OAuthHub: Mitigating OAuth Data Overaccess through a Local Data Hub

Qiyu Li
University of California San Diego
La Jolla, California, USA
qiyuli@ucsd.edu

Yuhe Tian
University of California San Diego
La Jolla, California, USA
yut009@ucsd.edu

Haojian Jin
University of California San Diego
La Jolla, California, USA
haojian@ucsd.edu

Abstract

Most OAuth service providers, such as Google and Microsoft, offer only a limited range of coarse-grained data access. As a result, third-party OAuth applications often end up accessing more user data than necessary, even if their developers want to minimize data access. We present OAuthHub, a development framework that leverages users' personal devices as the intermediary controller for OAuth-based data sharing between cloud services. The key innovations of OAuthHub are: (1) the insight that discretionary data access is largely unnecessary for most OAuth apps, which typically only require access at three well-defined moments—during installation, in response to user actions, and at scheduled intervals; (2) a development framework that requires explicit declarations of intended data access and supports the three common access patterns through intermittently available personal devices; and (3) a centralized runtime permission model for managing OAuth access across providers. We evaluated OAuthHub with three real-world apps on both PCs and mobile phones and found that OAuthHub requires moderate changes to the application code and imposes insignificant performance overheads. Our study with 18 developers showed that participants completed programming tasks significantly faster (9.1 vs. 18.0 minutes) with less code (4.7 vs. 15.8 lines) using OAuthHub than conventional OAuth APIs.

Keywords

OAuth Authorization, Data Overaccess, Personal Data Hub, Edge Processing

1 Introduction

OAuth is an authorization protocol that enables delegated access to resources without requiring users to share their credentials [32, 45]. For example, users can log in to applications (apps) like Uber and Zoom using their Google accounts and share their data, managed by Google, with these apps (Appendix F).

This prevailing protocol faces two key privacy challenges. First, most OAuth implementations offer limited control and transparency to users. In a typical OAuth 2.0 implementation, the third-party OAuth app redirects the user to a service provider's login page to authenticate and grant consent. After that, the app receives an access token to interact with the service provider's resource server on the user's behalf. Once access is granted, data sharing occurs between third-party apps (e.g., Zoom) and service providers (e.g., Google), leaving users with limited control and transparency over when app developers request their data.

To mitigate this concern, OAuth service providers started to allow users to access their personal data sharing under data protection law [15]. However, these efforts also face a few challenges. First, many OAuth providers do not offer this access management feature to users. Second, most OAuth providers do not expire an app's access unless the app has not accessed users' data for six months [9, 29, 30], and a few platforms never expire the access token [1]. Third, even if each service provider implements their own end-user review features, most users do not have the time to manage their granted accesses distributed across service providers [9, 62, 76].

Second, most OAuth service providers offer only a limited range of coarse-grained data access. As a result, third-party OAuth apps often end up accessing more user data than necessary, even if developers want to minimize data access. For example, Uber's AwardWallet feature needs to extract users' itineraries from flight confirmation emails, but it has to request access to users' all email messages since it is the only applicable option in Gmail API.

For the second concern, existing solutions often require service providers to modify their server-side implementations [11, 44, 49]. For example, Google Calendar API added a "free or busy" scope in addition to the all-or-nothing raw event scope [28]. Lodderstedt et al. advocate a new protocol called OAuth 2.0 Rich Authorization Request [49], which asks OAuth service providers to offer more granular access control mechanisms. Yet, implementing these protocol changes in practice is challenging [37]. Service providers need to adjust their APIs while ensuring backward compatibility and give long deprecation periods when changes are necessary.

This paper presents OAuthHub, a development framework that uses personal devices as the intermediary controller for OAuth-based data sharing between cloud services (Figure 1). OAuthHub uses personal devices (e.g., smartphones, PCs) as a data hub, which retrieves data from service providers (e.g., Google calendars) through OAuth (Figure 1 a–f), filters data locally, and allows third-party apps (e.g., Zoom, Uber) to access it also through OAuth (Figure 1 1–6). In doing so, the local data hub allows app developers to reduce the data access without service providers' cooperation and offers users a centralized runtime permission model for managing OAuth access across providers [41].

While a few projects have explored data filtering to achieve data minimization [11, 13, 23, 39], the unique challenge for OAuthHub is leveraging users' intermittently available personal devices as the local data hub. OAuthHub introduces three key design ideas.

First, common OAuth implementations allow app developers to access users' data at their discretion, often without notifying users. One extreme example is that Uber Award Wallet may monitor users'

Proceedings on Privacy Enhancing Technologies 2026(3), 604–624

https://doi.org/10.56553/popets-2026-0098

Figure 1: OAuthHub uses personal devices (e.g., smartphones, PCs) as an intermediary controller for data sharing between third-party apps (e.g., Zoom) and service providers (e.g., Google). OAuthHub retrieves data from service providers through OAuth (a-f), filters data locally, and allows third-party apps to access it through OAuth as well (1-6).

emails continuously after the initial access request. Our key insight is that **for most OAuth-based applications, granting developers discretionary access to user data is an unnecessary—and avoidable—privilege**. In analyzing 62 real-world OAuth apps, we found that most OAuth apps fall into three data access patterns: (1) Install-time, where the app only needs to pull data once to finish tasks like registering, linking, or setting up an account; (2) User-driven, where data access is initiated by the user's actions [60]; and (3) Scheduled time, where the app needs to access data periodically even when the user is not actively using their service. We discuss the few apps that OAuthHub does not support in Section 4.1.

Second, common OAuth implementations require two servers: an authorization server, responsible for managing login requests and URL redirections, and a resource server, which handles API requests from third-party apps. Both servers must always remain on and have static IPs so that third-party apps can establish communication and redirect URLs. However, personal devices typically lack static IPs and are not consistently online due to power limitations and network conditions. Our insight into this challenge is that **servers running on intermittently available personal devices can support the three common access patterns if developers explicitly declare the desired access pattern.** During both *User-driven* and *Install-time*, the user's device is already online, so the OAuth client can interact with the local device directly. In *Scheduled time* scenarios, users have to power on their devices to access content updated when they are offline, so we can defer data access until the device is online again.

Third, the local data hub allows us to build a **centralized runtime permission model** for managing OAuth access across providers, similar to how Android manages App permissions today [50]. Today, a user must grant Zoom full Google Calendar access (i.e., read and write) at installation time. In contrast, OAuthHub requires developers to explicitly declare the fine-grained access they need in a machine-readable manifest [2, 14, 39] and allows users to selectively grant permissions to OAuth apps as well as impose constraints on the usage of resources, similar to Apex [53]. For example, users may only grant Zoom read access to their calendar at the installation time and then grant write access when they need to create a calendar event through Zoom later.

The design of OAuthHub also aims to minimize disruptions to the existing OAuth ecosystem. OAuthHub does not require service providers to change their server implementations. The only adjustments are that (1) third-party developers need to integrate a new login option, and (2) users need to install a mobile app on smartphones or a Chrome Browser Extension on PCs. We developed a lightweight library to ease the integration[1], allowing app developers to integrate OAuthHub with ~50 lines of code. End users are motivated because it allows them to manage data access in a centralized manner and reduce unnecessary data disclosure [41]. We implemented the prototype of OAuthHub on both PC and mobile phones and will open-source the framework.

