Enabling Personal Dataflow Sovereignty via Bolt-on Data Escrow

Zhiru Zhu
The University of Chicago
Chicago, IL, USA
zhiru@uchicago.edu

Raul Castro Fernandez
The University of Chicago
Chicago, IL, USA
raulcf@uchicago.edu

Abstract

The digital economy is powered by a continuous and massive exchange of personal data. Individuals provide data to platforms in return for services, from social networking and search to health monitoring, entertainment, and access to LLMs. This exchange has created immense value, but it has also established a fundamental asymmetry of power: individuals possess only coarse-grained control over data access rather than fine-grained control over its purpose of use, creating a gap where data can be repurposed for undisclosed uses, e.g., platforms selling the data to data brokers, which results in a critical loss of *personal data sovereignty*.

This paper frames this socio-technical challenge as a *dataflow* management problem. We propose a *bolt-on data escrow* architecture through *delegated computation*. In our model, instead of data flowing to platforms, platforms delegate their computation to a trustworthy escrow. This inversion empowers individuals with transparency and control over their dataflows. We present four contributions: (1) a dataflow model that explicitly incorporates computational purpose as a first-class primitive; (2) a minimally invasive programming interface, RUN(ACCESS(), COMPUTE()), built on a unified relational interface that virtualizes on-device data sources and a computation offloading component; (3) a concrete implementation of our escrow within the Apple ecosystem, demonstrating its practicality; and (4) both qualitative and quantitative evaluations demonstrating that our solution is expressive enough to implement a wide range of dataflows from real-world applications and introduces minimal runtime overhead. In summary, our work serves as a stepping stone toward achieving *personal dataflow sovereignty*.

Keywords

Data Escrow, Dataflow, Personal Data Sovereignty

1 Introduction

Modern digital life runs on personal data. Every day, billions of individuals exchange sensitive data with platforms to enable services such as search, social media, health monitoring, entertainment, and access to LLMs. The prevailing model is simple: applications (*apps*) pull sensitive data (e.g., photos, contacts, location, health records) off individuals' devices and process the data on the platform's infrastructure. The moment a *dataflow* crosses the individual's *trust zone*—from the individual and their device to a remote platform—transparency and enforcement over *how* the data will be used are lost. While existing permission systems gate access to raw data, they fail to bind or expose the *purpose of its use*. This gap explains why seemingly benign requests (e.g., "share location for weather") can covertly support unrelated uses such as profiling or resale.

We frame this as a data management problem. Specifically, a *dataflow* management problem. We model a dataflow as a tuple $\langle A_s, D_s, f(), D_t, A_t \rangle$ that moves data D_s from a source agent A_s to a target agent A_t via computation $f()$ that produces D_t. This makes the *purpose* $f()$ explicit and enables *dataflow preferences* that subsume traditional privacy preferences: individuals want control not just over *what* data is shared, but *with whom and for what purpose*, as well as where and how that computation runs based on preferences over performance, battery, cost, and others.

Key idea: Bolt-on Data Escrow via Delegated Computation. We invert the prevailing model: instead of sending individuals' data to platforms, platforms *delegate* their computation to a trustworthy intermediary, a *data escrow* that resides within the individual's *trust zone*. App developers specify their computation using a minimally invasive programming interface, RUN(ACCESS(), COMPUTE()). The ACCESS() function is built on top of a *unified relational interface* that virtualizes on-device data sources containing individuals' sensitive data, allowing developers to declaratively specify their data needs using standard SQL. The COMPUTE() function contains the concrete implementation of the computation $f()$. The escrow executes both functions and returns only the result D_t. Developers never gain access to the raw D_s, and every dataflow is explicit and auditable by construction. The same interface also supports *offloading* COMPUTE() across devices and escrow-controlled servers—allowing principled trade-offs between privacy, performance, battery, and cost.

Why the Escrow is Deployable. Modern app ecosystems already concentrate data access behind Software Development Kits (SDKs) [54, 71, 99]. We design an escrow that is *bolted onto* Apple SDKs to enable practical path toward deployment. Apple can enforce the escrow as the bottleneck for data access through either its App Review process [13] or OS-level interventions. Specifically, by leveraging its existing enforcement mechanisms, such as entitlements [15], Apple can ensure that any app attempting to bypass the RUN() interface to access sensitive data directly is either rejected during app review or terminated at runtime. The bolt-on design ensures that the only parties that see the change are developers, who already constantly adapt to mandatory SDK changes. Unlike clean-slate solutions that require total ecosystem rewrites, our design integrates into the existing SDKs' update lifecycle.

Scope and limits. First, the main scope of our work focuses on establishing the *mechanism* (the escrow) required to enforce dataflow preferences. While navigating how individuals practically define and interact with these preferences (i.e., the *policy*) remains an open human-computer interaction challenge, addressing it requires a viable escrow architecture to exist first. Second, the escrow supports

Proceedings on Privacy Enhancing Technologies 2026(4), 837–851

https://doi.org/10.56553/popets-2026-0147

dataflows involving a single individual's data. If a computation calls third-party APIs, the escrow also makes this dataflow transparent (what leaves the device and why). However, fully supporting computation that combines multiple individuals' data or platform-confidential computation that cannot be disclosed remains future work. In these cases, our current model still improves the status quo by forcing platforms to use the escrow to implement a data copy that is subject to individuals' explicit approval.

Evaluation. We conduct a comprehensive evaluation that addresses three research questions regarding the escrow's feasibility, efficiency, and design. First, we perform a qualitative analysis, porting the dataflows of 10 popular, open-source iOS applications (e.g., Wikipedia[89], DuckDuckGo [35]) to our escrow model to demonstrate whether the escrow's programming interface is expressive enough for real-world use. Our analysis shows that most dataflows can be effectively reimplemented using the escrow. Second, to demonstrate the practicality of the escrow, we conduct a quantitative evaluation by comparing the runtime of the escrow-based app with a baseline app that uses native Apple SDKs directly. Our results show that the escrow introduces very little overhead, due to the efficiency of the escrow's relational engine implementation. Finally, we evaluate three different strategies for implementing the escrow's relational engine to identify the performance trade-offs. We found that by implementing classic database query optimization techniques such as predicate pushdown, one of our strategies can achieve superior performance while still preserving data freshness. Together, these experiments provide convincing evidence that our data escrow is not only expressive enough to capture the complex dataflows of real-world applications but also introduces negligible overhead, making it a practical and deployable solution for enabling individuals to gain transparency and control over their dataflows.

Contributions. This paper makes the following contributions:

- A *dataflow model* that elevates the *purpose* ($f()$) and *trust zones*, enabling purpose-based, resource-aware dataflow preferences.
- A *data escrow architecture* centered on a programming interface, RUN(ACCESS(), COMPUTE()), built on top of a unified relational interface and a computation offloading component.
- An *escrow design and implementation* in the Apple ecosystem. First, we map out the design space of the escrow and propose enforcement mechanisms to make the escrow's use mandatory for each design. We then explain the design of the escrow's relational engine; specifically, the relational schema design principles and three implementation strategies (Materialized Tables, Virtual Tables, and Virtual Tables with Pushdown). Finally, we provide detailed technical descriptions of the escrow's relational engine implementation using Virtual Tables with Pushdown and its end-to-end computation offloading pipeline.
- Both *qualitative and quantitative evaluation* showing that the escrow is expressive to implement dataflows from a wide range of real-world apps and introduces little performance overhead.

Outline. The rest of the paper is organized as follows. Section 2 introduces the dataflow model to reason about personal dataflow sovereignty. Section 3 presents the data escrow architecture and its programming interface. Section 4 describes the design of our escrow in the Apple ecosystem. Section 5 details the technical aspects of our escrow implementation. Section 6 evaluates the escrow's expressiveness and performance. Section 7 discusses the related work, and Section 8 concludes.

2 A Model for Personal Dataflow Sovereignty

To systematically address the problem of dataflow control, this section develops a model that allows us to reason about the flow of data between different entities. We begin by defining the primary *agents* and the concept of a *trust zone* in the app ecosystem (Section 2.1). We then introduce the dataflow as the core unit of analysis (Section 2.2). Building on this, we define dataflow preferences as a richer form of control that subsumes traditional privacy preferences (Section 2.3). Finally, we use this model to articulate the fundamental transparency and control deficits in the current app ecosystem that our work aims to solve (Section 2.4).

2.1 Agents and Trust Zones

We define an *agent* as any entity with the ability to store and process data. There are many such agents in today's app ecosystem. We offer a distilled view of the ecosystem that highlights the agents that play a major role in the problem outlined in the introduction and the solution we present in this paper.

- The **Individual** is the data subject, the person whose personal data is at the center of these interactions.
- The **Device** is the hardware owned and operated by the individual, such as a smartphone, wearable, or computer. It is the primary repository for much of the individual's sensitive data.
- The **Platform** is the entity providing a service, such as a social media company, a search engine, or a health application provider. Platforms are represented by the applications running on the device and the backend infrastructure they control.
- The **Developer** is the entity employed by the **Platform** to develop applications.

Central to our model is the concept of the *trust zone*. We define an individual's trust zone as the set of agents that the individual permits to access their raw data without restriction. In the prevailing model, this zone typically encompasses the individual and their personal devices. A dataflow is said to *cross the trust zone* when its target agent lies outside this boundary. For instance, when a user (Individual) interacts with a social media app such as Instagram on their iPhone, the iPhone (Device) is within the user's trust zone. However, Instagram is operated by Meta (Platform), which is outside the trust zone. When the app requests access to the user's location or photos and transmits that data to Meta's servers for processing, that dataflow crosses the trust boundary, moving out from a trusted domain. It is precisely these boundary-crossing dataflows that require more control.

Figure 1(a) provides a visual representation of today's app ecosystem, showing how an individual's data flows from their device, across the trust zone, to the platform's infrastructure where it is processed for purposes that are often opaque to the individual.

2.2 Defining and Characterizing Dataflows

Individuals share their data with platforms, inducing data to flow. We model such flow with a *dataflow*.

