in other jurisdictions.

This dissertation focuses primarily on systems and technical infrastructure to facilitate compliance with aspects[1] of these laws around respecting end-user consent and ensuring that they have control over their data. We omit discussion of other stipulations that are less technical in nature or orthogonal to web services' technical infrastructure, such as transparency and reporting requirements with respect to data breaches.

Access, Deletion, and Subject Access Requests (SARS). Most laws grant users control rights over their data. The GDPR, for example, requires services to honor *Subject Access Requests* (SARs). These requests allow a "*data subject*" (*i.e.,* an end user) to request a copy of their data (Right to Access, Article 15), to request the deletion of their data (Right to Erasure, Article 17), and to receive the data in a portable and machine-readable format (Right to Data Portability, Article 20). As the GDPR has become a model for

[1] This dissertation discusses the requirements of privacy laws in general and is not meant to be specific legal advice for any specific applications.

other privacy laws, many have adopted similar SAR-like requirements. The California Consumer Privacy Act (CCPA), for example, gives consumers the right to request "specific pieces of personal information [a business] has collected about the consumer" (§1789.110) and its deletion (§1789.105).

Complying with SARs requires the service provider ("data controller" in GDPR terms) to identify the information related to a data subject. If done naively, this can require significant additional metadata and impact application performance [SBW+20], in addition to being error-prone. Failure to comply correctly with SARs is a frequent source of fines. For example, a US AI company was fined €20 million in part for deactivating, rather than deleting, user accounts after requesting deletions [NOY22a], and a catering company was fined for not responding to access requests [NOY20d], among many others [NOY20a; NOY20b; NOY21; NOY22b].

Data Security. The GDPR and other laws also impose mandates for secure data handling, particularly encryption at rest (GDPR's Articles 25 and 32, and CCPA's §1798.150(a)(1)). These mandates avoid prescribing particular technologies: *e.g.,* the GDPR only requires that organizations take "appropriate technical measures" to secure personal data (Article 32), giving freedom to meet the requirement in different ways. In practice, encrypting data at rest with standard encryption schemes and deleting the encryption keys (referred to as "crypto-shredding"), *e.g.,* to make backups inaccessible is widely considered a compliant approach [Rob19], even though it suffers from potential risks against future adversaries that may be capable of breaking the encrypted backup (*e.g.,* quantum adversaries).

Furthermore, this requires applications to ensure that user data is only accessible to authorized users *e.g.,* employing proper user authentication and access control (*data confidentiality*, Article 5(f)). Violations of these requirements are a common source of fines for organizations with large and small applications alike. For example, two doctors were fined in France for not encrypting patient data and failing to protect it with proper authentication [Lou20].

Stricter requirements may apply to certain types of data or data subjects. For example, medical and education data is subject to additional protections from HIPAA [HIPAA96] and FERPA [FERPA74] in the US respectively, and the GDPR poses additional requirements on handling children's data (Article 8, Recital 38). This includes implementing age-appropriate default privacy settings, *e.g.,*, so that children's profiles are private and their emails and phone numbers are inaccessible publicly, something Instagram was fined €405 million for violating due to an application bug [Eur22].

Purpose Limitation and Restriction of Processing. The GDPR requires that services collect data for known and specific purposes, and that all present and future processing of that data be inline with those initial purposes (Article 5(b)). The CCPA also poses a similar requirement (§1798.100(b)). The GDPR also restricts data collection (*data minimization*, Article 5(c)) and storage (*storage limitation*, Article 5(e)) to what is necessary to carry out the desired purpose. Services often declare the purposes for which they collect and use user data in their privacy policies or terms of services. These are human-language documents that users of a service agree to, and applications must abide by the informal requirements set forth in these documents.

Currently, application developers must continuously reason about these informal requirements, including initial purposes, as they maintain their applications or add new features to it. This is an ad hoc process that may result in mistakes, especially in large applications with large development teams. Enforcement agencies actively pursue and fine services for violations of purpose limitation and their terms of service. For example, the FTC fined Twitter $150 million [Com] for using user phone numbers for targeted advertising, when it had been collecting them for the stated purpose of increasing account security with 2FA[2].

Finally, the GDPR allows data subjects to object to the processing of their data for specific purposes, such as using it for marketing or sharing with third parties (Article 21). The CCPA similarly allows users to opt out of the sale or sharing of their personal data (§1798.120). This makes ad hoc compliance even more error-prone, as different users may place different restrictions on how their data is processed.

Consent and Lawfulness of Processing. The GDPR sets several conditions that define when the collection and processing of user data is lawful (Article 6). A common basis for lawfulness is explicit consent provided by the data subject to whom the data belong. Alternatively, processing may be lawful if it is required to comply with other laws, to carry out contractual obligations, or for public or legitimate interest.

Crucially, user consent must be informed (Article 7). It is given relative to the stated purposes for which data is collected and the conditions under which it is processed. Thus, when data is used in an inconsistent manner, whether due to human error, negligence, or otherwise, consent becomes invalid.

[2]It is not clear whether this was a case of intentionally misleading users about the actual purposes their data is used for, or an omission due to carelessness or genuine human error in reusing data after it was collected for two-factor authentication.

When data is collected for multiple purposes, consent must be given for all of them, and users have the right to withdraw their consent freely at a later time.

Thus, ensuring consent is properly collected poses subtle technical challenges. Since the validity of the consent hinges on whether the informal terms described to users match how their data is used, or alternatively, whether application logic and data processing, including new features, are consistently checked against the informal specifications and privacy policies.

2.2 Privacy-Conscious Systems

Researchers built various systems that aim to improve end user privacy or to simplify upholding privacy requirements in web applications. We refer to these as *Privacy-Conscious Systems* and distinguish between two categories. The first consists of systems that can plausibly assist applications and their developers in complying with privacy laws. This may be either the primary goal of these systems, or a consequence enabled by their design, or implied by their guarantees. A common theme in this line of work is targeting a relaxed, but meaningful, threat model, where laws and their hefty fines are assumed to deter malicious behavior, and thus application developers have a strong incentive to comply with them correctly. These systems are closely related to the work presented in this dissertation. The second category consists of systems that aim to provide privacy guarantees that differ distinctly from those required by laws. These systems may offer orthogonal or complementary guarantees and privacy notions, aim to protect against stronger threat models, and often propose significant changes to the architecture and business model of the modern web ecosystem. Although less directly related to the work presented here, we discuss these systems for completeness and because they offer insight into alternative and sometimes complementary technical perspectives on what privacy may look like.

2.2.1 Privacy-Conscious Systems Related to Compliance

Storage Systems and SARs. Today, compliance with SARs requires application developers to write custom queries and maintain metadata to identify and track information related to each data subject [SBW+20]. The queries are tricky to get right and maintain as the application evolves. Thus, automatic and correct support for handling SARs is desirable, both to reduce the burden of compliance on companies as well as to improve data subjects' control over their data in practice. A natural place to add such built-in support is at the database level, since access and deletion are storage-level abstractions.

Adaptations of existing database systems can go some way towards complying with SARs, but can come at a steep performance cost. For example, Shastri et al. found that secondary indexes and strict metadata tracking impose overheads of up to 5× [SBW+20], leading to proposals to accelerate these operations in hardware [IPC21]. SchengenDB [KSB+19] outlines a design that provides GDPR compliance, but relies on extensive metadata and conservative, coarse-grained enforcement, *e.g.,* destroying entire virtual machine clusters when a data subject deletes their account.

To address part of this burden, some large companies built bespoke GDPR metadata stores [CKK+20, §1] and dedicated frameworks for data deletion, *e.g.,* Facebook's DELF [CDN+20]. DELF's design closely tracks Facebook's TAO graph database, where data objects are represented as rows whose relationships consist of edges between them. Application developers provide deletion specifications by annotating the nodes and edges that schematically define the database. Whenever an application instructs DELF to delete some object, DELF cascades that deletion to all related objects transitively, as specified by the annotations. Thus, if DELF is configured correctly, deleting the top-level object representing a data subject should result in deleting all of that subject's data as defined by the deletion specification.

However, DELF only solves part of the problem. It does not handle access requests, or help developers comply with other storage-related restrictions, such as encryption at rest. Furthermore, it provides identical deletion semantics for a deletion SAR request and regular application deletions, *e.g.,* the same cascading behavior for replies to a social media post when it is deleted as part of handling a SAR or handling a regular application point deletion of that post. This may be undesirable for applications where these semantics differ. Finally, DELF requires some application changes, as applications must use it explicitly for deletion. DELF's design and features are closely tied to Facebook's infrastructure, and most organizations lack the resources to build or deploy such a system themselves.

Access Control. Classic role-based access control (RBAC) [San98] enforces access control in databases. Several commercial and widely used databases have built in support for RBAC, such as MySQL. However, RBAC only expresses access control policies based on pre-defined roles and often leads to role explosion. Several extensions aim to address these limitations by supporting richer policies, such as attribute-based access control (ABAC) [HKF+15] and relation-based access control [Gat07].

RBAC is widely used in online services, although it is often used to manage the data and resources developers within an organization can access (*e.g.,* IAM in Amazon AWS [Ser25]), rather than enforce

access control for end users. Instead, when web applications write or read data from the database within some application flow (*e.g.,* in response to end-user actions), they tend to do so with ambient authority while enforcing authentication and access control in the application code itself.

Another approach for managing access is through object capabilities. This is a powerful model that prevents *confused deputy attacks*, *i.e.,* cases where the application, acting on behalf of some user, attempts to access data to which that user does not have access to *e.g.,* due to a bug in the application. Several operating systems, such as KeyKOS [BHP+92] and EROS [SSF99], use capabilities to ensure proper access control of system resources. Existing work that applies object capabilities to web applications includes Google Caja [MMT10], which safely integrates untrusted third-party libraries and code in web pages by using object capabilities to express and enforce security policies, and recent proposals to use object capabilities more systematically to improve user authentication in web applications [KKP+22]. In addition, primitive capabilities are a popular method for implementing authentication in web applications. For example, after a user successfully logs in, applications commonly authenticate that user by storing a cookie containing an API key or an OAuth token [Har12]. However, this still requires application developers to maintain an accurate account of what these capabilities or tokens allow users to do *e.g.,* by maintaining access control lists (ACL) and to correctly check them when handling user requests, similar to how UNIX operating systems maintain a C-list to which file descriptors refer.

Recent database-centric access and information flow control systems can express more complex access policies. Blockaid [ZSC+22] enforces access control and non-interference for database queries. It only allows querying information that a user is allowed to access, by checking that the information the queries output can be deduced from some given policy specified as a set of views. Daisy [GBS+19] supports row and column-based policies and enforces end-to-end information flow control (IFC) policies for database-backed applications, but does not support reasoning about aggregations and data linkage. Qapla [MEH+17] can reason about aggregation and linkage, but cannot provide end-to-end IFC guarantees, and comes at a steep overhead in end-to-end application performance (up to $6\times$).

These approaches cannot protect against application bugs, *e.g.,* leaking data via network calls or file I/O, and do not support policies beyond access control, even when these policies are required for compliance, such as for purpose limitation or restriction of processing.

Information Flow Control (IFC) and Privacy Enforcement. IFC enforces end-to-end security policies

in programs. IFC systems may rely on enforcement via runtime labels [KYB+07; YWZ+09; ZBK+06; ZBM08], compile-time enforcement via type systems [LKB+21; SCH08] and static analysis [DD77], or hybrid approaches [BVR15; CVM07; ML00; RPB+09]. Classical IFC systems rely on a security lattice made out of security labels and ensure that information flows only upward in that lattice. While sufficient to describe certain kinds of access control, this paradigm is not expressive enough to capture complex and dynamic privacy policies, such as around purpose limitation and consent. More recent systems allow expressing such dynamic privacy policies *e.g.,* by allowing developers to implement policies using arbitrary Python code [YWZ+09].

A common failure point of IFC systems is that they are challenging for developers to use [EK08]: some require the use of custom languages unfamiliar to web developers [BVR15; LKB+21; ML00; PYI+16], and others require developers to propagate complex security labels [ZBK+06; ZBM08]. They also frequently require significant changes to the application code. Static approaches can only express limited policies that developers must encode [LTB+24; SBR+11; SCF+11], and dynamic approaches often come with steep performance costs [YHA+16; YWZ+09]. Some systems, such as Riverbed [WKM19] avoid some usability and performance pitfalls but impose limitations on application functionality, such as separating user data into separate universes.

Compile-time enforcement approaches to IFC statically guarantee policy compliance. Application developers define their policies in a single, reviewable location and associate them with data. The compiler then checks that the program's execution cannot violate policies. Policies can range from classic IFC noninterference (*e.g.,* enforced through Rust types in Cocoon [LTB+24]) to more complex policies over SQL databases and their schemas (*e.g.,* in Ur/Flow [Chl10a], or via refinement types in Storm [LKB+21]). The latter approaches can express data-dependent policies, but only if they are exclusively defined via relationships over the database schema. They also require implementing applications in niche or specialized languages, such as Ur/Web [Chl10b] or LiquidHaskell [LKB+21], which limits adoption.

Dynamic policy tracking systems taint data with policies and maintain these taints with a modified language runtime. The runtime maintains a policy taint for every variable, propagating and combining the policy taints as the program runs. Developers write policies declaratively over a data model [YHA+16] or attach them dynamically to data [YWZ+09], and the runtime checks the associated policy before application sinks externalize tainted data. Runtime tracking imposes high performance overheads (*e.g.,*

33% [YWZ+09] to 75% [YHA+16]) and requires the use of interpreted languages, but minimizes developer effort.

Privacy Enforcement For Privacy Laws. Some recent systems specifically target enforcing policies mandated by privacy laws, such as purpose limitation. For example, RuleKeeper [FBS+23] combines static analysis of JavaScript code with runtime policy enforcement, but its static analysis is not sound and can miss policy violations. RuleKeeper protects against violations at limited components, *e.g.,* HTTP endpoints and database queries, but not against accidental leaks or custom sinks (*e.g.,* logging, file I/O).

Paralegal [AZZ+25] is a static analyzer for finding privacy-related bugs in web applications. Application developers use Paralegal by annotating code elements with *markers*, while policy engineers state a high-level policy specification that refers to these markers. Paralegal then statically checks the application code with respect to the specification by constructing and analyzing a corresponding program dependence graph (PDG) and reports any violations it finds back to the developer. Paralegal does not introduce any runtime overhead, as it relies on a purely static approach. It also does not require application modification. However, whether Paralegal is sound or complete depends on the policy being checked. Furthermore, Paralegal's policies are abstract and depend on the informal semantics attached to code markers. For example, Paralegal can check that user data flows into a deletion-marked routine, but it cannot check whether that routine is correct (or automatically synthesize a correct routine). It can also check for static, but not dynamic, access control and purpose limitation policies, *e.g.,* that a data type never flows to some undesired purpose, but not conditionally based on dynamic user consent or only after aggregation with data from sufficiently many other users.

PrivGuard [WKN+22] uses static analysis and runtime mechanisms to enforce privacy policies. PrivGuard aims to reduce human participation in auditing and reasoning about compliance of programs, but targets a narrow set of Pandas-like data analytics programs. PrivGuard ensures that programs submitted by analysts (*e.g.,* medical researchers), and executed by data curators (*e.g.,* hospitals with patient data), meet baseline privacy policies mandated by privacy regulations.

This Dissertation. In this dissertation, we present two systems: K9db §3 and Sesame §4.

In contrast to existing work (*e.g.,* DELF), K9db helps developers comply with both deletion and access requests. It also meets other storage requirements, such as encryption at rest, while providing

developers with the familiar interface of an SQL database. K9db redesigns the database to make correct privacy compliance a first-class property, without sacrificing performance and with moderate overhead.

Sesame aims to ensure that applications meet their desired custom privacy policies while reducing developer effort, application changes, and performance overhead. Sesame targets end-to-end enforcement of flexible, data-dependent policies for applications written in a widely used mainstream programming language (Rust). This means that Sesame can express richer policies than static enforcement systems, including policies that rely on dynamic information about the data or application, and makes it easy to adopt. Sesame achieves this without a custom taint-tracking runtime, in the presence of third-party libraries, with limited extra burden for developers, and at low overhead.

Furthermore, we show how applications can use K9db and Sesame in tandem to meet storage and processing requirements (§5). Both systems are guaranteed to be sound with respect to the policies specified by application developers. Together, K9db and Sesame provide strong end-to-end enforcement. For example, they protect against accidental leaks in custom sinks, unlike RuleKeeper. Furthermore, they provide more concrete guarantees than Paralegal, such as correct deletion of user data, including dynamic policies, such as those governing user consent or data aggregation.

2.2.2 Privacy-Conscious Systems Beyond Compliance

Privacy Enforcement in Untrusted Applications. Unlike earlier work described in the previous section, which aims to help application developers comply with desired privacy policies, this style of work guarantees to end-users that an application meets some privacy policies even when the application is untrusted and in some cases outright malicious.

Ryoan [HZX+16] has a strong threat model: it trusts neither the application nor the underlying cloud platform and assumes that both may be actively malicious and even colluding. Ryoan leverages sandboxing and trusted hardware enclaves to protect sensitive data. Furthermore, it assigns IFC-like security labels to data when it flows between remote services, in order to extend its guarantees to untrusted distributed applications. However, these strong guarantees come at significant runtime costs that render Ryoan impractical for most applications, and this adversarial threat model goes well beyond the requirements set forth by privacy laws.

Zeph [BKV+21] relies on cryptography to restrict the computations that a service can perform on

sensitive data from end users and data providers. Users submit their data along with their desired privacy policies, and Zeph ensures that any transformations applied to it are consistent with and allowed by the input privacy policies before releasing the output, while ensuring that data remain end-to-end encrypted. Zeph protects against an honest but suspicious service, *i.e.,*, a service that performs the computation correctly but monitors all communications and intermediaries to learn any information it can about the data. Because its threat model is weaker than Ryoan, Zeph exhibits better performance. However, it only supports restricted classes of numeric computations (*e.g.,* streaming sums).

Cryptographic Secure Computation. Cryptographic primitives such as fully homomorphic encryption (FHE) [Gen09] and secure multiparty computation (MPC) [BGW88; Sha79; Yao86] allow mutually distrusting parties to compute desired functions over sensitive private user data. Various existing systems provide implementations of these techniques for general-purpose computation [ACC+21; BLW08; DIL+19; LWN+15]. They have been used to outsource the computation of various statistics to untrusted parties [BCD+09; LJD+18]. However, a larger adoption of these systems faces several usability and performance challenges [HHN+19].

Recent systems provide domain-specific implementations of these techniques with better performance and usability. This includes systems for privacy-preserving SQL queries [VSG+19], time series analysis [FZL+23], private information retrieval [DIV+22; KC21], private Internet search [ASA+21; HDC+23], data analysis [MNL+23; PKY+21; RZH+20]. However, the performance of these systems remains much worse than their non-private and non-cryptographic counter parts, and while they offer strong data security guarantees in the presence of untrusted, or even malicious, parties, they are not comparable to the requirements of privacy laws. Laws rely on legal enforcement and fines, rather than formal cryptography, to deter malicious behavior, but aim to provide ensure user consent is respected and that users have control over their data and its uses. This is orthogonal to the cryptographic guarantees of MPC when data providers do not participate in the computation, as is often the case for performance and robustness.

Differential Privacy. Differential privacy (DP) [DMN+06] ensures that aggregates do not reveal too much information about individuals in their input sets, by adding noise from a carefully configured distribution prior to releasing the aggregate. Continuously computing and releasing aggregates over the same or overlapping datasets increases the leakage proportionally. Often, analysts must track this

total privacy loss to ensure that it remains below a desired privacy budget attached to the data. DP has seen increasing adoption in practice, for example in the US Census [Abo18]. DP is complementary to the goals and mechanisms of many of the systems described above. Recent systems help applications implement DP correctly by automatically tracking and checking privacy budgets *e.g.,* in streaming applications [LPT+21] and for targeted advertisement [TKM+24].

Decentralized Systems. Other proposals have advocated for a radical restructuring of web services to enforce users' privacy rights, but face barriers to adoption. Decentralized systems decouple data storage from the web application and put data storage under user control. Solid [MSH+16] allows users to store data in pods under their control and provides a set of APIs and protocols so that web applications, running in an end-user browser, can retrieve data from that user's and related users' pods and process it. Nostr [Nos23] provides a censorship-resistant decentralized platform for micro blogging, where users store replicas of their content across many relays, some of which may be under the control of that user. Decentralization gives users control of their data, but requires rewriting web applications, comes with restrictions (*e.g.,* all application logic must run in JavaScript in the browser), and is incompatible with today's advertising-based business model for web services. Furthermore, decentralization may make compliance with certain requirements from privacy laws trivial for some applications *e.g.,* when every user hosts their own dedicated Solid pod over which they maintain complete control. However, it may also make compliance more challenging in other scenarios *e.g.,* when data belonging to one user is replicated across many Solid pods or Nostr relays outside of that user's control.

CHAPTER 3

K9db: Privacy-Compliant Storage For Web Applications

In this chapter, we present K9db, a new SQL database that helps application developers comply with subject access requests, which are mandated by privacy laws. K9db's key idea is to make the data ownership and sharing semantics explicit in the storage system. This requires K9db to capture and enforce applications' complex data ownership and sharing semantics, which application developers express using a small set of schema annotations. K9db infers storage organization, generates procedures for data retrieval and deletion, and reports compliance errors if an application risks violating the requirements of privacy laws. Our K9db prototype successfully expresses the data sharing semantics of real web applications, and guides developers to getting privacy compliance right. K9db also matches or exceeds the performance of existing storage systems, at the cost of a modest increase in state size.

3.1 Motivation

As discussed in §2.1, privacy laws provide users with rights to issue subject access requests (SARs), including a *right to access*, which lets users request a copy of their data, and a *right to erasure*, which requires its deletion on request.

Achieving compliance can be onerous and expensive, however, particularly for small and medium-size organizations. These organizations must write custom queries and track metadata to identify and extract data related to a user, and continuously maintain this infrastructure as services evolve. Even well-intentioned developers sometimes get it wrong: for example, the ownCloud collaboration platform [own21b], though it claims GDPR compliance [own21a], retains a user's activity log after account deletion. Retrofitting compliance onto existing systems is tricky, as it still requires manual work [AGJ+21; LCG+21] and may harm performance [SBW+20].

Compliance with SARs is difficult, both manually and in automated systems, because web services often have complex ownership and data sharing semantics. Identifying data associated with a particular user ("data subject") is challenging. In relational databases, these associations are expressed as foreign keys; but data in many tables link to data subjects transitively via one or more intermediate tables, rather than directly. Multiple data subjects can be associated with the same data (*e.g.,* private messages), and sometimes this association is asymmetric and implies different rights for different data subjects (*e.g.,* a teacher and a student). Finally, many-to-many relationships introduce dynamically changing associations between data and a variable number of data subjects.

GDPR-like laws afford companies with some flexibility in handling SARs. Applications may keep data associated with the data subject (possibly in some anonymized form) after a deletion request due to legal or contractual obligations (*e.g.,* tax laws) or public interest [GDPR16, Article 17.3]. Data may also be retained depending on the purpose of its processing, including the interests of other users (Article 6.1, Article 17.1(b)). For example, Facebook's privacy policy specifies that Facebook deletes the comments that a withdrawing data subject made, but not the private messages they sent to a friend, unless that friend also deletes them [Fac]. Thus, the compliance policy and exact handling of SARs are application and data dependent.

We explore a new system design that achieves privacy compliance *by construction*. Our key idea is to make data ownership a first-class citizen in the database system itself. K9db, our new database system, tracks sufficient information to know, for each row in the database, what user (or users) have rights to it. This allows K9db to infer correct procedures for data retrieval and deletion, so that the database itself can handle requests under the rights to access or erasure, freeing the application developer from having to write or maintain custom scripts to handle these requests. The ownership information also allows K9db to encrypt data with per-user keys, which helps meet, *e.g.,* the GDPR's "Protection by Design and Default" requirement, which can be satisfied by encrypting at-rest data [GDPR16; NA15]. Finally, K9db uses ownership information to generate errors if the database schema or operations on database contents risk violating the GDPR.

To realize K9db, we had to address three challenges. First, K9db must understand and model the complex data ownership and sharing semantics of real applications. A user's data may span many tables with transitive relationships, may be shared in complex and data-dependent ways, and may require partial

redaction when returned or removed. Second, K9db must maintain and enforce compliance invariants matching these ownership semantics throughout application execution, and correctly respond to user access and deletion requests. Third, K9db should match the performance of today's databases that lack infrastructure for data ownership tracking, and must be both compatible with existing applications and easy for application developers to adopt.

K9db's design addresses these challenges as follows. First, K9db derives a *data ownership graph (DOG)* from a set of coarse-grained, declarative annotations on the database schema. Using a small number of primitives, the DOG models a wide range of complex data sharing relationships found in real-world applications. The DOG is central to K9db's storage organization, to its handling of users' access and erasure requests, and to K9db's ability to enforce privacy compliance. Second, K9db organizes data storage around data ownership to ensure that applications remain in compliance and handle access and deletion requests correctly by construction, without disrupting regular application operations. Third, K9db is a MySQL-compatible drop-in-replacement for existing databases, and requires few application changes beyond declarative schema annotations for normalized schemas. To accelerate complex queries, K9db provides an integrated, privacy-compliant in-memory cache based on materialized views. By integrating and managing materialized views, K9db provides the benefits of caching to applications, while relieving developers from ensuring compliance of cached data.

K9db structures the actual data storage as a set of user-specific logical "micro-databases" (μDBs), realized over a single physical RocksDB [Met22] store. Each user's μDB contains the data they own, and is encrypted with a user-specific key. K9db also helps developers use the system correctly by providing compliance-specific functionality not found in other databases. A new `EXPLAIN COMPLIANCE` SQL command gives the developer insight into the DOG and highlights possible schema annotation errors; and K9db supports *compliance transactions* that guard against dynamic compliance problems, such as data without an owner being left behind in the database. K9db provides ACID guarantees similar to those in default MySQL.

K9db provides out-of-the-box compliance for well-intentioned developers who want to comply with privacy laws, and helps developers avoid mistakes. We expect that fines for privacy violations (*e.g.,* the greater than 4% of annual turnover or €25M for GDPR violations) discourage intentional misuse.

In summary, we make the following contributions:

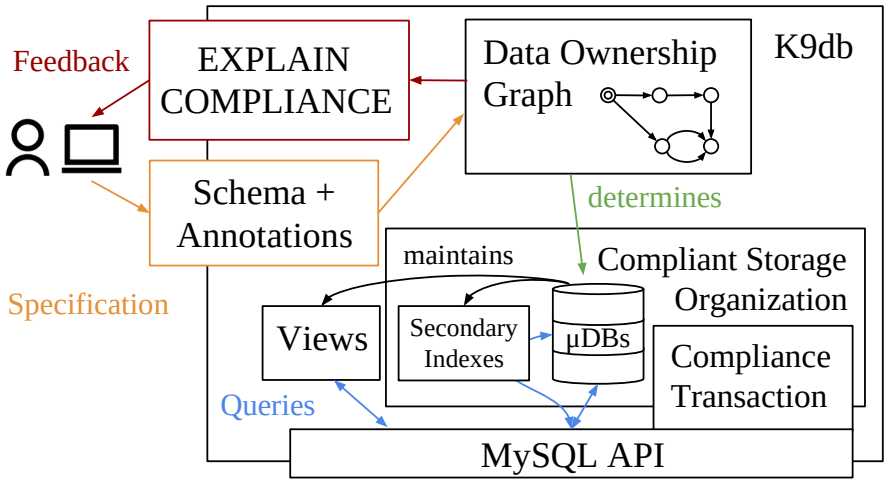


Figure 3.1: K9db provides privacy-compliant storage based on its data ownership graph, micro-databases (µDBs), and compliance helper mechanisms behind a MySQL interface.

1. The data ownership graph (DOG) for modeling ownership in a database, specified with schema annotations.
2. K9db, a new database that enforces compliance-by-construction based on the DOG and a compliant, ownership-aware storage organization.
3. Mechanisms that, based on the DOG, warn developers if schema annotations are insufficient or if the database becomes non-compliant at runtime.
4. An evaluation of K9db, demonstrating that a database centered around first-class data ownership and compliance-by-construction is practical.

We evaluate K9db with scenarios based on the Lobsters web application [Lob18b], the ownCloud document sharing platform [own21b], and the Shuup e-commerce platform [Shu18]. Our experiments show that K9db can express a wide variety of nuanced data sharing and ownership patterns found in these applications, and that K9db performs on-par with or better than MariaDB and the widely-used MariaDB/memcached stack when serving typical web application workloads.

3.2 K9db Overview

K9db is a relational database that makes data ownership an explicit first-class citizen. K9db targets typical web application workloads, which are dominated by reads and point lookup queries [GSB+18]. Its design goals are *(i)* to require few changes to application code, *(ii)* to capture and enforce the complex data ownership and sharing semantics of real-world applications, and *(iii)* to provide feedback that helps

developers get privacy compliance right.

Figure 3.1 shows an overview of K9db's components. K9db requires developers to extend their relational schema (*i.e.,* `CREATE TABLE` statements) with a small set of annotations that encode data ownership and sharing semantics. The annotated schema acts as an application-specific compliance policy that specifies how K9db handles SARs. From these annotations, K9db builds its key abstraction, the *data ownership graph (DOG)* (§3.3). The DOG lets K9db determine, for every row in the database, who owns it and who has rights to it. K9db uses the DOG to satisfy data subjects' SARs, to check that the database remains compliant after the application makes changes, and to warn the developer if their annotated schema and the compliance policy it encodes seem incomplete or contradictory.

Using information from the DOG, K9db organizes its storage in a user-centric way, storing each data subject's data in their own logical "micro-database" (μDB), a shard of the actual database. This design ensures that K9db enforces the developer-provided compliance policy by construction, lets K9db encrypt each data subject's data with a separate cryptographic key, and speeds up compliance-related enforcement and operations (§3.4). K9db maintains some additional secondary indices compared to a traditional SQL database, which help K9db efficiently resolve which μDBs store particular data. It also maintains materialized views that help simplify and accelerate execution of complex queries, while also providing an integrated, privacy-compliant in-memory cache (§3.5).

For normalized schemas, K9db requires little to no application code changes, except that developers may need to wrap certain operations in a compliance transaction (§3.4.5). Developers can use K9db as a drop-in replacement for MySQL.

3.3 Modeling Data Ownership and Sharing

K9db aims to provide correct-by construction compliance with privacy laws, which requires K9db to respond to SARs correctly and enforce several invariants over the data and its storage. Correct compliance has two prongs: *(i)* a compliance policy that is consistent with the privacy law in question, and *(ii)* correct enforcement of this policy when handling both regular application operations and SARs.

The compliance policy is application-specific and depends on the relationships in the underlying data. For a single application, multiple policies may achieve compliance, and laws afford developers some flexibility in choosing a policy that matches their application's semantics (§3.1).

(a) Ownership With FK.

(b) Ownership Against FK.

Figure 3.2: K9db's annotations on foreign keys (FKs; orange) indicate the direction of data ownership (black edge) between two tables. Circles are tables.

Annotation	**Example**
`DATA_SUBJECT`	`CREATE DATA_SUBJECT TABLE users (...)`
$T_A(x)$ `OWNED_BY` $T_B(y)$	`stories(author_id) OWNED_BY users(id)`
$T_A(x)$ `OWNS` $T_B(y)$	`member(gid) OWNS group(id)`
$T_A(x)$ `ACCESSED_BY` $T_B(y)$	`share(share_with) ACCESSED_BY user(id)`
$T_A(x)$ `ACCESSES` $T_B(y)$	`taggings(tag_id) ACCESSES tags(id)`
`ON DEL` $T_A(x)$ `{ANON (...) | DELETE_ROW}`	`ON DEL chat(receiver) ANON (receiver)`
`ON GET` $T_A(x)$ `{ANON (...) | DELETE_ROW}`	`ON GET review(paper_id) ANON (reviewer_id)`

Figure 3.3: K9db's table and column-level annotations. All annotations except `DATA_SUBJECT` and `ANON` imply a foreign key from column x in table T_A to column y in T_B.

In K9db, developers express their compliance policy using schema annotations, which K9db represents using the *data ownership graph (DOG)*: a directed, acyclic multigraph whose vertices represent database tables, and whose edges represent ownership relationships between rows in the tables.

3.3.1 K9db's Annotations

Developers use schema annotations on foreign keys to communicate their application's data ownership and sharing semantics to K9db. To communicate how the database represents human persons who have rights over data ("data subjects" in GDPR terms), the developer annotates one or more tables with the table-granularity `DATA_SUBJECT` annotation.

Foreign keys (FKs) relate rows in tables to each other, and often imply ownership—consider *e.g.,* a story pointing to its author. This is the simplest case: a story is owned by the row its FK value points to. K9db provides the `OWNED_BY` keyword for developers to annotate such FKs (Figure 3.2a; §3.3.3 discusses transitive cases). But foreign keys may also point in the *opposite* direction of ownership, as is the case *e.g.,* if a user table has a foreign key to their primary address. For such cases, K9db provides the `OWNS` annotation (Figure 3.2b).

In addition to ownership, an application may also have data that is owned by one data subject (who has the right to delete it when removing their account), but share it with others. For example, in the file sharing platform ownCloud [own21b], users want to share files with others, but when they remove their

account have the file be removed for everyone. K9db lets developers express this with the `ACCESSED_BY` annotation, and its dual for opposite-direction FKs, `ACCESSES`.

These annotations extend the semantics of foreign keys with compliance semantics, and while every annotation is applied to a foreign key, not every foreign key impacts ownership or needs to be annotated. For example, the foreign key connecting students in a university database with their declared majors carries no ownership information—the students do not own the majors—and should not be annotated.