We conducted detailed experiments to validate the design of OAuthHub. To understand the benefits and limitations of OAuthHub, we analyzed 218 OAuth apps collected from various marketplaces and examined how OAuthHub can mitigate data overaccess in these real-world use cases. We evaluated OAuthHub with 18 students who play the role of developers and found that participants completed programming tasks more efficiently (9.1 v.s. 18.0 minutes) with less code (4.7 v.s. 15.8 lines) using OAuthHub than the conventional OAuth APIs. We then recruited 100 participants on Cloud Research to test whether the fine-grained permissions enabled by OAuthHub can mitigate users' privacy concerns. Our results show that users were 56% to 78% less likely to deny the fine-grained permission requests through OAuthHub than the conventional OAuth permission requests. We developed three OAuth apps for PC and Android based on real-world scenarios by adapting sample source code provided by Google. Our experiments suggest that OAuthHub imposes modest performance overheads on users' local devices.

In this paper, we make the following contributions:

(1) A development framework that uses individual personal devices as the intermediary controller for OAuth-based data sharing between cloud services.
(2) A characterization of when OAuth apps need data access.
(3) A detailed prototype implementation and evaluation of OAuthHub on Android and PC.

2 Design Goal & Threat model

The key concept of OAuthHub is to use personal devices as the local data hubs to mediate the data sharing between OAuth service providers and third-party OAuth apps. Conventionally, these data-sharing activities are opaque to users and third-party auditors. By

[1] https://github.com/AISmithLab/OAuthHub

routing data through user-controlled devices and requiring developers to declare their data access in a machine-readable manifest, OAuthHub makes the process transparent, where app developers can assure users of their efforts in data minimization. It also enables a unified, centralized privacy management system (Section 5).

There has been sustained interest in building hubs to enhance users' privacy, leading to systems such as smart home hubs [39, 69], network privacy hubs like Pi-hole [57], and privacy-setting hubs [41]. OAuthHub extends this ecosystem by adding a new system primitive to hub developers' toolboxes. OAuthHub shares many of the strengths and limitations of these hub-based approaches: it can improve users' privacy by enabling data minimization and centralized data management, but third-party developers may be reluctant to adopt it because of incentives to monetize user data. We hope that OAuthHub can move the needle by offering two value propositions. First, it can increase the proportion of users who accept developers' access requests; for example, Google's Android team has found that exposing unnecessary information leads to unusually high denial rates [31, 66]. Second, as hubs support a wider range of protocols, they become more useful to users, increasing the likelihood that users will adopt the hub model in practice.

Threat model. We envision that some well-intentioned third-party OAuth developers can incorporate a new login option powered by OAuthHub in their websites. OAuthHub's goal is to assure users that these developers only obtain data they declare and users agree to. We make the following assumptions: (1) We assume that service providers (e.g., Google) will not disclose any personal data without appropriate credentials. Server-side data breach is an important problem, but orthogonal to the problem OAuthHub seeks to address. (2) We assume the user's local device is trusted. We conducted a detailed security analysis in Section 9.1.

System model. Users need to run a local OAuth server on each of their devices where they want to use OAuthHub. Our current implementation supports both mobile and web-based apps. OAuthHub is implemented as a Chrome extension for web apps or as a mobile app for mobile devices. For web apps, the Chrome extension uses background scripts to listen for requests from the webpage, then filters data locally and forwards the requested data to the application. OAuthHub requires network access to reach OAuth providers' endpoints and communicates with client apps through the local environment, such as via a browser extension's messaging APIs or a local service running on the user's device.

3 Understanding Data Access in OAuth Apps

To gain insights into the OAuth data access problem, we conducted an analysis of 62 real-world apps that utilize OAuth APIs.

Method. We investigated apps on Google Workplace Marketplace (GWM), which commonly utilize Google's OAuth APIs. We first selected a diverse range of apps across all categories (e.g., creative tools, task management) in GWM, including both highly popular apps and niche ones. We then manually installed the apps to identify the APIs and data types each app requested. We continued this exploration until we found no additional APIs or data types in the last ten apps. Ultimately, we collected 62 OAuth apps that utilize Google's OAuth APIs (see Appendix C). We then analyzed what data these apps request, and what they actually need, assuming each app's functionality based on its description. To complement this data analysis with actual data, we also studied a subset of 11 randomly selected apps using browser inspection and dynamic instrumentation tools (see Appendix A).

Since the source code of the apps is not available, and since our analysis focuses on the use cases of OAuth apps rather than their actual behavior, we inferred the necessary data access scopes from the descriptions of their functionalities. To mitigate potential bias, two authors independently labeled the results and cross-validated the labels, resolving any conflicts through discussion.

Results. We make the following key observations. First, **most apps request more data access than they need to perform their tasks** (50 out of 62, 81%). Most applications in our analysis require arbitrary read/write permission on the resource in their OAuth scopes, while they only need a small subset of the requested data (Appendix A). For example, apps that interact with Google Drive often request extensive write permissions, while they only need to modify specific folders or file types (e.g., Notability). This finding suggests that the granularity of data access in existing OAuth APIs could be improved.

Second, **data access needs are diverse and app-dependent**. For example, Wellybox requires receipt attachments for reimbursement purposes, and TripIt needs transportation and lodging information in emails to organize travel itineraries (Table 6). However, it would be challenging for service providers to implement and for developers to learn such a wide range of data granularities. Additionally, developers can minimize data access in many different ways, such as filtering only emails containing related information and removing unnecessary information in emails, while retaining the app's functionality. This finding motivates us to adopt a plugin architecture where developers can customize data access by chaining a set of operators.

4 Personal Devices as the Data Hub

OAuthHub uses a personal device as the intermediary controller by establishing two OAuth-based data-sharing sessions: one between the developer and the local hub (Figure 1 1-6), and another between the local hub and the service provider (Figure 1 a-f).

Enabling the OAuth data-sharing session between the developer and the local hub (Figure 1 1-6) is non-trivial. Conventionally, the OAuth protocol requires two key servers: an authorization server, responsible for managing login requests and URL redirections, and a resource server, which handles API requests from third-party apps. Both servers must always remain on and have static IPs so that third-party apps can establish communication and redirect URLs. However, personal devices typically lack static IPs and are not consistently online due to power limitations and network conditions. This section describes how we leverage a new understanding of when OAuth apps need to request data to tackle this challenge.

4.1 When do OAuth Apps need Data?

Our key insight is that **allowing developers to request data at their discretion is an unnecessary privilege for most OAuth apps**. Conventionally, once a user grants access to third-party apps, data sharing occurs between the apps and the service providers,

leaving users with limited control and minimal transparency. This design assumes that OAuth apps need to access user data at unpredictable times.