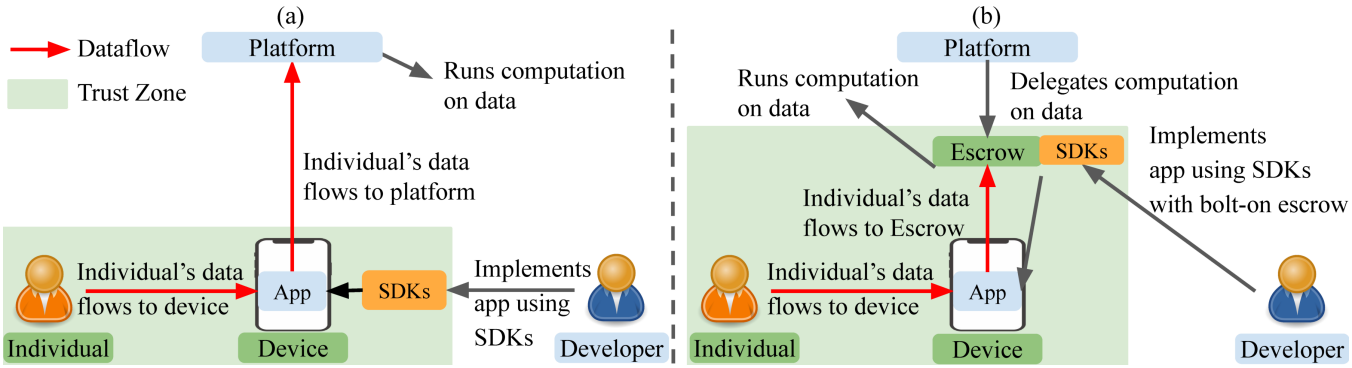

Figure 1: Dataflows in today's app ecosystem vs with the data escrow intermediary

Definition 1 (Dataflow). *A dataflow is a tuple* $< A_s, D_s, f(), D_t, A_t >$, *that indicates how the data* D_s *from a source agent* A_s *is transformed via function* $f()$ *to produce data* D_t, *which is accessible to a different target agent* A_t.

A dataflow captures *what* data is shared (D_s), with whom (A_t), and for what purpose[1] ($f()$). Using this model, we can characterize various common operations. A simple data copy, such as uploading a contact list to a platform's server, is a dataflow where $f()$ is the identity function and thus $D_s = D_t$. A more complex operation, like a weather app using an individual's precise location (D_s) to retrieve the local forecast (D_t), involves a dataflow where $f()$ is the API call to the weather service. The key insight is that the function $f()$ explicitly defines the purpose of the data use.

We are primarily interested in dataflows that involve individuals' sensitive personal data such as contacts, health, photos, and location data. While this data is used for purposes many individuals consider positive, such as sharing content with friends on social media or receiving localized weather forecasts, it has also been at the center of widespread privacy violations over the past two decades. Numerous high-profile incidents have revealed how platforms repurpose this data for undisclosed purposes: selling it to data brokers [32], enabling targeted surveillance [31], training AI models without consent [94], or enabling discriminatory practices in housing and employment [80]. The opacity of these dataflows, where individuals cannot verify that their location shared for weather updates isn't also sold to advertisers, or that their health data isn't used to adjust insurance premiums, represents a critical failure of the current ecosystem to respect individual preferences and societal values around data use.

2.3 Dataflow Preferences

Individuals have preferences on how their data should be used, which go beyond privacy preferences. By making the purpose $f()$ an explicit part of our model, we define a richer and more powerful form of individual preferences that *subsumes* traditional privacy preferences. Standard privacy preferences as presented by apps today are often binary, revolving around access to data (e.g., "Allow access to Location?") [5, 39, 85]. *Dataflow preferences*, in contrast, enable nuanced, purpose-based control over individuals' data.

We argue that an individual's privacy is preserved only when individuals' dataflow preferences are preserved. An individual may be perfectly willing to permit a dataflow where their location data (D_s) is used by a weather app (A_t) for the purpose of fetching a forecast ($f()$). However, they may wish to deny a different dataflow where the same location data is sent to an advertising broker (A_t) for the purpose of profiling ($f()$). This aligns with the privacy framework of Contextual Integrity [61], which states that adequate privacy protection can only be offered when information norms of a specific context are respected. Our dataflow model provides the formal structure needed to define and enforce these norms. It allows individuals to express preferences not just on what data is shared, but for what purpose, which is a more accurate reflection of real-world privacy expectations.

Furthermore, dataflow preferences extend beyond traditional privacy preferences. For instance, an individual might specify a preference to only allow computationally intensive dataflows (e.g., video processing) to run when their device is connected to Wi-Fi and charging, in order to conserve battery life and mobile data. Another individual might prefer that certain computations are offloaded to a server for faster performance. These dataflow preferences concerning resource consumption, performance, and cost are not related to privacy in the classic sense, but they are critical aspects of controlling one's data use. By capturing a broad set of dataflow preferences, our model allows us to reason about *personal dataflow sovereignty*—individuals' fundamental right to control their dataflows is the right to enforce their dataflow preferences.

2.4 The Transparency and Control Deficits

The dataflow model allows us to articulate two key challenges that result in the loss of *personal dataflow sovereignty*:

- **Lack of Transparency**: Individuals do not have clear visibility on what dataflows take place when using an app. When an app requests their data, the purpose $f()$ is at best implied and at worst intentionally obscured [5, 30, 72, 93, 95]. The terms of service that supposedly govern data use are notoriously broad and vague, failing to specify the concrete computations that will be performed on the data [62].

[1] Technically, $f()$ limits the purposes for which the data can be used, rather than exactly determining the purpose, but the difference is not important for the technical presentation of this paper.

- **Lack of Control**: Even if the purpose $f()$ were made transparent, individuals currently lack the technical mechanisms to enforce their preferences on the dataflows. The only available control is a blunt instrument: denying access to the source data D_s entirely. This often forces an undesirable trade-off, as denying access may break the primary, desired functionality of the app, leaving the individual with an all-or-nothing choice.

The goal of this paper is to design and implement the technical *mechanism* that endows individuals with both *transparency* and *control* over their dataflows, providing the necessary foundation for future *policy* design on eliciting individuals' dataflow preferences.

3 The Data Escrow Architecture

To address the transparency and control deficits identified in the current app ecosystem, this section details the design of our proposed *data escrow* architecture. We first introduce our key idea of employing the delegated computation model (Section 3.1). We then present a minimally invasive programming interface to enable delegated computation (Section 3.2). Next, we introduce computation offloading as a core architectural feature that enables moving computation across devices (Section 3.3). We then describe the data virtualization layer that provides a unified relational interface for transparent data access (Section 3.4). Finally, we provide a taxonomy of common dataflow patterns to clarify the capabilities and limitations of our solution (Section 3.5).

3.1 Key Idea: Delegated Computation Model

The key idea of our solution is to replace the traditional model of sending individuals' data to the platform, illustrated in Figure 1(a), with a *delegated computation* model. This architectural shift is visualized in Figure 1(b). Instead of the platform pulling an individual's data (D_s) from their device to run computation ($f()$) on its own opaque infrastructure, which crosses the individual's trust zone, the platform must delegate its computation to a trustworthy *data escrow* [91] intermediary (i.e., it must be in the individual's trust zone). This inversion of control is made possible because modern applications are not built from scratch; they are built using *Software Development Kits (SDKs)* provided by the ecosystem owner (e.g., Apple or Android). These SDKs provide the standard tools and APIs for all essential functions, including access to sensitive personal data. Our key insight is that an escrow can be *bolted onto* these existing SDKs. Developers, who already rely on these SDKs to write apps, would now use an escrow-based programming interface to specify their computation.

As shown in Figure 1(b), the developer implements their app against this new programming interface, effectively delegating their intended computation, $f()$, to the escrow. The escrow, operating as a trustworthy intermediary, then pulls the required data (D_s) from the device and executes the $f()$ on behalf of the platform. This execution only proceeds if the individual approves the entire dataflow. Because the escrow mediates all access to sensitive data, it becomes a natural *bottleneck*, making every dataflow inherently transparent and controllable by design. The fundamental trust assumption shifts: the individual no longer needs to trust every application platform but instead places their trust in the escrow provider.

3.2 The RUN(ACCESS(), COMPUTE()) Interface

To enable delegated computation, we design a minimally invasive programming interface centered on a higher-order function: RUN(ACCESS(), COMPUTE()), which has two key parameters:

- ACCESS(): The *data access function* that specifies the input data, D_s, required for the subsequent computation $f()$. As detailed in the next section, developers implement this function declaratively by writing a standard SQL query.
- COMPUTE(): The *computation function* that contains the concrete implementation of $f()$. It takes the output of ACCESS() as its input, and it can use any libraries or call third-party APIs.

The RUN() function is the core entry point executed by the escrow. When an application calls RUN(), the escrow orchestrates the entire dataflow. It first executes the ACCESS() function to retrieve D_s. It then executes the COMPUTE() function within an isolated execution environment managed by the escrow, passing D_s as the input, to produce the result D_t. The critical guarantee of this interface is that the platform and its developers never gain access to the raw data D_s. They provide the code for ACCESS() and COMPUTE(), and they receive the final output D_t, but D_s is only ever handled by the escrow within the individual's trust zone. This cleanly separates the specification of computation from the access to raw data.

Delegated Computation in Action. To make the delegated computation model concrete, consider a simple weather app. The app's goal is to use the individual's current location (D_s) to fetch the local weather forecast (D_t). Without the escrow, a developer would use the native SDK to directly fetch the device's location coordinates, which are considered sensitive, then send those coordinates across the individual's trust zone to the platform's server. A simplified code snippet (in Apple's Swift language) might look like this:

```
// Use native SDK to fetch raw location coordinates
let location = LocationManager.getCurrentLocation()
// Send raw data across the trust zone to the platform
let weather = PlatformAPI.fetchWeather(for: location)
```

In this common pattern, the raw location coordinates leave the individual's device. The individual has no technical guarantee that the platform is only using the data to fetch the weather and not for other purposes, such as selling it to data brokers. Next, we show how the developer writes the app using the RUN() interface.

```
import Escrow
// Declaratively specify the data needed
let locationSQL = "SELECT cllocation FROM Location
                   ORDER BY timestamp DESC LIMIT 1"
// Define the computation (purpose)
func getWeather(location: CLLocation) -> Weather {
    // Only sends an approximate region to platform
    let region = getSurroundingRegion(from: location)
    return PlatformAPI.fetchWeather(for: region) }
// Delegate computation to escrow
let weather = Escrow.run(locationSQL, getWeather)
```

In this revised implementation, the app never handles the raw location data. Since the platform only needs the approximate region around the individual's current location to fetch the weather, the app only sends the approximate region to the platform. The location SQL query and the `getWeather` function are submitted to

Pattern	Within individual's trust zone	Computation with external API call	Cross-individual computation	Confidential computation
Description	Computation runs fully within trust zone using a single individual's data	Computation calls third-party or platform API, sending individual's data outside the trust zone	Computation combines data from multiple individuals to produce aggregate results	Platform cannot or will not reveal computation
Example	An app scans individual's photo library to tag pictures of their pets using image classification model hosted on-device or in an escrow-controlled server	A weather app gets individual's location, calculates an approximate region based on the location, and sends the region to a third-party weather API to fetch forecast	A mapping service using location data from multiple individuals to calculate real-time traffic conditions	A social media app using a proprietary algorithm to recommend friends from individual's contacts
Supported by escrow	Fully supported	Supported with transparency - The escrow makes data transfer transparent	Currently supported by copying each individual's data to platform	Currently supported by copying each individual's data to platform

Table 1: Taxonomy of dataflow patterns and whether they are supported by the escrow's delegated computation model

the escrow. The escrow executes the query, passes the resulting `CLLocation` [7] object to the `getWeather` function, then computes and sends the approximate region to the platform to fetch weather, and finally returns only the `Weather` object to the app. The entire operation is transparent to the individual, who can verify the purpose and approve or deny this dataflow, with the understanding that their raw location coordinates never leave their trust zone. Alternatively, if the platform needs the raw location instead of the approximate region to fetch weather, the developer can still implement the same logic as before, which directly sends the raw data to the platform. This explicit data copy will be transparent to the individual via the escrow. Unlike the non-escrow version, where individuals do not know what happens to their data after they grant their permission for the app to access their location, they now have a clear idea that their raw location data is sent to the platform, and they can block the escrow from executing this data copy.