K9db also provides table-level annotations that allow developers to specify that columns in a table need anonymizing in the context of SARs. This is important because a row may need redacting before returning the row as part of a right-to-access request (`ON GET`), or because a row may need to be retained in anonymized form (*e.g.,* for tax compliance) after a data subject requests deletion of their data (`ON DEL`). Each anonymization annotation is associated with an ownership or access foreign key (*i.e.,* an outgoing edge from the table in the DOG). This allows for different anonymization behavior depending on how the data subject who issued a SAR is connected to the data. For example, in the HotCRP conference review system [Koh06], if a data subject who is both a reviewer and an author makes an access request, they should receive an unredacted copy of the reviews they wrote, but redacted, anonymized reviews for the papers they authored.

Figure 3.3 shows K9db's complete set of schema annotations.

3.3.2 Expressing Developers' Compliance Policies

We demonstrate how developers annotate their schema to express their desired compliance policy using two examples extracted from real applications: stories and messages in Lobsters (Figure 3.4), and file sharing in ownCloud (Figure 3.5).

In Lobsters, developers begin by annotating the `users` table, which records the application's end-users, with `DATA_SUBJECT`. A user may post several stories, and retains sole ownership of them: these stories must be retrieved or deleted when the user issues an SAR. Developers express this by annotating the `author` FK in `stories` with `OWNED_BY`. Lobsters also has a set of tags that represent discussion topics, *e.g.,* `games` and `programming`. Users can assign tags to stories they posted, and have complete ownership of these associations. Developers express this by annotating the `story_id` column in `taggings` with `OWNED_BY`. This makes the story the owner of its taggings, transitively making the data subject who owns the story (*i.e.,* its author) the owner of the associated taggings. But the tags themselves are not related to

```
CREATE DATA_SUBJECT TABLE users (id INT PRIMARY KEY, ...);
CREATE TABLE stories (
  id INT PRIMARY KEY, title TEXT, ...
  author INT NOT NULL OWNED_BY user(id)
);
CREATE TABLE tags (id INT PRIMARY KEY, tag TEXT, ...);
CREATE TABLE taggings (
  id INT PRIMARY KEY,
  story_id INT NOT NULL OWNED_BY stories(id),
  tag_id INT NOT NULL ACCESSES tag(id)
);
CREATE TABLE messages (
  id INT PRIMARY KEY, body text, ...
  sender INT NOT NULL OWNED_BY user(id),
  receiver INT NOT NULL OWNED_BY user(id),
  ON DEL sender ANON (sender),
  ON DEL receiver ANON (receiver)
);
```

Figure 3.4: Partial schema for Lobsters. Users own the stories they authored and their associations with tags. Messages are jointly owned by both sender and receiver.

any data subject. Thus, developers annotate `tag_id` with `ACCESSES` (and not `OWNS`). As a result, a data subject receives a copy of their stories and associated tags when they request access, while disassociating tags from their stories and removing the stories themselves when requesting deletion.

Similar to private messages in Facebook [Fac], messages in Lobsters are only deleted when both sender and receiver request deletion. Thus, developers annotate both `sender` and `receiver` with `OWNED_BY` (*i.e.,* joint-ownership), along with anonymization annotations that instruct K9db to hide the identity of the associated withdrawing user in surviving messages. An alternative policy could require deleting a message as soon as one of the associated users is deleted. Developers can express this via an `ON DEL ... DELETE_ROW` annotation.

ownCloud's data subjects are users in the `user` table, who can be members of a group (in the `group` table), as defined by the `member` association table. Users own their group memberships, so the developer annotates the `uid` column of `member` with `OWNED_BY`. The group and its associated resources are jointly owned by its members (ownCloud has no notion of group admins). Hence, the developer applies the `OWNS` annotation to the `gid` foreign key from `member` to `group`.

ownCloud's `share` table contains records of users sharing files with others. This table specifies the file's owner (*i.e.,* its original creator) via the `uid_owner` column, which is a direct FK to the `user` table. The developer thus annotates this column with `OWNED_BY`. The `share_with` and `share_with_group`

```
CREATE DATA_SUBJECT TABLE user (id INT PRIMARY KEY, ...);
CREATE TABLE group (id INT PRIMARY KEY, title TEXT, ...);
CREATE TABLE member (
  id INT PRIMARY KEY,
  uid INT NOT NULL OWNED_BY user(id),
  gid INT NOT NULL OWNS group(id)
);
CREATE TABLE share (
  id INT PRIMARY KEY, ...
  uid_owner INT NOT NULL OWNED_BY user(id),
  share_with INT ACCESSED_BY user(id),
  share_with_group INT ACCESSED_BY group(id)
);
```

Figure 3.5: Partial schema for ownCloud file sharing: users own their group membership, which owns the group; files have an owner and are shared with users who have access to them.

(a) Transitive Ownership.

(b) Variable Ownership.

Figure 3.6: Tables can have transitive ownership relationships (*: zero or more steps of indirection); if an edge follows a one-to-many or many-to-many relationship, it expresses variable ownership. Double circles indicate data subject tables.

columns are also FKs that eventually lead to the `user` table, but indicate that the file is shared with (rather than owned by) these users. The developer therefore annotates them with `ACCESSED_BY`.

3.3.3 Data Ownership Graph

K9db builds the DOG from developers' annotations by inserting DOG edges in the underlying FK direction for `OWNED_BY` and `ACCESSED_BY`, and against the FK direction for `OWNS` and `ACCESSES`. Thus, DOG edges always point towards a data subject table, unlike foreign keys.

When tables have a chain of annotated foreign keys, K9db adds an edge to the DOG that establishes a *transitive* ownership relationship (Figure 3.6a). For example, in Lobsters (Figure 3.7), a story's taggings have no direct references to the story's author. Instead, they refer to their story ②, which in turn refers to the author ①. Therefore, edges in the DOG always represent a single step towards a data subject.

The DOG is a multi-graph because two tables can have multiple foreign keys between them. For example, in Lobsters the `messages` table has two foreign keys, one to the sender ④ and one to the receiver of a message ⑤. Since sender and receiver jointly own a private message—*i.e.,* the message only disappears if both users delete their account—there are two annotated edges between `messages` and

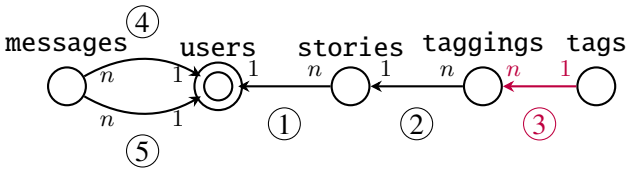


Figure 3.7: The DOG for stories and messages in Lobsters. Red indicates access-typed edges; 1 and n are cardinalities.

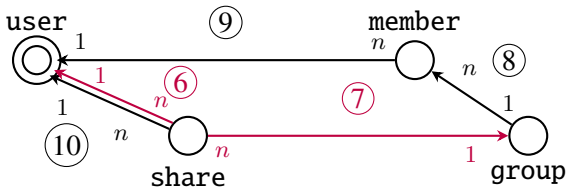


Figure 3.8: The DOG for ownCloud file sharing. Red edges are access-typed. Note the variable ownership (Fig. 3.6b) between `member` and `group`, as `member` rows are the group's owners.

`users`.

Access annotations on foreign keys also add edges to the DOG, but these edges are access-typed and distinct from owner-typed edges. For example, in ownCloud (Figure 3.8) a file is accessible but not owned by users it is shared with, either directly ⑥ or via a group ⑦. Differentiating ownership and access edges is important for K9db to correctly handle access and deletion requests.

If the destination of a DOG edge can contain multiple rows corresponding to a single row in the source table, then that row can have multiple owners or accessors. The DOG edge ⑧ from ownCloud's `group` to `member` is a one-to-many relationship, so a group may have many owners. This is an example of *variable ownership* (Figure 3.6b), as the number of owners varies depending on the data (*i.e.,* depending on the rows in `member`). Similarly, DOG edges may also express *variable access*, *e.g.,* a single tag in Lobsters may be accessed through many stories ③. This contrasts with the typical situation where the destination of a DOG edge is a primary key or unique column, making it a one-to-one or many-to-one relationship, both specifying a single owner (*e.g.,* ⑨ and ⑩). K9db's DOG metadata stores arity of relationships and K9db handles variable ownership and access appropriately.

3.3.4 Helping Developers Get Annotations Right

`EXPLAIN COMPLIANCE` gives the developer information about the DOG, including heuristic warnings and suggestions about how it may be improved. K9db runs a simple heuristic over the schema to discover column names which indicate user data such as variations on "name", "email" and "password". If a table

with such column names is not connected to a data subject in the DOG, K9db suggests to make it owned. This heuristic is most useful to discover missing data subjects, as their tables often contain columns with such names.

`EXPLAIN COMPLIANCE` also reports information that K9db derives from the DOG. For every table, it reports which data subject tables own it, and the paths through the DOG by which they own the table. This essentially shows the developer the closure over the DOG that K9db uses to handle SARs. `EXPLAIN COMPLIANCE` warns developers if a table is owned by many data subjects, *e.g.,* if a DOG path contains multiple variable ownership edges, which can result in multiplicatively many owners. Such liberal sharing is rare in practice and likely the result of a schema or annotation mistake.

3.3.5 Data Ownership Graph Properties

The DOG is *well-formed* if any path through it terminates at a data subject table. K9db rejects any schema that results in a DOG that is not well-formed.

Although the DOG is a graph of tables, its edges represent relations between rows in the source and destination tables based on the values of the underlying FK columns. Each DOG edge maps to a *relation* between rows in the two tables, where matching rows in the destination table own (or access) the rows in the source table. Intuitively, this relation can be evaluated as a query over the destination table, which yields exactly the owning row (or rows, in the case of variable ownership). Well-formedness guarantees that the transitive closure of these relations terminates at data subject tables.

Several key properties follow from this. First, if no matching rows exist in any destination table when evaluating the relations along *all* of the table's outgoing ownership edges, data is orphaned (*i.e.,* has no owner). This gives rise to the necessary (but insufficient[1]) *no orphaned data* compliance condition: any row in a database table connected to the DOG must resolve to ≥ 1 owning data subjects. Second, the transitive closure of relations corresponding to ownership edges in the DOG, starting from any row, identifies the set of data subjects that own this row. Third, the DOG's reverse transitive closure starting from a row in a data subject table yields:

1. the rows shared with and owned by that data subject, if considering accessor-typed and owner-typed edges; or
2. the rows owned by that data subject, if considering only owner-typed edges.

[1] Sufficiency would require the *correct* owners, not just any owner.

The former set corresponds to the data that needs returning from a right-to-access request, and the latter identifies the data that needs deleting for a right-to-erasure request, provided no other owners exist.

3.4 Compliant by Construction Storage

In principle, the DOG and its relations are sufficient to identify a data subject's data, and one could imagine adding it as a metadata layer over an existing database. But in practice, compliance is more complex. Although the DOG identifies all data owned by a data subject, K9db needs to take the correct actions on this data. For example, K9db must avoid prematurely deleting jointly-owned data, and deletion must cover backups outside the live database. K9db must also have efficient ways to decide if a given database operation will break compliance, *e.g.,* by violating the *no orphaned data* invariant, something that the DOG alone fails to provide.

K9db therefore introduces ownership as a first-class notion into the storage layer. This makes it simple for K9db to handle SARs, and to enforce invariants that must hold for compliance. Specifically, K9db's storage layer is organized around per-data subject logical "micro-databases" (µDBs), such that each µDB contains all of its data subject's owned data. For jointly-owned data, K9db stores copies of that data in the µDB of every data subject that owns it.

This design has several advantages. First, it ensures data deletion is correct relative to the DOG. When a data subject requests to delete their data, it is sufficient to delete their µDB. Data shared with other data subjects survives as copies in the other µDBs. Second, this design provides an easy way to check whether data is orphaned, as such data can only exist outside of all data subjects' µDBs. Third, this design lets K9db use a per-data subject key to encrypt data in each µDB. This simplifies deletion alongside external and replicated backups of the data, as deleting the owner's key makes all backups and copies inaccessible (*i.e.,* "crypto-shredding").

3.4.1 Storage Layout and Logical µDBs

K9db determines the µDBs to store each row in using the DOG. In a well-formed DOG, every table reaches at least one data subject table via its outgoing ownership edges. K9db splits the contents of such a table into different µDBs, each of which contains the rows owned by a particular data subject, and encrypts them with a key specific to that data subject. A table also includes an orphaned data section that may be used temporarily within sequences of operations (§3.4.5). A data subject's µDB therefore includes rows from every table that stores data owned by them. Note that even though µDBs store physical copies

of rows that have multiple owners, they are a logical abstraction and realized over a single underlying physical datastore (*e.g.,* RocksDB in our prototype).

Viewing the datastore as a whole, a previously single row in a table may now be multiple rows due to copies being stored in each owner's μDB. The value of the primary key of that row refers to all these copies. Internally, K9db identifies the different copies using a pairing of the data subject identifier (the value of its primary key in the data subject table) and the value of the primary key in the row.

K9db maintains on-disk secondary indexes separate from tables and μDBs, which K9db uses to execute queries efficiently. K9db creates an on-disk index for each unique and foreign key column and for the primary key. K9db on-disk indexes differ from traditional database indexes in two key aspects: they map keys to *(μDB identifier, primary key)*, and they point to all copies of any jointly-owned row that match the indexed key. K9db creates a special index for the primary key column(s) of owned tables, which maps the PK value to data subject identifiers that own the corresponding row.

K9db stores tables unconnected to the DOG in the same way as other databases. Such tables contain data that is not owned by any data subject, *e.g.,* all available tags in Lobsters or all majors in a university database, and thus are outside any μDB. Note that this is distinct from orphaned data, which are rows without owners in tables that *are* connected to the DOG.

3.4.2 μDB Integrity

The storage layer maintains an important invariant for compliance, *μDB completeness*: data owned by a data subject is exactly identical to the data stored in their μDB.

To maintain μDB completeness, K9db must identify the μDBs to insert new data into, and correctly apply application updates that change who owns rows. Changes to the data in a table may have cascading effects on who owns data in dependent tables connected to this table via some ownership path in the DOG. For example, changes to the `member` table in ownCloud affect who owns records in the `group` table. K9db utilizes the DOG to handle these situations correctly.

Inserting Data. When K9db receives an `INSERT` statement, it uses the DOG to identify the owners of this data. In particular, K9db analyzes the outgoing edges from the DOG vertex for the affected table. For a direct ownership edge, the data subject identifier is already present in the new row in the form of a foreign key. K9db determines this by introspection on the new row and without querying other

tables. If an edge indirectly leads to the data subject table, identifying the owner becomes more complex. K9db can find the owner(s) by querying the database along the transitive edges between the table and the data subject. But such a query may be expensive—for example, the DOG for the Shuup e-commerce application [Shu18] contains a chain of five edges from the payments table to the owning data subject. Instead, K9db memoizes the query by building and maintaining in-memory *ownership indexes*, which essentially provide "shortcut" relations over the DOG that point directly to the owning data subjects. In practice, K9db can often avoid or reuse ownership indexes (§3.5.1).

Cascading Updates. `INSERT`, `UPDATE`, or `DELETE` statements may have cascading effects on the ownership of records in dependent tables. After applying such statements to their target table, K9db identifies dependent tables from the DOG. It then queries the rows in each dependent table that match the updated row. K9db moves or copies the matched rows between µDBs appropriately, and cascades again into any further dependent tables. K9db requires no additional indexes to perform this matching efficiently, as it can rely on standard on-disk indexes over foreign keys' source and destination columns. In many cases, K9db avoids cascades via optimizations based on foreign key integrity (§3.5.1).

3.4.3 Handling Subject Access Requests

K9db needs to handle two types of SARs: the *right to access* and the *right to erasure*. K9db handles both with a similar high level procedure: *(i)* K9db traverses the DOG to identify all tables and edges connected to the data subject; *(ii)* K9db finds the data owned by the data subject in their µDB; *(iii)* K9db locates data accessed, but not owned, by the data subject in other µDBs; and *(iv)* K9db performs anonymization as specified by the developers in the schema.

For either type of request, K9db identifies the data subject's data by following paths in the DOG, starting from the data subject table, and moving against incoming edges. A path that consists solely of ownership edges signifies data owned by the data subject, while paths that contain one or more access edges reflect accessed data. K9db locates the relevant rows in a table before moving on to any dependent tables. For every incoming edge, K9db uses the rows it located in the parent table to identify dependent rows in the dependent table. K9db finds these either in the same µDB for ownership paths, or in other µDBs using on-disk indexes for access paths.

After traversing an edge and retrieving data in its source table, K9db selects the anonymization annotations in the schema that apply to that edge. The anonymization annotations specify the columns to

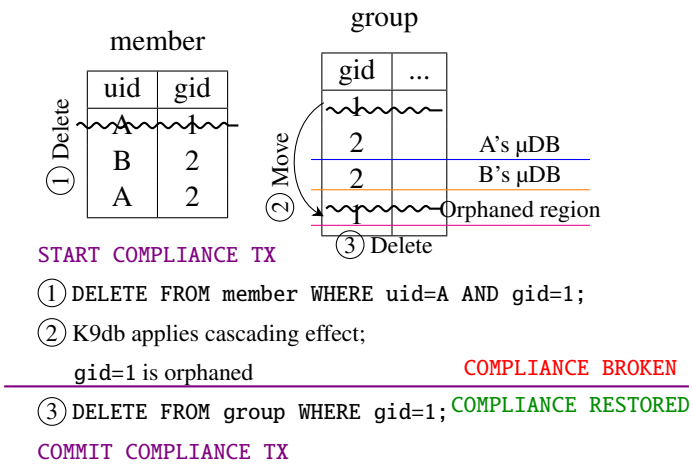


Figure 3.9: K9db's *compliance transactions* help developers check that the database is in a compliant state after multiple operations (here, (1) deleting the last owner of a group, and (3) then deleting the group). Without a compliance TX, K9db would report an error instead of applying step (2).

anonymize (*e.g.,* the sender of a chat message). For access requests, K9db anonymizes retrieved rows before sending them back to the client. On deletion requests, K9db removes the data subject's µDB from the database, and anonymizes any remaining copies of the data, which it locates in other µDBs using on-disk indexes.

3.4.4 Atomicity, Consistency, Isolation, and Durability

A single SQL statement may result in several underlying operations over K9db's storage, as it may update rows in several µDBs or cascade over dependent tables. It is critical for compliance that we ensure that these updates are all ACID, to avoid data races that could lead to a non compliant state (*e.g.,* by creating orphaned data, or breaking the µDB completeness invariant). Therefore, K9db executes every SQL statement as a single statement ACID transaction (similar to MySQL). This includes all underlying operations over any µDBs and all updates to on-disk secondary indices or the integrated in-memory cache (§3.5.2). Our prototype does not support general multi-statement SQL transactions yet (see §3.6).

K9db guarantees that concurrent SQL statements have *repeatable reads* isolation, which is the default in MySQL. Any weaker isolation level is insufficient for compliance, as it cannot guarantee that K9db's compliance invariants hold in the presence of concurrent updates.

3.4.5 Compliance Transactions

An application may itself perform operations that risk violating compliance. Consider the example from ownCloud shown in Figure 3.9: ① the application deletes user "A"'s membership in group 1,

of which "A" is the last remaining member. This deletion from `member` has a cascading effect on the dependent `group` table. Since the group with `gid` 1 no longer has any owners, K9db ② moves it into the table's orphaned data region. This breaks compliance, as it violates the DOG's *no orphaned data* invariant. A correct application must now perform some operation that restores compliance, *e.g.,* by deleting group 1 in a separate SQL operation, which ③ removes the orphaned row, restoring the invariant.

K9db supports this pattern with the idea of a *compliance transaction* (CTX). A CTX wraps a set of operations that may temporarily violate compliance, but commits only if the database is back to a compliant state at the commit point. Within a CTX, K9db stores orphaned data in orphaned regions attached to each table. On subsequent operations that reintroduce owners for this data, K9db migrates the rows from the orphaned regions to the corresponding µDBs; if deleted, K9db removes the data. At the end of a CTX, K9db ensures that every record moved to the orphaned region during the CTX has an owner again (or was deleted), and produces an error to the developer otherwise.

Finally, K9db forbids statements that write to the orphaned region unless they are part of a CTX. In particular, step ① in Figure 3.9 will error unless contained in a CTX. This means that developers need to modify applications that contain such patterns to use CTXs when necessary. Requiring such limited modification is desirable, as disallowing compliance-breaking changes outside of CTX helps developers identify issues and forces them to fix buggy and incompliant applications. For example, K9db would reject a buggy version of ownCloud that does not clean up groups with no members ③. Introducing a CTX allows an application to have benign temporary incompliance; if K9db instead required applications to only perform operations that move the database between compliant states (*e.g.,* deleting groups before deleting their last member), it would likely require more substantial rewrites.

CTX are different from regular SQL transactions, which serve to ensure consistency under concurrent execution. CTX are lightweight and required for compliance, while SQL transactions are expensive and web applications often (but not always) avoid them. In a privacy-compliant database with SQL transactions, each such transaction must also be a CTX.

3.5 Query Execution

When K9db executes a query, it must identify the µDBs affected to locate the relevant rows. Depending on the operation, this may involve finding one or all copies of shared rows.

Queries that refer to a single table, such as `DELETE` and `UPDATE` statements, and most `SELECT` queries

issued by web applications (*e.g.,* point lookups), run directly against K9db's µDBs with the aid of on-disk indexes. K9db analyzes the columns that appear in the `WHERE` condition of the query, and selects the index that matches the most columns. Like other databases, K9db finds all the rows that may match the query using the selected index, and then filters these rows with any remaining columns. If no index matches, K9db runs a scan over the table. Developers may create additional indexes using `CREATE INDEX`, similar to traditional databases.

When data has multiple owners, an index may refer to multiple copies of the same row. For `DELETE` and `UPDATE`, K9db atomically operates over all these copies, ensuring that all copies are consistent. K9db may need to remove or add some of the affected rows from/to µDBs, and may need to cascade into dependent tables as described in §3.4.2. For `SELECT` queries, K9db identifies a single copy of each matching row and skips any remaining index entries for other copies. This avoids overheads for deduplicating copies of the row.

K9db serves some complex `SELECT` queries from materialized view, described in §3.5.2.

3.5.1 Optimizations

K9db speeds up query execution and reduces its memory footprint with a set of optimizations designed to avoid deep cascades and to reduce the number of in-memory ownership indexes (§3.4.2) required. Some of these optimizations rely on *foreign key integrity*, which K9db enforces (like many other databases) to prevent application operations that result in dangling foreign keys. With FK integrity, rows cannot be inserted into a table if they contain references to non-existent rows in a destination table, and rows in the destination table cannot be deleted as long as source table rows refer to them.

Avoiding Cascades. K9db needs to cascade into dependent tables along incoming DOG edges to update dependent rows affected by a write (*i.e.,* those owned by a modified row). But FK integrity guarantees that no such rows exist when K9db handles `INSERT` and `DELETE` queries to a table T that is the destination of a FK from a dependent table. This lets K9db skip cascades along T's incoming DOG edges if the edge is in FK direction; otherwise, K9db must cascade.

Ownership Indexes. K9db relies on two techniques to reduce the number of ownership indexes. First, multiple incoming DOG edges that require an ownership index and point to the same column of a table (usually the primary key) may reuse the same index. Second, K9db omits ownership indexes for edges

in the DOG that correspond to `OWNS` annotations, such as the edge from `group` to `member` in ownCloud. These edges point in opposite direction to the underlying foreign key. FK integrity ensures that a row must exist at the source of such an edge (*e.g.,* `group`) before any rows referring to it can be inserted to the destination table (*e.g.,* `member`). Hence, K9db always inserts new rows from the source table into the orphaned region, and defers moving them to the correct μDB to future inserts into destination tables in the DOG (which must cascade), as discussed in §3.4.5. These optimizations, for example, help K9db create only one ownership index for Lobsters (which gets re-used three times), and avoid the need for any ownership indexes in ownCloud.

Queries With Inlined Owners. SQL Statements sometimes directly refer to the owner of their target rows, *e.g.,* by constraining a foreign key that corresponds to an ownership edge in the DOG. Queries that fit this pattern are common in the web applications: *e.g.,* in Lobsters, `SELECT * FROM stories WHERE author = ?` selects stories by their author, which is an annotated foreign key to `users`. K9db detects this situation by statically analyzing the `WHERE` condition and determines the relevant μDB without an on-disk index lookup.

3.5.2 Materialized Views

K9db serves complex `SELECT` queries, such as joins, aggregations, and those that reorder data, from materialized views. This design makes sense for two reasons. First, it is simple and avoids the need to engineer a sophisticated query planner that understands the nuances of ownership and indexes to efficiently execute these queries over K9db's μDBs. Second, developers often cache the results of complex `SELECT` queries in external systems (*e.g.,* memcached). Privacy compliance while using an external cache requires setting appropriate expiration policies for the cache [YYR21, §4.5] or explicit invalidation of cache entries related to a data subject if they request deletion of their data. This can be painful for developers and may require manually tracking metadata, *e.g.,* when caching aggregates over many data subjects' data. Instead, K9db provides an integrated privacy-compliant cache using materialized views.

When K9db receives a complex `SELECT` query for the first time, it creates a materialized view and serves further instances of the query from it, until the view is removed or times out. K9db keeps the materialized views up to date via an incremental, streaming dataflow computation triggered by writes to μDBs, as well as μDB deletion. This makes inserts, updates, and deletes more expensive, but speeds up reads. K9db updates the materialized views atomically prior to acknowledging the corresponding

operation to the client. This, along with our storage layer, ensures *repeatable reads* isolation for concurrent operations whether cached or not.

K9db's ownership indexes are special-case materialized views, maintained with the same dataflow infrastructure.

3.6 Implementation

Our K9db prototype consists of 35k lines of C++, 500 lines of Rust, and 2k lines of Java. It relies on RocksDB for µDB storage, on Apache Calcite [BCH+18] for query planning, and on libsodium [Den13] for encryption. Our implementation is similar to the MyRocks MariaDB storage engine [Mar22], but extends it with compliance and µDB capabilities.

MySQL Compatibility Layer. K9db exposes a MySQL binary protocol interface, so unmodified applications can treat K9db as a MySQL server. The interface to K9db's materialized views is primarily through prepared SQL statements: when an application registers a prepared statement, K9db creates a view if necessary and serves future executions of the prepared statement from it. Developers can also create additional views manually.

Storage. K9db relies on RocksDB for persistent data storage. Each table in the schema is a RocksDB column family. Rows in K9db are keyed by a combination of their owner and primary key, to uniquely identify each owner's copy of a row. Our prototype stores rows ordered by their owner identifier, and uses that identifier as a RocksDB prefix. This allows it to extract and delete µDBs using RocksDB prefix iterators. Our prototype creates and maintains on-disk indexes as RocksDB column families, and formats their content to allow writes to retrieve all the copies of a row, and reads to retrieve a single arbitrary copy, skipping the rest. Like MySQL, K9db creates indexes for primary, unique, and foreign keys.

An earlier version of K9db used MySQL for µDB storage, but its default storage engines do not scale to the number of tables or databases needed for K9db. (The MySQL community is actively working on this, motivated by GDPR compliance use cases [Rub18].)

Encryption at Rest. K9db uses hardware-accelerated AES256-GCM to encrypt all data in a µDB with the key of its owner. The key (µDB identifier, primary key) associated with every row is encrypted deterministically with a global key to allow consistent lookup. This has leakage, but is sufficient to

satisfy the GDPR's "security of processing" requirement (Art. 32), which is often interpreted to require encryption of data at rest [Ama23]. It is possible to use blind indexes [Arc17] which also allow consistent lookup but reduce leakage. K9db's design is independent of the particular encryption scheme used, and can benefit from future advances in searchable encryption. Information in materialized views and secondary indexes remains unencrypted, but K9db deletes it when deleting a user's data. K9db destroys the decryption key when a user removes their account, making any remaining backups inaccessible.

ACID. K9db executes each application SQL statement in a RocksDB transaction, which is based on row-level locking. This includes all updates to secondary indices (similar to MyRocks) and all μDBs and cascade operations. As in MyRocks, K9db serves reads from a consistent RocksDB snapshot. K9db also updates all relevant materialized views prior to committing. Unlike MyRocks, K9db enforces foreign key integrity and appropriately locks FK targets during execution. Overall, this ensures that concurrent SQL statements are atomic and consistent with *repeatable reads* isolation, which is the default in MySQL and MyRocks.

View Updates. K9db's materialized view updates follow a standard design akin to differential dataflow [MMI+13a; MMI+13b] and Noria [GSB+18]. Each table in the schema is associated with an input vertex in the dataflow graph, and when K9db performs updates to a table, it injects the updates into its dataflow input vertex. The dataflow processes the updates through a sequence of operators to derive an incremental update to the materialized view (or secondary index), and applies this update. Dataflow operators are stateless (*e.g.,* projections, filters, unions) or stateful (*e.g.,* joins, aggregations). K9db's materialized views are indexed for ordered and unordered lookups.

Limitations. Our prototype lacks support for general, multi-statement SQL transactions. These are rare in web applications, and can be supported using existing RocksDB primitives and techniques for versioned dataflow processing [MLS+20; MMI+13b]. While our prototype does not yet support schema changes, RocksDB is schema-oblivious, and our prototype's storage layer could be extended to support schema changes with some engineering effort, using similar techniques to MyRocks. Finally, K9db's dataflow graph operators sometimes store copies of a record; by using a record pool, our prototype's memory footprint could be reduced.

3.7 Evaluation

We evaluate K9db with three applications, Lobsters [Lob18b], ownCloud [own21b], and Shuup [Shu18]. We ask three questions:

1. What is K9db's impact on end-to-end application performance? (§3.7.1)
2. What is the impact of K9db's design features on performance? (§3.7.2)
3. What effort by application developers does using K9db require? (§3.7.3)

We run experiments on a Google Cloud `n2-standard-16` VM, storing databases on a local SSD. Our baselines use MariaDB v10.6.5 (a MySQL fork) with the RocksDB-based MyRocks storage engine, and memcached v1.6.10.

3.7.1 Application Performance

We start by analyzing K9db's performance with two applications: Lobsters and ownCloud.

Lobsters

Lobsters (lobste.rs) is an open-source discussion board, similar to Reddit. Lobsters currently lacks GDPR compliance [Lob18a], and has a schema that consists of 19 tables, which store posts, comments, nested replies, upvotes, invitations and other information. We annotated this schema for K9db with three `DATA_SUBJECT` tables, 14 `OWNED_BY`, one `ACCESSES`, and two anonymization annotations (details in §3.7.3). We use an existing open-source, open-loop benchmark for Lobsters based on public workload statistics [Har18]. The benchmark models ten endpoints in the Lobsters webapp that correspond to different pages and each issue between six and fifteen SQL queries, most of which are reads. We load the database with data that models the current production Lobsters deployment (15k users, 100k stories, 313k comments, and 416k votes) [Har18]. K9db therefore maintains 15k logical µDBs in this experiment. We compare MariaDB, and K9db with and without data encryption. (Encryption with per-user keys isn't possible in the MariaDB baseline.) Lobsters on most requests runs an expensive query to determine the user's recently read stories. This query joins four tables, including the (large) stories and comments tables. This query is slow in MariaDB (≈30ms) and dominates its latency for all endpoints, while K9db serves this query from a materialized view. To make the comparison fair, we remove the expensive query in the MariaDB baseline. A good result for K9db would show latencies comparable to MariaDB for all endpoints, and a low overhead for encryption.

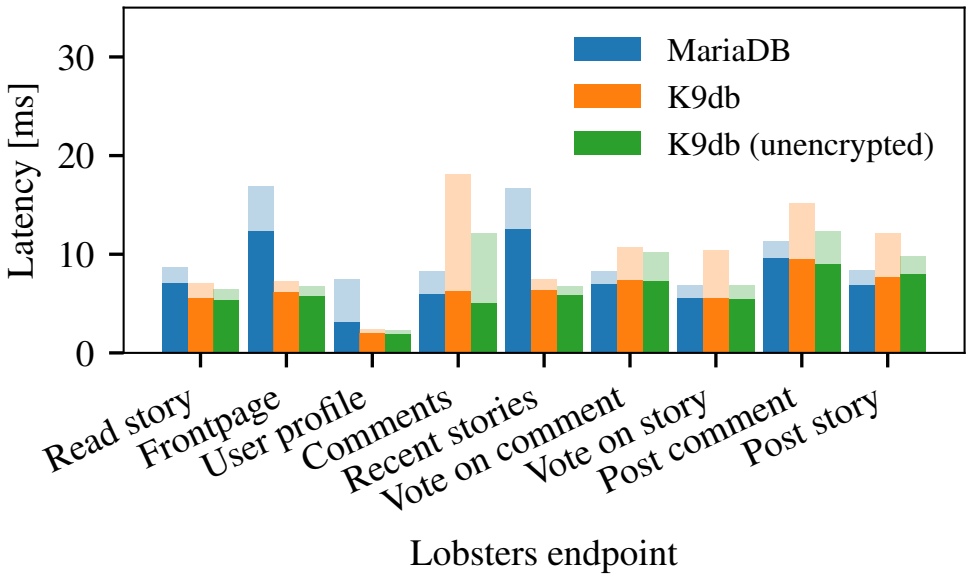


Figure 3.10: K9db matches or beats MariaDB's median (solid) and 95th percentile (shaded) latency on the Lobsters workload, and encryption has low overheads except on the "Comments" endpoint, which reads thousands of rows in the tail.

Figure 3.10 shows the results. Endpoints that mostly read (on the left) benefit from K9db's materialized views and are up to 2.1× faster than in MariaDB, but endpoints with many writes (on the right) are comparable in both systems. This makes sense, as K9db performs similar work to MariaDB, except that some read queries are served from materialized views, and writes need to be encrypted and must update any corresponding views. K9db without encryption is on-par with K9db in most endpoints. For the "Comments" endpoint, K9db is 2.1× slower than MariaDB and 1.5× slower than K9db without encryption in the 95th percentile. This happens when the endpoint retrieves comments and votes on a popular story from the database, which requires K9db to decrypt thousands of records. Developers could manually add materialized views in K9db to speed up this endpoint, at the cost of additional memory. Other endpoints read fewer rows or rely on (unencrypted) materialized views. This shows that K9db achieves good performance for a practical web application, and that encryption has acceptable cost. All further experiments show results for K9db with encryption enabled.