In analyzing 62 OAuth apps described in Section 3, we found that most OAuth apps only need to request data at three types of times: (1) **Install-time** (7/62), where the app needs to pull data once to finish tasks like registering, linking, or setting up an account. For example, when using Quora, users can use OAuth to log in. (2) **User-driven** (36/62), where data access is initiated by users' actions. For example, if a user edits a diagram in Draw.io, it would trigger Draw.io to save files to Google Drive. (3) **Scheduled time** (19/62), where the app needs to access data periodically. For example, Lenovo Smart Frame needs to pull new photos daily to display them.

Fortunately, all these common OAuth data access can be supported by a personal device, even if the device is not designed to be always online. For both install-time and user-driven access, the access is triggered when users are actively using the devices, so the device is online. While scheduled access may occur even when the personal device is not online, most OAuth apps can wait to fetch information until users actually need to use the service, so the device becomes online then. For example, a smart frame may remain off for several days and only needs to retrieve images when it is powered back on.

What cannot be supported? We later expanded our search to more marketplaces (see Section 9.2), actively seeking OAuth apps that cannot be supported by a personal device. We observed that an OAuth app requires discretionary access only when other users initiate actions that require access to a user's OAuth-protected resources. For instance, the Jenkins-GitHub Plugin [3] triggers builds based on GitHub repository events, such as code commits, pull requests, and merges, which are retrieved through the OAuth protocol. Imagine that a collaborator might open a pull request while the original user is offline. Delaying the build process until the owner returns online is undesirable. OAuthHub does not support these OAuth apps; fortunately, cases that genuinely require discretionary access are rare (<3%) among the apps we investigated.

4.2 Declarative Access & Declared Connections

We then made two key changes to the conventional OAuth implementations to support common OAuth access patterns using not-always-on users' devices and to enhance the control and transparency of OAuth-based data sharing. First, OAuthHub requires developers to explicitly declare when they will need to access user data through OAuth. Second, instead of letting app servers initiate connections to the authorization/resource servers, OAuthHub lets the authorization/resource servers connect to the app server at times declared by developers. Note that the IETF OAuth framework [33] only specifies the data flow (e.g., access tokens, data access), but not the network flows.

Declared connections. By having third-party apps declare when they will need access, OAuthHub can establish connections on demand at the appropriate times. Because the hub runs on the user's local device (often behind Network Address Translation barriers), external servers generally cannot initiate inbound connections to it. However, once an app declares its access timing, the hub can proactively initiate outbound connections to the resource server and the third-party app server as needed. For install-time access, the hub retrieves the data immediately after authorization and sends it to the app server. For user-driven access, the app client on the user's device (e.g., a website or mobile app) sends a request to the hub at the moment of access, and the hub responds by delivering the requested data to the app server. For scheduled-time access, developers need to submit the access schedule to the hub during the authorization process. This shifts the architecture from the server making trusted outbound calls to maintaining a public inbound listener. Consequently, developers must set up a public-facing API endpoint and implement robust DDoS protection, rate limiting, and signature verification to validate payloads from random IP addresses. In practice, common cloud endpoints (e.g., AWS) provide built-in DDoS protection and rate limiting configurations, while OAuthHub offers libraries to simplify payload signature verification.

How to implement an OAuthHub client? Third-party apps can integrate OAuthHub by adding a new option, "Sign in with OAuthHub" (Figure 1). We developed a lightweight TypeScript library to help third-party developers adopt OAuthHub (Figure 2). The library hides the complexity behind integrating OAuthHub, such as managing URL redirection and implementing API endpoints. Note that it is a common practice to implement OAuth clients using libraries due to the intrinsic complexity of handling access tokens.

```
//(1) generating authorization URL
const authorizationUrl = OAuthHubClient.generateAuthUrl({
    provider: service_provider, // e.g., google_calendar
    manifest: manifest,         // fine-grained data access
    redirect: redirect_url,     // callback URL after the OAuth
    accessType: access_type     // e.g., user_driven
});
//(2) exchanging authorization code for access token
OAuthHubClient.exchangeToken({
    provider: service_provider, // e.g., google_calendar
    manifest: manifest,         // fine-grained data access
    endpoint: endpoint_url,     // app server endpoint
    authCode: auth_code,        // auth code from callback URL
});
//(3) accessing or writing data through OAuthHub (user_driven access)
OAuthHubClient.query({
    provider: service_provider, // e.g., google_calendar
    manifest: manifest,         // fine-grained data access
    token: access_token,        // access token from step 2
    ...                         // other parameters
});
```

Figure 2: The OAuthHub library provides APIs for: (1) generating authorization URLs, (2) exchanging access tokens, and (3) accessing data through OAuthHub. Notably, developers declare the desired access type in the initial API call (1).

How OAuthHub works? If a user taps "Sign in with OAuthHub", the login web page will redirect the user to a locally hosted authorization interface. After the user grants permission, OAuthHub will behave differently based on the developers' declaration of when they will need to access user data.

For install-time access, the local hub immediately sends the requested data to the API endpoint. For scheduled time access, the local hub sets up a background process to periodically send data as scheduled. If a device goes offline during a scheduled period, it will automatically reconnect to registered services once it is back online. User-driven access is the only scenarios that require the API

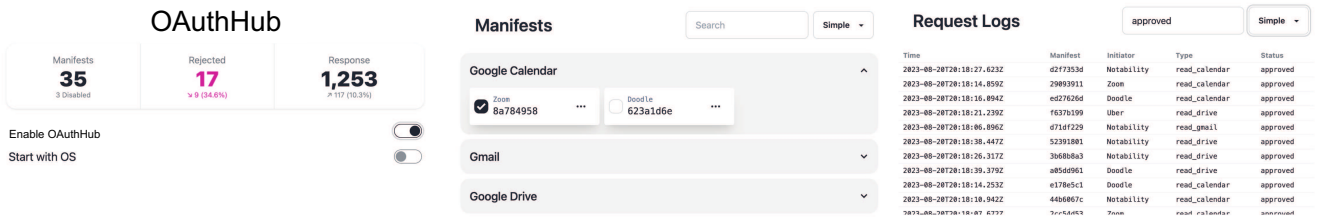


Figure 3: OAuthHub manifest management interface on the web. The global control panel (left) with the usage statistics of the runtime and toggles allows users to enable or disable OAuthHub. The manifests control panel (middle) can manage manifests across various services. The request logs interface (right) shows the access logs for user data via manifests for auditing.

(3) in Figure 2, where developers need to actively send requests to the local hub. For example, Draw.io may want to save a file through OAuth whenever a user edits a diagram.

The local hub will directly respond to a request if requested resources are readily available locally. If the authorization for OAuthHub is never granted or has expired, the user needs to go through a conventional OAuth authorization process with the resource provider (Figure 1 a-f). The local hub later stores all access tokens obtained from OAuth service providers, preventing the need for repeated user logins.

5 Runtime Permission Model for OAuth

The local data hub also allows us to build a centralized permission model for managing OAuth access across providers.