3.3 Computation Offloading

The delegated computation model naturally extends to support flexible execution locations. While many computation can run efficiently on-device, some may be too resource-intensive or require access to server-side resources. The practice of computation offloading—transferring intensive tasks to an external location—is a well-established technique to overcome device limitations such as computational power, storage, and energy [55]. Our escrow architecture allows COMPUTE() functions to be offloaded to a trustworthy, *escrow-controlled* server. This is achieved by extending the RUN() function with an optional SERVER_ID parameter: RUN(ACCESS(), COMPUTE(), SERVER_ID). When SERVER_ID is provided, the on-device escrow first executes the ACCESS() function to obtain the input data D_s. It then serializes D_s and transmits it, along with the signature of the COMPUTE() function to be executed, to the specified remote server via a secure channel. The remote server, which is managed by the escrow provider and thus remains within the individual's trust zone, executes the computation and returns the serialized result D_t. This offloading mechanism allows developers to balance on-device resources against the power of server-side processing. It aligns well with emerging industry trends such as Apple's Private Cloud Compute (PCC) [6], which provides a secure server infrastructure designed for privacy-preserving computation offloaded from mobile devices. Our escrow architecture provides a general framework that could be directly implemented on top of such infrastructure, leveraging its strong, hardware-enforced security guarantees.

3.4 Data Virtualization via Relational Interface

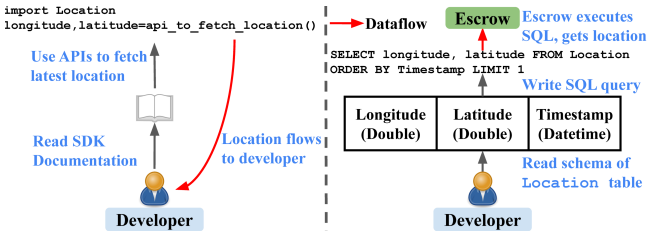

Figure 2: Accessing location using native SDKs (left) vs using the escrow's relational interface (right).

A key design decision is how developers should implement the ACCESS() function. Simply allowing developers to write arbitrary code would make the data access function opaque, effectively hiding a secondary computation within the ACCESS() function itself. To ensure transparency, we design a *declarative* interface based on the classic data virtualization approach [28].

The escrow virtualizes the device's heterogeneous data sources, such as contacts, photos, and location updates, into a unified relational schema. Instead of using numerous native SDK APIs to access data, developers now implement ACCESS() by writing a standard SQL query against this public schema. For example, to fetch an individual's most recent location, a developer would write an SQL query rather than importing the `Location` framework and writing imperative code to request location updates. Figure 2 contrasts these two approaches. The traditional approach (left) requires developers to learn SDK-specific APIs and gives them direct access to the resulting location. The escrow's relational interface (right) simplifies the data access to a declarative query and ensures the developer never handles the raw location data directly.

3.5 Taxonomy of Dataflow Patterns

The delegated computation model is designed to give individuals control over a wide range of dataflows, but its applicability varies depending on the nature of the computation. To clarify the scope of

the escrow, we provide a principled categorization of four common dataflow patterns in Table 1.

The escrow fully supports computations that are self-contained within an individual's trust zone, which includes computation on-device or in escrow-controlled servers through computation offloading. This category covers a wide range of common and important dataflows in today's apps. The escrow also supports computation that require calling external APIs, which makes the data transfer transparent. However, fully supporting dataflows that are cross-individual or confidential remains future work. For instance, computing real-time traffic conditions requires combining location data from many individuals, and platforms may be unable to delegate proprietary algorithms that are trade secrets. In those cases, the escrow still forces platforms to implement a data copy. This still represents a significant improvement over the status quo: the data copy becomes transparent and subject to the individual's explicit approval, empowering them with a meaningful choice that is often absent in today's ecosystem. Leveraging technologies such as Secure Multi-Party Computation (SMPC) [27] and Homomorphic Encryption (HE) [2] offers a theoretical pathway to support these patterns within the delegated model. For instance, if mobile hardware and network bandwidth evolve to support the overhead of these protocols efficiently [44], the escrow could orchestrate MPC dataflows where multiple devices jointly compute aggregate computation without revealing individuals' data, or evaluate HE-encrypted proprietary algorithms.

4 Escrow in Apple Ecosystem

To demonstrate the practicality of our proposed escrow architecture, we design and implement a data escrow within the Apple ecosystem. We begin by explaining our rationale for choosing the Apple ecosystem and how it provides a natural bottleneck for data access (Section 4.1). We then describe mechanisms to make the escrow's use mandatory for each escrow design (Section 4.2). Finally, we describe our schema design principles and three implementation strategies of the escrow's relational engine (Section 4.3).

4.1 Exploiting Apple SDKs as Bottleneck

Mobile devices are the primary repositories of sensitive personal data, and the two dominant ecosystems are Apple's iOS and Google's Android [100]. While our escrow architecture could be adapted to either, we choose to implement our prototype in the Apple ecosystem, with Apple acting as the trustworthy escrow, as its vertically integrated nature offers a clear path to deployment. A successful escrow must become the *bottleneck* to mediate all sensitive data access. The Apple ecosystem facilitates the creation of such a bottleneck for three key reasons: **First**, today, to access sensitive personal data such as health records, photos, contacts, or location, developers are required to use specific Apple SDKs (e.g., Contacts [9], PhotoKit [11], CoreLocation [14]). There are no alternative methods to access the data. **Second**, all apps distributed through Apple's App Store must undergo a rigorous review process before they can be published [13]. The App Review provides a centralized point of control to ensure that all data access in the app goes through the escrow. **Third**, individuals generally exhibit a high degree of trust in their Apple devices and, by extension, in Apple as the platform vendor. This pre-existing trust makes the platform vendor a natural and suitable candidate to assume the role of the data escrow, as it would seamlessly fit within the individual's existing trust zone. Combining these factors makes it feasible to create an escrow that can be adapted to the existing Apple infrastructure.

Adapting escrow to Android ecosystem. Enforcing the escrow on Android presents distinct challenges due to Android's support for third-party app stores, Original Equipment Manufacturer (OEM) customizations (i.e., companies such as Samsung and Xiaomi building customized Android operating systems on top of Google's standard Android and offering their dedicated app stores), and sideloading (i.e., individuals can install apps that are not published from an official app store) [59]. To adapt the escrow to the Android ecosystem, a centralized, trustworthy app distributor, such as Google Play or a dedicated privacy-focused app store, could mandate the integration of the escrow as a strict prerequisite for app distribution and review apps to ensure all data access goes through the escrow.

4.2 Escrow Design and Enforcement

We discuss two design choices for the escrow: a library-based integration and an OS-level system service. Because modifying proprietary systems by Apple is infeasible for third-party researchers, our escrow prototype implements the library-based design. However, we detail both designs to demonstrate how Apple enforces the escrow as a non-bypassable bottleneck for data access and how the escrow enforces control.

Library-Based Design. In this design, the escrow is a compiled library (e.g., an iOS framework) that developers link into their apps. Enforcement relies on Apple's App Review process, in which apps are reviewed and scanned for unauthorized API usage [13]; enforcing the escrow simply extends this process. Apps that attempt to call native data access SDKs directly, rather than declaratively describing data to access through the ACCESS() function in RUN(), are flagged and rejected during submission.

OS-Level Service Design. An alternative design, which Apple could adopt, integrates the escrow directly into the operating system as a privileged service. In the Apple ecosystem, access to sensitive data is governed by the Transparency, Consent, and Control (TCC) mechanism [19]. In this design, the OS uses its existing (TCC) mechanism to deprecate direct sensitive data entitlements [15] that apps require to access data. Only the privileged escrow service retains the entitlements to read sensitive data. If an app attempts to bypass the escrow and read protected resources directly from the filesystem or memory without the required entitlements, the OS kernel intercepts and terminates the process.

Enforcing Control. Once the escrow becomes the bottleneck for data access, it must enforce control in two areas: (1) controlling data access so an app cannot gain access to data it did not explicitly request via the ACCESS() function in RUN(), and (2) ensuring the COMPUTE() function does not contain malicious logic that extracts or transfers data indirectly, which would jeopardize the escrow's transparency. Apple's existing enforcement mechanisms naturally address both concerns. First, to control data access, developers must explicitly declare their required capabilities to access any protected resource [12], similar to how they do today. Depending on the integration level, this relies on standard entitlements (library-based

design) or escrow-specific capabilities (OS-level design). If an app declares a capability (e.g., accessing the Photos library) but lacks the corresponding ACCESS() function (e.g., querying the escrow's `Photos` table), the app is flagged during App Review. Conversely, attempting to access data without the required capability results in runtime denial. Second, to prevent malicious data extraction and transfer, developers must still provide a Privacy Manifest [16] that declares the data collected, its purpose, and any network domains used for data transfer. During Apple's mandatory App Review process [13], reviewers evaluate the app's RUN() implementation against these declarations. If the computation contains malicious logic to indirectly extract data or communicates with undeclared domains, the app is rejected prior to publication, ensuring the escrow's transparency remains uncompromised.

4.3 Relational Interface Design

The core component of the escrow is the data virtualization layer that provides the relational interface for the ACCESS() function. We detail the relational interface design of our escrow prototype to demonstrate a practical path toward deployment.