We chose the load in this experiment to saturate the hardware for the MariaDB baseline (≈ 760 pages/second, which results in 10k queries/second) and used the same load for K9db. K9db supports a up to a 4.8× higher load without latency degradation, thanks to its caching for complex queries via materialized views; we compare to a caching MariaDB+memcached baseline below.

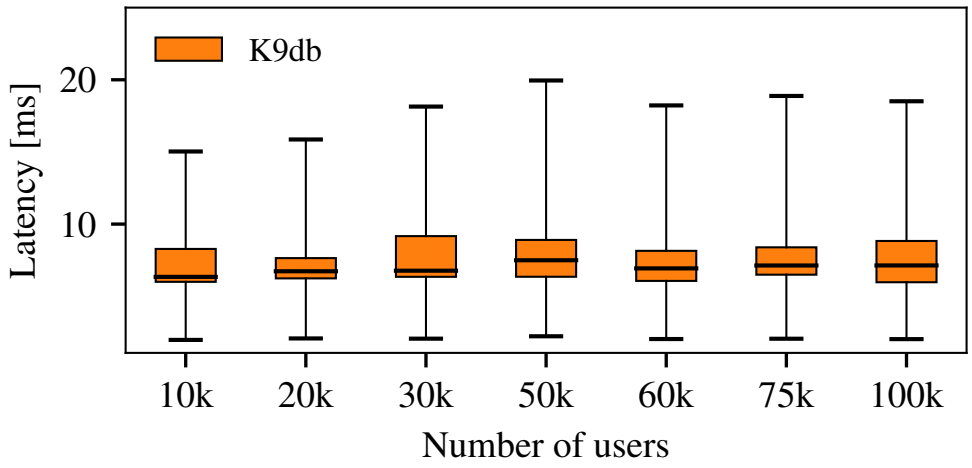


Figure 3.11: K9db's 95$^{\text{th}}$%ile latency on the Lobsters workload remains stable as the number of users (and thus, µDBs) increases. Each bar shows a distribution of endpoint latencies.

Subject Access Requests. We now measure the time required by K9db to satisfy SARs. We issue an access and a deletion request for each of the top 1000 users with most data in the database, and run these requests sequentially through K9db SARs API. Performance of SARs is secondary as they are rare operations and can be executed asynchronously. A good result shows that K9db handles SARs correctly (which it does by construction) and within reasonable time. In our experiment, K9db on average takes 1 ms to retrieve and 45 ms to delete the correct data for a user.

Scalability. We designed K9db to have performance independent of the number of µDBs. We confirm this using the Lobsters benchmark with different numbers of users. Adding users increases the number of µDBs and the amount of data in the database, but keeps the average amount of data per user constant. A good result for K9db would show latencies remaining constant as the number of users grows.

Figure 3.11 shows the results as box-and-whisker plots over the nine endpoints (*i.e.,* the bottom and top whiskers are the fastest and slowest endpoints, respectively). K9db's latency remains constant as the number of users—and, consequently, µDBs—grows, because K9db satisfies queries either from µDBs directly, via indexes, or from materialized views. These results confirm that K9db's logical µDB partitioning is practical for applications with large numbers of users.

Comparison to Caching Baseline. In the previous experiment, K9db had an unfair advantage over MariaDB: it serves some data from materialized views, while MariaDB recomputes queries every time. We now use one common query from Lobsters to compare three setups: *(i)* standalone MariaDB; *(ii)* MariaDB

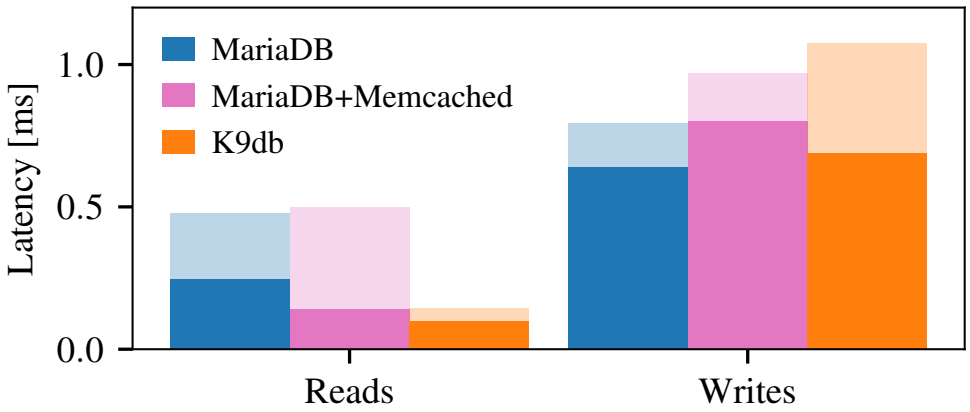


Figure 3.12: K9db matches MariaDB+memcached on a common Lobsters query (solid: median; shaded: 95$^{\text{th}}$%-ile).

with an in-memory cache ("MariaDB+Memcached"); and *(iii)* K9db. The MariaDB+Memcached setup is a demand-filled cache [NFG+13]: writes invalidate the cached query result in memcached, and the next read re-runs the query against the database when it misses in memcached. In K9db, writes update views via its dataflow graph. We generate a skewed workload with a Zipfian distribution ($s = 0.6$) where 95% of requests in the benchmark read the details of a random story and its vote count, and 5% of requests insert new votes. A good result for K9db would show competitive read performance with memcached and low overheads on write processing (since K9db does more work on writes); and MariaDB+Memcached and K9db would show lower latencies than MariaDB alone.

Our results are in Figure 3.12. For reads, MariaDB+Memcached and K9db are on par in the median, but K9db has a lower 95$^{\text{th}}$ percentile latency as K9db updates the cache via streaming dataflow, while MariaDB+Memcached queries the database on a read miss. All systems perform similarly on writes, as this query requires little dataflow update work in K9db and the caching baseline must make an extra RPC to invalidate memcached.

Memory Overhead. K9db's materialized views and ownership indexes add memory overhead compared to a traditional database. We measure this cost and compare it to a caching setup with memcached. We consider a setup that caches query results that developers would typically store in memcached, such as the output of expensive joins and aggregates. These queries are identical to the ones that K9db caches using materialized views. The experiment caches query results with the query parameters (`?` in prepared statements) as the key, and the concatenated records as the value. K9db stores additional in-memory data for internal dataflow state and ownership indexes. A good result for K9db would therefore show moderate

overheads compared to MariaDB+Memcached.

The Lobsters database is 61 MB on disk, and a typical memcached caching approach stores an additional 97 MB of in-memory state. K9db's memory footprint is 197 MB (3.3× DB size, and 2× memcached's footprint), which includes 6.5MB for the stories ownership index, and 56 MB for caching the expensive query we removed from MariaDB (without this query, K9db's overhead is 2.4× DB size/1.5× memcached). The overhead comes from K9db's dataflow state, which allows K9db to incrementally update materialized views.

ownCloud

ownCloud is a popular open-source application that allows users to upload files and share them with other users [own21b]. Recall ownCloud's schema (Figure 3.5): each file has a single owner—the original uploader—but users can share files with other users and with groups. Files shared with a group are accessible to all members of the group—*i.e.,* a many-to-many relationship between users and files (a pattern absent in Lobsters). We measure five common queries: *(i)* listing the files a user can access ("view files"); *(ii)* sharing a file with another user ("share with user"); *(iii)* sharing a file with a group ("share with group"); *(iv)* retrieving a file using its primary key ("Get"); and *(v)* updating the retrieved file ("Update"). Our setup uses 100k users who each own three documents; each document is shared uniformly at random with three users and two groups; and each group has five members. Our workload is 95% read and 5% writes, equally split among the two types of sharing and file updates. Reads and writes target users drawn from a Zipf distribution ($s = 0.6$). We batch ten reads and measure the per-request latency for the same setups as in the previous experiment. A good result for K9db would show comparable read latency to MariaDB+Memcached and low overheads on writes.

Figure 3.13 shows the results. "View files", which returns all files shared with a user (directly or via a group), involves five tables and three joins, which MariaDB executes on every read. MariaDB+Memcached and K9db serve precomputed results from memory instead, which is fast. The 95$^{\text{th}}$ percentile for MariaDB+Memcached suffers because it queries MariaDB on a cache miss, which occurs when a query retrieves files of user(s) invalidated by a previous write. K9db is fast and stable because it updates the views via dataflow on writes. All systems perform similarly for the two share queries—a good result for K9db, as it also updates views.

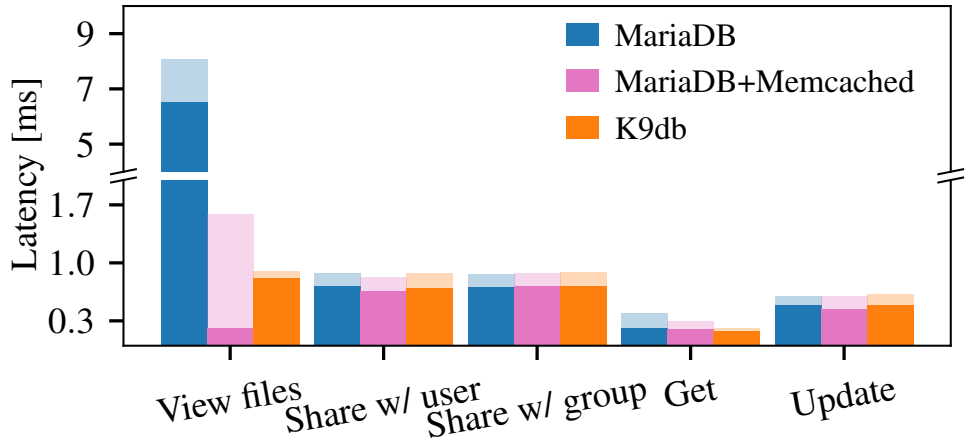


Figure 3.13: K9db matches the baseline setups' performance on the ownCloud workload (solid: median; shaded 95$^{\text{th}}$%-ile).

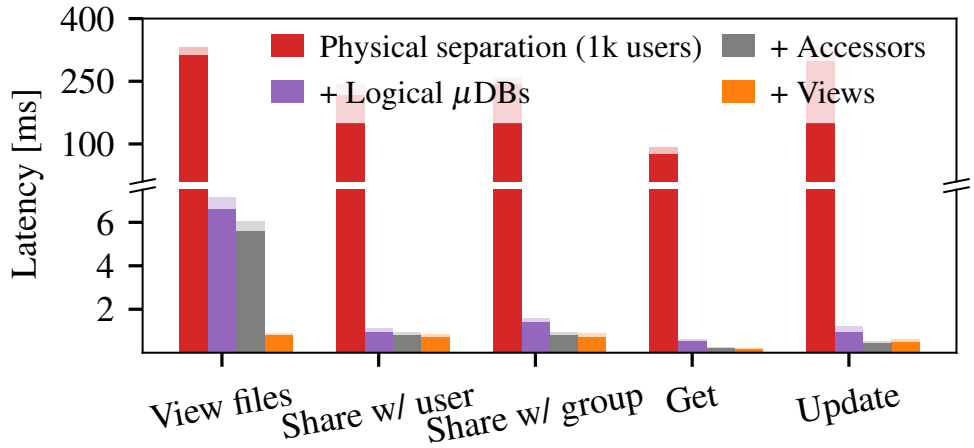


Figure 3.14: K9db design decisions and optimizations are critical for good performance on the ownCloud workload (solid: median; shaded 95$^{\text{th}}$%-ile).

3.7.2 K9db Design Drill-Down

To evaluate the impact of design decisions central to K9db, we run ownCloud workload from the previous experiment against versions of K9db that disable key components. We start with K9db set up to naïvely store every μDB in its own database (without cross-μDB indexes); without support for accessor edges in the DOG; and without materialized views (*i.e.,* queries always run over data in RocksDB). This guarantees strict separation of user's data, a solution sometimes adopted for GDPR compliance [Rub17; Rub18], although this lacks support for shared data (accessors) and anonymization. We then add separation into logical μDBs ("+ Logical μDBs"), accessor support ("+ Accessors"), and materialized views ("+ Views"). A good result would show that these features improve K9db's performance.

Figure 3.14 shows the results. The naïve µDB design is very slow because every query that K9db cannot statically resolve to the affected µDBs requires scanning all µDBs; we only ran this setup with 1k users (vs. 100k for the others). Making µDBs a logical abstraction much improves performance, justifying our design choice. Accessor-typed DOG edges are important for expressivity: without them, ownCloud would be restricted to a policy where users jointly own shared files. In addition, accessor support reduces the number of copies stored and the fan-out of writes, which slightly reduces query latency. Finally, materialized views improve latency of the "View files" query by 5×, as the results are cached in memory. Since the view update is cheap, writes do not suffer much overhead. The runtime of "View files" without no views is comparable to the runtime of the same query in MariaDB (Figure 3.13). This illustrates that views are beneficial, but not essential to good performance in K9db.

3.7.3 Schema Annotation Effort

To understand the developer effort K9db's schema annotations require, we now consider annotations for three applications (Lobsters, ownCloud, and Shuup [Shu18]) in detail,

Lobsters. The Lobsters schema contains 19 tables. To use K9db, we had to annotate the schemas for eight tables. Three tables (`users`, `invitations`, and `invitation_requests`) contain data subjects. We annotated two FKs in each of `hats`, `messages`, and `moderations` with `OWNED_BY` to model joint ownership. We annotated 8 other tables with a single `OWNED_BY`. For example, `votes` has multiple foreign keys that lead to the `users` table (one direct, two indirect), and thus requires a single `OWNED_BY` annotation to disambiguate and ensure votes are stored with the voter, rather than the author of the story or comment voted on. Finally, we used one `ACCESSES` in `taggings`, and two anonymization rules in `messages`, as shown in Figure 3.4.

ownCloud. ownCloud's schema has 51 tables. We focused on the file sharing core, which consists of six tables and has the most complex relationships. In addition to the annotations in Figure 3.5, we added an `OWNS` annotation to the FK in the `share` table that points to the corresponding file in the `file` table (omitted from Figure 3.5 for brevity).

ownCloud's original schema "overloads" the `share_with` column to either hold a user or a group ID, and includes a `share_type` column to distinguish these cases. K9db could support such de-normalized schema with more advanced conditional annotations; for our benchmarks, we modified the schema to

track users and groups in separate columns.

Shuup. Shuup [Shu18] is an open source e-commerce platform and supports customers with accounts, guests who do not have accounts, and shop owners, all of whom have GDPR rights. Shuup lets users request their account to be anonymized, but retains information for tax compliance, *e.g.,* payment data, customers' countries of residence, and tax ID numbers, a form of data retention allowed by the GDPR.

Shuup provides GDPR compliance via a manually-implemented module with 4k lines of Python code (2.7k lines of implementation and 1.3k lines of tests), developed in 137 commits over three years. At the time of writing, Shuup's anonymization behavior is inconsistent; it only anonymizes default shipping and billing addresses, but retains previous addresses in cleartext in the `mutable_address` table [Kil21b]. Moreover, downloading data for a user is not supported [Kil21a].

We implemented Shuup's anonymization policy in K9db using all annotations (Figure 3.3) over 17 of Shuup's 278 tables. We annotate `personcontact` with `DATA_SUBJECT`. This table stores natural persons, and has FKs to their contact information (in `contact`) and their logins (in `auth_user`) if they have accounts. Thus, `personcontact` contains users with and without accounts, *i.e.,* guests. Using K9db, Shuup correctly anonymizes data, lets users download the data and fixes the bug of not anonymizing previous default addresses.

Shuup's schema has several tables that might correspond to data subjects. K9db's `EXPLAIN COMPLIANCE` helps developers understand that they need to annotate `personcontact`. An incompliant (but plausible) alternative would be to annotate `auth_user`, the login details table. This results in `contact` being unconnected to the DOG, as there are no foreign keys to `auth_user`. The `personcontact` table has such a foreign key, but it is nullable (*e.g.,* for guests who lack accounts), and thus some of its rows will be stored in µDBs and others in the orphaned region.

`EXPLAIN COMPLIANCE` helps developers identify and rectify these issues:

```
1 Table "contact": GLOBAL
2  [Compliance Warning] Column "email" suggests personal data, but the table is not connected to
       any owners.
3 Table "personcontact": in µDB for auth_user.id
4  [Compliance Warning] Table has owners, but nullable foreign key may prevent correct deletion of
        data.
```

Application	Tables	Data Subject	Owner	Access	Anon
Commento [Cha18]	12	3	8	1	3
ghChat [aer20]	6	1	7	2	4
HotCRP [Koh06]	26	2	15	10	7
Instagram clone [Sha18]	19	1	18	1	0
Mouthful [Kuz18]	3	1	1	0	0
Schnack [sch17]	5	1	1	0	0
Socify [Mur21]	19	1	10	0	0

Figure 3.15: K9db requires few `DATA_SUBJECT`, ownership (`OWNED_BY` and `OWNS`), access (`ACCESSED_BY`, `ACCESSES`), and `ANON` annotations to support real web applications.

A developer might also annotate `contact` with `DATA_SUBJECT`, but that table includes entries for customers and companies. Annotating it makes companies into data subjects, which duplicates company-related tables across µDBs. `EXPLAIN COMPLIANCE` also alerts developers to this.

Other Applications. Our schema annotations were sufficient to express reasonable compliance policies for seven additional applications (Figure 3.15). We briefly highlight several interesting patterns in these applications.

In ghChat [aer20], a chat application for GitHub, and the Instagram clone [Sha18], a group is owned exclusively by its admin and accessed by its members. This is unlike ownCloud, which lacks group admins and has members jointly own the group.

Mouthful [Kuz18] is a commenting service that embeds in a host application (*e.g.,* a blog) to allow users to comment on the host content (*e.g.,* a blog post). Mouthful has no notion of users; instead, the host application provides a string that represents the user identity alongside the comment they posted. We added a `DATA_SUBJECT` table to store user identifiers, and created a FK constraint from the `Comment` table's `author` column to it.

Finally, the HotCRP [Koh06] review system associates data subjects to papers via a many-to-many `PaperConflict` table. The table has a `conflictType` column that specifies the relationship, such as "co-author" or "institutional conflict". While this schema is normalized in the traditional SQL sense, it is not normalized for ownership: rows with the co-author type signify ownership, while other conflict types do not imply any ownership or access rights over the paper. We resolved this by adding a new `PaperAuthors` table that only stores authorship associations, and refer to papers from it using `OWNS`. We reserve `PaperConflict` to record other conflict types with an un-annotated reference to papers.

Migrating Applications to K9db. We identify some common challenges when migrating applications to K9db. First, annotating an application schema requires knowledge of the application functionality and its compliance policy, but also summarizes the policy in an easy-to-maintain way alongside the schema. Many web applications also lack explicit FK constraints in their schema; developers must identify the columns that act as implicit FKs and annotate them if needed.

Second, applications often have schemas that are not normalized in the traditional SQL sense (*e.g.,* ownCloud's `share_with`) or with regards to ownership (*e.g.,* HotCRP's `PaperConflict`). Developers must normalize these schemas by introducing new columns or tables, and apply the corresponding changes to the application code. K9db could support such schemas via new annotations that condition on other columns, but this would complicate the annotation language and DOG model. Instead, K9db guides developers to good, normalized schema designs.

Finally, applications with variable ownership (*e.g.,* ownCloud, Shuup, HotCRP) often have endpoints that temporarily orphan data. Developers must wrap such endpoints in compliance transactions in order to use K9db. This modification is relatively unobtrusive, and K9db can be configured to automatically wrap sessions in a CTX. This alleviates the need to manually introduce CTX to applications that open new sessions for each endpoint or sequence of operations, but is not suitable for applications with long-lived sessions.

3.8 Discussion

Relationship to Garbage Collection. There are high-level similarities between data deletion in K9db and garbage collection in memory-managed programming languages. For example, one can view K9db's `GDPR FORGET` primitive as a kind of DOG-guided garbage collection. Consider a DOG with inverted edges, such that the edges point away from data subject to their corresponding data. In this case, records in data subject tables correspond to application-held active references, and (inverted) ownership and access edges correspond to strong and weak pointers, respectively. In this analogy, `GDPR FORGET` operates similarly to *(i)* deleting the top-level data subject row corresponding to the user that requests deletion (*i.e.,* the application giving up on a reference) and *(ii)* deleting all data that is no longer reachable from any data subjects (*i.e.,* garbage collection).

Since K9db stores copies of jointly owned data in each of its owners' µDBs, its deletion algorithm is akin to reference counting. On the other hand, data deletion in DELF [CDN+20] is more akin to tracing.

There is a crucial difference between garbage collection and data ownership in K9db. Imagine that an application consumes some reference in the middle of a transitive chain of pointers, causing its children to no longer be reachable. Garbage collectors can reclaim all these orphaned children without affecting the integrity of the application. However, K9db cannot similarly delete orphaned data created by some regular application operation, such as when the last member of a jointly owned group leaves that group. This is because such an operation may be a part of a larger transaction or may be the result of an unintentional or buggy application behavior. In both cases, deleting the data in question results in unrecoverable data loss, and violates the usual semantics of SQL database that developers are used to. This is why K9db rejects such operations unless they are part of a compliance transaction that addresses all orphaned data.

Alternative Designs. We experimented with alternative designs for K9db during our work. This included possible designs in which K9db acted as a shim on top of traditional SQL databases, such as MySQL and SQLite. A strawman design where the underlying database executes regular application operations without intervention from K9db cannot guarantee compliance, as regular application operations may create orphaned data and violate various ownership-integrity requirements.

We experimented with an improved strawman design, where K9db analyzes and rewrites application operations to ensure that they respect these integrity requirements prior to executing them via the underlying database. This design can indeed guarantee compliance and resulted in a smaller and simpler implementation. However, it exhibits high overheads, as it required building and maintaining complex metadata to enable analysis of application operations, and transformed even simple application operations into complex multi-table transactions. K9db's design overcomes these overheads by building a new storage layer organized by data ownership and the DOG, such that compliance invariants are simpler and cheaper to enforce.

We also investigated extensions that apply some of K9db's ideas, specifically its schema annotations and DOG, to non-SQL databases and to applications with heterogeneous storage. We discuss these in greater detail in §6.1.

3.9 Summary

K9db is a new database system that achieves compliance with the requirements of privacy laws by construction.

K9db models data ownership to capture the ownership patterns of real world applications, and handles requests for access and deletion correctly. K9db matches or exceeds the performance of a widely-used database and manual caching setup, and supports the privacy requirements of real-world applications. K9db is open-source and available at https://github.com/brownsys/K9db.

CHAPTER 4

Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions

This chapter introduces Sesame, a practical framework for end-to-end privacy policy enforcement. Sesame supports a wide range of expressive application-provided policies. These policies may correspond to requirements imposed by privacy laws, such as access control, purpose limitation, and user consent. They may also encode self-set and internal policies that organizations wish to enforce over their applications, such as policies governing data aggregation or security of authentication tokens and cryptographic secrets.

Sesame wraps data in policy containers that associate data with policies that govern its use. Policy containers force developers to use privacy regions when operating on the data, and Sesame combines sandboxing and a novel static analysis to prevent privacy regions from leaking data. Sesame enforces a policy check before externalizing data, and it supports custom I/O via reviewed, signed code.

We describe our experience porting four web applications to Sesame. Our results show that Sesame's automated guarantees cover 95% of application code, with the remaining 5% needing manual review. Sesame achieves this with reasonable application developer effort and imposes 3–10% performance overhead (10–55% with sandboxes).

4.1 Motivation

Modern web applications are subject to both self-imposed privacy policies and those required for compliance with privacy laws (*e.g.,* GDPR [GDPR16], HIPAA [HIPAA96], FERPA [FERPA74]). We provide an overview of the requirements of general purpose data protection laws in §2.1. Inadvertent

breaches of these policies can lead to significant penalties [Cal; NOY20c; NOY20d; NOY20e]. For example, Instagram was fined €405M for accidentally disclosing children's email addresses and phone numbers [Eur22]. Although Instagram had the correct internal policy and enforced it manually, developers overlooked an edge case where children had business accounts with public details.

Challenges. These problems are difficult to avoid because developers today lack *practical* frameworks to ensure that their code abides by their privacy policies. It's easy for a developer to misremember which policy applies to data, or to forget to apply appropriate checks throughout the application code. To reduce this burden, developers need small and clear regions of privacy-critical code on which to focus their attention, and automatic guarantees for the remaining code.

Existing systems that seek to provide policy compliance guarantees face the challenge of enforcing complex and application-specific policies over an entire codebase (see §2.2.1). Classic approaches to compile-time enforcement require developers to aid the analysis, *e.g.,* by reasoning about security labels or writing proofs [LKB+21; LTB+24]. Dynamic approaches, by contrast, often require custom runtimes and either suffer from high overhead [YHA+16; YWZ+09] or limit application functionality [WKM19].

Sesame. Sesame is a new framework for writing web applications that guarantees that the majority of application code upholds policies attached to data, and clearly highlights the remaining code that developers must audit. Sesame targets end-to-end enforcement of flexible, data-dependent policies for applications written in a widely-used mainstream programming language (Rust). This means that Sesame can express rich policies that rely on dynamic information about the data or application. Unlike earlier work, Sesame achieves this without a custom runtime, in the presence of third-party libraries, with limited extra burden for developers, and at low overhead. Sesame embraces key taint-tracking techniques from prior work on Information Flow Control (IFC) systems, but makes different tradeoffs to provide practical abstractions for developers. Sesame's key idea is to break the application into smaller, independent *privacy regions* that operate on sensitive data and "glue code" that connects these regions. This breakdown is possible because the Rust type system provides automated guarantees for the glue code, and it enables Sesame to apply a new hybrid approach to reason about privacy regions. Sesame checks privacy regions with a policy-independent static analysis, uses selective dynamic enforcement when static analysis fails, and taps into existing software engineering processes like code review as a fallback.

Policy Containers. Sesame needs to prevent code outside privacy regions from accessing sensitive data, and must track the association between data and policy. Sesame addresses this need with *policy containers*. A policy container is a wrapper type that protects the wrapped data and associates a policy object with it. Sesame relies on Rust's encapsulation of private members to restrict access to the sensitive data, and on Rust's static type system to propagate the policy taint, even as the policy container moves through data structures and glue code.

Privacy Regions. Of course, application business logic eventually needs to compute on the sensitive data. To do so, the developer uses a Sesame privacy region, realized as a closure that has access to the raw data. Sesame unwraps data in policy containers passed to the privacy region and re-wraps any returned data. This allows Sesame to distill the problem of arbitrary policy enforcement into enforcing a fixed, policy-independent *leakage-freedom* property over privacy regions. In particular, a privacy region must not—directly or implicitly—leak sensitive data into captured or global variables, via system calls, or through native or unsafe code. Sesame applies static analysis to privacy regions to detect such leakage.

Because no existing static analysis for Rust covers this leakage-freedom property, Sesame contributes SCRUTINIZER, a new static analyzer that soundly rejects leaking privacy regions. Rust's mutability, ownership, and lifetime information enable SCRUTINIZER's analysis, which is difficult to do precisely in other languages. Verified leakage-free regions that pass SCRUTINIZER run as-is and without runtime overhead.

SCRUTINIZER's static analysis is sound but incomplete, so it may reject non-leaking privacy regions that call into complex-to-analyze code or libraries with native code. Sesame executes such regions in a sandbox that prevents leaks.

Naturally, applications eventually do intentionally externalize data, *e.g.,* via database queries, HTTP RPCs, or emails. Sesame accommodates this via trusted Sesame-enabled libraries, which invoke policy checks before releasing data from policy containers. For custom I/O via arbitrary unsupported libraries, Sesame provides *critical regions*. A code reviewer must manually review and sign each critical region. Sesame's design makes critical regions infrequent ($\leq$5% of code), slim ($\approx$16 LoC), and clear to the reviewer; and Sesame enforces new reviews when the underlying code changes.

Thread Model. Sesame targets honest developers who make unintentional mistakes and assumes that

sanctions deter developers from malicious behavior. Thus, timing and side-channel attacks are out of scope. The Rust compiler, SCRUTINIZER's static analysis, the sandbox, and Sesame-provided libraries are trusted, and Sesame's guarantees for critical regions rely on good-faith and attentive code review. Developers should use Sesame-provided libraries to interact with external entities, such as an HTTP client or a database, to avoid frequent critical regions. Such mandatory use of specific libraries is already common practice in organizations today.

Results. We evaluated our Sesame prototype [DAA+24b] with four Rust web applications. Our experience suggests that Sesame can express a wide variety of policies and requires limited developer effort, and that effort is focused on critical regions that need careful attention. In our case studies, Sesame's automated guarantees cover 95% of application code, including the vast majority of privacy regions. Sesame's policy checks add 3–10% runtime overhead in the common case, while sandboxed regions have higher overhead (10–55%).

Contributions. In summary, we highlight four contributions:

1. Sesame, a practical framework that enforces policy compliance over data by construction, by breaking the application into glue code and privacy regions.
2. A new approach that composes Rust's guarantees with hybrid static/dynamic enforcement of a single, policy-independent leakage-freedom property.
3. A novel static analysis that checks Sesame's leakage-freedom property over privacy regions.
4. Dynamic enforcement of Sesame's guarantees using sandboxing where static analysis fails, and a design that focuses developer attention on critical regions that require human review.

4.2 Sesame Overview

Sesame is a framework for web application development. To illustrate how developers use Sesame, consider how a developer might write an HTTP endpoint for students to submit their answers in a homework submission application (Figure 4.1a, without Sesame). The code authenticates the user (line 2), retrieves their email address from the database (line 4), inserts the student's homework answer into the database (lines 6–7), and sends the student a confirmation email via a third-party library (lines 10 and 18).

This endpoint handles two types of user data: the student's email address and their submitted answer.

```
1  fn submit_homework(request: Request) {
2    let uid = authenticate(request);
3    // Get the user's email
4    let email = DB.lookup(uid);
5    // Get the answer from the request
6    let answer = request.answer;
7    DB.insert(answer);
8
9    // Format email body
10   let body = format!("submitted {}",
11                      answer);
12
13
14
15
16
17   // Email access control implicit
18   email::send(email, body);
19
20 }
```

(a) Without Sesame.

```
1  fn submit_homework(request: SesameRequest) {
2    let uid = authenticate(request);
3    // email: PCon<String, ...>
4    let email = SesameDB.lookup(uid);
5    // answer: PCon<String, AnswerAccessPolicy (§4.3.1)>
6    let answer = request.answer;
7    SesameDB.insert(answer);
8    // body: PCon<String, AnswerAccessPolicy>
9    let body = sesame::privacy_region(answer,          // §4.5.1
10     |raw_answer: String| format!("submitted {}", raw_answer)
11   );
12   // Sesame checks that email meets AnswerAccessPolicy.
13   let context = Context { email };                  // §4.3.2
14   sesame::critical_region(body, context,            // §4.5.3
15     #[signed(Kinan Dak Albab <babman@brown.edu>, 7459f3da..)]
16     |raw_body: String, context: Context::Out| {
17       // Reviewer checks context.email used as recipient
18       email::send(context.email, raw_body);
19     });
20 }
```

(b) With Sesame, privacy regions highlighted.

Figure 4.1: An HTTP endpoint for submitting homework answers, implemented (a) without Sesame and (b) with Sesame. Using Sesame, the developer must use privacy regions to operate on data in PCons. Sesame verifies the first region via static analysis (green), but the region that sends an email requires a critical region (yellow), which a code reviewer must sign.

Each type might be governed by a different policy. For example, a policy for the answer might allow only the student themselves, TAs, and the instructor to view the submission.

We now look at how the developer uses Sesame to implement this endpoint with compliance guarantees (Figure 4.1b). The developer invokes Sesame-enabled libraries, as mandated by their organization, to look up the email address in the database (line 4) and to access the answer in the HTTP request (lines 1, 6). As these libraries are Sesame-enabled, they return data wrapped in a *Policy Container* (PCon; see §4.4). They also accept PCons as input, *e.g.,* to insert the PCon-wrapped answer directly into the database (line 7). A PCon keeps the underlying data private and inaccessible to the application and associates it with a policy: *e.g.,* the answer is protected by an `AnswerAccessPolicy`. §4.3.1 explains how developers write policies and associate them with data.

Now, the developer needs to construct the email body. This is application-specific functionality unavailable in a Sesame-enabled library, and it operates on the answer data. However, accessing the answer directly causes a compiler error, as the answer is a private member in a PCon. Instead, the developer must use a privacy region. They invoke Sesame's `privacy_region` API, passing the PCon along with a closure to execute on the raw answer (lines 9–11). Sesame's static analysis (§4.5.1) verifies

that the closure is leakage-free, so it runs as-is on the raw data. Sesame wraps the output of the closure in a PCon with an identical policy to the input.

Finally, the developer emails the body they constructed to the student using a third-party email library. The contents of `body` are inside a PCon, so the developer could again invoke `privacy_region` with a closure that sends the email. However, this closure intentionally leaks data via email, so it is clearly not leakage-free. The developer informs Sesame of this by instead using a *critical region (CR)* (lines 14–19). Before executing a critical region, Sesame checks the associated policy relative to a developer-provided *context*. In our example, this context contains the recipient's email address, and Sesame checks that `AnswerAccessPolicy` allows sending the answer to that email address. A code reviewer, *e.g.,* a team lead or a policy engineer, must now manually review the critical region and ensure that it sends the email to the address in `context` that passed Sesame's policy check. §4.3.2 describes Sesame's policy check and the review workflow.

In general, reviewers must verify that the CR uses the context and acts in ways that are consistent with it, and then sign the region. During a release build, Sesame validates the signature (§4.5.3). If the CR's code or any of its dependencies change, validation fails. If the developer failed to see the need for a CR on line 14 and used `privacy_region` instead, Sesame's static analysis would reject this privacy region. The developer then would have to decide whether they believe the rejected region to be leakage-free. If so, they can run it in a sandbox (§4.5.2); if not, they use a CR as shown.

Developers may implement policies concurrently with the application, or use placeholder policies first and then add the policy logic later; §4.7 discusses porting existing applications.