5.1 On-demand Permission Granting

Current OAuth implementations often require users to grant all involved permissions upfront, similar to Android install-time permissions in the early days [70]. For example, a user today must grant Zoom full Google Calendar access (i.e., read and write) at the installation time.

Inspired by the evolution of Android Permissions [21, 22], OAuthHub enables selective permission granting, similar to the modern Android permission model [6, 55, 62]. With OAuthHub, Zoom needs to create separate manifests for read and write operations. Users can only grant reading access at the installation time and then grant writing access when they need to create a calendar event through Zoom for the first time.

This permission model is feasible because personal devices serve as intermediary controllers in OAuth-based data sharing. It would be challenging to enable this model in the conventional OAuth, because developers may request access while the user is offline.

Authorization interface. OAuthHub generates an authorization interface by analyzing the manifest associated with an OAuth app (Figure 4). The interface uses a multi-level design to balance detail and user comprehension. It provides high-level summaries for average users, while technical users can inspect processing steps and preview the actual data transmitted. The interface comprises three views: an overview, processing steps, and a live preview. The overview displays both original and processed data. We use a static analyzer to generate step-by-step descriptions of the manifest pipeline, translating operator names and parameters into natural language. The live preview shows the actual data to be transmitted to third-party apps, based on the execution results of the manifest.

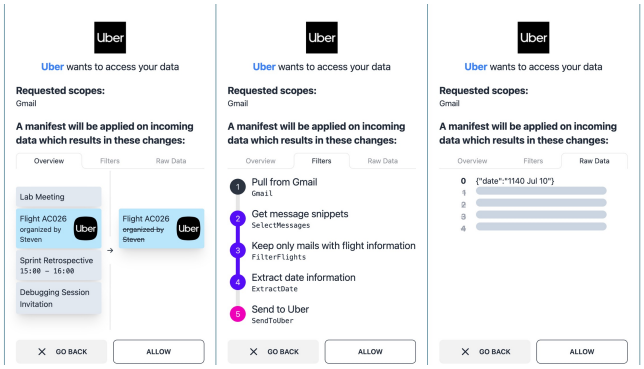


Figure 4: Example of an OAuthHub authorization interface on the Android device. OAuthHub runtime can generate centralized notice/control interfaces by parsing the machine-readable manifests: an overview (Left), the process steps (Middle), and a detailed data view (Right).

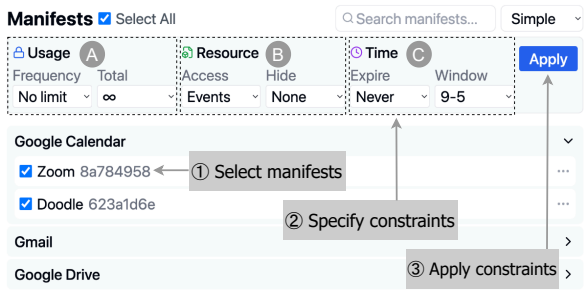


Figure 5: OAuthHub permission management interface on the web. OAuthHub allows users to specify three types of permission constraints through a centralized management interface: (A) Usage constraints: limit access frequency (e.g., twice per week) or total uses (e.g., one-time); (B) Resource constraints: restrict to specific resources (e.g., folders, file types) or obfuscate sensitive fields (e.g., names, emails); (C) Time constraints: set expiration duration (e.g., 24 hours) or restrict to specific time windows (e.g., business hours).

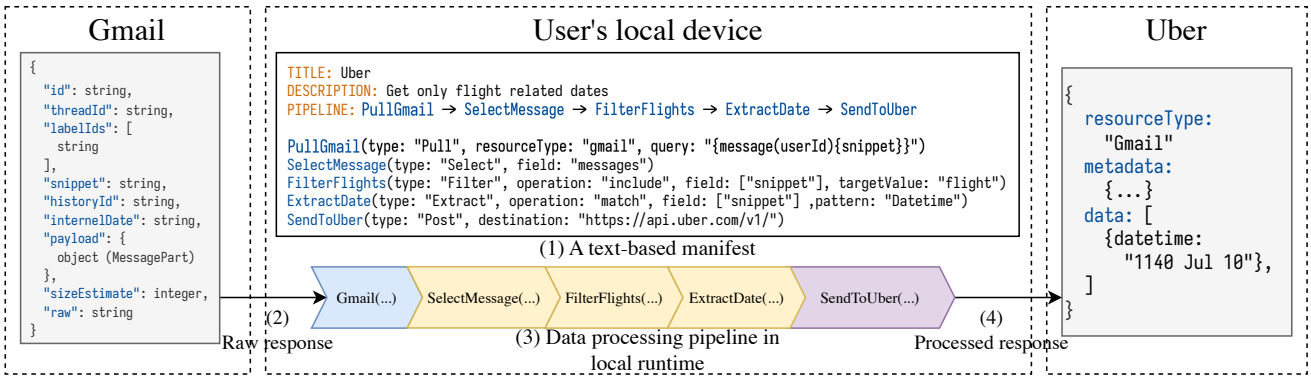


Figure 6: Uber wants to scan users' emails for flight information to book rideshares upon arrival. OAuthHub can help Uber developers access Gmail data via OAuth while minimizing data access. Uber can create a manifest by chaining five operators (i.e., Pull, Select, Filter, Extract, and Post). When responding to Uber's data requests, OAuthHub (1) pulls raw data from Gmail, (2) filters for flight-related emails, (3) extracts the flight dates, and (4) sends only the necessary information to Uber.

5.2 Centralized Privacy Management

The local data hub can also enable an open platform around the OAuth protocol, where well-intentioned third-party developers and privacy advocates can build centralized privacy management features. Because the manifests are machine-readable, OAuthHub can enable many privacy management features to enhance usability. We developed several features to demonstrate the feasibility.

Figure 3 Right shows an interface where users can review the actual access logs of all OAuth apps. The logs contain the time, the initiator, the manifest, and the action type for each third-party request. These logs offer valuable insights into the data usage patterns of OAuth apps, benefiting both auditors and end-users. Figure 3 Middle shows another interface that allows users to revoke their permissions in a centralized manner, which displays a list of manifests they've authorized. Once revoked, OAuthHub expires the corresponding access tokens app developers obtained previously. OAuthHub also allows users to create management policies across OAuth apps, including the number of access, the types of resources, and the time of access (Figure 5). Note that interfaces (Figure 4, 3, 5) are illustrative prototypes to demonstrate the feasibility of automatically generating centralized notice/control interfaces by parsing manifests. They do not represent the final user experience.

6 Fine-grained OAuth Data Access

OAuthHub adopts a Pipe-and-Filter architecture [39, 48, 56] to enable fine-grained OAuth data access.

6.1 Operator-based Manifest

OAuthHub specifies the OAuth access as a data pre-processing pipeline connected with a set of stateless operators with known semantics, each performing a specific type of transformation. We design these operators to reduce OAuth data access and implement them as an open-source library. Third-party app developers declare data access by composing these operators into a pipeline and specifying the pipeline in a text-based manifest, which the local runtime uses to generate permission prompts and execute the pipeline using pre-loaded operator implementations locally. So developers can only interact with OAuth resources in ways supported by the filters. The operators are designed to support common data minimization only, rather than replicate all processing logic.