4.3.1 Designing the Relational Schema. The escrow exposes a public relational schema that developers use to formulate their data access queries. The design of this schema is guided by the principle of minimizing friction for developers who are already familiar with the native SDKs. Rather than creating an entirely abstract schema, our tables and columns map closely to the objects and properties provided by Apple's SDKs [2]. For example, Apple's `PhotoKit` [11] framework represents a photo or video asset as a `PHAsset` [10] object. Our corresponding `Photos` table in the relational schema includes columns for standard attributes such as `id` (`String`), `mediaType` (`Integer`), and `creationDate` (`Date`). Crucially, it also includes a `phasset` column whose type is the native `PHAsset` object itself. This allows a developer to write a SQL query to filter and select assets declaratively, and then receive a sequence of native `PHAsset` objects in the COMPUTE() function, on which they can perform operations using the standard `PhotoKit` APIs they already know. This design choice makes the transition to the escrow model more intuitive and less disruptive for developers.

4.3.2 Designing the Relational Engine. The relational interface provides a logical view of the data to developers, and they no longer need to implement the concrete mechanisms to retrieve the data—this responsibility falls onto the escrow. We design, implement, and evaluate three approaches to implement the escrow's relational engine, each with different performance trade-offs.

Materialized Tables. In this baseline approach, the escrow upon initialization fetches all relevant data declared by the app's ACCESS() functions using the native Apple SDKs, and materializes it into tables within an on-device SQLite database. The ACCESS() function is then executed as a standard SQL query against this database. While this approach benefits from the performance of a native database engine, it suffers from two significant drawbacks. First, it creates a data freshness problem, as the materialized data can become stale if the underlying source changes. Second, it introduces significant overhead for complex data objects like `PHAsset`, which cannot be stored directly in SQLite. Instead, we must store a reference ID, and the escrow must perform a costly post-query lookup to map these IDs back to their corresponding native objects.

Virtual Tables. To address the issues of data freshness and object handling, our second approach leverages SQLite's virtual table mechanism [82]. A virtual table is a schema object that looks like a normal table but does not store any data on disk. Instead, when the database engine needs to scan a virtual table, it makes callbacks to custom C functions provided by our implementation. We implement these callbacks to make live API calls to the relevant Apple SDKs (e.g., `PhotoKit`, `Contacts`) to fetch all relevant data needed to populate the table. This data is then passed back to the SQLite engine, which performs any necessary filtering, sorting, or aggregation in memory. This approach ensures that query results are always up-to-date and avoids data duplication. Furthermore, because the data is only held in memory during query execution, we can pass pointers to complex native Swift objects (like `PHAsset`) through the C-to-Swift bridge, allowing the COMPUTE() function to receive them in their native format without any serialization overhead. However, this approach can be inefficient for selective queries, as it requires fetching all data from an API before filtering.

Virtual Tables with Pushdown. Our third approach enhances the virtual table implementation by applying classic database query optimization principles. The goal of predicate pushdown [48] is to filter data as early as possible to minimize data processing. We implement logic to push down query operations such as projections, predicates, sorting, and limits, from the SQLite query engine into the underlying native SDK API calls, since many of Apple's SDKs provide APIs that allow for more efficient, constrained data fetching. For example, consider the query `SELECT phoneNumber FROM Contacts WHERE givenName == 'Alice'`. A naive virtual table implementation would fetch *all* contact records from the device and let SQLite perform both the filtering on `givenName` and projection on `phoneNumber`. Our pushdown implementation, however, will translate this query into an efficient `CNContactFetchRequest` [8] with a predicate for the name 'Alice' and `keysToFetch` set to only the `phoneNumber`. This pushes down the filtering and projection operations from the SQLite engine into the native API, reducing the amount of data that needs to be fetched from the underlying data source and processed by the SQLite engine, leading to significant performance gains in many cases, as shown in our evaluation.

4.4 Bootstrapping Dataflow Preferences

While the primary contribution of our work is the technical mechanism required to enforce dataflow preferences (the escrow), an effective system must also practically elicit these preferences from individuals. We propose a concrete design that piggybacks on Apple's existing permission infrastructure to demonstrate a technically viable mechanism for purpose-bound control.

Currently, Apple elicits individuals' privacy preferences through its TCC subsystem. When an app requests sensitive data, TCC triggers a system-rendered pop-up (e.g., "App X would like to access your photos") alongside a developer-provided purpose string explaining why the app needs access [17]; developers are required to provide useful purpose strings so individuals using the app can

[2] Nevertheless, the *object–relational impedance mismatch* [53] always exists. Our goal is not to solve this mismatch, but to design the schema in a developer-friendly manner.

decide whether to allow or deny the app's access to their data, and App Review will reject apps that access protected resource without providing a purpose string [13]. Our design builds directly upon this existing practice. In the escrow model, the mandatory purpose string is elevated: instead of providing arbitrary explanations of why the app needs access to certain data, developers are required to provide a specific purpose string that directly maps to the logic of the COMPUTE() function for every RUN() invocation. During the App Review process, the reviewers verify that this declared purpose string accurately reflects the COMPUTE() logic; this is an extension of their existing review protocol. At runtime, the OS intercepts the RUN() call and generates an extended TCC prompt. Instead of a generic data request, this prompt displays the requested source data (D_s), the target agent (D_t) if the data leaves the device, and the vetted purpose string explaining the purpose ($f()$). This integration bridges the escrow architecture with Apple's existing UI, providing individuals with transparency and control over their dataflows without revamping the App Review process.

However, we explicitly note the limitations of this design. Prompting an individual for every distinct RUN() invocation, especially in complex apps with numerous dataflows, would inevitably induce permission fatigue as it already does today [24, 29, 65]. While a more comprehensive mechanism is needed to elicit dataflow preferences without overwhelming individuals, designing and rigorously evaluating such an interface is fundamentally a human-computer interaction (HCI) challenge. Consequently, solving this usability problem remains an important area for future work, one that relies on the technical foundation established by our escrow architecture.

5 Escrow Implementation

This section details the technical implementation of the escrow's two core subcomponents: the relational engine that powers the on-device data virtualization layer, and the secure offloading pipeline that enables delegated computation on escrow-controlled servers. We describe the detailed techniques we employ to implement the escrow's relational engine using Virtual Tables with Pushdown (Section 5.1) and its computation offloading pipeline (Section 5.2).

5.1 Relational Engine Internals

The core of our prototype is a relational engine that bridges the declarative world of SQL with the imperative, object-oriented APIs of Apple's native SDKs. We provide our implementation details of the Virtual Tables with Pushdown approach, as described in Section 4.3.2. The virtual table implementation for each virtualized data source follows a consistent pattern: a thin C layer interfaces directly with SQLite, while a Swift backend contains the primary logic for data fetching and pushdown optimization.

5.1.1 The C-to-Swift Virtual Table Bridge. We leverage SQLite's virtual table module [82] to build the relational engine. For each data source (e.g., Contacts, Photos, Location), we implement a standard `sqlite3_module` [83], a C structure that registers a set of callbacks with the SQLite engine. These C callbacks act as a lightweight bridge, delegating the main work to Swift functions exposed to the C runtime via the `@_cdecl` attribute. This hybrid architecture allows us to combine the low-level control of SQLite's C API with the high-level power of Swift for interacting with modern Apple SDKs. The query lifecycle is managed by these key C callbacks:

- `xConnect`: Invoked by SQLite when creating the virtual table. We use them to declare the virtual table's schema (e.g., the columns `givenName`, `familyName`, `phoneNumber` for the `Contacts` table).
- `xBestIndex`: This callback is the heart of our optimization logic. It is called by the SQLite query planner, which passes a representation of the query's constraints (i.e., the `sqlite3_index_info` structure [81]), including information such as WHERE and LIKE clause predicates, ORDER BY terms, and LIMIT counts. Our `xBestIndex` implementation parses these constraints to determine which operations can be pushed down to native Apple SDKs, and encodes the optimized indexing strategy into an integer bitmask (for predicates) and a formatted string (for projection mask and other parameters) by filling in the `idxNum` and `idxStr` fields of the `sqlite3_index_info` structure, which forms the reply to the SQLite query planner.
- `xFilter`: This function executes the data fetch according to the indexing strategy selected in `xBestIndex`. Its primary role is to bridge into our Swift implementation to perform the data fetch, passing the predicate values and other details of the indexing strategy; the Swift logic is responsible for translating the encoded indexing strategy into an efficient data fetch request using native Apple SDKs. `xFilter` receives back an opaque pointer to a Swift object that contains an in-memory snapshot of the query results.
- `xNext`/`xColumn`: After `xFilter` fetches the result snapshot, SQLite uses this pair of functions to iterate through the result set. `xNext` simply increments a row counter, while `xColumn` calls back into our Swift implementation to retrieve the data for a specific row and column from the snapshot.

5.1.2 Zero-Copy Native Object Handling. A key challenge is handling complex native Swift objects (e.g., `PHAsset` or `CLLocation`). A naive approach, such as the one used in our Materialized Tables baseline, would require storing object identifiers in the database and performing costly lookups to reconstruct the native objects post-query. Our virtual table implementation avoids this overhead. When the `xColumn` callback iterates through query results, it obtains an opaque pointer to the Swift object in memory (via `Unmanaged.passRetained(object).toOpaque()`). This pointer is passed across the C bridge and what SQLite stores for the object-typed column. Later, when the escrow processes the query results from SQLite, this pointer is passed back from the C runtime to Swift and safely converted back into a fully-typed Swift object (via `Unmanaged.fromOpaque(pointer).takeRetainedValue()`). This zero-copy mechanism allows passing native objects through the relational engine without any serialization or expensive re-fetching, which is critical for the performance gains shown in our evaluation.

5.2 Computation Offloading Pipeline

The escrow architecture supports offloading computation to escrow-controlled servers. This is enabled by an optional SERVER_ID parameter in the RUN function call.

5.2.1 Architecture and Security Model. The offloading pipeline runs on a client-server architecture where the on-device escrow acts

as the client and the server is designed to run in a Trusted Execution Environment (TEE) [74], analogous to systems like Apple's Private Cloud Compute [6]. Communication between client and server is encrypted using mutual TLS (mTLS), which authenticates both the device and server to protect data in transit. Furthermore, code integrity is enforced by requiring developers to pre-register and cryptographically sign any COMPUTE() function intended for offloading; the server verifies this signature against its registry before execution to prevent unauthorized code from running. These measures provide the technical foundation for securely extending the individual's trust zone to include remote infrastructure.