4.3 Design

Sesame's policy enforcement is concerned with the application's sources and sinks, which correspond to where data enters and leaves the application. When the application reads from a source, Sesame attaches a policy to the data. Sesame's design then ensures that policy remains attached to the data, including derived data, as it flows through the application (*i.e.,* taint-tracking). Sesame only allows data to leave the application at a sink if its associated policy check succeeds.

Sources. Sesame provides built-in support for common sources such as HTTP requests and SQL databases. Sesame reads data from those sources, places it in a policy container (PCon), attaches the

```
1  #[sesame::get("/view/<answer_id>", auth="student")]
2  pub fn view_answer(
3    student: Student, // authenticates via cookie
4    answer_id: PCon<u8, NoPolicy>,
5    context: Context) -> sesame::HTMLTemplate {
6    // rows : Vec<sesame::PConRow>
7    let rows = SesameDB.query(
8      "SELECT * FROM answers WHERE id = ?
9       AND author = ?", (answer_id, student.email),
10      context);
11   // answer: PCon<String, AnswerAccessPolicy>
12   let answer = rows[0]["answer"];
13   // Render the answer. Sesame automatically checks
14   // associate AnswerAccessPolicy here.
15   sesame::render("answer.html", answer, context)
16 }
```

Figure 4.2: An endpoint for students to view their answer uses `answer_id` from the HTTP request to look up the answer in the DB. Sesame only reveals the PCon-wrapped answer if the policy allows the signed-in student to view it.

appropriate policies to it, and returns it to the application. For example, Figure 4.2 shows an endpoint for students to view their homework answers. `answer_id` comes from a Sesame HTTP source (line 4), and `rows` comes from a Sesame database source (lines 7–10), so the application receives data in PCons. Attempts to read raw data from these sources will cause a compile-time type error.

Sources not in Sesame-enabled libraries provide raw data, which developers must manually place in policy containers and associate with policies. Organizational rules are responsible for ensuring that developers do this correctly: e.g., requiring developers to use a custom I/O module that associates policies to the data, rather than reading files directly.

Sinks. Similarly, Sesame has built-in support for common sinks, such as rendering HTML templates (line 15), or passing data to the DB for reads (line 9) or writes. Sesame receives policy containers from the application, checks the associated policies, and sends the data to the sink if the policy permits. Releasing data to custom sinks unsupported by Sesame requires critical regions and code review. For these sinks, developers explicitly provide Sesame a *context* that describes the intended use. Sesame checks that the policy associated with data accepts the context before running the critical region that touches the custom sink. Code reviewers must review and sign CRs to ensure that they match the context.

```
1  #[db_policy(table = "answers", columns = ["answer"])]
2  struct AnswerAccessPolicy {author: String, lecture: u64}
3  impl Policy for AnswerAccessPolicy {
4    fn check(&self, context: Context::Out) -> bool {
5      let email: String = context.email;
6      email == self.author || is_instructor(email)
7        || is_discussion_leader(email, self.lecture)
8    }
9    fn join(self, other: Self) -> Self { ... }
10 }
11 impl DBPolicy for AnswerAccessPolicy {
12   fn from_row(row: &MySQLRow) {
13     AnswerAccessPolicy {
14       author: row.get("author"),
15       lecture: row.get("lecture_id"),
16     }
17   }
18 }
```

Figure 4.3: The CHECK function (lines 4–8) of `AnswerAccessPolicy` only allows sending an answer to an email, from `context.email` (line 5), if it matches the answer's author, the instructor, or a discussion leader. Line 1 binds the policy to the `answer` column; lines 12–16 instantiate it.

```
1 let context = Context { email:  "someone@else.com" };
2 sesame::critical_region(body, context,
3   // Sesame invokes policy check with context prior
4   // to calling the closure. Policy check fails.
5   #[signed(..., 7459f3da..)]
6   |raw_body: String, context: Context::Out| {
7     email::send( context.email , raw_body);
8   });
```

(a) A signed critical region correctly uses its context, but the email in the context is unauthorized and Sesame's policy check fails.

```
1 let context = Context { email:  answer.author };
2 sesame::critical_region(body, context,
3   // Policy check would pass on this recipient, but the
4   // region uses an email address not from the context.
5   // Reviewer refuses to sign region.
6   |raw_body: String, context: Context::Out| {
7     email::send( "someone@else.com" , raw_body);
8   });
```

(b) Sesame's policy check passes on an authorized email, but the critical region mismatches its context, so the reviewer rejects it.

Figure 4.4: Buggy alternative implementations for Figure 4.1. Sesame rejects both via policy checks (a) or code review (b).

4.3.1 Policies

Developers express each policy type as a Rust struct and implement Sesame's POLICY trait for this struct. This trait requires providing a CHECK function that Sesame invokes before revealing data associated with that policy at a sink. The CHECK function may use the context as well as metadata stored inside the policy itself, and can execute arbitrary code (e.g., query a database). If the policy check fails, Sesame reports an error; otherwise, it releases the data to the sink.

Figure 4.3 shows a policy for student homework answers. The policy allows revealing an answer only to its author, the instructor, or students who are discussion leaders for the lecture (identified via a

database query).

Associating Policies with Data. Application developers must associate policies with the data read from application sources. Developers must explicitly associate insensitive data that is *intentionally* not covered by any policy with NOPOLICY (*e.g.,* Figure 4.2, line 4). For sources with a structured schema, application developers specify the policy associations decoratively for that schema: *e.g.,* Figure 4.3 associates `AnswerAccessPolicy` with the `answer` column in table `answers` (line 1). Applications declare the associated policies when they read data from unstructured sources, such as a cookie or GET parameter. Developers must implement constructors to create instances of a policy for each type of source to which they attach the policy. Since Sesame trusts policy code, it passes raw data to policy constructors.

Policy Conjunction. Sesame combines policies when combining PCONs, *e.g.,* when executing a privacy region over a vector of PCONs. If the policies have different types, Sesame generically combines them by *stacking* them. The stacked policy stores all source policies, and checks all of them in its CHECK function. If all the policies have the same type, Sesame combines them using a policy-specific JOIN, if policy developers implement one (Figure 4.3, line 9). Joining and stacking must be semantically equivalent, but joining may result in more compact policies that are faster to check.

4.3.2 Context and Policy Checks

Sesame invokes policy checks when: *(i)* a policy container reaches a Sesame-enabled sink; and *(ii)* before running a critical region on the data in a PCON.

Policy checks always happen relative to a *context*. Contexts contain summary information about the associated source or sink. They are immutable objects, either provided by Sesame or created by the developers themselves. The content and API of a context is application-specific: *e.g.,* the homework submission example identifies users by emails, but other application contexts might contain user IDs or OAuth tokens. Contexts can store sensitive information in PCONs, which Sesame replaces with raw data when invoking CHECK and critical regions (*i.e.,* CONTEXT::OUT; see §4.4 SESAMETYPE).

Contexts created by Sesame are trusted, *e.g.,* they correctly identify the authenticated user. Developers may pass them to built-in Sesame sinks, which use them for policy checks without any developer intervention (Figure 4.2, line 15). By contrast, Sesame only allows custom contexts with custom sinks, and relies on manual review to ensure the sink's behavior is consistent with the context.

Example. Figure 4.1b contains a critical region to send emails (a custom sink). The application creates a custom, untrusted context that indicates the *intended* recipient of the email (line 13). The application then invokes a critical region on `body` with that context (line 14). Since `body` is associated with `AnswerAccessPolicy` (line 8), Sesame first invokes that policy's CHECK function (Figure 4.3, lines 4–8) with the provided context, and executes the region only if that check succeeds. CHECK ensures that the `email` provided in the context belongs to the author, the instructor, or a discussion leader. A reviewer must ensure that the critical region indeed sends an email to the intended address (Figure 4.1b, line 18).

A buggy implementation might violate this policy in two ways. The application could provide Sesame with a context that contains an unauthorized email (Figure 4.4a). This causes the policy check to fail, so Sesame never executes the critical region. Or the application may provide a context with an authorized email when the code in the critical region sends an email to a different address (Figure 4.4b). This is an example of a custom context that misrepresents the sink. Here, the policy check would succeed, but a careful reviewer refuses to sign the critical region, because it does not email `context.email`, and Sesame errors on a release build.

Importantly, the reviewer merely verifies that the critical region uses `context.email`, which passed a policy check, and therefore Sesame guarantees it is authorized.

4.3.3 Guarantees and Threat Model

Sesame views policy definitions as ground truths that specify desired application behavior. This includes each policy's CHECK and JOIN functions, constructors, and associations with data at sources. Sesame also relies on the soundness of the Rust type system, its compiler, and the correct implementation of Sesame components, all of which are trusted. Subject to the assumptions below, Sesame guarantees that data can only be revealed at a sink whose context passes the policy check associated with the data's origin.

Proper Usage. Organizations must mandate the proper use of Sesame, *e.g.,* via code review, linters, or other best practices. Specifically, developers must *(i)* use Sesame's built-in sources when applicable, *(ii)* correctly wrap data from custom sources in policy containers, and *(iii)* compile their applications with Sesame's toolchain.

Accurate Review. Sesame assumes that reviewers carefully vet critical regions and only sign them after

ensuring that the regions are consistent with their specified contexts.

Unsafe Rust. Rust guarantees encapsulation for applications that operate solely within safe Rust. However, applications often, directly or in dependencies, invoke unsafe code for which Rust provides no encapsulation guarantees. Sesame implements additional protections to ensure data in PCONs remains inaccessible even to unsafe Rust code (§4.4), assuming that such code is buggy but not outright malicious. For example, Sesame protects against a logging library that uses unsafe code to byte-wise dump provided objects, but not unsafe code that dumps the entire memory of the application process (in line with other IFC systems [LKB+21; ML00; YWZ+09]).

Implicit Leakage. Sesame protects against direct leakage and implicit data-dependent control flow. Application code cannot perform control flow on data wrapped in policy containers without using privacy regions, which have mechanisms for mitigating data-dependent control flow (§4.5). This also ensures that observable data-dependent interactions (*e.g.,* with a database) only occur following a successful policy check. The one exception is certain FOLD APIs that Sesame provides for ergonomic reasons, which can leak information about the shape of some data types. Developers can disable them for data with restrictive policies (§4.4). Timing and micro-architectural side channels are out of scope.

4.4 Policy Containers

Sesame's policy containers are a generic data type, PCON<T, P>, that wraps two private members: data of type T, and a policy object of type P. The data wrapped by a PCON is private and can only be accessed or manipulated by Sesame and never by application code, except through a privacy region. This guarantees application code cannot leak such data without going through Sesame's checks. Using policy containers also guarantees that a policy associated with data at a source remains associated with that data, including derived data, throughout the application.

PCONs are regular Rust data types. Application code can pass or return PCONs to and from functions. It can store PCONs inside vectors and other collections, and it can define structs that have PCON fields. While PCONs simplify policy tracking and checking, they prevent application code from directly computing over the wrapped data. Therefore, a core challenge in Sesame is to provide application developers with tools and abstractions to allow them to operate on data inside PCONs. PCONs provide builtin functions for common operations, such as type conversions (*e.g.,* from a PCON<U32, P> to a

API Level	Supports	Guarantees	Root of Trust	App. Developer Effort
Built-in	Common primitives	Static (taint tracking) Runtime (policy checks)	Rust + Sesame	Use PCon<T, P> instead of T with compiler guidance
Verified Region (VR)	Statically-verified leakage-free closures	Static analysis (sound but incomplete)	Rust + Sesame	Check that Sesame accepts closure as leakage-free
Sandbox Region (SR)	Leakage-free closures that cannot be verified statically	Runtime (sandbox)	+ RLBox [NDG+20]	Engineering setup
Critical Region (CR)	Arbitrary closures including sinks	Code signing	+ reviewers	Authorized developers review and sign closure

Figure 4.5: Sesame's API levels. Built-in libraries and verified regions require a minimal root of trust. Sandboxed regions support more complex code at the cost of runtime overhead. Critical regions support arbitrary code, but rely on code review.

PCon<String, P>). Developers must use privacy regions to perform more complex tasks on data in PCons.

PCon Layout. Rust guarantees that safe Rust code cannot access the private members of a PCon. However, unsafe Rust code can circumvent these protections by using casts or accessing the bytes of the policy container directly. Library code sometimes does this for legitimate reasons, *e.g.,* a data structure that uses `memcpy` for efficient resizing, or a logging library that dumps the bytes of arbitrary objects. To ensure that such libraries cannot accidentally leak the data wrapped by PCons, Sesame stores the data on the heap and references it within the PCon using an obfuscated pointer. Sesame XORs this pointer with a random global secret. This prevents unsafe application or library code from accidentally leaking the data, but cannot protect against actively malicious code that intentionally undoes the obfuscation or scans memory contents, both of which are out of scope. Finally, this layout has performance implications: operations on the PCon require an additional XOR and pointer dereference, and vectors and collections of PCons have worse cache locality than their plain counterparts. These overheads affect wrapping and unwrapping data at privacy region entry and exit, but not the body of a privacy region, which accesses the data directly. This indirection adds a 1.7–2.1× overhead in microbenchmarks, but is negligible for real workloads (§4.8).

SesameType. Sesame handles types with arbitrary nested PCons, such as Vec<PCon<T, P>> or a custom struct with nested PCon fields, by relying on the SesameType trait. For each type X that implements it, this trait defines the corresponding out-type $X::Out$. $X::Out$ mirrors the structure of X but replaces every nested PCon<T, P> with the corresponding T. The trait also defines private conversion functions between the two types, which only Sesame can invoke. Sesame implements this trait

for various monads, tuples, and collections that may contain PCONs, *e.g.,* Vec<X> and Option<X> with out-types Vec<X::Out> and Option<X::Out>. Because this trait deals with raw data, Sesame disallows applications from manually implementing it for their custom types (enforced via custom lints, §4.6). Instead, Sesame provides a macro that applications can use to derive trusted implementations of this trait and its out-type for their custom types.

Fold. For better ergonomics, Sesame provides a FOLD API, which allows applications to move PCONs in and out of data structures. Suppose the application has d: PCON<X, P> where X is some application struct containing $\{a$: $T1$, b: $T2\}$. Applications can use FOLD$(x).d$ to retrieve the field a: PCON<T1, P> ("folding in"). Alternatively, applications can FOLD PCONs "out", *e.g.,* to transform Vec<PCON<T, P>> to a PCON<Vec<T>, P> whose policy is the conjunction of all the input policies. Folding out is always safe, but folding in may leak information about the shape of the underlying data, *e.g.,* the length of a vector or whether an `Option` is NONE or SOME. If undesirable, policy developers can annotate a policy with NOFOLDIN to prohibit folding in on its associated data.

4.5 Privacy Regions

Sesame provides different ways for applications to operate on data wrapped by PCONs. Figure 4.5 summarizes them: "built-in" describes Sesame-enabled libraries, while the other three API levels correspond to privacy regions. Privacy regions allow application code to execute on the raw data. Sesame's static analysis helps the developer determine what type of privacy region to use. At a high level, the analysis checks that a closure cannot leak input or derived data, e.g., by writing to a file or modifying a global or captured variable.

Depending on the static analysis outcome, different types of privacy regions are appropriate:

1. If the static analysis passes, the privacy region is a *verified region (VR)* and runs as-is (§4.5.1).
2. If the static analysis fails, but the developer expects the region to be leakage-free, they can choose to use a *sandboxed region (SR)*, which runs the code in a constrained environment that prevents leakage (§4.5.2).
3. If the developer expects a failing region to have intentional side effects (e.g., because it interacts with a custom sink), they use a *critical region (CR)*, which requires manual code review (§4.5.3).

In general, the `privacy_region` APIs accept a SESAMETYPE X and a closure F: X::$Out \rightarrow Y$ as

arguments. Sesame executes the closure over the input after replacing PCONs with their underlying data. Sesame wraps the result of the closure in a PCON, and associates it with the conjunction of all the policies in the input. Critical regions are the exception; they can return data with a different policy or no policy at all.

4.5.1 Static Analysis and Verified Regions

Sesame's static analyzer, SCRUTINIZER, checks whether a closure passed to a privacy region could leak some of its arguments outside the region. SCRUTINIZER is sound but incomplete: it never accepts leaky privacy regions, but may conservatively reject leakage-free ones.

SCRUTINIZER searches the application code for instances of Sesame's `privacy_region` call. We refer to the closure passed to the privacy region as the *top-level function*. SCRUTINIZER considers each argument to the top-level function to be *sensitive*. Top-level functions may also capture external variables from their environment, but captured variables are not sensitive. SCRUTINIZER accepts a function only if it concludes that the function cannot leak any of its arguments or any data derived from them. This includes leakage via external side effects (*e.g.,* printing to stdout, changing the file system) or via mutating captured variables that other parts of the application can observe (*e.g.,* global variables). SCRUTINIZER checks the top-level function and its callees, direct or indirect, including those in external libraries.

Information Flow. SCRUTINIZER computes a sound over-approximation of the information flow of the arguments and captured variables in the top-level function all the way through call chains to helper and library functions. SCRUTINIZER uses Flowistry [CPA+22] to find flows from a sensitive variable to any aliases within a single function body. We extend SCRUTINIZER with additional analysis to track sensitive variables as they are passed to and returned from other functions. SCRUTINIZER uses dataflow analysis to soundly resolve dynamic dispatch with good accuracy. Thus, SCRUTINIZER identifies all aliases or variables derived from sensitive variables, either directly or implicitly via control flow. SCRUTINIZER considers all such variables to be sensitive as well.

Analysis. SCRUTINIZER checks that the information flow of sensitive variables is contained entirely within the analyzed code. Within Sesame's threat model, information can flow outside a function in three ways:

1. via mutably captured variables or variables derived from such captures;

2. via any unsafe mutation primitives applied to captured variables and variables derived from captures, regardless of their mutability;
3. via functions with unknown bodies that SCRUTINIZER cannot conservatively approximate, such as native code or unresolved generics, unresolved dynamic dispatch, and unresolved function pointers.

SCRUTINIZER catches all three cases. Mitigating the first case ensures that the function cannot mutate external variables in ways that depend on (and thus leak) sensitive arguments. Mitigating the second case covers all forms of interior mutability in Rust, which ultimately rely on unsafe mutation (via transmute or raw pointer dereferences). Mitigating the third case ensures that SCRUTINIZER rejects functions that leak sensitive data via external side effects, such as writing to files or sockets, as they must invoke native code.

Allow list. SCRUTINIZER allow-lists some trusted functions, including certain Rust intrinsics and low-level functions for string formatting and panics. We manually confirmed that these functions are leakage-free.

SCRUTINIZER also allow-lists functions in standard library collections (*e.g.,* `Vec::push`) that take the `Self` parameter as a mutable reference. This is sound, as invoking such a function on a captured collection would require a mutable reference to it, which can only be acquired by mutably capturing it (violating case 1) or via an unsafe conversion from an immutable capture to a mutable one (violating case 2). Since SCRUTINIZER rejects such code, these functions can only be called on local variables, which external code cannot observe. The only remaining risk is the allow-listed functions directly leaking arguments (*e.g.,* by writing to a file), which standard library collections do not. This assumption makes `std::collections` part of Sesame's TCB, an acceptable risk in practice.

Details. SCRUTINIZER first collects Rust's MIR representation of all available function bodies via `rustc` dataflow analysis framework, including all possible variants for dynamic dispatch. Second, SCRUTINIZER ensures all captures are immutable and then uses Flowistry to track information flow through the collected call tree. If it encounters any violations of cases 1–3, SCRUTINIZER rejects the privacy region. Appendix B describes the analysis in more detail.

Discussion. SCRUTINIZER is sound but incomplete for three reasons. First, SCRUTINIZER conservatively rejects functions if it fails to resolve their information flow in its entirety. For example, a leakage-free function will be rejected if it contains dynamic dispatch that SCRUTINIZER cannot resolve. Second,

SCRUTINIZER checks for stronger (but easier to detect) variants of the three cases above. For example, SCRUTINIZER rejects *all* functions that capture via mutable reference, even if they never mutate them based on sensitive values. Third, Flowistry itself over-approximates information flow [CPA+22].

SCRUTINIZER uses Flowistry to propagate sensitivity labels within a function body, but it contributes new dataflow and type analyses that *(i)* propagate labels across functions, *(ii)* handle unsafe code, generics, and dynamic dispatch, and *(iii)* detect mutation, including in implicit, data-dependent ways.

4.5.2 Sandboxes

Developers may choose to run a region that SCRUTINIZER rejects as a sandboxed region, which relies on runtime protections to enforce that the region never leaks sensitive data.

Sesame's sandboxed regions use RLBox [NDG+20], a lightweight sandbox used in Firefox to execute untrusted native libraries. RLBox relies on software-based fault isolation (SFI), which uses inline dynamic checks to restrict memory reads and writes to a memory region allocated at sandbox creation time. This isolates the sandboxed region's memory from the rest of the application. In addition, RLBox forbids system calls, so sandboxed regions cannot externalize data via I/O.

Extending RLBox. RLBox is designed to isolate untrusted libraries from a trusted host application (Firefox). Hence, RLBox allows the application to access the sandbox's outputs and lets the sandbox print to stdout and stderr for debugging. In Sesame, the application is untrusted and the sandbox must not leak any sensitive information to it. We thus align RLBox with Sesame's requirements by: *(i)* modifying the RLBox runtime to forbid printing, and *(ii)* building infrastructure around sandbox invocations to compute the conjunction of all policies associated with the sandbox inputs and wrapping the sandbox output in a PCON with the conjoined policy.

Optimizations. RLBox allocates the entire sandbox memory on sandbox creation, which makes creating and destroying sandboxes expensive. Firefox overcomes this by reusing the same sandbox for invocations in the same trust domain (*i.e.,* the same library and HTTP origin). Sesame sandboxes process data with different policies, making such reuse unsafe: earlier invocations with stronger policies could affect sandbox state that influences later invocations with weaker policies. Instead, Sesame uses a pool of pre-allocated sandboxes, and wipes the sandbox memory after each use to ensure isolation across invocations. This involves zeroing out the sandbox stack and heap, and restoring global data and metadata

to their initial state from a checkpoint.

Because of sandbox memory isolation, Sesame must copy all input data into the sandbox memory, and vice-versa for its outputs. However, the same datatype may have an incompatible size and memory layout across the application and sandbox, as RLBox runs sandboxes in 32-bit WASM and offsets its address space for isolation. For example, a vector type in the application may store three 64-bit fields: the pointer to the underlying buffer, a length, and a capacity, while the vector type in the sandbox stores all three in 32-bit variables with an offset pointer. Sesame provides a SANDBOXCOPY trait for quickly deep-copying Rust objects to/from sandbox memory, while altering their memory layout and offsetting any pointers (*i.e.,* "pointer swizzling"). Sesame implements this trait for primitives, strings, and vectors, and provides developers with macros to derive the trait for their custom types. For types that do not implement this trait, Sesame falls back on serializing and deserializing data.

4.5.3 Critical Regions

Developers must use a *critical region (CR)* to send data to custom sinks. They may also use a CR to execute a leakage-free region that SCRUTINIZER conservatively rejected, and in rare cases where that region is incompatible with sandboxing or the runtime overheads of sandboxing prove undesirable. Code reviewers manually review CRs for unintentional leakage and for correct use of the region's context.

Review Process. CRs should be concise, single-purpose, and self-contained. Reviewers should reject CRs that are unfocused or overly complex and request that authors simplify them, similar to regular code review.

Sesame executes a CR only after a successful policy check on the input data, given the provided context (§4.3.2). Reviewers thus do not need to reason about the semantics of the associated policies (*e.g.,* which emails are allowed). Instead, they reason about the code of the CR and the semantics of the context object it receives (*e.g.,* does the CR send an email to the address specified in the context). They also need to ensure the CR has no unintentional leaks, *e.g.,* via logging.

Reviewing a CR includes reasoning about library code it calls into. Large companies have existing procedures for approving and updating dependencies; for example, Google curates approved dependencies and versions that developers are allowed to use [Goo23]. A reviewer of a CR that invokes an approved dependency need only check that the usage of the dependency is consistent with its documented API, and

relies on the curation process to ensure the sanity of the docs. Open source projects or smaller companies may not have the capability to perform such detailed vetting. In such cases, reviewers must look up each dependency they encounter in a CR, and decide how strictly they review it based on reputation, bug reports, or other criteria.

Sesame requires reviewing and re-signing a CR when its code or dependencies change. Since Rust locks the dependency versions in an application's `Cargo.lock` file, dependency updates are explicit, intentional, and generally rare.

Signatures. Reviewers indicate to Sesame that they approve a CR by signing it. Signatures serve two purposes: *(i)* they verify to Sesame that the CR has indeed been approved by an authorized reviewer, and *(ii)* that the CR and its dependencies are unchanged since the review. During review, Sesame produces a hash of the CR. Reviewers sign that hash and attach their signature to the CR. During each subsequent release build, Sesame reproduces a hash for that CR using the same procedure, and checks that the signature attached to the CR is a valid signature for that hash. The hash differs if the CR changed since review, including changes to helper and library functions. This in turn invalidates the signature, which prompts Sesame to reject the CR and require a reviewer to re-vet and re-sign it.

Hashing. Sesame builds a call graph for the CR similar to §4.5.1, but stops at calls to external dependencies. Sesame concatenates the source code of all functions in the call graph into a normalized string (*e.g.,* without comments and extraneous white spaces). Sesame then records the external dependencies the region calls into and traverses the `Cargo.lock` file to find the exact versions of these dependencies and any transitive dependencies. Sesame augments the CR string with the dependency information, and hashes it. Changes to the application portion of the CR, or to direct or transitive dependencies, result in a different hash. Updates to dependencies or application code unrelated to the CR do not affect the hash and avoid superfluous review.

Ergonomics. Sesame omits signature checks in debug mode, which allows developers to implement and test their CRs without review. Then, authors request signatures from reviewers, who must review these regions, *e.g.,* as part of a pull request. This mirrors existing industry practices that require approval by authorized reviewers for merging code.

Our prototype uses GitHub as a key provider and for identity management. Sesame pulls public keys

for reviewers from GitHub to validate the signature of each CR during release builds. A reviewer's privileges can be revoked, and Sesame can either invalidate their signatures immediately, or preserve existing signatures while disallowing future signatures if privilege revocation and signatures are timestamped.

4.6 Implementation

We implement a Sesame prototype in 12k LoC in Rust, including 4.2k LoC for SCRUTINIZER, 2.1k for Sesame's web framework, and 0.5k for Sesame's MySQL library. The web framework and MySQL library mirror the APIs of Rocket.rs [Ben24] and mysql [bla18], modified to accept PCONs at sinks and generate PCONs at sources. Sesame also has partial support for SeaORM [TB21]. All this code is trusted, as is RLBox (for SRs).

RLBox is primarily designed for sandboxing C++ functions; Sesame generates the necessary wrappers and bindings to use it with Rust. Our prototype uses RLBox with WebAssembly (WASM), and thus does not support code and libraries incompatible with WASM in sandboxed regions.

Sesame provides mock versions of its built-in sources and sinks for end-to-end application tests. These versions strip policy containers from application outputs, and allow testing code to create synthetic contexts to test policy CHECK functions. Sesame uses Rust's conditional compilation to ensure these features are only available in tests.

Finally, Sesame includes linting rules it checks when compiling in release mode. These forbid developers from manually implementing SESAMETYPE on their custom types, and instead force them to use Sesame's `[#derive]` macros to generate automatic and correct implementations.

Our prototype's hashing of CRs is sensitive to some syntactic changes to code that have no semantic effect (*e.g.,* renaming a variable), which invalidate signatures. Better stable code hashing techniques could improve precision [DRF+17].

4.7 Application Case Studies

We applied Sesame to four web applications: *(i)* WebSubmit [Sch20] and *(ii)* Portfolio [JP22] are pre-existing applications we ported to Sesame; *(iii)* Voltron is an application from Storm [LKB+21] that lets group of students collaboratively edit a piece of code with instructor oversight; and *(iv)* YouChat is a simple chat application for individuals and groups. The original versions of WebSubmit and Portfolio have 1.3k and 5.1k LoC. We built a Rust version of Voltron with comparable functionality to the original

LiquidHaskell application. For Voltron and YouChat, we first built idiomatic Rust implementations without Sesame in mind (0.5k and 0.8k LoC) and then ported these implementations to Sesame. This section describes the applications' policies and our process porting them to Sesame, while §4.8.1 quantifies developer effort.

Policies. We added policies for access control, purpose limitation, user consent, and aggregates to the applications.

YouChat has a single access control policy: users can only view messages that they sent or received, or messages from groups they are members of.

For **Voltron**, we implemented all of the policies from Storm [LKB+21]: *(i)* only admins can enroll new instructors; *(ii)* students can only be enrolled into a class by that class instructor; and *(iii)* code buffers can only be read or modified by students in the corresponding group or by the class instructor. The last policy turns into two Sesame policies that cover reads and writes. We also added two additional policies: *(i)* Firebase authentication headers from HTTP requests may only be used for read database queries; and *(ii)* endpoints may only use the email address of the authenticated user.

WebSubmit is a homework submission system similar to our example in §4.2. Prior to porting to Sesame, we extended WebSubmit with a machine learning model over students' grades (training and inference), and with aggregate statistics for university managers and employers, as well as a user consent choice to release the latter. WebSubmit has six policies: *(i)* a student's answer is only accessible to the student, instructor, and discussion leaders for the corresponding lecture; *(ii)* an individual grade is only accessible to the student and instructor; *(iii)* a student's average grade and email are only released to employers if the student consents; *(iv)* a student's data can only be used for model training if the student consents; *(v)* university administrators cannot aggregate over protected demographic data; and *(vi)* aggregate grade data released must contain grades from at least k different students (min-k).

Portfolio is a high school admissions system deployed in the Czech Republic [JP22]. Candidates create accounts, input personal information, and upload PDF documents for admission review; Portfolio encrypts the stored data at rest. We add two policies to Portfolio, which cover the most sensitive data it handles: *(i)* sensitive candidate data, in either plain or ciphertext form, is accessible only to the candidate themselves and to school administrators reviewing their application; and *(ii)* private keys are never

revealed outside of the DB, except in cookies to their respective owners.

Porting Experience. To port these applications to Sesame, we first implemented the policies, then associated policies with data at sources, and adapted application code to use PCONs instead of raw data. Finally, we used Sesame to check verified regions, and reviewed and signed any CRs.

Swapping Sesame-provided libraries for the web framework (Rocket) and database connector (SeaORM in Portfolio, MySQL in others) made the applications fail to compile, as the libraries now provide PCONs in, *e.g.,* HTTP request data, but the application expects them to be raw types. To fix these compiler errors, we replaced these raw types with PCONs with an associated policy: for structured data, this was a quick update to database schemas; for unstructured data, we had to change code that obtains it from Sesame libraries.

These changes sufficed for simple endpoints (*e.g.,* Figure 4.2), but endpoints that compute on data in PCONs still had compiler errors. We fixed these errors by introducing privacy regions, akin to Figure 4.1b. Rust made this process of lifting code into privacy regions easy, as idiomatic Rust already encourages closures (*e.g.,* in iterator chains).

With the application building and tests passing in debug mode, we compiled in release mode to invoke SCRUTINIZER and Sesame's lints. SCRUTINIZER accepted most regions as verified regions; for the remainder, we found the distinction between sandboxed regions and CRs obvious (*e.g.,* hashing a password vs. sending an email). SCRUTINIZER rejected six regions that use an encryption library that relies on async Rust, even though they are in fact leakage-free. We attempted to make them sandboxed regions, but the library is incompatible with WebAssembly, so we had to turn these regions to CRs and review them manually. Replacing the library with a compatible alternative could avoid these CRs.

Anti-Patterns. We found two problematic code patterns. First, because SCRUTINIZER currently lacks support for async Rust, it rejects regions that contain `await`. To overcome this, we perform operations inside the privacy region without `await` and return a PCON that wraps a future. PCON has an API to `await` a wrapped future outside of privacy regions; this is safe because the result remains wrapped in a PCON.

Second, some endpoints in Portfolio and WebSubmit early-return, *e.g.,* on failed form validations. But early-return checks inside privacy region closures cannot return from the surrounding function.

Application	Policy count	App LoC	...of which policy	...of which CHECK
YouChat	1	1.1k	118	38
Voltron	6	1.2k	425	121
Portfolio	2	6.7k	305	85
WebSubmit	6	2.2k	373	72

Figure 4.6: Policy code size scales with the complexity and number of policies, rather than the size of the application.

Instead, we return a RESULT<T> from these privacy regions, which Sesame wraps in a PCON with the appropriate policy. Sesame's FOLD API (§4.4) lets the application fold it into a RESULT<PCON<T, P>>. This allows early-return when the RESULT is an error, but creates a channel for implicit leakage. Policies can disable this if unacceptable, forcing the remaining code to operate on the RESULT monad instead and defer the early return.

4.8 Evaluation

We evaluate Sesame with four applications: YouChat, Voltron, Portfolio, and WebSubmit. We ask three questions:

1. What developer effort does using Sesame require? (§4.8.1)
2. What is the impact of using Sesame on end-to-end application performance? (§4.8.2)
3. What is the impact of key Sesame components on its correctness and performance? (§4.8.3)

Our benchmark machine has a Xeon E3-1230v5 CPU (3.4GHz) and 64 GiB RAM. We use Ubuntu 20.04, Rust nightly-2023-10-06 for sandboxes and nightly-2023-04-12 for static analysis.

4.8.1 Developer Effort

We evaluate the developer effort required to implement applications with Sesame or to port them as described in §4.7.

Implementing Policies. A critical component of writing an application with Sesame is implementing policies. Ideally, the size of these policies and the effort required to implement them should depend on the complexity of the policies themselves, rather than application size.

We thus measured the size, in LoC, of the policies in our four applications. Policy code includes the policy structs, constructors, the `Policy` trait implementations, and the CHECK functions. The CHECK

Application	Region	# of regions	Total % of App	Size (LoC)
YouChat	VR	3	<1%	1–5
Voltron	VR	3	<1%	1–2
	CR	2	<1%	3–7
Portfolio	VR	43	1.2%	1–8
	SR	6	<1%	1–5
	CR	20	1.4%	1–27
WebSubmit	VR	17	2.0%	1–9
	SR	2	1.0%	4–19
	CR	2	1.3%	8–22

Figure 4.7: Counts and sizes of each Sesame privacy region used. This accounts for the size of the top-level region closure, but not helper functions that require no porting effort.

function typically dominates the developer effort for policy authoring.