Figure 6 presents a pre-processing pipeline for Uber that extracts flight information from Gmail to offer recommendations based on the user's itinerary. The process begins with the "PullGmail" operator, which retrieves the user's Gmail content, followed by "SelectMessages" to extract relevant email messages. Next, "FilterFlights" uses regex matching to identify emails containing flight information, and "ExtractDate" extracts the date and time details. Finally, "SendToUber" sends the data to the Uber service.

OAuthHub operators are application-agnostic. When the OAuthHub loads JSON data from the service provider, the hub runtime parses the data according to the schema defined by the service provider. All OAuth data is represented in JSON, enabling filtering via a small set of SQL-like operators. In doing so, the same operators can interact with data streams from different providers. Developers only need to adjust the operator properties to reuse them across applications. For example, a filter operator in Figure 6 is specified as [type: "Filter", operation: ">", field: "start.dateTime", targetValue: NOW]. Compared to SQL operators, these operators have well-defined semantics related to privacy.

Operators v.s. Attributes. One alternative design to implement fine-grained data access is through attribute-based data access [36]. For example, OAuthHub may introduce a declarative attribute-based data access, such as flight confirmation email access, to handle the Uber scenario. We decided to use the operator-based approach because it can enable more flexible data access management. For example, OAuthHub may further reduce the access in Figure 6 by inserting operators that remove attachments from the email and obfuscate names in the email text. In contrast, it would be onerous for service providers to implement numerous potential attribute-based data accesses and for developers to learn these APIs.

6.2 Operator Design

We used best practices in API design [10] to guide the design of these operators. We started with operators in [39, 48] and used

them to implement the use cases of OAuth apps collected in Section 3. We then iteratively expanded the operator design to accommodate additional use cases. Finally, we removed the unused operators. This process resulted in thirteen operators grouped into five categories: `provider`, `reduction`, `transform`, `network`, and `utility` (Appendix G).

Data exchange. Since OAuthHub serves as a proxy for processing raw data requests, it needs to mediate data between OAuth resource servers and third-party applications. For OAuth apps retrieving data from OAuth resource servers, we provide the "Pull" operator to fetch only the necessary data. We allow developers to specify exactly what data they need from an OAuth resource using a GraphQL query, a widely-used data query language for declarative API fetching [35]. For example, in Figure 6, the "Gmail" operator specifies only the snippets of email messages are needed. This approach offers three benefits. First, it reduces the risks of data overaccess and improves network efficiency by transmitting only the necessary data. Second, it enhances transparency by exposing what specific data is being used. Third, it reduces the burden on developers to adapt to new API versions, as the data representation tends to remain compatible across API evolution [25]. Developers can use the *Post* operator to send the processed data to their application services.

To allow apps to perform actions on the OAuth resource server, we provide *Receive* operator to handle requests from third-party applications. We require developers to encode their requests in JSON to maintain uniform data representation with OAuth data. This includes an `action` field representing the API name of the OAuth resource (e.g., create, update, delete), a `body` field for the HTTP request body (e.g., file content), and a `parameters` field listing all parameter names and values (e.g., metadata). The *Write* operator acts as a wrapper, invoking the API of the OAuth resource with the specified parameters to perform the actions.

Data filtering and transformation. Due to the coarse nature of OAuth scopes, OAuth data often includes irrelevant information for the third-party application's needs. For example, Uber only needs emails with flight information rather than all emails. While developers can minimize data by specifying necessary fields using GraphQL, we also provide `Reduction` and `Transform` operators to support more complex logic. The `Reduction` operators trim data to only preserve the necessary information. For example, *Filter* operator can be used to exclude irrelevant data based on specific conditions, such as removing emails unrelated to flights. *Extract* operator can retrieve information based on particular data patterns, like extracting flight dates from related emails. The `Transform` operators process the data to produce derived or aggregated results. *Aggregate* operator combines multiple data items to generate statistical reports, such as counting the number of responses for a specific Google form. *Map* operator transforms individual data items, such as converting birth dates into age categories for demographic analysis. *Anonymize* operator modifies data to enhance privacy, like adding noise to ages in user profiles.

Specifying action constraints. Since action requests are also represented in JSON format, well-intended developers can use the *Filter* operator to restrict their actions similarly. For example, they can limit file paths for write operations by filtering requests where the file's parent ID matches the destination folder's file ID. If there is no match, the *Write* operator will not execute any action with null input. This additional sanity check helps mitigate the risks of data overaccess and maintain data integrity.

7 Case Studies

We present three example OAuthHub use cases.

Zoom accesses Google Calendar. Zoom's current integration with Google Calendar requires users to grant Zoom permission to view, edit, share, and permanently delete all the calendars they can access. OAuthHub can help Zoom mitigate this over-privilege access using a manifest that filters calendar events that do not contain Zoom meeting links. We implemented the client using the example code provided by the Google Calendar API. We then added 44 lines of code and removed 20 lines to support OAuthHub.

Uber accesses Gmail content. Uber wants to access users' flight confirmation emails to offer travel personalization. Its current integration with Gmail requires users to permit Uber to read all their emails and settings. OAuthHub can help Uber mitigate this overaccess by extracting only the relevant flight date information using the manifest in Figure 6. We implemented the client using the example code provided by Gmail API. We then added 45 lines of code and removed 28 lines to support OAuthHub.

Notability writes to Google Drive. Notability is a note-taking application that combines handwriting, photos, audio and typing in a customizable interface. Notability's backup feature requests permissions to read, write and delete files in all the folders on Google Drive. OAuthHub can prevent Notability from writing to arbitrary paths in Google Drive by filtering the parameters of requests. We implemented the client using the example code provided by Google Drive API. We then added 51 lines of code and removed 36 lines.

8 Implementation

The development framework of OAuthHub comprises a developer-facing library and a local hub runtime.

Developer-facing library. We developed a lightweight TypeScript library ($\approx$ 300 lines of code) to help third-party OAuth app developers integrate OAuthHub. The library hides the complexity of implementing OAuth protocols, ensuring that developers only have to make minimal code modifications. To support OAuthHub, developers only need to generate redirect URLs and call APIs to retrieve and consume access tokens (Figure 2).

We also developed an Integrated Development Environment (IDE) to assist developers in authoring and debugging the manifest. Our IDE features syntax highlighting, auto-completion, and built-in documentation. Developers can insert a *Debug* operator in the manifest pipeline to output intermediary results for debugging.

Local hub runtime. We developed a hub runtime to enable the permission model on users' personal devices. The hub runtime executes operator-based manifests, hosts local-hosted authorization web pages, manages access tokens from OAuth service providers, and offers data access management interfaces. We implemented the core runtime using Node.js, and then developed a Chrome extension for PCs and an Android app for mobile phones. Since Node.js is cross-platform, we reused most of the code across PC and mobile.