5.2.2 End-to-End Offloading Pipeline. When an app calls RUN(ACCESS(), COMPUTE(), SERVER_ID), it executes the following steps:

1. Local Data Access. The ACCESS() always executes locally on the device via the relational engine to obtain the source data, D_s. This ensures raw data access remains under the local escrow's control.

2. Serialization. The resulting Swift objects are serialized into a portable binary format (e.g., Protocol Buffers [45]).

3. Secure Invocation. The on-device escrow establishes an mTLS connection to the server specified by SERVER_ID. It then transmits a request payload containing the serialized D_s and a unique identifier for the pre-registered COMPUTE() function.

4. Remote Execution. The server authenticates the device's TLS certificate and verifies the integrity of the requested function. Upon success, it deserializes D_s and invokes the corresponding COMPUTE() function within a sandboxed process, passing the data as input.

5. Result Return. The final result, D_t, is serialized by the server and sent back to the device over the secure mTLS channel. The on-device escrow deserializes the result and returns it to the app.

This pipeline provides the technical foundation for enforcing resource-aware dataflow preferences, and enables the escrow to act as a scheduler, dynamically placing computation based on individuals' preferences, device state (e.g., battery, network), and the computational cost of the dataflow.

6 Evaluation

In this section, we answer three research questions:

- **RQ1**: Can real-world applications be ported to the escrow? (i.e., is the proposed escrow solution feasible?) We collect 10 representative apps and study their implementation on the escrow-based programming interface (Section 6.1).
- **RQ2:** Are escrow-based applications efficient? We measure the overhead the escrow introduces compared to applications written without using the escrow (Section 6.2).
- **RQ3:** What are the performance trade-offs of different implementations of the escrow's relational engine? We evaluate the performance of three approaches: Materialized Tables, Virtual Tables, and Virtual Tables with Pushdown (Section 6.3).

6.1 RQ1: Dataflows in Real-World Apps

We design a rubric to collect a representative sample of iOS apps containing various dataflows involving sensitive personal data, and we describe how those dataflows can be reimplemented using the escrow's programming interface.

Rubrics for selecting apps. We selected 10 representative apps based on the following rubric [3]:

- Each app needs to be open-sourced in GitHub so we can find dataflows in its codebase, and it should be currently maintained (i.e., not archived and has commits in the recent month)
- Each app needs to be relatively popular for it to be representative, thus we only select apps with over 1000 Github stars.
- Apps need to contain dataflows covering different types of data.
- Apps need to be diverse, spanning multiple categories.

Analysis. We analyze each app's source code, identify dataflows involving sensitive personal data, and summarize our findings in Table 2. We include one row describing the dataflow involving contacts, location, and image/video respectively, since most apps contain dataflows that involve at least one of the three data types. We also include one row describing other dataflows.

Location dataflow. Six apps use individuals' current location for various purposes. Five of them (DuckDuckGo [35], Signal [78], Wikipedia [89], Home Assistant [22], WordPress [90]) share the location outside the individuals' devices with servers. For instance, Wikipedia finds and recommends articles that individuals might be interested in based on their surrounding region. The app makes a search request to Wikimedia (the parent company of Wikipedia) server with the individual's surrounding region as input, and the server returns relevant articles. We show a simplified version of how the dataflow can be implemented using the escrow based on Wikipedia's current implementation:

```
Escrow.run("SELECT longitude, latitude FROM Location
ORDER BY timestamp LIMIT 1") {
coordinates -> SearchResults in
    let region = getSurroundingRegion(
        location: coordinates, radius: 1000)
    // Create search request using Wikimedia's API
    let url = wikiMediaAPI(region: region)
    let results = fetchArticles(url: url, limit: 50)
    return results }
```

Although the exact details of how Wikimedia finds articles based on a given location are unknown, implementing the dataflow using the escrow makes it transparent exactly *what* data is leaving the individual's device and its purpose through COMPUTE(). In the case of Wikipedia, the search request only needs the approximate region that centers around the individual's current location by a specified radius, and the precise location coordinates or other identifying information should not leave the device. The app only has access to the fetched articles as the output of RUN and nothing else.

Contact dataflow. Two apps, Signal [78] and Simplenote [79], contain dataflows involving contacts. Signal performs "contact discovery" by fetching the individual's contacts (specifically phone numbers) and finding which contacts are Signal users. Notably, the computation takes place on Signal servers, which run in a secure hardware enclave with oblivious RAM to ensure Signal or anyone else does not gain access to individuals' contact data [77]; essentially, the computation is equivalent to a private set intersection

[3] We include Covid Alert (Canada's COVID contact tracing app) [76], which does not fit all rubrics but has interesting dataflows that are worth discussing in the paper.

	Bluesky[23]	**Covid Alert**[76]	**DuckDuckGo**[35]	**Nextcloud**[60]	**Signal**[78]	**Simplenote**[79]	**VLC**[87]	**Wikipedia**[89]	**WordPress**[90]	**Home Assistant**[22]
Category	Social media	Contact tracing	Browser	File Management	Messenger	Notes	Media player	Wikipedia	Blogs	Home automation
Contact dataflow	N/A	N/A	N/A	N/A	Find other users from contacts	Share notes with other contacts	N/A	N/A	N/A	N/A
Location dataflow	N/A	N/A	Share location with websites to obtain location based results	Show location on a map	Reverse geocode, search and share location	N/A	N/A	Search and recommend articles based on location	Add current location to posts	Send location to local Home Assistant server
Image/ video dataflow	Set profile pictures, create posts	N/A	Perform voice search	Upload image/ video to user's cloud	Send image/ video	N/A	Control Apple system event to resume/pause media	N/A	Add photo/media to posts	Perform voice assist
Other dataflow	N/A	Record bluetooth keys and match keys with positive Covid diagnosis	N/A	N/A	Transfer app data to another device via LAN	N/A	Access Desktop/ Documents/Downloads folder, and network/ removable volume	N/A	N/A	Sends motion data and focus status to Home Assistant

Table 2: Dataflows of representative apps. N/A means no dataflow exists in the app

between individuals' contacts and Signal's users [67]. A simplified implementation of this dataflow using the escrow may look like:

```
Escrow.run("SELECT phoneNumber FROM Contacts") {
phoneNumbers -> SignalAccounts in
    let url = SignalEnclaveURL()
    let matched = contactDiscovery(phoneNumbers, url)
    return matched }
```

While technically the mechanisms Signal implemented for contact discovery would already ensure individuals' contact data remain private and not accessible to anyone else, implementing this dataflow via the escrow makes it transparent that this computation *only* takes place in Signal's enclave and the data is not used for other purposes. In addition, it is conceivable that the contact discovery can take place in an escrow-controlled server with similar enclave technologies, so individuals gain more control over the specific usage of their data. Lastly, it is worth noting that such emphasis on privacy-preserving computation, like what Signal has implemented, is *not* the current industry norm [101].

On the other hand, Simplenote lets individuals pick which contact they would like to share their notes with, then sends a request to the contact (e.g., via email). This dataflow can also be implemented using the escrow; for instance, ACCESS() fetches necessary contact records (e.g., name and email), and COMPUTE() implements the interface that allows the individual to pick which contact to share with, then sends the request to the contact. The escrow makes it transparent what data Simplenote has access to (in this case it would only have access to the specific contact the individual picked) and how the data is used.

Image/video dataflow. Seven apps have read access to the device's Photos Library. Most apps access the Photo Library for functionalities such as setting avatars (Bluesky [23]), creating posts (Bluesky, WordPress [90]), sending / uploading messages / files (Nextcloud [60], Signal), or playing videos (VLC [87]). We concentrate on one unique use case from two apps (DuckDuckGo, Home Assistant), which is speech recognition. Generally, there are two ways to perform speech recognition: one is to transcribe a recording that's already stored on-device, and the other is to transcribe live speech. Implementing the former task using the escrow is straightforward; for example, to transcribe the most recently stored audio recording, developers can write:

```
Escrow.run("SELECT asset FROM Photos WHERE mediaType
=='audio' ORDER BY creationDate LIMIT 1") {
recording -> String in
    let recognizer = SpeechRecognizer()
    let text = recognizer.transcribe(recording)
    return text }
```

The SQL query fetches the most recently created asset stored in the Photos library that has an audio media type. The query returns a native `PHAsset` object that can be directly consumed by the `SpeechRecognizer`. On the other hand, transcribing a live speech is generally achieved by storing the individual's live speech in a buffer in memory. When the buffer is full (or after a small time interval), the speech recognizer will transcribe the new buffer (along with the previously stored buffers), thus achieving the effect of transcribing speech on the fly. Since no speech data is physically stored on-device, the escrow cannot access the data, thus we consider implementing this dataflow to be outside the scope of this paper. However, it is possible that by integrating with Apple's infrastructure the escrow will be able to access structured data that is stored in memory and enable dataflows such as live speech transcription.

Other dataflows. Four apps contain dataflows that do not involve contact, location, or image/video data, each with its distinct purpose. All dataflows from Covid Alert, VLC, and Home Assistant can be implemented using the escrow except Signal, since Signal's dataflow involves app-specific data that is not accessible using Apple SDKs. We explain how to implement the dataflow from Covid Alert, which we consider to be the most interesting one among the four apps.