Figure 4.6 shows the results. Policy complexity varies based on the diversity of application user roles and purposes of data use. For example, Voltron (1.2k LoC total) is a small application, but it contains a complex hierarchy of user roles, so its policy code is comparatively large (425 LoC). By contrast, Portfolio is the largest application (6.7k LoC), but it has fewer, simpler policies, as it collects broad data (academic history, letters, demographics) but for the same purpose (viewing by admissions officials). This indicates that effort for writing Sesame policies reflects the complexity of the application's data-use semantics, rather than application size.

We compare the three policies Storm [LKB+21] implements for Voltron with the equivalent policies in Sesame. In Sesame, the CHECK functions for these policies consist of 88 Rust LoC, compared to 37 and 17 LiquidHaskell LoC for policy and "non-trivial type annotations" in Storm. This suggests that writing policies for Sesame requires comparable effort to existing work. In addition, Sesame can express more diverse policies, such as min-k or k-anonymity, that depend on runtime state.

Using Policy Containers. Writing a Sesame application requires developers to use PCONs to associate data with policies, to change types to PCONs where necessary (*e.g.,* in function signatures), and to add privacy regions. To quantify this effort, we consider our experience porting Portfolio to Sesame. Porting an existing application is more work than writing a new one that already anticipates these abstractions.

Porting Portfolio took 30 person-hours. We changed types in five ORM and four JSON payload structs to associate policies with structured data. For unstructured data, *e.g.,* cookies and GET parameters, we

Applications	LoC	# CRs	Burden %	Avg Burden
YouChat	1.1k	0	–	–
Voltron	1.2k	2	<1%	5 LoC
Portfolio	6.7k	20	5.0%	16.8 LoC
WebSubmit	2.2k	2	1.5%	16.5 LoC

Figure 4.8: Critical region count in applications. Burden % indicates the fraction of code that needs review; average burden is the average size of in-crate code for CRs.

declared the policy type on each read. We spent the majority of porting time adjusting function signatures to use PCons. This is largely mechanic and guided by compiler type errors that indicate where changes are needed. Compiler errors related to PCons also pointed us to application logic that requires raw data, and we lifted this logic into privacy region closures. Across our applications, this required no structural changes (*e.g.,* moving code between functions or changing control flow). Sesame requires a few dozen privacy regions for Portfolio, and smaller applications require fewer (Figure 4.7). Overall, we had to lift 1–4.3% of application code into region closures.

Critical Region Review. We now consider the review effort for critical regions. A good result for Sesame would show that critical regions are slim and infrequent.

Figure 4.8 shows the number of critical regions and their review burden in terms of in-crate code to audit. In all applications, critical regions are small and shallow: the review burden in Portfolio makes up 5% of application code including the CR closures and all their in-crate helpers. Portfolio has 20 CRs with an average review burden of 17 LoC each.

CRs often invoke external dependencies in addition to in-crate code, so the review burden extends to auditing these dependencies. Reviewers may leverage organizations' existing supply chain audit protocols for approving dependencies and updates to reduce review burden.

These results suggest that Sesame focuses reviewer attention on infrequent, small, and contained code regions.

4.8.2 Application Performance

Next, we evaluate Sesame's impact on end-to-end application performance using WebSubmit and Portfolio. PCon encapsulation adds overhead due to pointer indirection and the additional memory footprint of policies; runtime policy checks and the use of sandboxes may also incur overhead. We

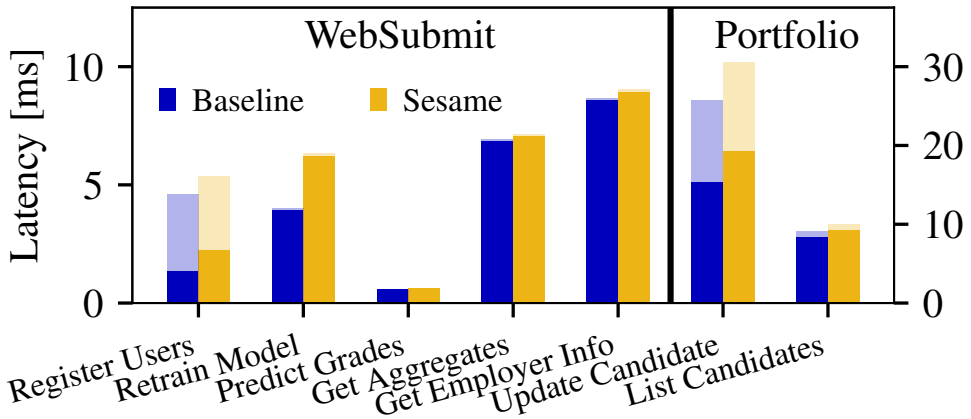
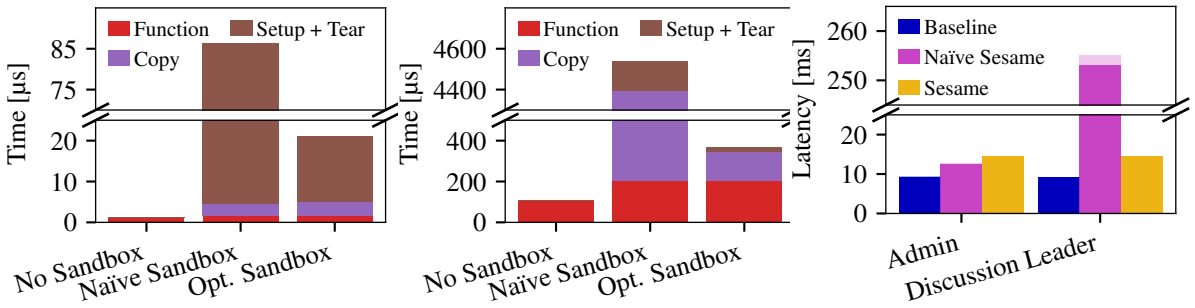


Figure 4.9: Sesame has reasonable performance overheads for WebSubmit and Portfolio (solid: median; shaded 95th %-ile).



(a) Reuse optimizations speed up a sandbox running a cheap hash function by 4×.

(b) Copy optimizations speed up a sandbox running ML training by 11×.

(c) Policy composition avoids repeated policy checks (median; 95th%-ile shaded).

Figure 4.10: Drill-down experiments: Sesame benefits from sandbox reuse reducing setup/teardown overhead (a) and from direct copies avoiding serialization of data passed into the sandbox (b); policy composition reduces check overheads (c). Functions run 2× slower in the sandbox because of the WASI runtime [Cla19] and inserted dynamic checks, *e.g.,* on pointer dereferencing.

compare *(i)* the baseline application, and *(ii)* application with Sesame. We use a database with 100 students and 100 homework questions for WebSubmit and an admission cohort of 1k candidates (one application each) for Portfolio. We measure endpoint latency for HTTP endpoints that use privacy regions and/or require policy checks. A good result for Sesame would show comparable latencies with and without Sesame and acceptable overhead for sandboxed operations.

Figure 4.9 shows the results. WebSubmit performs API key hashing during user registration ("Register Users") and ML model training ("Retrain Model") in sandboxed regions. These endpoints with sandboxed regions see 10% and 55% overhead, most of which is due to the cost of copying data into the sandbox. User registration (10% overhead) copies a single record into the sandbox (tail latency is due to DB disk

I/O); retraining the ML model transfers the grades of all consenting users to the sandbox (55% overhead). "Get Aggregates", which computes statistics over the whole class, and "Get Employer Info", which computes average grades for release to employers, have a 1–3% overhead. Both endpoints combine many students' data, which can have different policies that cannot be folded together into a single policy, so Sesame must perform repeated policy checks. "Predict Grades" has a 10% overhead, albeit with low absolute latency as grade inference operates on an in-memory model without I/O.

Portfolio's "Update Candidate" writes candidate demographics to the DB on disk. It performs JSON serialization in a sandboxed region, resulting in a 25% overhead. "List Candidates" has admins retrieve a paginated list of 20 applications. Sesame uses FOLD to combine the candidates' policies into a single policy check, resulting in a 10% overhead.

These results indicate that PCONs and policy checks impose low overheads, while the cost of sandboxing scales with the size of the data copied into the sandbox.

4.8.3 Drill-Down Experiments

We now evaluate key components of Sesame—SCRUTINIZER, sandboxes, and policy composition—in isolation.

SCRUTINIZER. Sesame's guarantees depend on sound rejection of leaking regions. Simultaneously, if SCRUTINIZER accepts genuinely leakage-free regions, this avoids sandboxing overhead and manual review. We evaluate SCRUTINIZER on 98 privacy regions across our four applications: 80 that we manually verified as leakage-free, and 18 that we know to be leaking. In a good result, SCRUTINIZER would accept most leakage-free regions and reject all leaking regions.

Figure 4.11 shows the results of running SCRUTINIZER over regions we know to be leakage-free. SCRUTINIZER successfully verified 66 of 80 regions, but conservatively rejected two regions in WebSubmit and twelve in Portfolio. Six of the rejected regions use async code not currently supported by SCRUTINIZER. The remaining eight regions perform cryptographic hashing, encryption, ML training, and CSV serialization via external libraries that dereference raw pointers for performance. With extra engineering effort, SCRUTINIZER should be able to verify some of these as leakage-free. SCRUTINIZER correctly rejects all 18 leaking regions.

We also evaluated SCRUTINIZER on methods from standard library containers—a challenging test, as

Application	Privacy Regions: *Leak-free*	Privacy Regions: *... of those, accepted*	Functions Analyzed	Time
YouChat	3	3	823	2.31s
Voltron	3	3	11	0.80s
Portfolio	55	43	774,624	711.61s
WebSubmit	19	17	332,326	500.52s

Figure 4.11: SCRUTINIZER accepts the majority of leakage-free regions, avoiding sandboxing overheads or the burden of manual review. SCRUTINIZER rejected all leaking regions.

the standard library extensively uses unsafe code for performance. SCRUTINIZER rejected all leaking methods, and rejected two out of 57 leakage-free methods.

Sandboxes. To measure the cost of sandboxing, we consider the sandboxed regions in WebSubmit: "Register Users" and "Retrain Model". The former computes a hash over a short string and the latter fits a linear regression model to thousands of rows. We compare the runtime of executing the two privacy regions *(i)* without a sandbox (the baseline); *(ii)* in a sandbox without any optimizations ("Naïve Sandbox"); and *(iii)* in a sandbox with reuse and copy optimizations (§4.5.2).

The results are in Figures 4.10a and 4.10b. Compared to the baseline, the naïve sandbox adds substantial overhead: sandbox setup and teardown dominate the (fast) hashing sandbox's runtime (Figure 4.10a), while data copying via serialization dominates for ML training, which operates over more data (Figure 4.10b). With optimizations, the overhead of sandboxes decreases substantially. Reusing sandboxes after erasing their memory improves the hashing sandbox's runtime by 4×. Copying data and swizzling pointers reduces data copy time by 29× and overall runtime by 11× compared to an naïve, serialization-based ML training sandbox.

In both cases, the actual code of the region itself ("Function") takes roughly twice as long as without sandboxing, in line with overheads reported by RLBox [NDG+20].

Policy Composition. We now measure the performance impact of policy composition using FOLD. We use two endpoints from WebSubmit that display homework answers to course staff and discussion leaders, for a setup with 100k answers (100 students, 100 lectures). Releasing homework answers requires evaluating the `AnswerAccessPolicy`: answers are shared only with authors, admins, or discussion leaders. The list of admins is part of the application's in-memory configuration, while discussion leaders

must be retrieved with a database query. In the discussion leader case, each policy check requires a database query, and joining policies that have the same discussion leaders reduces this policy check to a single database query. The experiment measures the impact of this policy join, and its overhead for the admin case, where the policy check is inexpensive.

Figure 4.10c shows the results: retrieving answers without policy composition for the admin endpoint incurs a $1.4\times$ performance overhead compared to a policy-free baseline, while retrieving answers with policy composition incurs a $1.6\times$ overhead. For the discussion leaders endpoint, retrieving answers without policy composition has a $27\times$ overhead, while the same operation with policy composition incurs a $1.5\times$ overhead. This experiment indicates that policy composition is a worthwhile abstraction: while it incurs some additional overhead when policy check execution is cheap, it cuts the cost of policy checks when execution is expensive.

4.9 Discussion

Critical Regions and Declassification. Classic IFC approaches support declassification of data, which exceptionally allows data to be revealed to principals who otherwise would not be allowed to access it. Declassification is required because *(i)* noninterference is often stricter than the real policies desired by some applications, and *(ii)* IFC systems are almost always too conservative since it is undecidable to precisely capture information flow in real programs [Zda04]. Policy enforcement systems also face similar problems when applications interact with components they cannot analyze or whose semantics are too low-level or tedious to reflect in the policies, *e.g.,* when using libraries. As a result, such systems also provide applications with escape hatches similar to declassification (*e.g.,* filter objects in Resin).

Thus, declassification supports two distinct cases: *(i)* explicit and intended violations of the policy, and *(ii)* behavior that is semantically in line with the policy, but is too difficult for the system to analyze. In either case, application developers must ensure that they use the data as intended after declassification on their own without system support. Another pitfall of this approach is that developers may think that their intended behavior falls under the second case, when, in fact, it violates the policy. In some scenarios, developers may revise that behavior if they are made aware of this discrepancy.

In its design of critical regions (CRs), Sesame aims to tease these two cases apart. By default, Sesame performs an automatic policy check before invoking a CR. This ensures that *the intended behavior* of the CR, as defined by the corresponding context object, is allowed by the policy and thus corresponds

to the second declassification case above. Application developers can use a different Sesame API to execute a CR without a policy check, which corresponds to the first declassification case. In our case studies, we only needed to use this API in Portfolio to intentionally relax the policy associated with a user's admission application after encrypting it, so that we can write the cipher to a file. We discuss future ideas that aim to streamline and simplify such policy transformations in §6.2.3. All other CRs in our case studies correspond to the second declassification case, such as the CR in WebSubmit that sends email receipts when a student submits an answer.

For both CR types, Sesame helps developers ensure that a CR's implementation matches what the developers intuitively intended. First, CRs have a clearly delineated scope and declare their intended behavior in the corresponding context object, which simplifies their review. Second, Sesame's code signature mechanisms protect against inadvertent future changes that may affect a CR or its helper functions after it has been reviewed.

False Positives in Scrutinizer. SCRUTINIZER is sound but incomplete. It never accepts a region that leaks sensitive information, but it may incorrectly reject regions that are in reality leakage-free. We refer to such regions as false positives and give some representative examples taken from our case study applications.

The first category of false positives we encountered is code that uses highly-optimized native libraries. This includes the hashing and encryption libraries used in WebSubmit and Portfolio, respectively, which rely on native implementations and hardware instructions that SCRUTINIZER cannot access or analyze. The second category includes regions where sensitive data flows into `unsafe` Rust. Portfolio contains one such region that performs JSON-serialization via `serde`. The region serializes sensitive data into a string, which Sesame then places into a PCON, and thus is leakage-free. However, internal serialization processes involve incremental writing of bytes to a raw `unsafe` buffer that SCRUTINIZER conservatively rejects, because it cannot determine whether the buffer points to a global variable.

One particularly troublesome category of false positives involves string formatting in Rust. The underlying code in the standard library contains type-erased dynamic dispatch, where the formatter gets transformed into a raw function pointer with a common signature. Instead of rejecting such regions and falling back to sandboxing, we elected to allow-list specific string formatting APIs in the standard library in order to improve developer ergonomics. We manually confirmed that these APIs are leakage-free.

Object Capabilities (OCaps) and PCons. On the surface, one can view a PCons as a kind of imperfect capability. Specifically, if an application has a PCon<T, P> that contains some data $t : T$ and a policy instance $p : P$, then this is proof that an application is allowed to externalize t according to p. Although a policy instance (and therefore their enclosing PCon) may contain state, it often does not contain the entire state needed to determine how its associated data may be externalized. For example, the `AnswerAccessPolicy` shown in Figure 4.3 stores the author's email, but not the discussion leaders that are allowed to access the data.

However, there is a deeper connection between the object capabilities model [Mil06] and Sesame. Sesame forces applications to adhere to the following code pattern: *(i)* Sesame provides applications with input data wrapped in PCons along with their corresponding policies, *(ii)* Sesame forces applications to structure their code into privacy regions by virtue of the PCon API design and the Rust type system. The outputs of privacy regions are also wrapped in PCons with the same (or stronger) policies as their inputs. *(iii)* Sesame allows applications to externalize data out of a PCon after checking the associated policy.

Thus, application components may only posses a PCon containing some data and policy by reading data from sources via Seasme or via a sequence of privacy regions starting from such a source. In OCaps terminology, the first case corresponds to *initial conditions* and *parenthood*, while the later represents a sequence of *endowments* and *introductions*, corresponding to invocations of privacy regions with one or many arguments, respectively. Furthermore, by combining several PCons via privacy regions or Sesame's `Fold` API (§4.4), the application can create a new PCon with a more restrictive conjunction policy, which it can then pass to other application components with attenuated privileges.

We envision supporting more expressive policy transformations that allow for strengthening policies in custom application-defined ways beyond conjunction. These transformations may help in extending Sesame's end-to-end guarantees to applications that invoke remote services (see §6.2.3), and would mirror the membrane forwarder pattern in OCaps.

Several implementations of OCap systems ensure unforgeability by modifying popular languages to prevent various "loop holes". For example, Joe-E [MWC10] is a subset of Java without reflection and native functions, and Caja [MMT10] is a subset of Javascript without global free variables. In contrast, Sesame relies on static analysis and sandboxing to ensure leakage-freedom (which implies unforgeability of PCons) at the level of individual privacy regions, instead of language- or program-wide restrictions.

```rust
fn make_vector(x: &String) -> Vec<String> {
  let y: String = x.clone();
  let mut v: Vec<String> = Vec::new();
  v.push(y);
  v
}
```

(a) A function that clones a String and returns a vector containing the copy.

```rust
fn make_vector<T: Clone>(x: &T) -> Vec<T> {
  let y: T = x.clone();
  let mut v: Vec<T> = Vec::new();
  v.push(y);
  v
}
```

(b) A polymorphic variant of the same function that accepts any cloneable type.

Figure 4.12: Two examples of functions that consist of glue code. a is implemented with concrete types, even-though that concreteness is not required. b provides equivalent functionality but is implemented polymorphically over any cloneable type.

Interestingly, SCRUTINIZER checks for invocations of native and `unsafe` functions, mutation of global variables, and potentially dangerous type casts, which are similar to the restrictions imposed by OCap languages. An open question is whether Rust could enable OCap systems with less restrictions or better ergonomics *e.g.,* by relying on a SCRUTINIZER-like static analysis.

What Constitutes Glue Code? A key aspect of Sesame's design is that it enforces policies over the vast majority of application code with little to no developer intervention by hiding sensitive data inside PCONs. Instead, Sesame focuses its static analysis and sandboxing mechanisms, as well as the developer intervention, on the remaining code. We call the former *glue code* and refer to the latter as *privacy regions*. Note that the notion of glue code characterizes whether a developer needs to use a privacy region, and thus the amount of developer intervention required to implement some functionality using Sesame. Thus, glue code corresponds precisely to code that can be invoked with arguments of type PCON<T, P> and T interchangeably with no more than simple syntactic changes to type declarations in it. This is closely related to the parametricity of the code, regardless of whether the code is actually implemented parametrically or whether the type parameter is "imagined".

Consider a valid Rust function f that passes the Rust type checker. For simplicity, assume that the function takes a single argument of type T and returns a value of type R_T. As a first attempt, let us say that f consists of glue code if we can invoke it on arguments of type PCON<T, P> to acquire results of type $R_{PCon<T,P>}$ (for any P) without any changes. It is easy to see that any f that is parametrically polymorphic over T without any restrictions satisfies this. In Rust, such functions would have the signature `fn f<T>(t: T) ->` R_T.

Furthermore, f also satisfies this definition if it poses type constraints on T that are also met by

PCON<T, P>. For example, Sesame's PCONs implement `Clone` and `Copy` when their underlying types do. In addition, PCONs also inherit certain implementations of `Into` and `From` for standard library types. Thus, a Rust function with signature `fn f<T: Clone>(t: T) ->` R_T also consists of glue code.

Note that application developers often implement their functions with concrete, rather than polymorphic, types even when that concreteness is superfluous. For example, a developer may implement a function as shown in Figure 4.12a, even though they really could have implemented it in a more generic and reusable way as shown in Figure 4.12b. The latter function clearly constitutes the glue code as it can be invoked with arguments of types `String` and PCON<STRING, P> interchangeably.

On the other hand, to invoke the first function on arguments of type PCON<STRING, P>, developers would need to modify its signature to `fn make_vector(x: &PCon<String, P>) -> Vec<PCon<String, P> >` and then update its body accordingly. However, updating the function's body in this case is syntactic: It merely changes the type declarations on lines 2 and 3. Furthermore, given that developers often omit these types and rely on Rust's type inference instead, updating the body may be unnecassary in practice. Thus, we consider the first function to also be glue code with an "imagined" type parameter `T: Clone`.

Contrast this to a function that adds two integers together or prints a string to the screen. Neither function can be invoked on PCONs with only syntactic type changes to its signature and body, since PCONs do not implement addition or the `Debug` trait required to print to the screen.

Sesame's design could be extended to expand what it counts as code glue and thus can handle without developer intervention. Specifically, we can allow PCONs to structurally inherit all traits that their underlying types implement, by applying the trait functions on the underlying data and wrapping the output in a new PCON, akin to a monad. However, this requires further engineering to automatically apply SCRUTINIZER to such trait implementations first to ensure that they are leakage-free, and to avoid inheriting rejected implementations.

4.10 Summary

Sesame is a framework to help developers enforce application-specific privacy policies over user data across an application. Privacy regions allow developers to operate over policy-protected data by leveraging runtime policy checks, static analysis, sandboxing, and human code review.

We show that Sesame requires modest developer effort, incurs acceptable overheads, and provides practical end-to-end privacy compliance guarantees.

Sesame is open-source software and available at:

https://github.com/brownsys/sesame.

CHAPTER 5

Case study: GDPR Compliance in Practice

In previous chapters, we looked at the design and technical capabilities of K9db and Sesame. K9db provides automatic procedures for handling GDPR-style access and deletion requests, and enforces storage invariants to ensure data is encrypted at rest and that the integrity of ownership and access is preserved. Sesame assists developers in ensuring that their application code and business logic adhere to the privacy policies they specify. Sesame supports diverse and rich privacy policies that range from access control to consent, purpose limitation, and aggregation policies.

K9db and Sesame provide building blocks that can ensure that applications comply with the privacy policies that their developers set. These policies may express low-level and internal properties that developers would like their applications to have, but we also envision them including application-specific interpretations of the requirements imposed by privacy laws. For example, the GDPR requires applications to allow users to request the deletion of their data, which K9db can automatically do *if used and configured properly*. GDPR also requires that applications collect data for specific purposes and only use that data for these purposes (purpose limitation), *e.g.,* that applications that state that they only collect phone numbers for two-factor authentication do not, in fact, use them for a different purpose. Sesame can enforce such a policy throughout the entire application end-to-end if the policy is encoded and associated to the relevant data correctly.

In this chapter, we take a closer look at *(i)* how application developers use K9db and Sesame to get compliance guarantees with privacy laws in practice, and *(ii)* how that compares with the current practice which relies on ad hoc and manual enforcement. Specifically, we look at the configuration and socio-technical work that applications need to do in order to comply with laws using these systems.

Generally, this work falls into two categories:

1. Correctly configuring K9db and Sesame with annotations and policies that encode and contextualize the high-level requirements of privacy laws with respect to their application-specific concepts and business logic (§5.1).

2. Integrating K9db and Sesame into their application workflows, including how they interact and interface with end users without a technical background (§5.2).

This chapter showcases what these categories entail by looking at a case study application: WebSubmit. We describe the work we had to do in each of these categories for WebSubmit in order to comply with the specific GDPR requirements discussed in §2.1. We also discuss a hypothetical approach to achieving comparable compliance in WebSubmit without system support and compare the developer burden it imposes to that of K9db and Sesame (§5.3).

This exposition is informed by our experience deploying a K9db-powered version of WebSubmit for the fall 2023[1] and 2024[2] offerings of CS2390, our graduate seminar on privacy-conscious systems, and building a deployment-ready version of WebSubmit that uses both K9db and Sesame that we will deploy for the 2025 offering of the course.

5.1 Configuring K9db and Sesame for Compliance

5.1.1 Schema Annotations

Application developers must annotate their schema with the appropriate K9db ownership, access, and anonymization annotations. K9db handles deletion and access requests according to these annotations and enforces their integrity throughout the execution of the application. Thus, whether an application complies with GDPR-style rights to access and deletion depends on whether the annotations encode reasonable ownership and access semantics consistent with the law.

We note that for a given application, there are usually several compliant ownership-semantics (and thus K9db annotations) that application developers may choose. The law intentionally gives applications some leeway in determining whether to retain some of the user's data and, if so, the extent of its anonymization. For example, a social media application can choose to delete a direct message between two users after either or both of them request deletion and still be compliant, but it is likely to be incompliant if it never

[1] https://cs.brown.edu/courses/csci2390/2023/index.html
[2] https://cs.brown.edu/courses/csci2390/2024/index.html

```
CREATE TABLE lectures (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title TEXT,
    ...
);
CREATE TABLE questions (
    id INT PRIMARY KEY AUTO_INCREMENT,
    question TEXT,
    ...
);

CREATE DATA_SUBJECT TABLE users (
    email TEXT PRIMARY KEY,
    apikey TEXT UNIQUE,
    is_admin INT,
    consent_employers INT,
    consent_ml INT,
    is_remote INT,
    ...
);

CREATE TABLE discussion_leaders (
  id INT PRIMARY KEY AUTO_INCREMENT,
  lecture_id INT REFERENCES lectures(id),
  email TEXT OWNED_BY users(email)
);
CREATE TABLE answers (
  id INT PRIMARY KEY AUTO_INCREMENT,
  lecture_id INT REFERENCES lectures(id),
  question_id INT REFERENCES questions(id),
  author TEXT OWNED_BY users(email),
  answer TEXT,
  grade INT,
  ...
);
```

Figure 5.1: Excerpt from WebSubmit schema showing the K9db annotations we applied. The users table stores profile information for students and teaching staff and represents data subjects. It also stores the user's consent preferences for releasing data to potential employers and participating in our ML experiment. These fields configure our consent-related Sesame policies for WebSubmit. An answer is owned by its author. The `discussion_leaders` table is a many-to-many mapping between students and the lectures whose discussion they are assigned to lead. Records in this table are owned by the student they refer to.

deletes the message. Ultimately, whether a particular way of annotating the schema is compliant is a case-by-case determination that depends on the application business logic, the sensitivity of the data, and the legitimacy of other factors that influence data retention (*e.g.,* compliance with other laws, contracts, or public interest). We discuss a general blueprint that can serve as a starting point for selecting the underlying ownership semantics and annotating the schema for an application.

Blueprint for Access and Deletion Rights. In general, data that is directly provided by a user (*e.g.,* by

submitting a form) or derived from interactions with that user (*e.g.,* interaction history) should be owned by that user. This is why we chose to annotate the `author` foreign key with `OWNED_BY` in the `answers` table in WebSubmit's schema, which is shown in Figure 5.1. When a table only has one ownership annotation point to a single owner, that user has complete and exclusive deletion rights over the data in the table.

Some data is unrelated to any users. This may include configuration data and data that are not derived from user interactions. For example, a list of country codes or currency exchange rates in a shopping application may fall into this category. Another example is the lectures and questions created by the teaching staff, which are required for the core operation of WebSubmit, and should never be deleted.

Data may also be related to other secondary users. Developers can identify this by looking at outgoing foreign keys in the table that stores this data. Developers have three options for foreign keys that lead to such secondary users. First, they can annotate these foreign keys with `OWNED_BY` or `OWNS` if the developers determine that there is a particularly pronounced and legitimate reason to retain such data for the benefit of these secondary users. This bar is higher the more sensitive and explicitly identifying the data is, *e.g.,* it is more reasonable to retain an upvote on some user's post than the phone number associated with that user's account. Second, they can annotate foreign keys that lead to a secondary user with `ACCESSED_BY` or `ACCESSES` if the data is somewhat related to or derived from the secondary user, but not to a degree that gives these secondary users the right to control its retention. In either of these two cases, the developers need to determine whether to anonymize some data on deletion or access, respectively. A starting general principle is to anonymize identifying information about a user that requests deletion of jointly owned data, or about other related users when some user issues an access request. Lastly, developers can choose not to annotate such foreign keys to indicate that secondary users have no deletion or access rights, *e.g.,* a re-share or fork of a social media post or code repository. An example of this is the implicit relationship between answers and discussion leaders via the `lectures` table. In our view, discussion leaders do not have access or deletion rights over records in `lectures` or the answers submitted by other students to questions assigned to that lecture.

We believe that the above blueprint is sufficient for a wide range of application scenarios with some exceptions. These exceptions include data that should be retained regardless of the status of any of its related users, *e.g.,* transaction histories for tax compliance. The application developers should make

sure they have a legitimate reason for retaining such data, as enumerated by the law. If so, they can express this by not annotating any of the related foreign keys with `OWNED_BY`, and adding anonymization annotations for sensitive fields that they do not have a reason to retain, if any. Another exception is data that should be deleted as soon as any of its related users request deletion, *e.g.,* a secret chat in a messaging app. Developers can express this using `ON DEL <foreign_key> DELETE_ROW`.

Compliance with Data Security. K9db encrypts the data at rest, which directly satisfies a portion of the data security requirements required by privacy laws, such as GDPR. Furthermore, K9db uses user-specific encryption keys for that user's data, provided that the ownership annotations attached to the schema are compliant as described above. As a result, K9db's deletion mechanism is compliant even in the presence of (encrypted) backups, including cases where these backups are not directly updated during deletion (*e.g.,* for performance reasons). Nevertheless, we recommend that such backups be periodically deleted or updated, but this can happen on a longer time scale (*e.g.,* years), *e.g.,* in order to protect against future adversaries with possible encryption breaking capabilities (*e.g.,* much more powerful future machines or quantum adversaries).

In addition, privacy laws require that applications put other technical measures in place to ensure that data can only be accessed by authorized users. This requirement goes beyond the scope of K9db, and instead falls in the purview of Sesame, as it governs application code, except for one subtle issue. Application developers may mistakenly annotate their schemas with overly-lax access annotations, which may result in K9db returning data that a user should not have access to when they issue an access request. Sesame can also protect against this at runtime if the access request workflow is properly integrated into the application, we discuss this later in §5.2. Furthermore, we propose future extensions that statically find such inconsistencies between the GDPR access semantics declared by K9db annotations and the application-level access control policies used by Sesame (§6.2.2).

5.1.2 Sesame Policies

Application developers can enforce complex privacy policies using Sesame. In our experience, this is powerful enough to express many privacy laws requirements as well as other security and business logic properties that go beyond compliance. To ensure compliance, application developers need to translate the high-level legal requirements to concrete Sesame policies over specific application concepts, *e.g.,* columns in the database or application purposes. At a high level, the policies necessary for compliance within

scope in this dissertation fall under one of data security, purpose limitation, or user consent requirements.

Access Control and Data Security. Laws commonly require ensuring proper user authentication and that data is only accessible to authorized users. As shown in §4.7 and §4.8, developers can express and enforce such policies with reasonable effort. Sesame enforces access control throughout the application, including data that is rendered to end users in response to HTTP requests, as well as custom sinks such as sending an email or invoking a third-party API, *e.g.,* for targeted advertisement.

Application developers may elect to define a "blanket" access control policy that applies to many types of data, *e.g.,* in WebSubmit, nearly all data related to an answer is accessible to the teaching staff and the student who submitted the answer. Then, they can augment that policy with additional stipulation for specific data types, either to make it accessible to more users, *e.g.,* the answer text is additionally accessible by the relevant discussion leaders (Figure 4.3), or to make access more stringent. Common examples of this pattern include data that applications collect for internal purposes (*e.g.,* authentication), data that falls into protected categories (*e.g.,* health data protected under HIPAA), or data about protected individuals (*e.g.,* children). Developers can perform this augmentation either by defining a new policy specific to the data in question, and assigning it as the sole policy for those data, or by combining multiple sub-policies using `PolicyAnd` or `PolicyOr`.

Generally, we recommend having a smaller number of policies that can be composed together for different data types, rather than a new standalone policy per each combination. This is because our results show that a large part of the developer effort when using Sesame is in policy design, which scales with the number of policies in the application.

Finally, we note that, in some cases, whether a user or entity has access to some data depends on the consent of the user or the purpose of access. Developers can express this by combining their access control policies with purpose or consent policies, which we discuss next.

Purpose Limitation. Privacy laws require that services collect data for specific purposes and only process the data for the stated purposes. End-user facing privacy policies and terms of services often enumerate these purposes in relatively vague prose and describe the types of data they apply to. To enforce such policies using Sesame, developers need to do two things.