Security. One potential concern is that local devices may become lucrative targets for attackers, as OAuthHub centralizes the management of OAuth tokens. To mitigate this risk, OAuthHub leverages platform-specific security mechanisms. For Chrome extensions, OAuthHub invokes the `chrome.identity.getAuthToken` API for authorization, which eliminates the need to store the client secret within the extension, maintains tokens in Chrome's internal secure storage, and automatically obtains and refreshes access tokens. For Android, OAuthHub uses the AppAuth SDK [8] to perform OAuth authorization with servers, which follows OAuth security best practices for native apps [18]. To reduce the risk of a single point of compromise across multiple services, OAuthHub encrypts tokens using unique keys for each service and stores the keys in secure OS storage such as Android's KeyStore [7].

9 Evaluation

This section presents detailed experimental evaluations of OAuthHub, including its security analysis (Section 9.1), effectiveness in mitigating overaccess risks in OAuth apps (Section 9.2), usability for developers (Section 9.3), how it addresses users' privacy concerns (Section 9.4), and system performance (Section 9.5).

9.1 Security Analysis

Using personal devices as intermediaries introduces additional attack surfaces.

User identity impersonation. Since OAuthHub lets third-party apps communicate with the user-controlled device rather than directly with the authoritative OAuth provider, third-party apps need to verify the user's identity during authorization and in subsequent communications to prevent false claims of another user's account. OAuthHub obtains identity tokens (e.g., JWTs) from the service provider, and applications verify these tokens with the provider as usual to confirm user identity. As a result, OAuthHub cannot falsely claim access to an unrelated account, because the tokens cryptographically bind the user to the account. During authorization, OAuthHub generates a unique key pair and shares the public key with the app; it then signs all subsequent payloads with the private key, enabling the app to verify authenticity from random IP addresses.

Physical device loss/theft. A concern is that users could lose their personal device, allowing an unauthorized person with physical access to access all users' data across all OAuth service providers. OAuthHub employs an on-demand loading strategy: tokens and data are stored only when explicitly requested and automatically expire according to the session policies of service providers.

Local device compromise. Attackers may steal OAuthHub tokens by compromising the user devices (e.g., rooting the OS). OAuthHub mitigates this by only storing encryption keys and encrypted tokens in platforms' secure storage.

Man-in-the-middle attacks. Malicious third-party libraries or apps may intercept tokens or authorization data in transit. OAuthHub mitigates this by implementing PKCE [52], which requires a matching code verifier for token exchange to prevent authorization code interception attacks. Additionally, OAuthHub verifies web page origins and app signatures to ensure request integrity, and rotates access tokens (i.e., issuing a new token after each access and invalidating the prior one) to prevent token replay attacks.

UI Spoofing: Malicious apps may impersonate OAuthHub to trick users into granting access. OAuthHub mitigates this by performing authorization through platform's secure UI framework (e.g., `chrome.identity.launchWebAuthFlow`), ensuring users interact only with system-rendered consent screens.

Users do not read the interface. Abusive developers may request OAuth data excessively without filtering (e.g., fetching raw data every second), and users may approve without reading the interface. Because manifests are machine-readable, future work can develop automated algorithms that analyze access patterns, detect malicious or overly aggressive behavior, and potentially auto-deny such requests.

Security benefit. One benefit is that OAuthHub manages refresh tokens on behalf of developers, which may prevent developers from mistakenly exposing refresh tokens in the front-end code, as found in prior studies [63, 75].

9.2 Mitigating Overaccess in OAuth Apps

We examined how OAuthHub mitigates the overaccess risks in 218 OAuth apps qualitatively.

Method. We investigated a broader set of OAuth apps to assess how well OAuthHub generalizes to different use cases. We first collected 21 OAuth app marketplaces and grouped them into four categories: platform, productivity, communication, and programming. We then selected the most popular marketplaces for each category: Google, Trello, Slack, and Github. For each marketplace, we selected the apps similarly to the study in Section 3. This search concluded with 218 apps in total. Two authors then collaboratively implemented a manifest for each app function to explore ways to mitigate data overaccess. We then independently and manually coded the manifest and discussed high-level categories for the resulting codes to agree on a selected scheme.

Results. In our analysis of 218 apps, only six apps require discretionary access, which our system cannot support. These apps fall into three categories: (1) CI pipeline triggers (e.g., Jenkins, Google Cloud Build) that trigger builds when code changes occur, (2) Security scanning (e.g., GitGuardian) that detects exposed secrets for each pull request, and (3) Task sync tools (e.g., Gitlab) that maintain consistency across project management platforms across team members. For the remaining apps, we examined how OAuthHub minimizes their data access. We created a codebook with 9 common methods for mitigating data overaccess, grouped into three categories: data minimization, action restriction and access timing restriction (Table 1).

OAuthHub provides four data minimization methods when retrieving data from OAuth resource servers: data filtering, content selection, data anonymization, and data aggregation. Data filtering is the most common method (161 out of 218 apps), which screens related data items based on specified criteria, such as file types or app-specific tags. Content selection is also very common (139/218). Since many apps only need specific types of information in the data, we can extract only necessary information from complex OAuth data representations, such as retaining only related information (i.e.,

localId, email and language) within user profiles for Google Colab. Data anonymization (51/218) enhances user privacy when the apps need to access Identifiable Information (PII), such as hashing user emails. Data aggregation is currently available for only 7 out of 218 apps. Data aggregation is applicable to only 9 out of 218 apps, as the remaining apps require raw or individual-level data. We currently support 7 of those 9 apps, as the remaining two depend on complex aggregation operations (e.g., conditional sums and nested aggregations) that our system does not yet support. We anticipate our supported operations will expand over time to accommodate more use cases.

For apps interacting with OAuth resources, OAuthHub offers two common types of action restriction: limiting action types and constraining action conditions. The first (176/218) controls which API methods third-party apps can use, like allowing only updates to documents but not creation or deletion in Google Docs. The second (155/218) imposes specific constraints on actions, such as limiting file paths or accepted file types for upload.

OAuthHub also enforces access timing restrictions based on data access patterns. For install-time apps (9/218), it only allows one-time use after installation. For user-driven apps (135/218), background access is not feasible when the user's device is offline. For scheduled time apps (68/218), it limits access duration and frequency.

9.3 API Usability for Developers

We conducted an IRB-approved study with 18 developers to evaluate the usability of our API.

Method. We aimed to study how OAuthHub can support developers to access and process personal data. Specifically, we are interested in (1) whether OAuthHub is faster and easier to use than the conventional OAuth APIs for app developers and (2) whether OAuthHub helps developers effectively minimize data access.

We asked participants to complete programming tasks involving personal data. We derived four tasks from common OAuth app use cases in Section 3 (see Table 2). The study was a within-subjects design, where participants used both conventional OAuth APIs and OAuthHub to complete two tasks each. We counterbalanced the presentation order of APIs and tasks across participants. We did not disclose that one of the APIs was developed by the authors to prevent introducing potential bias.