Covid Alert is Canada's official Covid contact tracing app. The app uses Apple's Exposure Notification framework [20]. It works by recording each individual's on-device Bluetooth keys when their device exchanges Bluetooth beacons with other devices. When an individual receives a positive COVID diagnosis, they can voluntarily share their keys with the government-controlled remote key server, which stores all Bluetooth keys from individuals who have recently tested positive for COVID. Periodically, the app downloads keys from the key server and tries to find whether the individual's keys from the last 14 days match any of the downloaded keys (from those who tested positive). If there is a match, the app notifies the individual. The two important dataflows—sharing keys with the key server when an individual tests positive, and matching the individual's keys with keys downloaded from the key server, can both be implemented using the escrow. For example, an approximate implementation of the former dataflow may look like:

```
Escrow.run("SELECT * FROM KeyTable ORDER BY data LIMIT
14") { keys in let url = keyServerURL()
    shareKeysWithServer(url) }
```

	Data Access	Computation
Contacts	Get the phone number of the contact whose given name is 'uniqueName' `SELECT phoneNumber FROM Contacts WHERE givenName = 'uniqueName'`	Check if the result is a valid US phone number
Photos	Get the most recent 100 images (as `PHAssets`) from the Photos library `SELECT phasset FROM Photos WHERE mediaType=='image' ORDER BY creationDate DESC LIMIT 100`	Transform the result to a list of `NSImage`
Location	Get the most recent location (as `CLLocation`) `SELECT cllocation from Location ORDER BY timestamp DESC LIMIT 1`	Get weather using OpenWeatherMap [63] API

Table 3: Dataflows run by the escrow-based and baseline app, which uses SQL query and Apple SDKs to access data respectively

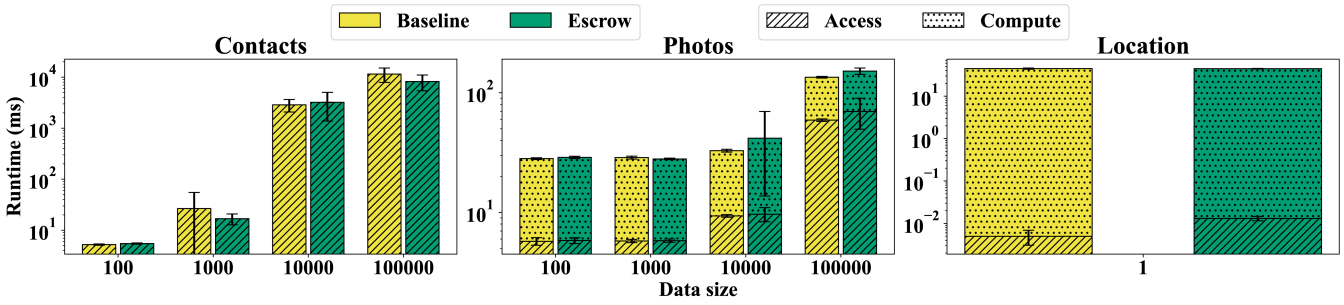

Figure 3: Average access and compute runtime (with std) by escrow-based app vs baseline app over 10 runs

Using the escrow provides transparency of the dataflow and ensures that the individual's private health information is not exposed to anyone else other than the government-controlled key server.

In summary, we show that it is feasible to re-implement many dataflows in real-world apps via the escrow's programming interface. We note that since the apps we collected are all open-source, they are a "biased" sample in the sense that the apps are already designed with transparency as a priority. Many closed-source apps have undisclosed dataflows for purposes not covered in our analysis (e.g., profiling, resale). Enforcing those dataflows to be implemented using the escrow would enable individuals to gain transparency and control over these dataflows.

6.2 RQ2: Runtime Efficiency of the Escrow

We measure the overhead introduced by the escrow by comparing the time to access data and run computation on the data using the escrow's RUN function versus without the escrow (i.e., the baseline) for different data sizes. Table 3 shows the dataflows used in our evaluation, which involve three data types that are representative in many of today's apps, as we have shown in Section 6.1.

Implementation. We implement an escrow-based app, which uses the escrow's RUN function to access data and run computation, and the baseline app, which uses Apple SDKs to access data. The computation functions in both apps are implemented in a similar way and the output of the computation are exactly the same.

The escrow-based app uses the escrow as a static singleton instance (note that the escrow prototype is deployed as a library). We implement its relational engine using Virtual Tables with Pushdown. The schemas of the virtual tables are:

- `Contacts`: `id TEXT, givenName TEXT, familyName TEXT, phoneNumber TEXT`
- `Photos`: `id TEXT, phasset PHAsset, mediaType INT, creationDate DATE, collectionID TEXT, collectionName TEXT`
- `Location`: `cllocation CLLocation, longitude REAL, latitude REAL, horizontalAccuracy REAL, timestamp TIMESTAMP`

We implement projection pushdowns for all three virtual tables. In addition, for Contacts table, we implement predicate pushdown for all four columns, For Photos table, we implement predicate pushdown for `id`, `mediaType`, `collectionID` and `collectionName`, and order by / limit pushdown for `creationDate`. For Location table, we implement order by / limit pushdown for `timestamp`.

Data preparation. We prepare and run the experiments for different data sizes. For contacts, we add 100, 1k, 10k, and 100k contacts records to the Contacts app; each record has a unique given name, family name, and phone number. For photos, we add 100, 1k, 10k, and 100k photos to the Photos library; each photo is 16KB. Location data is updated in real time.

Configurations. The apps are implemented as multi-platform (iOS/macOS) apps in Swift 6.1. We deploy the app as a macOS app and run the experiments using a 2021 MacBook Pro with Apple M1 Pro chip, 16 GB memory, and macOS 15.6.1, since iPhones cannot handle the large data volumes we will insert.

Result. We run each app 10 times for each dataflow and data size, and we show the average access and computation time separately (with standard deviation) in Figure 3. In short, the escrow introduces very little overhead. For contacts dataflows, the runtime is dominated by the data access time, since checking whether a given number is a valid US phone number only involves regex matching and is very fast. Because the escrow executes the SQL query via the Contacts virtual table, which implements pushdown for projections and predicates, the database does not perform additional data filtering, hence the overhead is small compared to fetching data directly using Apple's Contacts SDK. For photos dataflow, as data size increases, the majority of runtime shifts from computation to data access, since the number of photos to process stays the same

	Full	Projection	Predicate	Order By / Limit
Contacts	`SELECT * FROM Contacts`	`SELECT familyName, givenName FROM Contacts`	`SELECT * FROM Contacts WHERE givenName = 'uniqueName'`	\
Photos	`SELECT * FROM Photos`	`SELECT phasset FROM Photos`	`SELECT * FROM Photos WHERE collectionName = 'uniqueAlbum'`	`SELECT * FROM Photos ORDER BY creationDate DESC LIMIT 1`
Location	\	\	\	`SELECT * FROM Location ORDER BY timestamp DESC LIMIT 1`

Table 4: Queries run by the escrow's relational interface

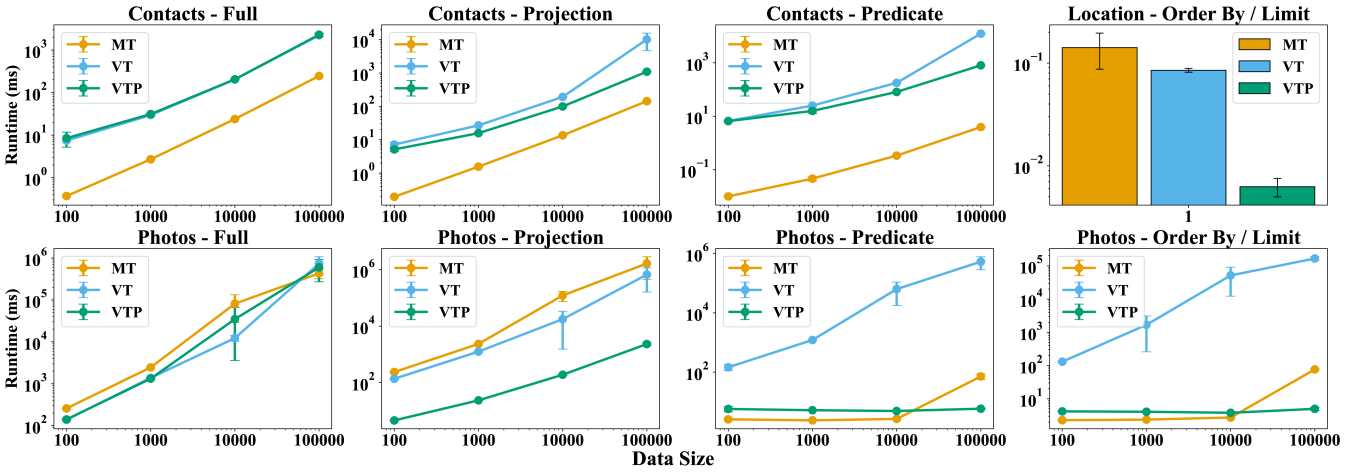

Figure 4: Average runtime (with std) to execute queries over 10 runs using Materialized Tables (MT), Virtual Tables (VT), and Virtual Tables with Pushdown (VTP)

in the computation function. Similarly to Contacts, the Photos virtual table also implements pushdown so there is no additional data filtering on the database side. For Location dataflows, the runtime is dominated by the computation time, since data is transferred to and from the OpenWeatherMap server.

In summary, the escrow's overhead comes from the relational engine and any post-processing the escrow needs to perform to transform the query result to a unified format so it can be processed by the subsequent computation function. Implementing pushdown in virtual tables reduces the amount of data filtering the database needs to do, and implementing the zero-copy mechanism for passing native Swift objects through the relational engine ensures there is no serialization overhead, thus reducing the overall overhead.

6.3 RQ3: Relational Engine Comparison

We compare the runtime performance of three implementations of the escrow's relational engine (Section 4.3): Materialized Tables, Virtual Tables, and Virtual Tables with Pushdown. Table 4 shows the queries used in the evaluation. We categorize the queries into full, projection, predicate and order by / limit queries in order to evaluate how each implementation performs for each category.

Setup. All three implementations are exposed to the same schema as described in the previous experiment (Section 6.2). We also use the same data preparation and experiment configurations as in the previous experiment. We create a unique album with one photo inside, in addition to the photos we have already added. We only measure the performance of running the *data access* function and there is no computation function involved. All three implementations produce query result in the same format.

	size=100	size=1000	size=10000	size=100000
Contacts	10.85	32.68	279.28	3402.41
Photos	290.72	1714.91	19150.86	130038.08

Table 5: Time (ms) to materialize Contacts and Photos table

Result. We run each query 10 times, and we show the average runtime with standard deviation in Figure 4. In addition, for Materialized Tables, we report the time to materialize the Contacts and Photos table for each data size in Table 5. The time grows linearly with the data size. Materializing the Photos table takes considerably more time since it needs to enumerate each photo asset and fetch its collection ID and name. The runtime performance of the three implementations varies for different categories of queries. As expected, Virtual Tables with Pushdown perform better than Virtual Tables for all except full (SELECT *) queries, in which case the two implementations have roughly the same runtime since no pushdown can be applied. Notably, Virtual Tables with Pushdown performs magnitudes faster for highly selective queries (Photos queries with predicate and order / limit), because it asks the `PhotoKit` framework for a single album with only one photo inside or the single most recent photo, avoiding the materialization of unnecessary data, whereas Virtual Tables implementation must filter or sort all photos. It is also the fastest for Photos queries

with projection since the other two implementations both need to fetch collection information for each `phasset`, which is costly. Materialized Tables implementation is about an order of magnitude faster than the other two for Contacts queries but not for Photos queries, because for Contacts queries the data is already in the database and SQLite can employ any internal optimization it needs to execute the query efficiently, but for Photos queries the escrow needs to map the result `phasset` IDs to `phasset` objects when the query finalizes, which introduces overhead. Both Virtual Tables implementations use zero-copy mechanism to pass `phasset` objects through the database thus do not incur serialization overhead.