First, they must label application sinks with their corresponding purpose. One approach is to attach

```
#[db_policy(table = "answers", columns = ["grade", "author"])]
struct EmployersReleasePolicy {
  author: String
}
impl Policy for EmployersReleasePolicy {
  fn check(&self, context: Context::Out) -> bool {
    let purpose: String = context.purpose;
    if purpose == "emp" {
        return DB.query("SELECT consent_employers FROM users WHERE email = {}", self.author);
    }
    return false;
  }
}
impl DBPolicy for EmployersReleasePolicy {
  fn from_row(row: &MySQLRow) {
    EmployersReleasePolicy {
      author: row.get("author")
    }
  }
}
```

Figure 5.2: Policy governing release of students emails and grades to potential employers. This policy rejects releasing grade or emails unless it is for the `'emp'` purpose and with explicit user consent. In WebSubmit, we also attach other policies to these columns using a `PolicyOr`, *e.g.,* to allow releasing data to authorized users, such as the teaching staff, or for other purposes.

one or more purpose labels to application endpoints. For example, we attach the `'emp'` and `'ml'` purposes to the WebSubmit endpoints that show student information to employers and perform our grade prediction machine learning experiment, respectively. Sesame adds these purposes to the `Context` object prior to performing any policy checks. Earlier work follows a similar approach, *e.g.,* RuleKeeper [FBS+23] requires application developers to map endpoints to purpose labels in a manifest file. Second, developers must attach a list of permitted purposes to specific data types or database columns. They do so by implementing custom policy types that only accept contexts with one of the allowed purposes, or by using a generic purpose-limitation policy configured with the specific permitted purposes.

Web applications often contain *core* purposes that users consent to by using the application and cannot opt out of. For example, the core purpose of WebSubmit is to collect homework answers and grade them. For such purposes, developers can write simple, unconditional Sesame policies that merely check the purpose labels in the context object.

Consent and The Right to Object. Web applications also commonly contain *non-essential* purposes that users can opt out of while continuing to use the core components of the application. Privacy laws require that many forms of third-party data sharing, including advertisement, fall in this category, *e.g.,*

the CCPA's right to opt-out of sale or sharing. For example, WebSubmit only shares student emails and average grades with potential employers with that student's explicit consent. This may also apply to custom, application-specific non-essential purposes that laws do not explicitly identify. For example, WebSubmit requires users to give explicit consent before using their data in the grade prediction machine learning algorithm.

In our experience, application developers use consent to conditionally allow using data for a purpose or sharing data with a particular user or entity, *e.g.,* conditionally allow the `'emp'` or `'ml'` purposes given the user consent in WebSubmit. In this case, developers need to look up the user's consent in their policy `check` function alongside the purpose or access control checks, or combine a consent-checking policy with the other policies using an `And`. A common approach is to store the consent value in the database and retrieve and check it in the policy `check` function, as shown in Figure 5.2.

Finally, we note that cookie-related consent and banners, which are widely required by privacy laws, constitute a special case of consent-based policies. Cookies are a built-in sink in Sesame. Thus, when the application attempts to externalize the data via a cookie, Sesame adds all information related to that cookie to the `context` object prior to invoking the policy check. Policy developers can then perform additional consent checks during their policy checks for cookies to ensure that the end-user's cookie preferences are respected, including for guest users without accounts.

5.2 Integrating Compliance Into Application Workflows

Application developers can follow the guidelines described above to configure K9db and Sesame with annotations and policies that reflect their application's legal privacy requirements. Following that, they must ensure that these systems interface correctly with the application's end-users.

5.2.1 Human-Readable Privacy Policies

Privacy laws require that their applications inform their users of a summary of their privacy policies in a human-readable form. Usually, this takes the form of human-language documents that spell out what types of data the application collects, what purposes it collects and uses them for, the lawful basis for this processing (*e.g.,* user consent), and which data the application may share with third parties.

Developers should ensure that these human-readable policies are consistent with how they configured K9db and Sesame. One benefit of Sesame is that policies are centralized and defined declaratively,

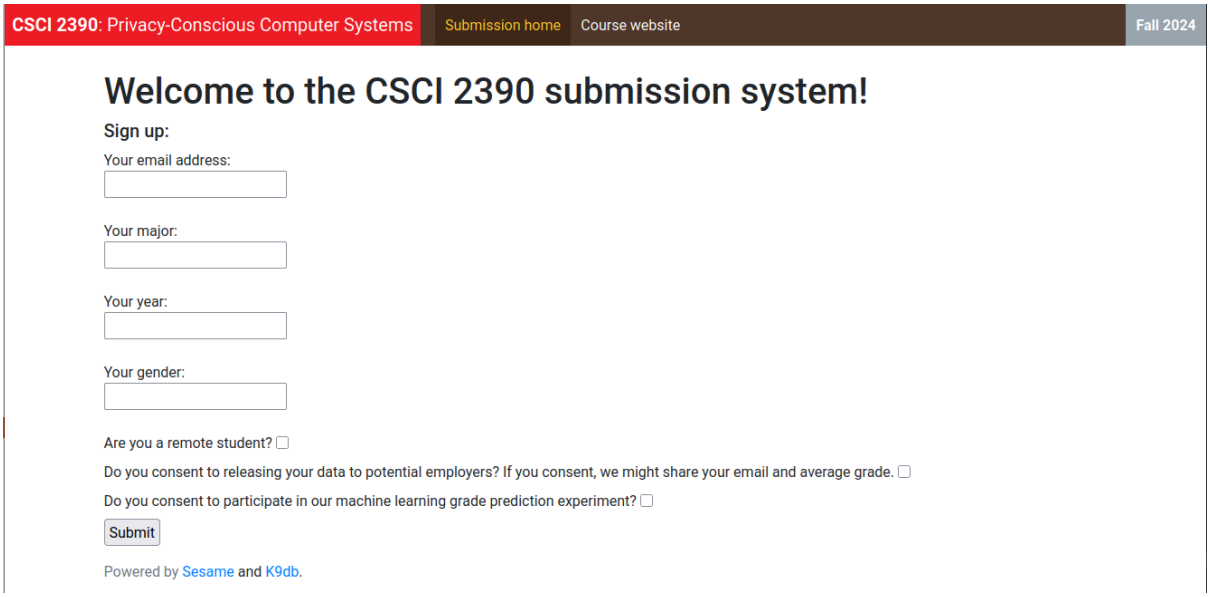


Figure 5.3: The account creation page in WebSubmit allows students to specify their consent preferences. WebSubmit stores these preferences in the database and checks them in the relevant Sesame policies, *e.g.,* releasing students emails and grades to employers if they consent (Figure 5.2).

including their associations with data types and database columns (*e.g.,* through the `db_policy` macro). Thus, developers have a single ground truth they can refer to to determine all the conditions that allow a certain data type to be used or released by the application, *e.g.,* the authorized users that can access it, and the purposes and consent conditions for which the data is collected and used.

Finally, applications sharing data with third parties often do so via *(i)* special application endpoints that third parties query to "pull" the data from the application, or *(ii)* "push" the data to third parties by invoking third-party API in the application code. In Sesame, the latter case can only occur within privacy-critical regions, making them easy to exhaustively identify, *e.g.,* using a lint. This can help developers compile lists of third parties with whom they share data and under what conditions.

5.2.2 Consent and Other Policy Preferences

Application developers must create user interfaces that allow users to specify their policy preferences, these include consent for various non-essential purposes, data sharing with third parties, and other parameters for Sesame policies that may go beyond mere compliance with privacy laws, *e.g.,* a minimum k to require before releasing aggregates or a differential privacy budget.

In WebSubmit, students have two such preferences: consenting to releasing data to potential employers and participating in the grade prediction machine learning experiment. Our user interface allows students

to choose these preferences during account creation. We show a screenshot of that interface in Figure 5.3. We also allow students to update these preferences later through their account profile page.

Applications need to retrieve these preferences from the user interface and store them, *e.g.,* in the database. Then, they can use these preferences in their Sesame policies by either *(i)* retrieving them from storage during the policy check or *(ii)* retrieving them at policy construction time and storing them as metadata in the policy type. For example, we look up the consent preference of a user during the `EmployersReleasePolicy` `check` function (Figure 5.2).

Alternatively, some of these preferences can be saved in cookies and controlled using cookie banners, *e.g.,* for applications with guest users that do not have accounts. Such applications would need to supply constructors for their Sesame policies that read the policy metadata from the cookies.

Although out of scope for this work, it is important to note that the visual design of user interfaces themselves plays a role in compliance and how users set their preferences. For example, applications sometimes use dark patterns [BLN+24] to discourage users from withdrawing consent to certain purposes and, when extreme, enforcement agencies have found such applications to be incompliant.

5.2.3 Endpoints for Data Access and Deletion

Finally, application developers must create user interfaces that allow users to request access to or deletion of their data. Developers must also build dedicated application endpoints that these interfaces invoke. These endpoints issue the corresponding `GDPR GET` and `GDPR FORGET` commands to K9db with the corresponding value of primary key of the relevant `DATA_SUBJECT` table. The endpoints then retrieve the results from K9db and render them in a human-readable format. Figure 5.4 shows a screenshot of the interface that renders student data after they request access. The screenshot is taken from the version of WebSubmit that we deployed for the Fall of 2024 offering of CS2390.

These endpoints may perform pre- and post-processing needed to correctly handle a user's access or deletion request. For example, in WebSubmit, the endpoint first ensures that the user is authenticated by looking up their API key from the request cookies. Additionally, the endpoint may perform further formatting of the output to increase its readability, such as visualizing it via graphs or tables.

When the application also uses Sesame, these endpoints are governed by the same exact Sesame enforcement as any other application endpoints. Specifically, Sesame automatically enforces the policies

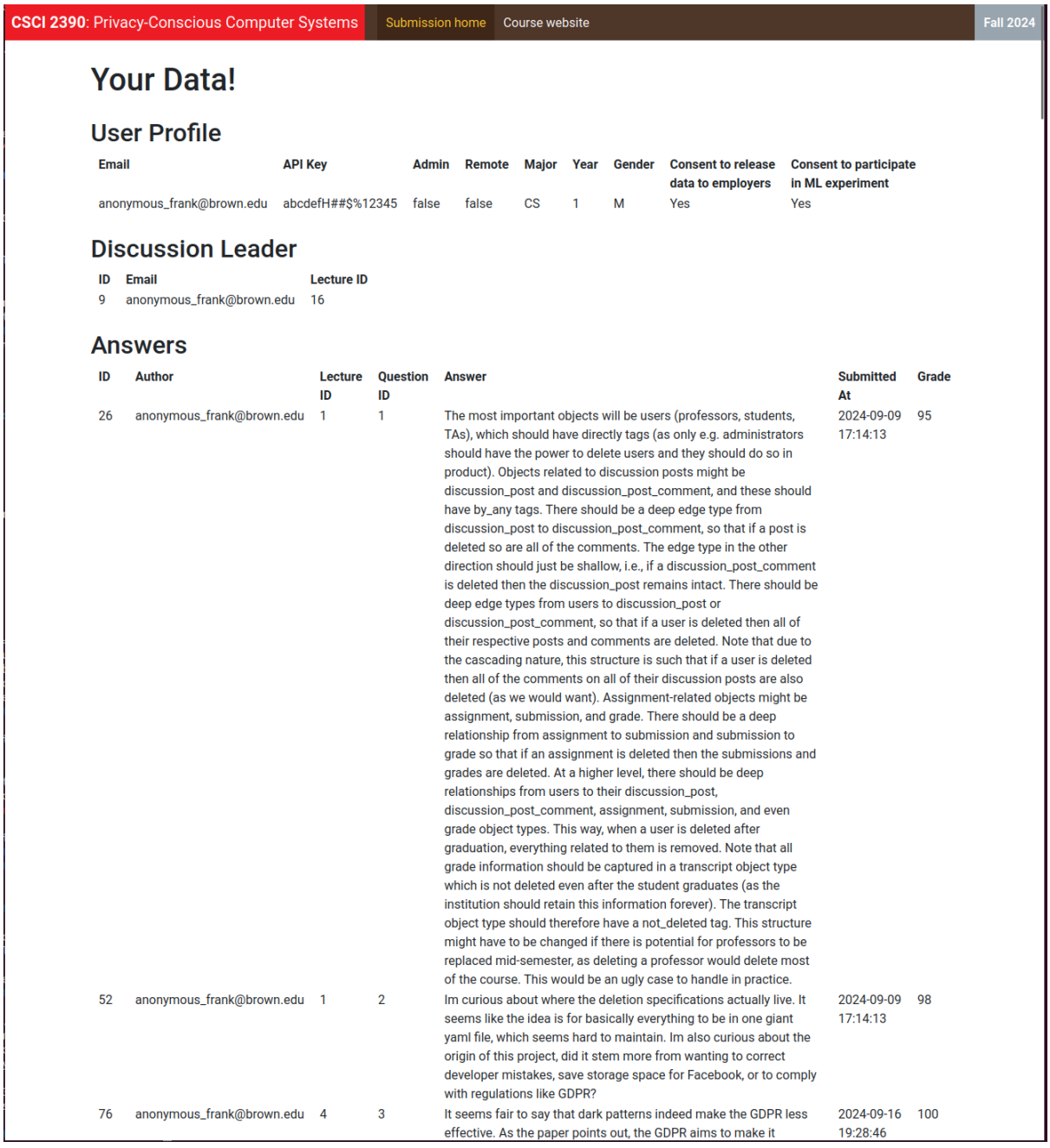


Figure 5.4: WebSubmit provides a user interface for users to issue data access request and to display that data in a human-readable form.

associated with the data returned by K9db. This would detect and disallow the release of data when K9db's access annotations are too lax compared to Sesame's access control policies.

Endpoints can perform additional sanity checks and forbid deletion when necessary. For example, students should not be able to delete their data in WebSubmit during the semester (which includes their homework submissions), because retention of their account and data is required for a legitimate contractual reason (their enrollment in the course at the university). However, they should be able to delete their data after a period of time since the completion of the course or if they drop the course. We resolve this by not allowing students to directly delete their data in WebSubmit. Instead, we run a background task that automatically issues `GDPR FORGET` statements for each student one year after the course is completed, *e.g.,* to ensure enough time has passed to resolve any grading complaints.

In general, applications may use K9db's `GDPR ACCESS` and `GDPR FORGET` primitive commands as a sub-procedure within larger workflows, *e.g.,* periodic deletion of inactive accounts, or to allow authorized relatives of users who pass away to delete their accounts (*e.g.,* in a social media application).

5.3 Compliance Without System Support

We now contrast the developer effort required to achieve compliance guarantees using Sesame and K9db described above with current approaches that do not rely on system-level compliance support. The current approaches rely on manual, ad hoc application modifications that explicitly or implicitly ensure the application logic satisfies the requirements of privacy laws. In addition to the initial effort required to correctly implement this logic, developers need to also maintain and adapt that logic as the application evolves, either in response to new features or to changes in the application's privacy policies and terms of service.

We note that application developers must perform a similar integration effort with this manual approach to what we described earlier in §5.2. They must create and maintain human-readable privacy policies. They also need to collect, store, and manage users' consent and policy preferences, and create user interfaces and back-end endpoints to facilitate data access and deletion. However, they must do so without being able to rely on system-level support, *e.g.,* without having the ability to issue `GDPR GET` and `GDPR FORGET` commands to the database.

5.3.1 Manually Supporting Access and Deletion Requests

Rather than annotating the database schema and relying on K9db to automatically handle access and deletion requests, application developers must implement procedures that explicitly *(i)* identify user data in the database, *(ii)* access or delete that data and determine whether to retain shared data, *(iii)* apply any necessary anonymization, and *(iv)* update backups and caches.

Access and Deletion in WebSubmit. WebSubmit has a simple schema that alleviates the need for many of these steps, and it does not use backups and caches. The application developers must write a sequence of queries, glued together by application code, that retrieves or deletes user data from the table `users`, `answers`, and `discussion_leaders`. These require performing two joins between the `users` and `answers` and the `users` and `discussion_leaders` tables.

Due to the simplicity of the schema, we estimate that these procedures can be implemented in a handful of developer hours. However, developers still need to maintain and update them with application changes. For example, WebSubmit initially did not support assigning students to be discussion leaders. Developers would have needed to update the access and deletion procedures when introducing support for discussion leaders.

In contrast, using K9db only required three simple annotations of the database. When introducing the `discussion_leaders` feature and table, developers would have only needed to annotate its `email` column with `OWNED_BY`, and K9db would automatically update how it handled access and deletion. We selected and applied these annotations to WebSubmit in a matter of minutes.

Other Examples. Generalizing to more complex schemas and applications, we note that the complexity of manual access and deletion implementations scales with the complexity and the number of relevant ownership relationship (*e.g.,* foreign keys) in the schema, which may scale super-linearly in the number of tables. Thus the size, complexity, maintenance effort, and vulnerability to implementation error of manual implementations quickly explodes as the application increases in complexity. For example, §3.7.3 shows that for Shuup, a real e-commerce application, the manual implementation of access and deletion requests include 4k LoC of Python code (including 1.3k LoC of tests) developed over three years and with at least two known bugs we found upon deeper inspection. In another data point, it took about a month and approximately 400 LoC for a student enrolled in CS2390 to implement manual access and

deletion routines for SignMeUp [RFE15], a small web application to facilitate students signing up for TA office hours. In contrast, the number of required K9db annotations often scales linearly with the size of the schema (*e.g.,* as shown in Figure 3.3), and developers can use it to support schemas of similar complexity to SignMeUp with a few annotations and developer hours.

Orphaned Data. WebSubmit does not have long transitive ownership chains and does not use the `OWNS` annotation. Therefore, it does not directly benefit from K9db ownership-integrity guarantees and does not use any compliance transactions. However, applications with more complex schemas, including Shuup, risk creating orphaned, owner-less data via regular application operations. Developers should ensure that their application logic and queries properly cascade into other tables when appropriate *e.g.,* by explicitly deleting a group chat after its last user leaves it. On the other hand, K9db automatically protects against this by detecting and rolling back buggy application operations that cause orphaned data and notifying developers about them.

5.3.2 Manually Enforcing Application-Level Policies

Today, application developers need to manually ensure that their application logic abides by their desired privacy policies, such as purpose limitation and access control. Without system support, developers need to explicitly or implicitly enforce these policies in their logic on a per-endpoint basis.

User Authentication. Every endpoint invocation in WebSubmit authenticates the user via a cookie that contains the user's API key. If that cookie is not set or contains an invalid API key, the endpoint redirects to the login page.

This approach is reasonably effective for simple authentication, especially given that modern web frameworks (including Rust's rocket) provide nice abstractions for defining reusable authentication models. However, it quickly becomes tedious and error-prone for more complex forms of authentication or other policies. For example, some endpoints in WebSubmit are only available to admins and thus must further check that the user's API key corresponds to an admin. Others require that the user satisfy a certain predicate or possess some capability, *e.g.,* only the author of the answer is allowed to edit.

Cross-Cutting Policies. Some policies, such as respecting purpose limitation and user consent, require that developers intertwine explicit or implicit policy-related code and checks with application logic. For example, application developers need to ensure that their database queries perform correct joins and

filters by user consent preference when retrieving data to share with potential employers. An application purpose or functionality may span several endpoints and code path, and developers may need to duplicate the relevant policy-related code and checks across them. For example, WebSubmit may feed data to the machine learning grade prediction experiment in two ways. First, it may "push" a single data point (*i.e.,* assignment submission and grade) to the model upon data creation. Second, it may also "pull" all relevant data and retrain the model on demand. The latter is implemented as a dedicated endpoint with a single purpose. Thus, developers are more likely to include the appropriate policy checks there. The former, however, is part of the grade submission endpoint, which may be developed at an earlier stage or by a different developer that may push data to the model via a hook or helper function without giving it much thought, thus potentially omitting checking user consent and violating the application's policies.

Another example of this relates to access control of the submitted answer text. WebSubmit shows the answers to users via several application endpoints that show specific answers or all answers to a specific question or lecture. It also sends an email receipt containing the answer text to the author, teaching staff, and discussion leaders. Application developers must ensure that all of these paths only reveal the answer to authorized users with access to it.

In contrast, Sesame tracks the policies with the data they apply to throughout application code and checks these policies whenever they flow into an application sink. Thus, Sesame enforces cross-cutting policies end-to-end, without burdening developers with keeping a mental model of which policies apply where or with having to manually enforce them throughout their application.

Manual Enforcement in Evolving Applications. Finally, let us look at how application policies and enforcement can evolve along with the application *e.g.,* when adding new features. Earlier versions of WebSubmit did not support assigning discussion leaders to a lecture or showing them answers to questions for that lecture. We added this feature in two stages. First, we added support to track which students are discussion leaders for a lecture and included them as recipients of the answer submission receipt email. Later, we added application user interfaces and endpoints to allow discussion leaders to view answers through the website. Specifically, we added two endpoints: the first shows a list of lectures the student is assigned to read, and the second shows answers for a specific lecture. The desired flow is that a student first lists the lectures they lead and then clicks on one of them in the user interface to retrieve its answers. That flow implicitly ensures that the student is a discussion leader and thus allowed

```
#[rocket::get("/lectures", auth="student")]
pub fn list_lectures_leader(
  student: User,
) -> rocket::HTMLTemplate {
  let lectures = DB::query(
    "SELECT lecs.id, lecs.title
    FROM lectures as lecs
    JOIN discussion_leaders as dls
    ON dls.lecture_id = lecs.id
    WHERE dls.email = {}",
    student.email
  );
  rocket::render(
    "list_lectures.html",
    lectures
  )
}
#[rocket::get("/<lec_id>", auth="student")]
pub fn show_answers_leader_buggy(
  lec_id: u32,
  student: User,
) -> rocket::HTMLTemplate {
  let answers = DB::query(
    "SELECT answer
    FROM answers
    WHERE lecture_id = {}",
    lecture_id
  );
  rocket::render(
    "list_answers.html",
    answers
  )
}
```

(a) The first endpoint lists the lectures a student is assigned to lead. The second endpoint lists all answers to questions for a given lecture. The second endpoint implicitly assumes users only invoke it on one of the lectures returned by the first endpoint. It merely checks that the authenticated user is a valid student, but not that they are a discussion leader for the given lecture.

```
#[rocket::get("/<lec_id>", auth="student")]
pub fn show_answers_leader(
  lec_id: u32,
  student: User,
) -> rocket::HTMLTemplate {

  // Get all lectures the authenticated
  // user is assigned to lead.
  let lectures = DB::query(
    "SELECT lecture_id
    FROM discussion_leaders
    WHERE email = {}",
    student.email
  );

  // Check that requested lecture is one
  // of the ones assigned to the user.
  if !lectures.contains(lec_id) {
    return rocket::error(UNAUTHORIZED);
  }

  // Carry out the functionality.
  let answers = DB::query(
    "SELECT answer
    FROM answers
    WHERE lecture_id = {}",
    lecture_id
  );

  rocket::render(
    "list_answers.html",
    answers
  )

}
```

(b) A correct endpoint implementation for showing all answers to questions for a given lecture to a student that is assigned to lead that lecture. The endpoint first explicitly checks that the student is a discussion leader, then retrieves the answers from the database and renders them via HTML.

Figure 5.5: Two ways of adding support for retrieving answers for discussion leaders to WebSubmit. a shows a buggy implementation that omits checking that the user is indeed a discussion leader and b shows a correct implementation.

to access the answers. Figure 5.5a shows this implementation. However, note that this approach is buggy. Users may manually invoke the second endpoint and supply it with a lecture ID they are not assigned to lead, in which case they would be able to erroneously see the answers for it. To protect against this, simply authenticating the user is not sufficient, and the endpoint code should explicitly ensure that the user is a discussion leader for the given lecture, as shown in Figure 5.5b.

In contrast, consider how we would add such a feature in the Sesame-protected version of WebSubmit. First, we update the access control policy assigned to the `answer` column to also allow us to reveal the answer to the relevant discussion leaders. Second, we would implement the two endpoints, as shown in Figure 5.5a, without the need to add explicit checks for whether the student is a discussion leader, since Sesame checks that automatically. Thus, Sesame reduces the risk that application developers may omit needed checks or cause inadvertant policy violations when adding new features or modifying existing ones.

Sesame Policy Burden vs. Manual Checks. Comparing the Seasme-protected WebSubmit to a correct Sesame-less WebSubmit where policies are enforced manually, it turns out that all the functionality we implemented within the `check` functions of Sesame policies appears in some similar form in the Sesame-less WebSubmit. For example, consider the `AnswerAccessControl` policy shown in Figure 4.3. The policy contains three stipulations, the latest of which checks that the target user is assigned as the discussion leader to the corresponding lecture. In our implementation, we notice that this implementation is identical to the manual version of this shown in Figure 5.5b (lines 9-20). The other two stipulations for checking that the target user is the author or part of the teaching staff also appear in similar forms in other endpoints.

Another example is `EmployersReleasePolicy` (Figure 5.2) whose check function contains a query to check the consent preference of the corresponding user. A variation of this query appears in the manually enforced version of WebSubmit in the employers release endpoint. Specifically, it shows up as part of a large join query that retrieves the data and filters by consent preference in one go.

Sesame declares each of these policies once in a central policy repository and then enforces them across the entire application code. The manual approach scatters this logic through out application endpoints, with some identical or similar checks duplicated across several endpoints. Thus, using Sesame reduces developer effort in both *(i)* implementing policy-related logic, and *(ii)* tracking code locations where this logic must be applied.

CHAPTER 6

Discussion and Future Work

This chapter discusses ideas for building on K9db and Sesame in three future directions:

1. We discuss how adaptable the ideas and design of K9db and Sesame are to other types of commonly used databases and programming languages (§6.1).

2. We look at direct extensions of K9db and Sesame that aim to further improve ergonomics and reduce application developer effort (§6.2).

3. We provide a brief discussion of some of the ways in which the insights learned from these systems can enhance or complement other privacy notions that go beyond consent and control (§6.3).

6.1 Systems for Compliance with Other Databases and Programming Languages

K9db and Sesame target SQL-databases and web applications built in the Rust programming languages, respectively. There are good reasons for this: SQL schemas have a relational structure that simplifies the annotations needed to express ownership semantics, and Rust provides encapsulation and memory safety guarantees that make Sesame's PCONs and its leakage-freedom static analysis feasible. Now, we consider whether systems with comparable guarantees, performance, and ergonomics can be built for other databases and languages.

Storage Compliance for non-SQL Databases

DELF [CDN+20] shows that it is possible to specify and enforce deletion semantics over graph database, like Facebook's TAO. Although DELF does not provide a complete compliance solution, as it lacks support for data access, encryption at rest, and enforcing the integrity of data ownership at runtime (*e.g.,* no compliance transactions), DELF's existence suggests that the design ideas of K9db can be applied

to graph databases with comparably low performance overheads and with low application developer specification and annotation effort.

NoSQL Databases. NoSQL databases lack the schematic structure of SQL to specify and reason about ownership. This makes it more challenging for developers to express their ownership semantics to the database and also complicates enforcing these semantics by the database. We believe this can be address in two ways. First, developers can implement their ownership semantics using custom program logic *e.g.,* such as deletion hooks that can determine which documents to cascade to dynamically. Second, systems can leverage the structure imposed by object relational mapping (ORMs) that sit on top of the database.

We investigated both approaches with several students in their CS2390 course projects, our graduate seminar on privacy-conscious systems. We found that the two approaches offered interesting trade-offs. The first approach yielded good performance and can express a wide range of semantics, even in cases where the data has very little structure. However, it imposed significantly more burden on application developers compared to K9db, as they had to implement a handful of hooks, each containing tens of lines of code, for small-to-medium sized applications. The second approach resulted in application performance and developer burden in line with K9db but for simpler applications, and we strongly suspect that with further engineering, it can provide guarantees and features comparable to K9db. However, it does require application developers to use a custom ORM.

Applications with Heterogenous Storage. DELF handles deleting user data scattered across multiple storage systems. DELF requires application developers to declare the schema of data stored in each of these systems, as well as the logical relationship between them, in a storage-agnostic domain-specific language.

An alternative design is to create a system with a universal API, through which applications insert, update, and delete data in any of the underlying storage systems. Applications invoke this API with their data and update, the target storage system, and a list of zero or more owners. The system maintains the required metadata that maps between users and the data they own across all the storage systems. The system can use this metadata to delete or retrieve that data when a user requests deletion or access. An advantage of this design is that it does not require any modeling or specification effort by the application developers.

Students in CS2390 investigated this design and built a small prototype compatible with MongoDB, SQLite, and Redis. This prototype exhibited good performance with low overheads resulting from tracking the required metadata. However, it also showed that the design is vulnerable to application bugs *e.g.,* when application edge cases result in an incorrect list of owners for a piece of data.

Privacy Enforcement with Other Web Frameworks

Our Sesame prototype is closely tied to the Rocket web framework [Ben24], but it can be extended to support other Rust-based web frameworks by implementing shims over them. These shims are responsible for *(i)* invoking policy constructors when the application accesses user-provided data in HTTP requests, and *(ii)* invoking policy checks prior to externalizing PCONs via HTTP responses. We investigated adding one such shim to support Axum [LP25], another recent and popular Rust web framework, and found that doing so seems feasible with more engineering effort.

Privacy Enforcement in Other Programming Languages

Overall, Sesame relies on Rust in three distinct ways: *(i)* The encapsulation of data and policy within PCONs, *(ii)* The analysis and enforcement of the three types of privacy regions, and *(iii)* The ergonomics of restructuring applications into glue code and privacy regions.

Other languages may be able to support what Sesame needs, though perhaps with additional developer effort or possible loss of precision.

Encapsulation and PCONs. PCONs are central to Sesame's design. They ensure that data and policies remain attached to each other throughout application execution and force application developers to use privacy regions to interact with the underlying data. Because of Rust's encapsulation guarantees, it *nearly* suffices to define the data and policy members as private within Sesame's PCON type definition. However, because unsafe code is prevalent in the Rust ecosystem, which is able to violate encapsulation and access private members of structs, we augment encapsulation with a pointer obfuscation protection mechanism (§4.4).

Several languages thought to provide strong encapsulation guarantees in reality have escape hatches (similar to unsafe in Rust) that allow application code to access private members. For example, applications can access private members in Java and C# using reflection. They can also access private members in C++ in a variety of ways, including casting and pointer arithmetic. Thus, a PCON-like implementation

in these languages must also rely on pointer obfuscation (or similar techniques).

Several popular programming languages do not provide direct support for private members in a type, such as Python and Javascript. We believe the best recourse to implement PCON-like objects in these languages is a pointer-like mechanism to hide the data. It may be possible to use closures (which provide encapsulation) to disallow external access to the data in Javascript. Alternatively, a Python-based Sesame-like system could store all protected data and policies in some map or vector structure within it, and merely store handles (*e.g.,* obfuscated indices or keys) within PCONs.

Analysis of Privacy Regions. Sesame relies on several properties of safe code in Rust to enable its analysis of privacy regions. First, it relies on lifetime information to resolve aliases and compute information flows via Flowistry [CPA+22]. Second, it combines information flows with a custom inter-procedural analysis and compiler-provided information (*e.g.,* monomorphization, identifying implementors of a trait within scope) to resolve dynamic dispatch. This is further aided by the fact that a large chunk (but not all) of dynamic dispatch in Rust is carried out through trait objects or typed closures, as opposed to free and type-less function pointers. Overall, this enables Sesame (specifically, SCRUTINIZER) to construct a monomorphized call graph (MCG) that represents a sound and reasonably accurate approximation of all functions invoked in a privacy region, directly or indirectly via other dependencies.

Sesame uses this MCG to enable two types of privacy regions. First, verified regions (VRs) simply check that the MCG does not contain flows between sensitive information and constructs with observable side effects. Second, critical regions (CRs) hash this MCG and use that as the basis for code signing, which ensures that any future changes to the CR or its dependencies cause the hashes to diverge and thus the signature to become invalid.

JVM-based languages such as Java and Scala may be closest to Rust with respect to constructing and analyzing the MCG. First, they are statically typed and mostly realize dynamic dispatch via class-based inheritance, rather than unrestricted function pointers which, even in Rust, is challenging for SCRUTINIZER to resolve accurately. Second, they do not expose raw pointers and pointer arithmetic to applications, akin to how Rust aims to hide such unsafe operations behind safe interfaces. Finally, it is possible to detect or enumerate operations with effects in these languages since interactions with hardware resources and I/O must pass through specific JVM or native interfaces and are thus easy to detect. These languages allow runtime reflection and invocations of native code, but code using these

features may be detected and conservatively rejected without further analysis. This analysis may also be possible in other languages that also exhibit the above properties, such as C# or Haskell.

The construction or the analysis of the MCG may be more difficult in other languages that do not meet some of the above properties. It may be feasible to construct a sound MCG in applications that *exclusively* use *modern* C++, where pointer arithmetic is rare, and dynamic dispatch occurs mostly via inheritance. However, analyzing such a C++ MCG for side effects is tricky, as side effects themselves are harder to identify, unlike in Rust. For example, all I/O interactions (*e.g.,* network, files) in Rust must go through clearly delineated unsafe intrinsics or native functions, and thus SCRUTINIZER can look for and reject any sensitive flows into such functions. Furthermore, mutations of captured and global variables are also easy to detect. In C++, this delineation is less clear, although detecting certain calls to the standard library or syscalls could serve as proxy for detecting I/O and effects. Additionally, constructing the MCG in dynamic languages like Python or Javascript is much harder, especially with soundness and good accuracy. For example, related work that tried to perform somewhat similar static analysis for privacy in Javascript is neither sound nor complete [FBS+23].

However, there are alternative mechanisms for privacy regions that may be more suitable for such languages. For example, sandboxing via WASM might be ideal practical as the primary mechanism (rather than a fallback) for leakage-free regions in Javascript. The overhead of sandboxing in Sesame comes from having to change the memory layout of arguments to the sandbox, and from the increased overhead of using WASM compared to un-instrumented Rust code. Both of these overheads are *relatively* lower for Javascript: WASM often can make Javascript code run faster. Similarly, it may be possible to achieve a similar effect in Python via a modified runtime. The runtime can detect when it is executes a region (*e.g.,* by setting a flag), and then error out if the interpreter attempts to execute native-facing APIs or modify global memory. Although this would likely require significant engineering effort, it may result in a lighter-weight solution with better performance than taint-tracking approaches, such as Resin [YWZ+09] and Riverbed [WKM19], which modify the Python runtime to track and combine policies with every operation.

Ergonomics. Rust applications often exhibit two properties that simplify *porting* existing applications to Sesame. First, they often rely on type inference and thus mostly write out explicit types in function signatures. This makes it easier to update an unprotected variable or piece of data in the existing

application to a PCon. Second, idiomatic Rust encourages chain closures, *e.g.,* in iterator chains or with monads such as `Option` and `Result`. Thus, this makes it easier to upgrade such code to a privacy region, which is also a closure. Although these properties make porting applications to Sesame easier, we suspect that they do not play as much of a role in developing applications using Sesame from scratch.