Since CS students are considered acceptable substitutes for developers [67], we recruited 18 CS students from multiple universities. Participants in the study (13 identified as male, 5 identified as female, aged 19-25) were undergraduate and graduate students with at least 3 years of programming experience. Each participant received a 20 USD gift card as compensation for their time. The study was performed in our lab or remotely over the Zoom meeting. We obtained participants' consent electronically via Google Forms at the beginning of each study.

The study took about 90 minutes for each participant, including two 45-minute sessions. Each session included a 10-minute walk-through, a 30-minute programming period and a 5-minute debriefing. During the walk-through period, we guided participants through warm-up tasks to get them familiar with OAuthHub or conventional OAuth APIs. In the programming period, we instructed participants to complete the assigned tasks. They were free to search the Internet for documentation and solutions. For a fairer comparison, we did not require the participants to declare the scopes using conventional OAuth APIs, as OAuthHub automatically requests and manages OAuth scopes. During the debriefing period, participants completed the NASA Task Load Index (NASA-TLX) questionnaire [34] to rate six aspects of cognitive load on a 7-point Likert scale. After the study, participants participated in a semi-structured interview to discuss their experiences using two APIs.

Results. Table 3 shows the results of our developer study. All participants completed the tasks successfully for both APIs. Regarding the developers' efficiency, participants spent less time (average of 9.1 v.s. 18.0 minutes) to complete the tasks using OAuthHub than using the conventional OAuth APIs. The Holm-Bonferroni corrected Mann-Whitney U test showed significant differences in completion time between the two APIs ($p < 0.05$ for all tasks). We also counted the lines of code (LOC) in the final programs. On average, participants used less code (4.7 v.s. 15.8 lines) to complete the tasks with OAuthHub than the conventional OAuth APIs. Participants could finish tasks 1 to 3 using OAuthHub with only 5-7 lines of code, and task 4 in just 2 lines.

To quantify the privacy benefits of OAuthHub, we tested the manifests created by participants using a real-world dataset curated from the authors' personal data, to avoid collecting sensitive data from participants. The dataset consists of 500 emails, 97 calendar events, and 9 form responses. We measured the percentage of data egress reduction in JSON entries and bytes. On average, OAuthHub filtered out 96% of emails, 97% of calendar events, and 44% of form responses, resulting in data reduction rates of 95% for Zoom and over 99.9% for both WellyBox and FormLimiter. The observed data reduction results from data pre-processing on the edge. For example, Zoom previously retrieved all raw Google Calendar events. In contrast, OAuthHub filters data locally on users' devices and only relays the processed results.

Additionally, we measured the perceived usability of APIs (Figure 7). Participants perceived lower mental load, effort and frustration, along with better performance with OAuthHub than conventional OAuth APIs. These differences were statistically significant under a Wilcoxon Signed-Rank test (all $p < 0.01$).

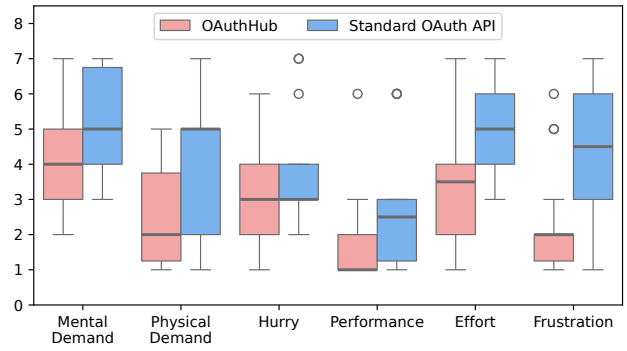


Figure 7: The cognitive load measured by NASA-TLX. Participants perceived significantly less mental load, effort and frustration using OAuthHub than conventional OAuth APIs.

Category	Method	Usage Scenario	# Support
Data Minimization	Data Filtering	Apps require only a portion of data that is relevant	161 / 218
	Content Selection	Apps only need partial data in specific fields	139 / 218
	Data Anonymization	Apps require Personally Identifiable Information (PII)	51 / 218
	Data Aggregation	Apps only need aggregated results	7 / 218
Action Restriction	Restricting Action Type	Apps only need to perform specific types of operations	176 / 218
	Restricting Action Conditions	Apps only need to perform actions satisfying specific conditions	155 / 218
Timing restriction	Allowing only once	Apps are intended for one-time use during installation	9 / 218
	Allowing only while using	App only need access when user is actively using the app	135 / 218
	Allowing only at scheduled times	Apps need to access data periodically	68 / 218
Not supported	–	Apps require discretionary access	6 / 218

Table 1: We implemented manifests for 218 OAuth apps, analyzed the types of methods for mitigating data overaccess in these manifests, and examined the applicable usage scenarios.

Task ID	App	Task description
Task 1	Zoom	Get all upcoming Zoom meetings from Google Calendar
Task 2	WellyBox	Extract financial receipt attachments from Gmail
Task 3	FormLimiter	Get the number of valid responses from Google Forms
Task 4	Notability	Backup notes to Google Drive

Table 2: The programming tasks used in the developer study.

Task ID	Conventional OAuth API		OAuthHub	
	Time	LOC	Time	LOC
Task 1	16.0 ± 5.8	19.2 ± 6.2	**9.3 ± 3.6**	5.3 ± 1.0
Task 2	25.1 ± 10.9	13.6 ± 6.1	**14.5 ± 5.5**	6.7 ± 0.5
Task 3	13.3 ± 4.7	8.2 ± 2.8	**9.1 ± 2.8**	5.9 ± 0.8
Task 4	18.1 ± 3.8	24.6 ± 9.9	**3.8 ± 1.8**	1.6 ± 0.7
Average	18.0 ± 5.4	15.8 ± 5.7	**9.1 ± 3.0**	4.7 ± 1.4

Table 3: In the developer study, participants spent less time and used less code to complete the programming tasks with OAuthHub than conventional OAuth APIs. Format: mean ± standard deviation.

In the post-study interview, participants provided more positive feedback on our API. When asked about their preferences between the two APIs, nearly all participants (16/18) leaned towards OAuthHub for programming, citing its ease of use over the original API provided by Google. The rest opted for conventional OAuth APIs because they were more familiar with the programming language or preferred procedural programming. However, they also envisioned that developers could complete programming tasks more efficiently with OAuthHub once getting familiar with its syntax, as it requires less code and is more concise.

Participants noted several features of OAuthHub that contributed to their positive feedback. Most participants (12/18) appreciated the simple syntax of our manifest language, describing it as "intuitive" and "more readable" since each operator conveyed the high-level semantics of data transformation. Many participants (8/18) also appreciated its modular structure, likening it to SQL, and mentioned its advantages for analysis and debugging due to the clear separation of functions. Besides, some participants (6/18) valued the clarity of the documentation, finding it easy to navigate and helpful for grasping the language despite the initial learning curve. However, several participants (4/18) also noted the semantic limitations of our API language, which are intentionally designed to restrict data processing behaviors and mitigate potential vulnerabilities.

9.4 Mitigating User Privacy Concerns

We evaluated whether OAuthHub can mitigate users' privacy concerns using an online survey based on three scenarios from our case studies (Section 7).