In summary, while Materialized Tables offers speed for full scans, it suffers from high initialization costs and data staleness. By applying classic query optimization principles, Virtual Tables with Pushdown provides the data freshness of on-the-fly data virtualization with performance that is competitive with, and in some cases superior to, querying materialized data.

7 Related Work

Personal Data Sovereignty. The concept of personal data sovereignty has emerged as a critical paradigm for rebalancing control in the digital economy [37, 49, 56, 66]. Our work's central goal of *Personal Dataflow Sovereignty* is a direct, technical instantiation of this principle, providing a concrete mechanism for individuals to exercise control over how their data is used. While legal frameworks like the GDPR [38] provide a foundation for personal data sovereignty, their practical implementation often falls short [73], highlighting the need for a technical foundation like the one we propose. The data management community has a rich history of engaging with the core technical problems under personal data sovereignty, from *Hippocratic Databases* [4] that articulates a vision for database systems designed with privacy as a core principle, to seminal work on purpose-based access control databases [26], to more recently a data escrow system [91] that enables delegated computation.

Data Governance Frameworks. The concept of rebalancing power in the data economy has given rise to various data governance frameworks [1]. These include Data Unions, Data Trusts, and Data Cooperatives, which advocate for collective organizations that would manage data and negotiate on behalf of individuals [33, 51, 70]. A related economic perspective reframes individual-generated data as a form of digital labor, arguing that individuals should be compensated for their contributions to the data economy, a concept known as Data as Labor [21, 68]. While these frameworks provide compelling visions for a more equitable data ecosystem, they often lack a concrete technical foundation for their implementation. Our data escrow architecture can be viewed as a critical piece of enabling the technical infrastructure that serves as the trustworthy intermediary required for such frameworks to operate effectively.

Privacy design and enforcement mechanism. A large body of research focuses on usable privacy and policy design in apps. For instance, many works focus on privacy designs to guide individuals toward informed privacy choices [3, 41, 46, 75, 96–98]. Recent research has also explored the use of Large Language Models (LLMs) to facilitate privacy comprehension [36, 40]. However, a critical gap exists: usable privacy solutions frequently lack the underlying technical feasibility to be deployed effectively in practice [25, 34, 43, 52, 64]. Our escrow provides the technical *mechanism* required to enforce individuals' dataflow preferences, thereby motivating and providing the necessary foundation for future *policy* design on eliciting those preferences. In addition, existing industry practices offer alternative approaches to data minimization, such as data pickers that allow granting apps access to only the subset of data individuals have picked (e.g., Apple's Photo picker [18]). However, they fundamentally constrain the types of computations a developer can implement. The escrow model differs by allowing developers to specify arbitrary computation over data.

Relational model over heterogeneous data sources. The challenge of providing a unified query interface over multiple heterogeneous data sources is a classic problem in the database community [47]. Data virtualization techniques are pioneered by early data integration systems like Garlic [28], where source-specific wrappers translate queries against a global schema into the native languages or API calls of the underlying sources. This principle is now widely adopted in production systems, for instance, through PostgreSQL's Foreign Data Wrapper (FDW) [69]. The escrow's relational interface applies this same principle of data virtualization, creating a unified schema over disparate, programmatic APIs to access data. Our approach is also analogous to how TinyDB [58] overlays a relational model over data collected from sensor networks.

Trusted Execution and Computation Offloading. A core capability of the escrow is offloading computation while keeping data within the individual's trust zone, necessitating mechanisms for secure execution. Therefore, we contextualize our architecture alongside prior systems that secure data and computation against untrusted environments. In cloud settings, systems like Sieve [88] and Iron [42] leverage functional encryption and cryptographic access controls, while Slalom [86] and Telekine [50] use hardware enclaves to secure remote workloads. On mobile devices, frameworks such as TinMan [92], ShadowNet [84], and SeCloak [57] rely on hardware isolation to shield peripheral data and execution from the host OS. While the escrow architecture provides the orchestration layer, which dictates whether a dataflow is permitted based on its purpose, these prior systems provide the execution layer, which dictates how it is securely computed. When the escrow executes a COMPUTE() function on local hardware or offloads it to an escrow-controlled server, it could leverage these technologies to ensure the execution remains trustworthy.

8 Conclusion

In this paper, we contribute a solution for a critical problem in the modern data economy: the loss of personal data sovereignty. We propose a data escrow architecture that leverages delegated computation. We design and implement a minimally invasive programming interface and an efficient data virtualization layer to make this model practical and developer-friendly. Our concrete implementation in the Apple ecosystem demonstrates a feasible path to deployment. Finally, our comprehensive evaluation shows both qualitatively and quantitatively that our escrow solution is expressive enough to implement dataflows in a wide range of real-world applications, and that it is efficient.

Acknowledgments

We thank the anonymous reviewers for their detailed and constructive feedback, and the Data Ecology Research Initiative of the Data Science Institute for their support.

References

[1] Rene Abraham, Johannes Schneider, and Jan Vom Brocke. 2019. Data governance: A conceptual framework, structured review, and research agenda. *International journal of information management* 49 (2019), 424–438.
[2] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. 2018. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)* 51, 4 (2018), 1–35.
[3] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, et al. 2017. Nudges for privacy and security: Understanding and assisting users' choices online. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–41.
[4] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 143–154.
[5] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. 2015. Your location has been shared 5,398 times! A field study on mobile app privacy nudging. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 787–796.
[6] Apple. 2024. *Private Cloud Compute: A new frontier for AI privacy in the cloud.* https://security.apple.com/blog/private-cloud-compute/
[7] Apple. 2025. *CLLocation.* https://developer.apple.com/documentation/corelocation/cllocation
[8] Apple. 2025. *CNContactFetchRequest.* https://developer.apple.com/documentation/contacts/cncontactfetchrequest
[9] Apple. 2025. *Contacts Framework.* https://developer.apple.com/documentation/contacts
[10] Apple. 2025. *PHAsset.* https://developer.apple.com/documentation/photokit/phasset
[11] Apple. 2025. *PhotoKit.* https://developer.apple.com/documentation/photokit
[12] Apple. 2026. *Adding capabilities to your app.* https://developer.apple.com/documentation/xcode/adding-capabilities-to-your-app
[13] Apple. 2026. *App Review Guidelines.* https://developer.apple.com/app-store/review/guidelines/
[14] Apple. 2026. *Core Location Framework.* https://developer.apple.com/documentation/corelocation
[15] Apple. 2026. *Entitlements.* https://developer.apple.com/documentation/bundleresources/entitlements
[16] Apple. 2026. *Privacy manifest files.* https://developer.apple.com/documentation/bundleresources/privacy-manifest-files
[17] Apple. 2026. *Requesting access to protected resources.* https://developer.apple.com/documentation/uikit/requesting-access-to-protected-resources
[18] Apple. 2026. *Selecting Photos and Videos in iOS.* https://developer.apple.com/documentation/photokit/selecting-photos-and-videos-in-ios
[19] Apple. 2026. *Transparency, Consent, and Control (TCC) Target Flag.* https://security.apple.com/bounty/target-flags/#tcc
[20] Apple and Google. 2025. *Privacy-Preserving Contact Tracing.* https://covid19.apple.com/contacttracing
[21] Imanol Arrieta-Ibarra, Leonard Goff, Diego Jiménez-Hernández, Jaron Lanier, and E Glen Weyl. 2018. Should we treat data as labor? Moving beyond "free". In *aea Papers and Proceedings*, Vol. 108. American Economic Association 2014 Broadway, Suite 305, Nashville, TN 37203, 38–42.
[22] Home Assistant. 2025. *Home Assistant for Apple Platforms.* https://github.com/home-assistant/iOS
[23] Bluesky. 2025. *Bluesky Social App.* https://github.com/bluesky-social/social-app
[24] Cristian Bravo-Lillo, Lorrie Cranor, Saranga Komanduri, Stuart Schechter, and Manya Sleeper. 2014. Harder to ignore? revisiting {Pop-Up} fatigue and approaches to prevent it. In *10th Symposium On Usable Privacy and Security (SOUPS 2014)*. 105–111.
[25] Carolyn Brodie, Clare-Marie Karat, John Karat, and Jinjuan Feng. 2005. Usable security and privacy: a case study of developing privacy management tools. In *Proceedings of the 2005 symposium on Usable privacy and security*. 35–43.
[26] Ji-Won Byun and Ninghui Li. 2008. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal* 17, 4 (2008), 603–619.
[27] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively secure multi-party computation. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 639–648.
[28] Michael J Carey, Laura M Haas, Peter M Schwarz, Manish Arya, William F Cody, Ronald Fagin, Myron Flickner, Allen W Luniewski, Wayne Niblack, Dragutin Petkovic, et al. 1995. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings RIDE-DOM'95. Fifth International Workshop on Research Issues in Data Engineering-Distributed Object Management*. IEEE, 124–131.
[29] Hanbyul Choi, Jonghwa Park, and Yoonhyuk Jung. 2018. The role of privacy fatigue in online privacy behavior. *Computers in Human Behavior* 81 (2018), 42–51.
[30] Andreas Claesson and Tor E Bjørstad. 2020. Technical report:'out of control'—a review of data sharing by popular mobile apps. *Report for the Norwegian Consumer Council* 14 (2020).
[31] Kevin Collier and Minyvonne Burke. 2022. Facebook turned over chat messages between mother and daughter now charged over abortion.
[32] Matthew Crain. 2018. The limits of transparency: Data brokers and commodification. *new media & society* 20, 1 (2018), 88–104.
[33] Sylvie Delacroix and Neil D Lawrence. 2019. Bottom-up data trusts: Disturbing the 'one size fits all'approach to data governance. *International data privacy law* 9, 4 (2019), 236–252.
[34] Verena Distler, Matthias Fassl, Hana Habib, Katharina Krombholz, Gabriele Lenzini, Carine Lallemand, Lorrie Faith Cranor, and Vincent Koenig. 2021. A systematic literature review of empirical methods and risk representation in usable privacy and security research. *ACM Transactions on Computer-Human Interaction (TOCHI)* 28, 6 (2021), 1–50.
[35] DuckDuckGo. 2025. *DuckDuckGo iOS.* https://github.com/duckduckgo/ios
[36] Lea Duesterwald, Ian Yang, and Norman Sadeh. 2025. Can a cybersecurity question answering assistant help change user behavior? an in situ study. Symposium on Usable Security and Privacy (USEC) 2025-NDSS Symposium.
[37] Jens Ernstberger, Jan Lauinger, Fatima Elsheimy, Liyi Zhou, Sebastian Steinhorst, Ran Canetti, Andrew Miller, Arthur Gervais, and Dawn Song. 2023. Sok: data sovereignty. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 122–143.
[38] European Parliament and Council of the European Union. 2016. *Regulation (EU) 2016/679 of the European Parliament and of the Council.* https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679
[39] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*. 1–14.
[40] Yuanyuan Feng, Abhilasha Ravichander, Yaxing Yao, Shikun Zhang, and Rex Chen. 2024. Understanding how to inform blind and {Low-Vision} users about data privacy through privacy question answering assistants. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2065–2082.
[41] Yuanyuan Feng, Yaxing Yao, and Norman Sadeh. 2021. A design space for privacy choices: Towards meaningful privacy control in the internet of things. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.
[42] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 765–782.
[43] Simone Fischer-Hübner and Farzaneh Karegar. 2024. Overview of Usable Privacy Research: Major Themes and Research Directions. *The Curious Case of Usable Privacy: Challenges, Solutions, and Prospects* (2024), 43–102.
[44] Idoia Gamiz Ugarte, Cristina Regueiro Senderos, Óscar Lage Serrano, Eduardo Jacob Taquet, and Jasone Astorga Burgo. 2025. Challenges and future research directions in secure multi-party computation for resource-constrained devices and large-scale computations. *International Journal of Information Security* 24 (2025).
[45] Google. 2025. *Protocol Buffers.* https://protobuf.dev/
[46] Hana Habib and Lorrie Faith Cranor. 2022. Evaluating the usability of privacy choice mechanisms. In *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. 273–289.
[47] Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10, 4 (2001), 270–294.
[48] Joseph M Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 267–276.
[49] Patrik Hummel, Matthias Braun, Max Tretter, and Peter Dabrock. 2021. Data sovereignty: A review. *Big Data & Society* 8, 1 (2021), 205395172098201 2.
[50] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. 2020. Telekine: Secure computing with cloud {GPUs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 817–833.
[51] Ada Lovelace Institute. 2021. Exploring legal mechanisms for data stewardship. *Ada Lovelace Institute and UK AI Council* (2021).
[52] Yasha Iravantchi, Pardis Emami-Naeini, and Alanson Sample. 2025. Sok:(un) usable privacy: the lack of overlap between privacy-aware sensing and usable