Finally, Rust provides an ecosystem that allows plugging in various static analyses via compiler plugins and lints. It also provides derive macros and build script support that simplify tasks like generating FFI-bindings (*e.g.,* for sandboxed regions). Achieving similar ergonomics in other languages without such an ecosystem remains possible but requires more engineering effort.

6.2 Extensions to K9db and Sesame

We discuss three ongoing extensions that aim to improve ergonomics and reduce application developer effort in common application scenarios. The first extension, SesameBun, seamlessly extends the scope of Sesame's enforcement to complex database queries by tracking Sesame policies within the database. The second, SesaSpec, aims to help applications combine K9db and Sesame through a simple unified specification language. Finally, the last extension, Tahini, aims to provide end-to-end enforcement guarantees for applications with remote and microservices.

6.2.1 Tracking Sesame policies in the database using SesameBun

Sesame's enforcement rests on its policy containers (PCons). Rust's encapsulation guarantees ensure that application code cannot directly access or externalize the data within PCons. Instead, applications must use privacy regions when manipulating this data, and may only externalize it via Sesame-enabled sinks or a critical region. Thus, Sesame ensures that the externalization of data within a PCon respects its associated policy, assuming proper review of any relevant critical region.

This leads to the question of how the data is associated with a policy object and stored in PCons to begin with. Generally, application developers are responsible for constructing the desired policy objects and placing them along input data in PCons at custom application sources, such as reading from a file. Furthermore, Sesame provides support for two common sources: HTTP requests and the database.

Recall the access control policy Figure 4.3 for homework answers in WebSubmit: Answers can only be accessed by their authors, the teaching staff, or the students assigned to lead the discussion in the corresponding lecture. After implementing this policy as a Rust struct, application developers must

```sql
CREATE VIEW aggregate AS
SELECT users.is_remote,
  AVG(answers.grade) as avg
FROM users JOIN answers
ON users.email = answers.author
GROUP BY users.is_remote;
```

(a) The unmodified `aggregate` query as written in WebSubmit prior to porting to Sesame. It computes the average grade over all submissions grouped by author remote enrollment status.

```sql
CREATE VIEW aggregate_modified AS
SELECT users.is_remote,
  AVG(answers.grade) as avg,
  COUNT(DISTINCT answers.author) as c
FROM users JOIN answers
ON users.email = answers.author
GROUP BY users.is_remote;
```

(b) The modified query after porting WebSubmit to Sesame. This query additionally tracks the number of distinct users with grades in each enrollment status. The `MinK` Sesame policy uses this count to determine whether to allow releasing the aggregate value.

Figure 6.1: The queries for computing average grade per student enrollment status in WebSubmit. a shows the unmodified query in the original WebSubmit implementation, and b shows the modified query after porting WebSubmit to Sesame.

```rust
#[db_policy(table = "aggregate_modified", columns = ["avg"])]
struct MinK { k: u64 }
impl Policy for MinK {
  fn check(&self, context: ...) -> bool {
    self.k >= MIN_K
  }
}
impl DBPolicy for MinK {
  fn from_row(row: &Row) {
    MinK { k: row.get("c") }
  }
}
```

Figure 6.2: The `MinK` policy assumes the count `c` computed by `aggregate_modified` (Figure 6.1b) is correct and confirms it is larger than the minimum allowed K for release.

perform two steps to associate that struct with the answers in the database, as shown earlier in Figure 4.3: *(i)* they must use the `#[db_policy(...)]` annotation to declare which columns in the database this policy applies to, and *(ii)* they must implement the `from_row` policy constructor that creates a policy object given a row in the database. Sesame provides the constructor with the entire row rather than merely the cell to which the policy applies. This is important as policies often depend on metadata adjacent to the data they govern. For example, the WebSubmit access control policy depends on the adjacent `author` and `lecture_id` columns. This approach works well for simple database queries, such as point lookups or scans. However, it runs into issues when handling complex database queries that project away important columns or aggregates over multiple rows.

Consider how WebSubmit computes the average grade by the remote enrollment status of students.

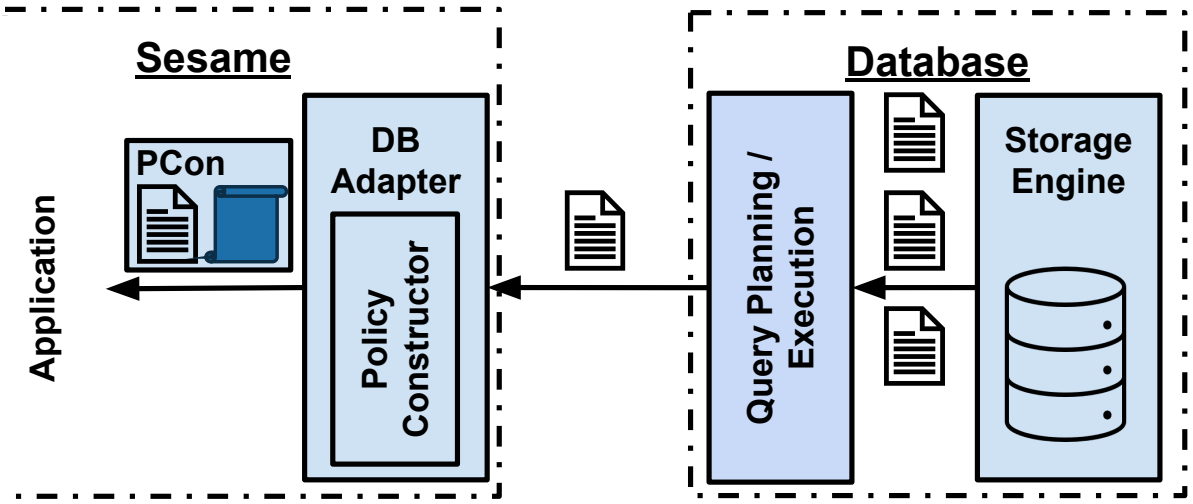


Figure 6.3: How Sesame interfaces with a "black-box" SQL database. Here the application issues a query (*e.g.,* an aggregation). The database reads relevant individual rows from the underlying storage engine, and then perform the query (*e.g.,* aggregation) on them, returning the result to Sesame. Sesame database adapter creates policy objects (shown as a blue scroll) for the result using developer-provided policy constructors. The query results must contain all the needed information to construct the policies, which may require manual query rewriting by application developers.

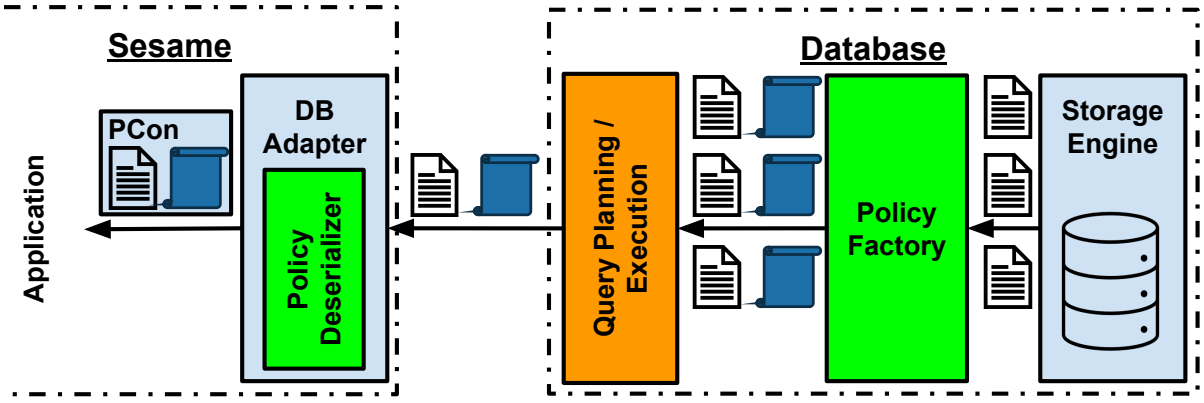


Figure 6.4: Overview of SesameBun's design. SesameBun constructs Sesame policy objects within the database as soon as it reads individual rows from the underlying storage engine (*e.g.,* aggregation inputs). SesameBun modifies query planning and execution such that it tracks and combines the policy objects associated (*e.g.,* aggregates policies along with data). Now the database returns the query results associated with its policies. Application developers no longer need to rewrite their queries or provide policy database constructors. New and modified components are shown in green and orange, respectively.

Figure 6.1a shows the query that computes this aggregate prior to porting WebSubmit to Sesame. This version of the query contains a subtle privacy threat that Sesame must address. Imagine a case where only one of the students in the class is remote: WebSubmit should not reveal the remote average grade in that case, as it corresponds to the grade of that remote student. Encoding such a policy in Sesame

requires tracking the `answers.author` values in each aggregation group to ensure that there are more than K students in each aggregate value. However, the query projects away that information and only retains `users.is_remote` and the average grade. Thus, the Sesame database adapter and the policy's `from_row` constructor do not have enough information to construct correct policy objects for this query. This is a consequence of viewing the database and its queries as a "black-box", as shown in Figure 6.3. Sesame interfaces with the database via the adapter, which only has access to the final query output (at which point relevant metadata may be lost), and not intermediate records used during query execution within the database (where the relevant metadata still exist).

To overcome this, we modified the query while porting WebSubmit to Sesame, so that it tracks the distinct count of students in each aggregation group (Figure 6.1b). Then, we attach a `MinK` policy to the output of that query and retrieve the count in its `from_db` constructor (Figure 6.2). This correctly enforces the desired policy. However, it suffers two drawbacks: *(i)* it requires application modification and thus increases developer effort, and *(ii)* it makes the policy dependent on the query (specifically, the count), and thus makes the query part of the trusted policy code. For example, we could have incorrectly modified the query to compute the count without `DISTINCT`, which would have changed the semantics of the policy to require a minimum of K records from any number of users, rather than records from at least k distinct users.

To address these drawbacks, we now sketch SesameBun, a *database extension* of Sesame that tracks Sesame policies within the database (Figure 6.4), such as when handling aggregates. In this design, a SesameBun-compatible database creates policy objects for the raw records it reads from the underlying storage engine, prior to any aggregation, projection, or other lossy transformations. It then tracks and combines these policies as appropriate as it performs operations on these records. Finally, it outputs both the query result and its associated policies, which Sesame's database adapter deserializes into PCons, without the need for any policy constructors.

Preliminary Prototype

We implemented a preliminary prototype of SesameBun for K9db. Our prototype extends K9db with the following components:

1. A new "`SET POLICY <policy_name> FOR <table_name>.<column_name>`" SQL command. Applications issue this command to the database at schema creation time to indicate that the

provided column is governed by the given policy, identified by name. We implemented a C++ version of each of the policies used in WebSubmit and linked them with K9db.

2. A policy factory that *(i)* maintains a mapping between columns and their associated policies, and *(ii)* constructs and attaches policy objects to cells in records K9db reads from its underlying storage engine.

3. A modified query execution and dataflow engine that preserves policies associated with its input records. Specifically, it computes the conjunction of the policies associated with cells that are aggregated or joined, and attaches that conjunction to the result.

4. A policy serializer that transmits query results along with their policies via the MySQL protocol.

We chose to implement this prototype on top of K9db, rather than, say, MySQL, for two reasons. First, as discussed in §5, applications can use Sesame with K9db to get privacy compliance guarantees that cover both application logic and storage and SAR requirements. A tighter integration between Sesame and K9db facilitates their combined use. Second, K9db tracks the owners of each piece of data in order to ensure correct compliance with SARs. This gives us an opportunity to associate Sesame policies with data based on its owners in the future, in addition to schema- or column-based associations.

One possible pitfall in this proposal is that the database must now serialize policy objects along with query results. These policy objects are associated with specific cells in the query result. Thus, the database may transmit as many policy objects as cells in the result set. If the policy objects contain significant metadata, such as an access control list, this may result in a significant increase in transmitted bandwidth, and thus performance overhead. To avoid this, we implemented an optimization in our prototype that computes the column-wise conjunction of all policies in the result set prior to transmitting the output. This results in having one policy object per column, as opposed to per cell. It is also sound, as that policy is as strong as all the cell-level policies combined.

Preliminary Results

We can judge the effectiveness of SesameBun's approach and design based on the following criteria. First, using SesameBun as a back-end should not result in any enforcement degradation compared to the current black-box approach to databases in Sesame. Second, SesameBun should simplify policy construction and enforcement with complex database queries and result in fewer query modifications

compared to the current Sesame approach. Finally, SesameBun should provide end-to-end application performance comparable to that of black-box approaches.

Our preliminary evaluation of the SesameBun K9db prototype shows promising results. We used K9db's SesameBun extension as the back-end database for our Sesame port of WebSubmit (§4.7). We found that: *(i)* we were able to successfully enforce all of WebSubmit's policies using the SesameBun backend, *(ii)* SesameBun alleviated the need to manually modify aggregation queries for policy construction, and *(iii)* SesameBun had comparable (and in certain cases better) performance to non-SesameBun baseline. These are promising results that provide a good preliminary indication that this approach is viable and may result in simpler integration between Sesame and the database and thus a lower application developer effort.

Relation To Related Work

Policy enforcement systems with goals comparable to Sesame face similar issues when it comes to application queries. For example, Resin [YWZ+09] has a similar database adapter (*i.e.,* a "filter" object) that constructs policies for cells in the results of the database query. One difference is that Resin persists policies along with the data when it is written to the database, and then reloads that policy when data is read. Sesame does not persist the policy, and instead reconstructs it from scratch *e.g.,* using metadata from adjacent columns. However, both approaches face similar limitations when dealing with aggregates, such as the one shown in Figure 6.1. Cocoon [LTB+24] relies on application developers manually wrapping data from sources, including the database, in Cocoon's `Secret` type. This manual approach is in line with many other IFC systems [ML00; RPB+09]. The overall design and goals of SesameBun could benefit these other systems, including Resin and Cocoon, by associating and tracking their policies or labels during query execution within the database.

6.2.2 SesaSpec: Common Specification Language for K9db and Sesame

In Sesame, application developers implement their policies as Rust types that their application and Sesame can use *e.g.,* with PCons. With SesameBun, they must also inform K9db about these policies. Specifically, developers must specify the columns the policies are associated, how to construct instances of them for database rows, and their `join` functions, which are responsible for policy conjunction.

Furthermore, K9db itself requires schema annotations that application developers must provide. These annotations express data ownership relationships in the database and govern how K9db handles GDPR

access and deletion requests. Although K9db's schema annotations and Sesame's policies relate to different privacy and compliance requirements, they sometimes partially overlap. First, K9db annotations identify the owners of a piece of data, and the Sesame policies attached to that data are often configured by its owners' preferences, *e.g.,* whether the author of an answer consents to releasing their answer grades to potential employers. Second, schema annotations and Sesame policies often express overlapping but distinct notions of "access".

As a result, there is often overlap between the specifications that application developers must provide to K9db and Sesame, which take the form of schema annotations and Rust policy definitions, respectively. A unified specification language could govern both K9db and Sesame together, with the goal of reducing specification duplication and effort, and creating additional sanity checks that can help application developers ensure their specifications indeed encode their desired policies. We sketch the design of one such language, SesaSpec, below.

The SesaSpec Unified Specification Language

SesaSpec is a JSON-like schema languages that defines *(i)* the SQL schema including tables and columns, *(ii)* the GDPR ownership and access relationships between records in that schema, and *(iii)* the associations between columns and policies that govern how their contents may be used by the application. We show an example of this language applied to WebSubmit in Figure 6.5.

Policy Templates. Built-in policy templates can simplify policy specification by providing application developers with configurable and reusable templates that correspond to common application policies. For example, `AccessControl`, `Aggregate`, and `Consent` are all policy templates in the above example. Application developers configure them via custom fields in SesaSpec, such as the `allow` list or `purpose`. Importantly, the values of these fields can be expressions that refer to other adjacent cells (*e.g.,* `$self.author`), database queries (*e.g.,* all admins), or fields associated with their data owners (*e.g.,* `$owner.consent_employers`).

Custom Policies. Applications may contain a number of custom policies that cannot be expressed using built-in policy templates. Application developers can express such policies as Rust types as in regular Sesame. We envision providing Rust macros that allow developers to register these custom policies so that they can refer to them by name in SesaSpec. We also envision automatically compiling and

```
"table": "users", "data subject": true, "columns": [
  { "name": "email", "ty": "text", "primary key": true },
  { "name": "apikey", "ty"; "text", "unique": true, "policy": { "name": "CookieOnly" } },
  { "name": "is_admin", "ty": "int" },
  { "name": "consent_employers", "ty": "int" },
  { "name": "consent_ml", "ty": "int" },
  { "name": "is_remote", "ty": "int" },
  { "name": "gender", "ty": "text" }
]

"table": "discussion_leaders", "columns": [
  { "name": "id", "ty": "int", "primary key": true },
  { "name": "lecture_id", "ty": "int", "foreign key": "lectures.id" },
  { "name": "email", "ty": "text", "owned by": "users.email" }
]

"table": "answers", "columns": [
  { "name": "id", "ty": "int", "primary key": true },
  { "name": "lecture_id", "ty": "int", "foreign key": "lectures.id" },
  { "name": "question_id", "ty": "int", "foreign key": "questions.id" },
  { "name": "author", "ty": "text", "owned by": "users.email" },
  { "name": "answer", "ty": "text", "policy": {
      "name": "AccessControl",
      "allow": [ $self.author, ${SELECT email FROM users WHERE is_admin = 1},
                 ${SELECT email FROM discussion_leaders WHERE lecture_id = $self.lecture_id} ]
    }
  },
  { "name": "grade", "ty": "int", "policy": {
      "Or": [
        { "name": "AccessControl",
          "allow": [$self.author, ${SELECT email FROM users WHERE is_admin = 1}] },
        { "name": "Aggregate", "count": $self.author, "min_k": 10 },
        { "name": "Consent", "purpose": "emp", "field": $owner.consent_employers },
        { "name": "Consent", "purpose": "ml", "field": $owner.consent_ml }
      ]
    }
  }
]
```

Figure 6.5: An example specification for WebSubmit written in SesaSpec. The specification defines the database schema, its ownership semantics (for handling GDPR requests), and policy association (for application-level policy enforcement).

packaging these policies into a shared library that can be dynamically loaded and used by K9db, similar to a user defined function (UDF) in traditional databases. The `CookieOnly` policy, which only allows externalizing a user's apikey via a cookie to that user, is one example of such custom policies.

Sanity Checks. Future tooling on top of SesaSpec can help application developers perform sanity checks and identify specification bugs, by checking that the application-level and GDPR-level access policies are consistent. Developers specify the former using the SesaSpec's `AccessControl` policy template, while K9db's data ownership graph (DOG) captures the latter. These two concepts are different,

but they overlap: In general, a user may be allowed to access some data via the application interfaces without having GDPR access rights to it. For example, users of a social media application can see public comments on public posts via the application but do not get a copy of all such comments (of which there could be millions) when they request GDPR access to their data. However, `AccessControl` policies should not forbid access to data that a user has GDPR access to. Such contradictions imply an inconsistency and thus an error in the specification.

Human-readable Privacy Policies. Applications need to provide end-users with human-readable privacy policies to comply with the GDPR and similar laws. At a minimum, these documents describe the types of data that the application collects, the legal basis for this collection (*e.g.,* user consent), the purposes for which it is used and any third parties with whom it may be shared. Note that these concepts overlap with common policies templates *e.g.,* for consent, purpose limitations, or third-party sharing. We envision that SesaSpec, which associates these templates with different columns and data types, can be used as a basis for generating such human-readable documents. For example, synthesis or LLM based techniques could generate such documents in a semi-automatic way from SesaSpec. Alternative approaches could check whether these documents are consistent with SesaSpec, *e.g.,* by generating examples of good and bad flows and comparing them with each.

Prototype and Preliminary Results

We implemented an initial prototype of SesaSpec capable of supporting the features, keywords, and policies shown in Figure 6.5. The prototype takes as input developer-provided specifications written in SesaSpec's format. The prototype automatically generates the corresponding SQL schema along with its K9db annotations and uses it to setup K9db. It also sends `SET POLICY` commands to our SesameBun prototype to configure it with the policy associations described in the specification. The prototype provides the three policy templates shown in the example, but does not have streamlined support for custom policies, such as `CookieOnly`. Instead, we had to manually implement and link a C++ version of `CookieOnly` to a fork of K9db and SesameBun. We plan to add better support for custom policies with further engineering in the future.

Using this prototype, we were able to enforce all WebSubmit policies and support SARs for it. Our preliminary results show that using SesaSpec results in a $2.6\times$ reduction in overall specification size, including SQL schema, K9db annotations, and Sesame policies code and constructors. We further observe

that SesaSpec also reduces the complexity of the specification, as it results in a $10\times$ reduction in the portion of the specification written in Rust: the regular Sesame port of WebSubmit included 373 LoC of trusted Rust policy code, while we only needed 33 LoC of Rust code for custom policies using SesaSpec. This provides a good first indication that SesaSpec may result in simpler specifications that are easier to write and reason about.

6.2.3 Extending Sesame to Distributed and Microservices Applications with Tahini

Web applications often invoke remote services, which may be run by the same organization or by untrusted third parties. Either type of remote invocation requires externalizing data outside the application process boundary *e.g.,* by invoking a custom remote procedure call (RPC). As a result, for Sesame-protected data, such invocations must be wrapped in a reviewed privacy critical region. This guarantees that application developers cannot inadvertently call remote services on sensitive data, since Sesame checks the associated policies, and authorized reviews must deem the invocation desirable and sign its code. However, it also means that Sesame's enforcement guarantees are conditional on the remote service behaving within the expectations of the policy check and human review, *e.g.,* that a remote plagiarism detection service does not publicly reveal students' homework answers. Even when the remote service is trusted, this is vulnerable to human error, especially when the remote service evolves, *e.g.,* by adding new behavior.

Tahini

We sketch a possible extension of Sesame, Tahini, with support for such distributed settings. Tahini aims to provide end-to-end enforcement guarantees when both the application and remote service use Sesame properly. For tightly integrated micro-services developed by the same company, Tahini adds a built-in Sesame-enabled RPC library that *(i)* serializes the policies associated with the data prior to RPC invocation, and *(ii)* automatically deserializes the policies and attaches them to the corresponding data via PCons on the remote service side. Crucially, such an policied-RPC abstraction must be transparent to applications to ensure good ergonomics.

For remote services provided by untrusted third parties, this is a little more complicated. First, third parties may be deploying Sesame incorrectly, *e.g.,* they may be too lax in how they review the critical regions. Second, they may not be able to encode or enforce the policies associated with the data by the application (and vice versa). For example, when these policies, and in particular their `check` functions,

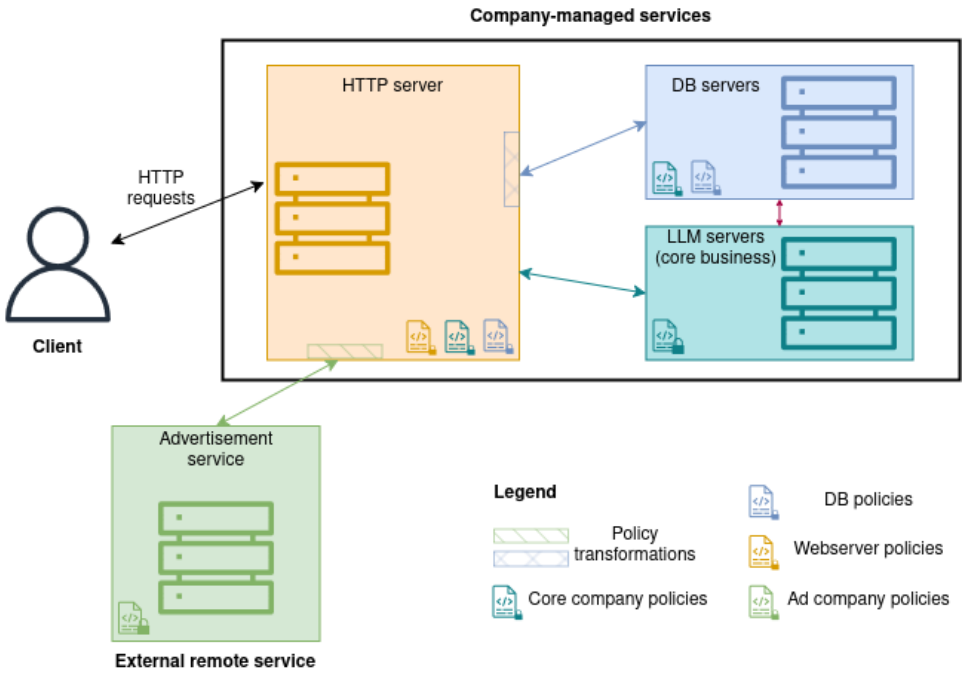


Figure 6.6: Example LLM chatbot application that consists of several micro-services and uses Tahini for end-to-end policy enforcement. Data is governed by shared core company policies through out company managed services. Furthermore, the HTTP and DB servers impose additional policies on the data (*e.g.,* whether the user allows persisting message history in the database). The ad company is governed by an independent set of policies. Tahini allows the application to seamlessly invoke these micro-services and perform the required policy transformations, provided their conditions are met.

refer to application-specific concepts such as by making a database query.

For such scenarios, Tahini adds support for policy transformations. These transformations seamlessly replace the Sesame policy attached to some data with a stronger but simpler policy that does not refer to any internal application concepts and thus is compatible with remote services. Alternatively, these transformations can allow weakening of the policy provided some contextual conditions are met, *e.g.,* a specific service is being invoked, or that the data has been transformed in some predefined way, *e.g.,* hashed, encrypted, or anonymized.

Remote attestation techniques, such as Intel TDX [Int23], may be useful to attest to the correct use and configuration of Tahini by the remote service. Such attestation techniques may to be extended to reason or show information about relevant critical regions, while keeping the rest of the service private *e.g.,* when the code is proprietary.

Preliminary Prototype

We implemented an early-stage prototype of Tahini that seamlessly supports invoking RPCs on PCONs along with a prototype implementation of policy transformations. The policy developers express what transformations over that policy are allowed and under what conditions by implementing certain Tahini-provided traits for that policy.

We applied our Tahini prototype to a simple LLM chatbot application that we built for evaluation. Figure 6.6 shows an overview of the application design. The application consists of an HTTP, database, and LLM servers, each implemented as a standalone micro-service. It also invokes remote services for advertisement that are managed by an independent untrusted entity. This provides an early indication that this approach is feasible.

6.3 Complementary Notions of Privacy

We discussed two other complementary notions of privacy in §1.1.1: cryptographic secure computation and statistical privacy. These notions complement the focus of this dissertation on respecting end-user consent and control over their data. We now discuss some future directions for combining these techniques to obtain stronger privacy guarantees inspired by our earlier work on building and deploying secure computation in practice [DIL+19; LJD+18].

Outsourced Secure Computation. Secure computation, such as secure multiparty computation [BGW88; Sha79; Yao86] or holomorphic encryption [Gen09], allows mutually distrusting parties to jointly compute a function over sensitive input while keeping these inputs private. A common deployment scenario for these technologies is the *outsourced model*, where a large number of *data contributors* provide secret shares or homomorphic ciphers of their data to a small number of *compute parties*. The compute parties then perform the desired computations over the secret data, while only revealing the final output. This reduces communication and computation overhead compared to a straw-man solution in which data contributors perform the computation. It also allows for asynchronous computation or long-running analytics, where data contributors are not required to be constantly available or online. However, it also results in a loss of control, since the compute parties can decide to run whatever computation they agree upon, without involving the data contributors.

Secure Data Deletion in Outsourced Secure Computation. An approach to increasing data contributor

control over their data in this outsourced model is to provide them with verifiably correct and secure data deletion mechanisms. A challenge is that it can be difficult to determine the source or owners of such data, since they are in secret-shared form, *e.g.,* data may be associated with an identified or encrypted shared user or email. Furthermore, the data deletion mechanism must itself be secure and oblivious. Otherwise, it may constitute a side channel through which adversaries can attribute the deleted data to its owners, and use that to deduce information about that data given prior released outputs.

Policy Enforcement in Outsourced Secure Computation. Another promising approach is to rely on policy enforcement *e.g.,* in the style of Sesame or other related work. In this setting, data contributors attach privacy policies to each of their shares that govern what functions the compute parties may run. As long as some threshold of computer parties honestly check these policies, secure computation guarantees that the outputs of violating functions cannot be reconstructed.

One challenge is that this makes the policy check a part of the secure computation itself and thus requires it to be executed using the same underlying cryptographic techniques. This is slow when policy checks contain non-linear operations, *e.g.,* checking that the number of users that contributed data to some specific aggregate is greater than a threshold (`min_k`). Furthermore, the policy check itself may depend on the underlying secret inputs. Thus, by observing whether a policy check succeeds or fails, adversaries can learn information about the inputs.

Guaranteeing Data Contributor Differential Privacy Preferences. Statistical privacy techniques, such as differential privacy (DP) [DMN+06], are often deployed in a centralized model, where a single analyst collects the input data and performs their statistical analysis over it. The analyst then adds the required noise to ensure that differential privacy's guarantees are upheld. That is, the probability of any external adversary successfully determining whether a particular user's data were in the data set or not is bound by a function of two configurable parameters, ϵ and δ.

Data contributors similarly face a loss of control in this scenario. They have no guarantees that the analysts added noise correctly[1]. Furthermore, they have no guarantees that analysts did not overuse their data, by releasing multiple aggregates over it over time. This is problematic, as repeated release results in proportionally lower overall privacy.

[1] In fact, sampling noise correctly to meet a desired ϵ and δ is non-trivial even for trusted parties [Mir12; NSN+24].

Existing work [LPT+21] provides systems to track and control the overall loss of privacy, measured by a *privacy budget*, in cases where data can be analyzed over time. However, this line of work assumes that the self-declared privacy consumption of each workload is accurate. In other words, it assumes that each workload correctly configured the noise distribution and declares the resulting ϵ and δ. It also does not protect against bugs and errors in the system, its configuration, or how it is deployed.

An interesting line of research is to build systems that can provide end-users and data contributors with end-to-end guarantees in this setting, for example, by using infrastructure with a stronger policy enforcement guarantee, such as Sesame, to power the tracking and accounting of various users' privacy budgets. This could also be combined with static analysis approaches to validate or deduce the actual ϵ and δ induced by some code, or assist in configuring and sampling the underlying noise distribution.

CHAPTER 7

Conclusion

Application developers often implement buggy code that may misuse user data in violation of privacy laws such as the GDPR as well as their own self-set privacy policies and terms of service. This not only results in a loss of privacy for affected individuals, but also in large financial and reputation damages for companies via fines and privacy scandals. A common reason for these violations is the lack of practical systems and tools to assist application developers in ensuring their applications meet the desired privacy requirements.

This dissertation presented familiar and practical systems that web applications can use to get automatic compliance assurances for many requirements from privacy laws. The dissertation focuses in particular on storage and processing requirements around respecting end-user consent and ensuring that users have control over their data. The storage requirements include correct handling of access and deletion requests from end users, often called subject access requests (SARs), as well as encryption at rest. The processing requirements govern the application code and business logic, specifically to ensure proper authentication and access control, purpose limitations, restriction of processing for certain purposes, and respecting user consent. We discussed these requirements and their legal basis in GDPR and GDPR-like laws in more detail in §2.1, and provided an overview of some existing privacy-conscious systems and approaches that aim to help applications comply with them or related privacy properties in §2.2.

The main technical contributions of this dissertation are the design, implementation, and evaluation of two practical systems: K9db (§3) and Sesame (§4). K9db is a privacy-compliant database that supports compliance with GDPR-style access and deletion requests. K9db relies on declarative schema annotations to capture application-specific ownership semantics and automatically handles access and

deletion requests as built-in primitives. K9db rethinks database design around data ownership as a first-order principle, by implementing a new storage layer organized into per-user micro-databases and providing compliance transactions to ensure data ownership integrity is preserved throughout application execution.

Sesame enforces application-level privacy policies over application code. These include access control, purpose limitation, and user consent. They also include additional privacy policies that go beyond what is strictly required for compliance but may be desirable for applications to enforce internally, such as policies around aggregation, securing cryptographic keys, among others. Sesame combines recent advances in memory safe languages and runtime sandboxing with code review processes in software engineering to achieve practical enforcement with reasonable developer effort.

Finally, we looked at how application developers can use K9db and Sesame to meet the consent and control requirements of privacy laws in practice. We compared this experience to the status quo approach to compliance, where application developers must manually implement any required features and ensure that the required policies are upheld via explicit checks or implicit application logic scattered across the application code. We explored this in §5 using WebSubmit, an open-source web application for homework submission that we deployed in various courses at Brown, as a case study. We discussed a general blueprint for how to configure K9db and Sesame to enforce the requirements of the law, and how applications integrate their features, such as data access and deletion, within their larger workflows and user interfaces.

We evaluated each of the systems presented in this dissertation using realistic case studies and application workloads. These experiments confirm that the systems meet their respective goals: *(i)* the systems require reasonable developer effort to use correctly, and *(ii)* web applications that use these systems exhibit end-to-end performance comparable to the current status quo, where compliance is manual and underlying web systems do not offer built-in support for compliance. This provides evidence to support the central thesis statement of this dissertation. That is, it demonstrates that it is possible to build familiar and compatible privacy conscious systems that simplify compliance for well-intentioned application developers, including with privacy requirements mandated by privacy laws and beyond, while reducing application developer effort and with reasonable application performance.

Bibliography

[Abo18] John M. Abowd. "The U.S. Census Bureau Adopts Differential Privacy". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '18. London, United Kingdom: Association for Computing Machinery, 2018, page 2867 (cited on pages 6, 22).

[ACC+21] Abdelrahaman Aly, Karl Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P Smart, Titouan Tanguy, et al. *Scale-mamba v1. 14: Documentation*. https://nigelsmart.github.io/SCALE/Documentation.pdf. 2021 (cited on page 21).