privacy research. *Proceedings on Privacy Enhancing Technologies* (2025).
[53] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*. IEEE, 36–43.
[54] Simon Koch, Manuel Karl, Robin Kirchner, Malte Wessels, Anne Paschke, and Martin Johns. 2025. The Impact of Default Mobile SDK Usage on Privacy and Data Protection. *Proceedings on Privacy Enhancing Technologies* (2025).
[55] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. 2013. A survey of computation offloading for mobile systems. *Mobile networks and Applications* 18 (2013), 129–140.
[56] Florian Lauf, Simon Scheider, Jan Bartsch, Philipp Herrmann, Marija Radic, Marcel Rebbert, André T Nemat, Christoph Schlueter Langdon, Ralf Konrad, Ali Sunyaev, et al. 2022. Linking data sovereignty and data economy: arising areas of tension. (2022).
[57] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 1–13.
[58] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.
[59] Matthew Forsythe, Director Product Management, Android App Safety. 2026. *Android developer verification: Balancing openness and choice with safety.* https://android-developers.googleblog.com/2026/03/android-developer-verification.html
[60] Nextcloud. 2025. *Nextcloud iOS ap.* https://github.com/nextcloud/ios
[61] Helen Nissenbaum. 2004. Privacy as contextual integrity. *Wash. L. Rev.* 79 (2004), 119.
[62] Jonathan A Obar and Anne Oeldorf-Hirsch. 2020. The biggest lie on the internet: Ignoring the privacy policies and terms of service policies of social networking services. *Information, Communication & Society* 23, 1 (2020), 128–147.
[63] OpenWeatherMap. 2025. *OpenWeatherMap API for current weather data.* https://openweathermap.org/current
[64] Shidong Pan, Zhen Tao, Thong Hoang, Dawen Zhang, Tianshi Li, Zhenchang Xing, Xiwei Xu, Mark Staples, Thierry Rakotoarivelo, and David Lo. 2024. A {NEW}{HOPE}: Contextual privacy policies for mobile applications and an approach toward automated generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5699–5716.
[65] Martin Pielot, Amalia Vradi, and Souneil Park. 2018. Dismissed! a detailed exploration of how mobile phone users handle push notifications. In *Proceedings of the 20th international conference on human-computer interaction with mobile devices and services*. 1–11.
[66] Radim Polčák and Dan Jerker B Svantesson. 2017. *Information sovereignty: data privacy, sovereign powers and the rule of law.* Edward Elgar Publishing.
[67] Raluca Ada Popa. 2024. Confidential Computing or Cryptographic Computing? Tradeoffs between cryptography and hardware enclaves. *Queue* 22, 2 (2024), 108–132.
[68] Eric Posner and Eric Weyl. 2018. *Radical markets: Uprooting capitalism and democracy for a just society.* Princeton University Press.
[69] PostgreSQ. 2025. *Foreign Data.* https://www.postgresql.org/docs/current/ddl-foreign-data.html
[70] Mozilla researc. 2020. *What Does it Mean? | Shifting Power Through Data Governance.* https://foundation.mozilla.org/en/data-futures-lab/data-for-empowerment/shifting-power-through-data-governance/
[71] David Rodriguez, Joseph A Calandrino, Jose M Del Alamo, and Norman Sadeh. 2025. Privacy settings of third-party libraries in android apps: A study of facebook sdks. *Proceedings on Privacy Enhancing Technologies* (2025).
[72] David Rodriguez, Jose M Del Alamo, Celia Fernández-Aller, and Norman Sadeh. 2024. Sharing is not always caring: Delving into personal data transfer compliance in android apps. *IEEE Access* 12 (2024), 5256–5269.
[73] Mark Ryan, Paula Gürtler, and Artur Bogucki. 2024. Will the real data sovereign please stand up? An EU policy response to sovereignty in data spaces. *International Journal of Law and Information Technology* 32 (2024), eaae006.
[74] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, Vol. 1. IEEE, 57–64.
[75] Florian Schaub, Rebecca Balebako, Adam L Durity, and Lorrie Faith Cranor. 2015. A design space for effective privacy notices. In *Eleventh symposium on usable privacy and security (SOUPS 2015)*. 1–17.
[76] Canadian Digital Service. 2025. *COVID Alert Mobile App.* https://github.com/cds-snc/covid-alert-app
[77] Signal. 2022. *Technology Deep Dive: Building a Faster ORAM Layer for Enclaves.* https://signal.org/blog/building-faster-oram/
[78] Signal. 2025. *Signal iOS.* https://github.com/signalapp/Signal-iOS
[79] Simplenote. 2025. *Simplenote for iOS.* https://github.com/automattic/simplenote-ios
[80] Chandler Nicholle Spinks. 2019. Contemporary housing discrimination: Facebook, targeted advertising, and the Fair Housing Act. *Hous. L. Rev.* 57 (2019), 925.
[81] SQLite. 2025. *Virtual Table Indexing Information.* https://www.sqlite.org/c3ref/index_info.html
[82] SQLite. 2025. *The Virtual Table Mechanism Of SQLite.* https://www.sqlite.org/vtab.html
[83] SQLite. 2025. *Virtual Table Object.* https://www.sqlite.org/c3ref/module.html
[84] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. 2023. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1596–1612.
[85] Joshua Tan, Khanh Nguyen, Michael Theodorides, Heidi Negrón-Arroyo, Christopher Thompson, Serge Egelman, and David Wagner. 2014. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 91–100.
[86] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).
[87] VLC. 2025. *VLC media player.* https://github.com/videolan/vlc
[88] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 611–626.
[89] Wikipedia. 2025. *Wikipedia iOS.* https://github.com/wikimedia/wikipedia-ios
[90] WordPress. 2025. *WordPress for iOS.* https://github.com/wordpress-mobile/WordPress-iOS
[91] Siyuan Xia, Zhiru Zhu, Chris Zhu, Jinjin Zhao, Kyle Chard, Aaron J Elmore, Ian Foster, Michael Franklin, Sanjay Krishnan, and Raul Castro Fernandez. 2023. Data station: delegated, trustworthy, and auditable computation to enable data-sharing consortia with a data escrow. *arXiv preprint arXiv:2305.03842* (2023).
[92] Yubin Xia, Yutao Liu, Cheng Tan, Mingyang Ma, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. TinMan: Eliminating confidential mobile data exposure with security oriented offloading. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.
[93] Yue Xiao, Zhengyi Li, Yue Qin, Xiaolong Bai, Jiale Guan, Xiaojing Liao, and Luyi Xing. 2023. Lalaine: Measuring and Characterizing {Non-Compliance} of Apple Privacy Labels. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1091–1108.
[94] Angela Yang. 2024. More than 11,000 creatives condemn unauthorized use of content for AI development.
[95] Jinyan Zang, Krysta Dummit, James Graves, Paul Lisker, and Latanya Sweeney. 2015. Who knows what about me? A survey of behind the scenes personal data sharing to third parties by mobile apps. *Technology Science* 30 (2015), 2–1.
[96] Shikun Zhang, Yuanyuan Feng, Yaxing Yao, Lorrie Faith Cranor, and Norman Sadeh. 2022. How usable are ios app privacy labels? *UMBC Faculty Collection* (2022).
[97] Shikun Zhang, Lily Klucinec, Kyerra Norton, Norman Sadeh, and Lorrie Faith Cranor. 2024. Exploring {Expandable-Grid} Designs to Make {iOS} App Privacy Labels More Usable. In *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*. 139–157.
[98] Shikun Zhang and Norman Sadeh. 2023. Do privacy labels answer users' privacy questions. In *Network and Distributed System Security Symposium*.
[99] Yifan Zhang, Zhaojie Hu, Xueqiang Wang, Yuhui Hong, Yuhong Nan, XiaoFeng Wang, Jiatao Cheng, and Luyi Xing. 2024. Navigating the Privacy Compliance Maze: Understanding Risks with {Privacy-Configurable} Mobile {SDKs}. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6543–6560.
[100] Zimperium. 2025. *2025 Zimperium Global Mobile Threat Report.* https://lp.zimperium.com/hubfs/Reports/2025%20Global%20Mobile%20Threat%20Report.pdf
[101] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1296–1310.