[ADS17] Paul C Attie, Kinan Dak Albab, and Mouhammad Sakr. "Model and program repair via sat solving". In: *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2017), pages 1–25 (cited on page 11).

[aer20] aermin. *ghChat (react version)*. 2020. URL: https://github.com/aermin/ghChat (visited on 05/02/2021) (cited on page 51).

[AGJ+21] Archita Agarwal, Marilyn George, Aaron Jeyaraj, and Malte Schwarzkopf. "Retrofitting GDPR Compliance onto Legacy Databases". In: *Proceedings of the VLDB Endowment* 15 (Dec. 2021) (cited on page 23).

[Ama23] Amazon Web Services. *Navigating GDPR Compliance on AWS: Encrypt Data at Rest*. 2023. URL: https://docs.aws.amazon.com/whitepapers/latest/navigating-gdpr-compliance/encrypt-data-at-rest.html (visited on 05/05/2021) (cited on page 42).

[ARA+19] Brooke Auxier, Lee Rainie, Monica Anderson, Andrew Perrin, Madhu Kumar, and Erica Turner. *Americans and Privacy: Concerned, Confused and Feeling Lack of Control Over Their Personal Information*. Nov. 2019. URL: https://www.pewresearch.org/

internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/ (cited on page 1).

[Arc17] Scott Arciszewski. *Building Searchable Encrypted Databases with PHP and SQL*. May 2017. URL: https://paragonie.com/blog/2017/05/building-searchable-encrypted-databases-with-php-and-sql (cited on page 42).

[ASA+21] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. "Coeus: A system for oblivious document ranking and retrieval". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 2021, pages 672–690 (cited on page 21).

[AZZ+25] Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Nathan Harbison, Mithi Jethwa, Will Crichton, Malte Schwarzkopf, and Shriram Krishnamurthi. "Paralegal: Practical Static Analysis for Privacy Bugs". In: *Proccedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2025. Forthcoming (cited on page 19).

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. "Secure Multiparty Computation Goes Live". In: *Financial Cryptography and Data Security*. Edited by Roger Dingledine and Philippe Golle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 325–343 (cited on pages 6, 21).

[BCH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, Texas, USA, 2018, pages 221–230 (cited on page 41).

[Ben24] Sergio Benitez. *rocket*. 2024. URL: https://docs.rs/rocket/latest/rocket/ (visited on 04/19/2024) (cited on pages 73, 107).

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computation". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pages 1–10 (cited on pages 6, 21, 121).

[BHP+92] Alan C Bomberger, Norman Hardy, A Peri, Frantz Charles, R Landau, William S Frantz, Jonathan S Shapiro, and Ann C Hardy. "The KeyKOS nanokernel architecture". In: *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. 1992 (cited on page 17).

[BKV+21] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. "Zeph: Cryptographic Enforcement of End-to-End Data Privacy". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 387–404 (cited on pages 3, 20).

[bla18] blackbeam. *mysql*. 2018. URL: https://docs.rs/mysql_common/latest/mysql_common/ (visited on 04/19/2024) (cited on page 73).

[BLN+24] Nataliia Bielova, Laura Litvine, Anysia Nguyen, Mariam Chammat, Vincent Toubiana, and Estelle Hary. "The effect of design patterns on (present and future) cookie consent decisions". In: *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*. 2024, pages 2813–2830 (cited on page 97).

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A framework for fast privacy-preserving computations". In: *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, MáLaga, Spain, October 6-8, 2008. Proceedings 13*. Springer. 2008, pages 192–206 (cited on page 21).

[Bra19] National Congress of Brazil. *Lei Geral de Proteção de Dados [Brazilian General Data Protection Law]*. English translation by Ronaldo Lemos, Daniel Douek, Sofia Lima Franco, Ramon Alberto dos Santos and Natalia Langenegger. 2019. URL: https://iapp.org/media/pdf/resource_center/Brazilian_General_Data_Protection_Law.pdf (visited on 06/11/2020) (cited on pages 1, 12).

[Bru21] Graeme Bruce. *Privacy and Big Tech: Gauging attitudes around the world*. Mar. 2021. URL: https://today.yougov.com/technology/articles/35025-privacy-and-big-tech-gauging-attitudes-around-worl (cited on page 1).

[BVR15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing static and dynamic typing for information-flow control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Vancouver, British Columbia, Canada, Aug. 2015, pages 289–301 (cited on page 18).

[Cal] California Attorney General. *Privacy Enforcement Actions*. URL: https://oag.ca.gov/privacy/privacy-enforcement-actions (visited on 07/31/2023) (cited on page 56).

[Car23] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. 2023. URL: https://pypl.github.io/PYPL.html (visited on 04/29/2025) (cited on page 3).

[CCPA18] California Legislature. *The California Consumer Privacy Act of 2018*. June 2018. URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375 (cited on pages 1, 5, 12).

[CDN+20] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. "DELF: Safeguarding deletion correctness in Online Social Networks". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. Banff, Canada, Aug. 2020 (cited on pages 16, 52, 105).

[Cha18] Adhityaa Chandrasekar. *Commento*. 2018. URL: https://github.com/adtac/commento (visited on 05/02/2021) (cited on page 51).

[Chl10a] Adam Chlipala. "Static checking of dynamically-varying security policies in database-backed applications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 105–118 (cited on page 18).

[Chl10b] Adam Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, 2010, pages 122–133 (cited on page 18).

[Cho25] Julian Chokkattu. *Your Next AI Wearable Will Listen to Everything All the Time*. Jan. 2025. URL: https://www.wired.com/story/bee-ai-omi-always-listening-ai-wearables/ (cited on page 1).

[CKK+20] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. "Unearthing inter-job dependencies for better cluster scheduling". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020, pages 1205–1223 (cited on page 16).

[Cla19] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Mar. 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 08/12/2024) (cited on page 79).

[Com] Federal Trade Commission. *FTC Charges Twitter with Deceptively Using Account Security Data to Sell Targeted Ads*. URL: https://www.ftc.gov/news-events/news/press-releases/2022/05/ftc-charges-twitter-deceptively-using-account-security-data-sell-targeted-ads (visited on 07/31/2023) (cited on page 14).

[CPA+22] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. San Diego, California, USA, 2022, pages 1–14 (cited on pages 68, 70, 108, 148).

[CVM07] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: enforcing confidentiality and integrity in web applications". In: *Proceedings of 16th USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2007 (cited on page 18).

[DAA+24a] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2024, pages 709–725 (cited on pages 2, 10).

[DAA+24b] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. *Sesame*. Sept. 2024. URL: https://github.com/brownsys/sesame (visited on 09/14/2024) (cited on page 58).

[DD77] Dorothy E. Denning and Peter J. Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), pages 504–513 (cited on page 18).

[DDH+22] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. "SwitchV: automated SDN switch validation with P4 models". In: *Proceedings of the 36th ACM Special Interest Group on Data Communication Conference (SIGCOMM)*. 2022, pages 365–379 (cited on page 10).

[Den13] Frank Denis. *The Sodium cryptography library*. June 2013. URL: https://download.libsodium.org/doc/ (cited on page 41).

[Des24] Damien Desfontaines. *A list of real-world uses of differential privacy*. Sept. 2024. URL: https://desfontain.es/blog/real-world-differential-privacy.html (cited on page 6).

[DIL+17] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Azer Bestavros, and Nikolaj Volgushev. "Scalable secure multi-party network vulnerability analysis via symbolic optimization". In: *Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2017, pages 211–216 (cited on page 11).

[DIL+19] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, and Ira Globus-Harris. "Tutorial: Deploying secure multi-party computation on the web using JIFF". In: *Proceedings of the 4th IEEE Secure Development Conference (SecDev)*. 2019 (cited on pages 2, 10, 11, 21, 121).

[DIV+22] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. "Batched differentially private information retrieval". In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. 2022, pages 3327–3344 (cited on pages 2, 10, 21).

[DMN+06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. "Calibrating noise to sensitivity in private data analysis". In: *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*. Springer. 2006, pages 265–284 (cited on pages 6, 21, 122).

[DRF+17] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. "cHash: Detection of Redundant Compilations via AST Hashing". In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. Santa Clara, California, USA, July 2017, pages 527–538 (cited on page 73).

[DSA+23] Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, and Malte Schwarzkopf. "K9db: Privacy-Compliant Storage For Web Applications By Construction". In: *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2023, pages 99–116 (cited on pages 2, 10).

[EK08] Petros Efstathopoulos and Eddie Kohler. “Manageable Fine-grained Information Flow”. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. Glasgow, Scotland, United Kingdom, 2008, pages 301–313 (cited on page 18).

[Eur22] European Data Protection Board. *Binding Decision 2/2022 on the dispute arisen on the draft decision of the Irish Supervisory Authority regarding Meta Platforms Ireland Limited (Instagram) under Article 65(1)(a) GDPR*. July 2022. URL: https://www.edpb.europa.eu/system/files/2022-09/edpb_bindingdecision_20222_ie_sa_instagramchildusers_en.pdf (visited on 09/14/2024) (cited on pages 2, 13, 56).

[Fac] Facebook. *Permanently Delete Your Facebook Account*. URL: https://www.facebook.com/help/224562897555674?helpref=faq_content (visited on 05/21/2023) (cited on pages 24, 30).

[FBS+23] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. “RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks”. In: *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*. San Francisco, California, USA, May 2023, pages 2817–2834 (cited on pages 19, 94, 109).

[FERPA74] *Family Educational Rights and Privacy Act of 1974*. United States Code of Laws, 20 U.S.C. § 1232g. Aug. 1974 (cited on pages 13, 55).

[FZL+23] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. “TVA: A multi-party computation system for secure and expressive time series analytics”. In: *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. 2023, pages 5395–5412 (cited on page 21).

[Gat07] Carrie Gates. “Access control requirements for web 2.0 security and privacy”. In: *IEEE Web* 2.0 (2007), pages 12–15 (cited on page 16).

[Gaz19] Thailand Government Gazette. *Personal Data Protection Act*. Unofficial English translation. 2019. URL: https://thainetizen.org/wp-content/uploads/2019/11/thailand-personal-data-protection-act-2019-en.pdf (visited on 06/11/2020) (cited on page 12).

[GBS+19] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. “Information-Flow Control for Database-Backed Applications”. In: *Proceedings of the*

4th 2019 IEEE European Symposium on Security and Privacy (EuroS&P). Stockholm, Sweden, June 2019, pages 79–94 (cited on page 17).

[GDPR16] European Union. "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)". In: *Official Journal of the European Union* L119 (May 2016), pages 1–88 (cited on pages 1, 5, 12, 24, 55).

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pages 169–178 (cited on pages 6, 21, 121).

[Gho24] Bijit Ghosh. *LLM Privacy and Security*. Oct. 2024. URL: https://medium.com/@bijit211987/llm-privacy-and-security-56a859cbd1cb (cited on page 1).

[Goo23] Google, Inc. *Google Open Source: Third-Party*. Mar. 2023. URL: https://opensource.google/documentation/reference/thirdparty (visited on 09/16/2024) (cited on page 71).

[GSB+18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 213–231 (cited on pages 26, 42).

[Har12] Dick Hardt. *The OAuth 2.0 authorization framework*. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749 (visited on 04/29/2025) (cited on page 17).

[Har18] Peter Bhat Harkins. *Lobste.rs access pattern statistics for research purposes*. Mar. 2018. URL: https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b (visited on 03/12/2018) (cited on page 43).

[HDC+23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. "Private web search with Tiptoe". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*. 2023, pages 396–416 (cited on page 21).

[HHN+19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. "Sok: General purpose compilers for secure multi-party computation". In: *Proceedings of the IEEE*

symposium on security and privacy (SP). IEEE. 2019, pages 1220–1237 (cited on pages 6, 7, 21).

[HIPAA96] *The Health Insurance Portability and Accountability Act of 1996*. United States Public Law 104-191. Aug. 1996 (cited on pages 13, 55).

[HKF+15] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. "Attribute-Based Access Control". In: *Computer* 48.2 (2015), pages 85–88 (cited on page 16).

[HZX+16] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, 2016, pages 533–549 (cited on pages 3, 7, 20).

[Ind19] PRS Legislative Research India. *The Personal Data Protection Bill, 2019*. 2019. URL: https://www.prsindia.org/billtrack/personal-data-protection-bill-2019 (visited on 06/11/2020) (cited on pages 1, 12).

[Int23] Intel. *Intel Trust Domain Extensions (TDX)*. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intetrust-domain-extensions.html (visited on 04/17/2025) (cited on page 120).

[IPC21] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. "Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach!" In: *Proceedings of the VLDB Endowment* 14.7 (Mar. 2021), pages 1167–1174 (cited on page 16).

[JDL+17] Frederick Jansen, Kinan Dak Albab, Andrei Lapets, and Mayank Varia. "Brief Announcement: Federated Code Auditing and Delivery for MPC". In: *Stabilization, Safety, and Security of Distributed Systems: 19th International Symposium, SSS 2017, Boston, MA, USA, November 5–8, 2017, Proceedings 19*. Springer. 2017, pages 298–302 (cited on page 11).

[JFD+18] Mohamad Jaber, Yliès Falcone, Kinan Dak Albab, John Abou-Jaoudeh, and Mostafa El-Katerji. "A high-level modeling language for the efficient design, implementation, and testing of Android applications". In: *International Journal on Software Tools for Technology Transfer* 20 (2018), pages 1–18 (cited on page 11).

[JP22] Vojtěch Jungmann and Sebastian Pravda. *Portfolio*. 2022. URL: https://github.com/admisio/Portfolio (visited on 04/12/2024) (cited on pages 73, 74).

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. "Private blocklist lookups with checklist". In: *Proceedings of the 30th USENIX security symposium (USENIX Security)*. 2021, pages 875–892 (cited on page 21).

[Kil21a] Benjamin Kilimnik. *GDPR - Download Data button doesn't return any data*. 2021. URL: https://github.com/shuup/shuup/issues/2614 (visited on 12/13/2021) (cited on page 50).

[Kil21b] Benjamin Kilimnik. *GDPR - shuup_mutaddress rows not anonymized*. 2021. URL: https://github.com/shuup/shuup/issues/2612 (visited on 12/13/2021) (cited on page 50).

[KKP+22] Michael Koppmann, Christian Kudera, Michael Pucher, and Georg Merzdovnik. "Utilizing Object Capabilities to Improve Web Application Security". In: *Applied Cybersecurity & Internet Governance* 1 (2022) (cited on page 17).

[Koh06] Eddie Kohler. *HotCRP conference review software*. 2006. URL: https://github.com/kohler/hotcrp (visited on 07/22/2020) (cited on pages 29, 51).

[KSB+19] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. "SchengenDB: A Data Protection Database Proposal". In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019, pages 24–38 (cited on pages 7, 16).

[Kuz18] Viktoras Kuznecovas. *Mouthful*. 2018. URL: https://github.com/vkuznecovas/mouthful (visited on 05/02/2021) (cited on page 51).

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information flow control for standard OS abstractions". In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, 2007, pages 321–334 (cited on page 18).

[LCG+21] Connor Luckett, Andrew Crotty, Alex Galakatos, and Ugur Cetintemel. "Odlaw: A Tool for Retroactive GDPR Compliance". In: *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*. Chania, Greece, Apr. 2021 (cited on page 23).

[LDI+19] Andrei Lapets, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, Azer Bestavros, and Frederick Jansen. “Role-based ecosystem for the design, development, and deployment of secure multi-party data analytics applications”. In: *Proceedings of the 4th IEEE Cybersecurity Development (SecDev)*. IEEE. 2019, pages 129–140 (cited on page 10).

[LJD+18] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. “Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities”. In: *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies (COMPASS)*. 2018, pages 1–5 (cited on pages 6, 11, 21, 121).

[LKB+21] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. “STORM: Refinement Types for Secure Web Applications”. In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 441–459 (cited on pages 7, 18, 56, 65, 73, 74, 77).

[Lob18a] Lobste.rs. *Privacy: Lobsters*. 2018. URL: https://lobste.rs/privacy (visited on 05/01/2021) (cited on page 43).

[Lob18b] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: https://lobste.rs (visited on 03/02/2018) (cited on pages 26, 43).

[Lou20] Privacy Out Loud. *CNIL announces two fines against two doctors amounting to EUR 9,000 for failure to ensure the security of patient data*. https://privacyoutloud.ro/2020/12/22/cnil-announces-two-fines-against-two-doctors-amounting-to-eur-9000-for-failure-to-ensure-the-security-of-patient-data/. Accessed: 16-03-2022. 2020 (cited on page 13).

[LP25] Carl Lerche and David Pedersen. *Axum*. 2025. URL: https://docs.rs/axum/latest/axum/ (visited on 04/27/2025) (cited on page 107).

[LPT+21] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. “Privacy budget scheduling”. In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2021, pages 55–74 (cited on pages 22, 123).

[LTB+24] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024) (cited on pages 7, 18, 56, 115).

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. "ObliVM: A Programming Framework for Secure Computation". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pages 359–376 (cited on page 21).

[Mar22] MariaDB. *MyRocks – MariaDB Knowledge Base*. 2022. URL: https://mariadb.com/kb/en/myrocks/ (visited on 12/06/2022) (cited on page 41).

[MEH+17] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy compliance for database-backed systems". In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, British Columbia, USA, Aug. 2017, pages 1463–1479 (cited on page 17).

[Met22] Meta Platforms, Inc. *RocksDB: A persistent key-value store for fast storage environments*. 2022. URL: http://rocksdb.org/ (visited on 12/10/2022) (cited on page 25).

[Mil06] Mark Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. Johns Hopkins University, 2006 (cited on page 84).

[Mir12] Ilya Mironov. "On significance of the least significant bits for differential privacy". In: *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*. 2012, pages 650–661 (cited on pages 6, 122).

[ML00] Andrew C. Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pages 410–442 (cited on pages 18, 65, 115).

[MLS+20] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Mothy Roscoe. "Shared Arrangements: practical inter-query sharing for streaming dataflows". In: *Proceedings of the VLDB Endowment* 13.10 (June 2020), pages 1793–1806 (cited on page 42).

[MMI+13a] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013 (cited on page 42).

[MMI+13b] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455 (cited on page 42).

[MMT10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Object Capabilities and Isolation of Untrusted Web Applications". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pages 125–140 (cited on pages 17, 84).

[MNL+23] Elizabeth Margolin, Karan Newatia, Tao Luo, Edo Roth, and Andreas Haeberlen. "Arboretum: A planner for large-scale federated analytics with differential privacy". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*. 2023, pages 451–465 (cited on page 21).

[MSH+16] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. "A Demonstration of the Solid Platform for Social Web Applications". In: *Proceedings of the 25th International Conference Companion on World Wide Web (WWW)*. Montréal, Québec, Canada, 2016, pages 223–226 (cited on pages 3, 7, 22).

[Mur21] Sudharsanan Muralidharan. *Socify: open source social network using Ruby on Rails*. 2021. URL: https://github.com/scaffeinate/socify (visited on 05/02/2021) (cited on page 51).

[MWC10] Adrian Mettler, David A Wagner, and Tyler Close. "Joe-E: A Security-Oriented Subset of Java." In: *NDSS*. Volume 10. 2010, pages 357–374 (cited on page 84).

[NA15] European Network and Information Security Agency. *Privacy and data protection by design: from policy to engineering*. 2015. URL: https://data.europa.eu/doi/10.2824/38623 (cited on page 24).

[NDG+20] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting fine grain isolation in the Firefox renderer". In: *Proceedings of the 29th USENIX Security Symposium*. Virtual Event, Aug. 2020, pages 699–716 (cited on pages 66, 70, 81).

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and

Venkateshwaran Venkataramani. “Scaling Memcache at Facebook”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398 (cited on page 46).

[Nor20] Noria Contributors. *Noria Lobsters benchmark*. June 2020. URL: https://github.com/mit-pdos/noria/tree/3edd3ad55d2564493f7456d27abb41abf0169def/applications/lobsters (cited on page 147).

[Nos23] Nostr. *Nostr*. https://github.com/nostr-protocol/nostr. 2023 (cited on pages 3, 22).

[NOY20a] NOYB: European Center for Digital Rights. *GDPRHub: AEPD PS/00448/2020, Xfera Móviles S.A.* 2020. URL: https://gdprhub.eu/index.php?title=AEPD_(Spain)_-_PS/00448/2020 (visited on 03/15/2025) (cited on page 13).

[NOY20b] NOYB: European Center for Digital Rights. *GDPRHub: APD/GBA 81/2020*. 2020. URL: https://gdprhub.eu/index.php?title=APD/GBA_-_81/2020 (visited on 05/06/2021) (cited on page 13).

[NOY20c] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-008*. 2020. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-008 (visited on 05/06/2021) (cited on page 56).

[NOY20d] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-018, Nestor SAS*. 2020. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-018 (visited on 05/06/2021) (cited on pages 13, 56).

[NOY20e] NOYB: European Center for Digital Rights. *GDPRHub: GPDDP 9485681, Vodafone Italia*. 2020. URL: https://gdprhub.eu/index.php?title=Garante_per_la_protezione_dei_dati_personali_-_9485681 (visited on 05/06/2021) (cited on page 56).

[NOY21] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2021-008, Brico Privé*. 2021. URL: https://gdprhub.eu/index.php?title=CNIL_(France)_-_SAN-2021-008 (visited on 03/04/2022) (cited on page 13).

[NOY22a] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2022-019, Clearview AI*. 2022. URL: https://gdprhub.eu/index.php?title=CNIL_(France)_-_SAN-2022-019 (visited on 03/15/2025) (cited on page 13).

[NOY22b] NOYB: European Center for Digital Rights. *GDPRHub: GPDP 9746068, T.S.M.* 2022. URL: https://gdprhub.eu/index.php?title=Garante_per_la_protezione_dei_dati_personali_(Italy)_-_9746068 (visited on 03/15/2025) (cited on page 13).

[NSN+24] Ivoline C Ngong, Brad Stenger, Joseph P Near, and Yuanyuan Feng. "Evaluating the usability of differential privacy tools with data practitioners". In: *Proceedings of the 20th Symposium on Usable Privacy and Security (SOUPS)*. 2024, pages 21–40 (cited on pages 6, 7, 122).

[OFl22] Kate O'Flaherty. *Apple Slams Facebook And Google With Bold New Privacy Ad*. May 2022. URL: https://www.forbes.com/sites/kateoflahertyuk/2022/05/25/apple-slams-facebook-and-google-with-bold-new-privacy-ad/ (cited on page 5).

[own21a] ownCloud GmbH. *GDPR compliant cloud storage*. 2021. URL: https://owncloud.com/gdpr (visited on 12/01/2021) (cited on page 23).

[own21b] ownCloud GmbH. *owncloud – share files and folders, easy and secure*. 2021. URL: https://owncloud.com (visited on 12/01/2021) (cited on pages 23, 26, 28, 43, 47).

[PKY+21] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. "Senate: a maliciously-Secure MPC platform for collaborative analytics". In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 2021, pages 2129–2146 (cited on page 21).

[PYI+16] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. "Type-Driven Repair for Information Flow Security". In: *CoRR* abs/1607.03445 (2016). arXiv: 1607.03445 (cited on page 18).

[QLJ+19] Lucy Qin, Andrei Lapets, Frederick Jansen, Peter Flockhart, Kinan Dak Albab, Ira Globus-Harris, Shannon Roberts, and Mayank Varia. "From usability to secure computing and back again". In: *Proceedings of the 15th USENIX Symposium on Usable Privacy and Security (SOUPS)*. 2019, pages 191–210 (cited on pages 6, 11).

[RFE15] Dhruv Rawat, Anamta Farook, and Mohsan Elahi. *Sign Me Up*. 2015. URL: https://github.com/signmeup/signmeup (visited on 04/28/2025) (cited on page 101).

[Rob19] Brent Robinson. *Crypto shredding: How it can solve modern data retention challenges*. Jan. 2019. URL: https://medium.com/@brentrobinson5/crypto-shredding-

how-it-can-solve-modern-data-retention-challenges-da874b01745b (cited on page 13).

[RPB+09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. "Laminar: practical fine-grained decentralized information flow control". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, 2009, pages 63–74 (cited on pages 18, 115).

[Rub17] Alexander Rubin. *One Million Tables in MySQL 8.0*. 2017. URL: https://www.percona.com/blog/2017/10/01/one-million-tables-mysql-8-0/ (visited on 05/03/2021) (cited on page 48).

[Rub18] Alexander Rubin. *40 million tables in MySQL 8.0 with ZFS*. 2018. URL: https://www.percona.com/blog/2018/09/03/40-million-tables-in-mysql-8-0-with-zfs/ (visited on 05/03/2021) (cited on pages 41, 48).

[RZH+20] Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C Pierce. "Orchard: Differentially private analytics at scale". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pages 1065–1081 (cited on page 21).

[San98] Ravi S Sandhu. "Role-based access control". In: *Advances in computers*. Volume 46. Elsevier, 1998, pages 237–286 (cited on page 16).

[SBR+11] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. "Logical attestation: an authorization architecture for trustworthy computing". In: *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 2011, pages 249–264 (cited on page 18).

[SBW+19] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. *Understanding and Benchmarking the Impact of GDPR on Database Systems*. 2019. arXiv: 1910.00728 [cs.DB] (cited on page 7).

[SBW+20] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. "Understanding and Benchmarking the Impact of GDPR on Database Systems". In: *Proceedings of the VLDB Endowment* 13.7 (Mar. 2020), pages 1064–1077 (cited on pages 7, 13, 15, 16, 23).

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. "Secure distributed programming with value-dependent types". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pages 266–278 (cited on page 18).

[SCH08] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. "Fable: A language for enforcing user-defined security policies". In: *Procedings of the 29th IEEE Symposium on Security and Privacy (S&P)*. Oakland, California, USA, May 2008, pages 369–383 (cited on page 18).

[sch17] schnack! *schnack.js*. 2017. URL: https://github.com/schn4ck/schnack (visited on 05/02/2021) (cited on page 51).

[Sch20] Malte Schwarzkopf. *websubmit-rs: a simple class submission system*. 2020. URL: https://github.com/ms705/websubmit-rs (visited on 06/03/2020) (cited on page 73).

[Ser25] Amazon Web Services. *Policies and permissions in AWS Identity and Access Management*. 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html (visited on 04/29/2025) (cited on page 16).

[Sha18] Faiyaz Shaikh. *React-Instagram-Clone-2.0*. 2018. URL: https://github.com/yTakkar/React-Instagram-Clone-2.0 (visited on 05/02/2021) (cited on page 51).

[Sha79] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pages 612–613 (cited on pages 6, 21, 121).

[Shu18] Shuup Commerce, Inc. *Shuup Open-Source E-Commerce Platform*. 2018. URL: https://github.com/shuup/shuup (visited on 12/05/2021) (cited on pages 26, 36, 43, 49, 50).

[SKK+19] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "Position: GDPR Compliance by Construction". In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019 (cited on page 8).

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: a fast capability system". In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '99. Charleston, South Carolina, USA: Association for Computing Machinery, 1999, pages 170–185 (cited on page 17).

[Sta24] Federal Trade Commission (FTC) Staff in the Office of Technology. *AI Companies: Uphold Your Privacy and Confidentiality Commitments*. Jan. 2024. URL: https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2024/01/ai-companies-uphold-your-privacy-confidentiality-commitments (cited on page 1).

[SWC19] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. "How Design, Architecture, and Operation of Modern Systems Conflict with GDPR". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. July 2019 (cited on page 12).

[Sza22] Gábor Szabó. *Programming language popularity: Rust*. 2022. URL: https://szabgab.com/programming-language-popularity-rust (visited on 04/29/2025) (cited on page 3).

[TB21] Chris Tsang and Chan Billy. *SeaORM: An async & dynamic ORM for Rust*. 2021. URL: https://crates.io/crates/sea-orm (visited on 09/17/2024) (cited on page 73).

[Tho19] Griffin Thorne. *GDPR Meets its Match ... in China*. July 2019. URL: https://www.chinalawblog.com/2019/07/gdpr-meets-its-match-in-china.html (visited on 06/04/2020) (cited on pages 1, 12).

[TKM+24] Pierre Tholoniat, Kelly Kostopoulou, Peter McNeely, Prabhpreet Singh Sodhi, Anirudh Varanasi, Benjamin Case, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. "Cookie Monster: Efficient On-device Budgeting for Differentially-Private Ad-Measurement Systems". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 2024, pages 693–708 (cited on page 22).

[VSG+19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. "Conclave: secure multi-party computation on big data". In: *Proceedings of the 14th EuroSys Conference*. EuroSys '19. Dresden, Germany, 2019 (cited on page 21).

[WKM19] Frank Wang, Ronny Ko, and James Mickens. "Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Feb. 2019, pages 615–630 (cited on pages 7, 18, 56, 109).

[WKN+22] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. "PrivGuard: Privacy regulation compliance made

easier". In: *Proceedings of the 31st USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2022, pages 3753–3770 (cited on page 19).

[Yao86] Andrew Chi-Chih Yao. "How to generate and exchange secrets". In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 1986, pages 162–167 (cited on pages 6, 21, 121).

[YHA+16] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, dynamic information flow for database-backed applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, 2016, pages 631–647 (cited on pages 18, 19, 56).

[YWZ+09] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Improving Application Security with Data Flow Assertions". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pages 291–304 (cited on pages 7, 18, 19, 56, 65, 109, 115).

[YYR21] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter". In: *ACM Transactions on Storage* 17.3 (Aug. 2021) (cited on page 40).

[ZBK+06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Maziéres. "Making Information Flow Explicit in HiStar". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006, pages 263–278 (cited on pages 7, 18).

[ZBM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, California, USA, Dec. 2008 (cited on page 18).

[Zda04] Steve Zdancewic. "Challenges for information-flow security". In: *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*. Volume 6. 2004 (cited on page 82).

[ZSC+22] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. "Blockaid: Data access policy enforcement for web applications". In: *Proceedings of the*

16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2022, pages 701–718 (cited on page 17).

Appendix A

K9db Artifact

Abstract

Our open source artifact contains our prototype implementation of K9db. It also includes the harnesses and scripts for running and plotting the experiments described in this dissertation and the K9db paper from OSDI 2023.

Our prototype provides a MySQL-compatible interface layer, which applications and developers can use to issue SQL statements and queries to and retrieve their results. Our prototype is compatible with the standard MySQL connectors and drivers for several languages, including C++, Rust, and Java. It is also compatible with the command line MySQL and MariaDB clients.

Scope

Our prototype serves as a demonstration of the following:

1. The application scenarios described in §3 work with K9db and its schema annotations.
2. K9db's system design and guarantees can be realized with a familiar MySQL-compatible interface suitable for web applications.
3. The performance of compliant-by-construction databases is comparable to traditional databases, such as MariaDB.

Contents

K9db. The artifact includes our prototype implementation and its MySQL-compatiblity layer. The artifact contains instructions for building, running, and using this K9db.

Application Harnesses. The artifact includes harnesses for Lobsters, a Reddit-like discussion board (§8.1.1), and ownCloud (§8.1.2), a file sharing application. The harnesses create the database schema and load the database with data; they also execute loads with representative queries, and measure the time required to process them. We used these harnesses to evaluate our prototype and the baselines shown in our experiments. The Lobsters harness is a pre-existing open source harness that we adapted to work with our prototype [Nor20].

Documentation. The artifact wiki on GitHub contains a tutorial on using K9db and its schema annotations. The artifact also includes unit and end-to-end tests that validate that our prototype handles application SQL operations correctly and provides correct compliance with SARs.

Hosting

Our artifact is hosted on GitHub at `https://github.com/brownsys/K9db`.

The version of the repository corresponding to the experimental results shown in this dissertation is available at `https://github.com/brownsys/K9db/releases/tag/osdi2023`, with commit hash *df2bcdffa05f70f508fad95a11e2a6de8a7efe14*.

The corresponding wiki commit hash is *c720b085ca34edc16246f296991e623a29933f9b*.

Requirements

We developed our prototype on x86-64 machines running Ubuntu 20.04 and 22.04. Our prototype is built using Bazel 4.2.1. We provide a Docker container that includes the necessary software dependencies. We ran our experiments on Google Cloud using `n2-standard-16` machines with a local SSD.

Appendix B

Scrutinizer

SCRUTINIZER Code. SCRUTINIZER is open-source software at https://github.com/brownsys/scrutinizer. Our experiments in this dissertation and the Sesame paper from SOSP 2024 used the version tagged `sosp24`.

SCRUTINIZER Analysis Details. SCRUTINIZER follows a two-stage approach to check the properties in §4.5.1.

First, SCRUTINIZER builds a call tree of all functions and code that may be executed by the top-level function under analysis. SCRUTINIZER uses Rust's dataflow analysis framework to traverse function bodies recursively in execution order. This discovers all possible function bodies that the top-level function could call, and organizes them into a call tree. When it encounters dynamic dispatch, SCRUTINIZER attempts to construct a superset of all concrete functions the dynamic dispatch may resolve to, and analyzes all of them. If SCRUTINIZER cannot construct such a set, it rejects the function. SCRUTINIZER keeps track of functions it visited to avoid unnecessary recomputation. This stage finishes when SCRUTINIZER discover no more new function calls.

Second, SCRUTINIZER begins the analysis stage. SCRUTINIZER rejects a top-level function if it captures any variables with a mutable reference, since such sensitive data could leak into such variables. For top-level functions that pass this check, SCRUTINIZER labels the arguments to the function as "sensitive". It then traverses every statement in the call tree while simultaneously propagating the "sensitive" label to aliases and derived variables using Flowistry [CPA+22]. This ensures that SCRUTINIZER keeps track of sensitive arguments as they are passed, aliased, and derived from throughout the call tree. If

SCRUTINIZER encounters a function call into native or otherwise unresolvable code that sensitive variables flow into, it rejects. If SCRUTINIZER encounters a function call to an allow-listed function, or with arguments that lack the sensitive label, it skips it. Otherwise, SCRUTINIZER analyzes the function's body. If SCRUTINIZER encounters an unsafe mutability mechanism, such as a raw mutable pointer dereference, it rejects. If SCRUTINIZER finishes analyzing the call tree without rejecting, it accepts the top-level function.