Method. The survey used a between-subjects design where each participant experienced three scenarios in a randomized order. For each scenario, we randomly presented each participant with one of the two interfaces. We started by checking if they understood the implications of granting access to gauge their understanding of the permission requested by OAuth apps. Then, we asked how likely they would grant access to the OAuth app on a 5-point Likert scale. We included an open-ended question at the end of each scenario for users to explain the rationale behind their choices. The full survey questions are provided in Appendix E.

We recruited 100 participants through Cloud Research. We chose participants with an approval rate greater than 90% who reside in the U.S. To ensure the quality of responses, we included an attention check at the end of the survey and filtered out 4 responses that did not pass. On average, the survey took each participant 4-6 minutes to complete. Each participant received a compensation of 1 USD for their time. The study was approved by our institution's IRB.

Results. Figure 8 presents the results of the survey. On average, 97%, 94% and 91% of participants understood that conventional OAuth permissions would grant apps full access to personal data for Zoom, Uber and Notability, respectively. With OAuthHub, 97%, 87% and 96% of participants correctly identified the restricted scope of permissions granted for these apps. The results suggest that users can clearly understand the permissions requested by OAuth apps to make informed decisions across different scenarios.

Over 40% of users were more likely to deny the original OAuth permission requests for all three scenarios. We looked at open-ended responses to understand their decision rationale. Many participants were concerned that granting the permissions would allow the app to access more data than needed, especially when it involves sensitive information. For example, P16 noted that "*I don't think Uber needs access to all of my emails, some of which contain confidential material. I would allow them to have access to a separate folder that I send travel info to, but not all of my emails.*" Participants also

voiced concerns about the potential of modifying unrelated data: "*Given the fact that it can view and edit all of my calendar events, I would be worried about mistakes or errors it would make with my calendars. I would need to be able to view the changes that it makes in order to ensure that it is only focused on Zoom meetings and changes that are made in relation to that*" (P62).

With OAuthHub, the likelihood of denial requests (i.e., somewhat or very unlikely to grant permissions) decreased from 42.5% to 12.0% in the Zoom scenario, with a reduction rate of 72%. Similarly, the denial rates dropped by 56% for Uber and 78% for Notability. These results suggest that users are less likely to deny the fine-grained permission requests of OAuthHub than the original OAuth requests.

Participants' feedback also suggests that OAuthHub can help to alleviate users' privacy concerns. They appreciated the idea of limiting access to only the necessary data. P38 mentioned that "*It's not excessive and restricts itself to the links/contents that it should be concerned with, i.e., zoom links, not other info that it has no right to access.*" Further, participants seemed to feel more comfortable with the fine-grained permissions enabled by OAuthHub: "*It is restricted to the content that is relevant, not all content which it is not entitled to access to use. If I needed to use the service / feature, then I would be more likely to proceed with this sort of limited access*" (P46) and "*If I needed to use this app for a specific functionality / purpose, then I would feel comfortable being able to select a narrow scope of access versus allowing carte blanche access to everything*" (P55).

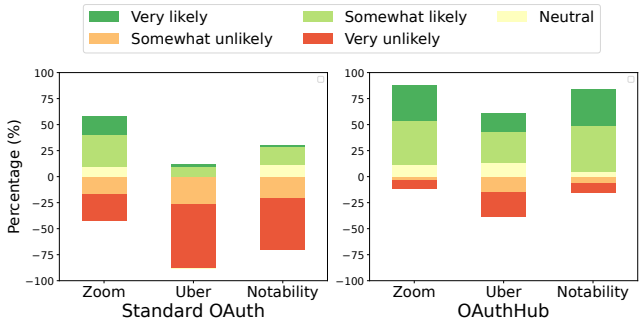


Figure 8: Users' likelihood of granting permissions for conventional OAuth requests and OAuthHub across three OAuth applications. Participants were more likely to approve the restricted permissions enabled by OAuthHub than conventional OAuth requests for all three OAuth applications.

9.5 System Performance

We used three example cases to quantify OAuthHub's system overhead in memory, CPU utilization and latency.

Method. We tested all use cases on both PC and mobile settings. For the PC, we used an Apple M1 MacBook Pro running Node.js v20.3.1. For mobile, we used a Huawei ELE-AL100 smartphone with 6GB memory, running Android 10. For each use case, we requested the data from the client side of OAuthHub and measured the CPU utilization rates and peak memory usage during this process. We also measured the end-to-end latency from the moment the request was initiated until when the response was received. We executed each request 100 times to reduce potential inaccuracies caused by network fluctuations. We then compared the results with the latency when directly using the APIs provided by Google under the same test conditions.

For **energy consumption**, we charged the smartphone to full and connected it to a POWER-Z KT001 Portable USB-C Fast Charging Tester. We set the power plan of the smartphone to 'performance first' to minimize measurement errors from system optimizations. We then ensured the target application remained active for at least one hour, during which we recorded the average hourly energy usage. We conducted power consumption tests under four different conditions: (1) complete idle state, when all applications were shut down; (2) OAuthHub idle, when only OAuthHub was running but not executing any manifests; (3) OAuthHub Google Calendar | Gmail | Google Grive active, when OAuthHub was executing the manifest every 10 seconds; and (4) only a popular messenger app was running in the background. All tests are done with screen on at 50% brightness level.

Results. Table 4 shows the CPU and memory usage of OAuthHub under PC and mobile settings. OAuthHub consumed less than 1% of CPU time and around 90MiB of memory on PC, and less than 5% of CPU time with a peak memory usage of 200 MiB on the mobile device. Our results suggest that OAuthHub introduces modest CPU and memory overheads on the user's local device.

Figure 9 presents the latency of OAuth access with OAuthHub (dot bars) and without OAuthHub (slash bars). OAuthHub introduced relative overheads of 1.30, 1.25 and 1.16× on PC, and 1.33, 1.22 and 1.47× on mobile for Google Calendar, Gmail and Google Drive respectively. Google Drive exhibits the highest overhead (1.47×) on mobile due to its resource-intensive nature, which might not be well-suited for mobile devices with lower processing power. We observed a lower relative overhead in Gmail on mobile because this app is more expensive as it handles a large volume of email messages, making the overhead less noticeable. On average, OAuthHub introduced an average latency of 195ms on PC and 849ms on the mobile device.

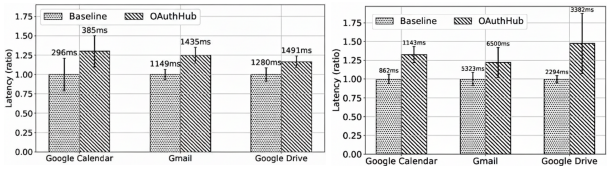
The energy overhead results are shown in Table 5. OAuthHub causes a 12% to 39% energy overhead when running on a mobile phone, depending on the specific workload. The seemingly high percentages are mostly a result of our run-nothing blank control group. Actually, in the same testing environment, this overhead is similar to the one caused by running a messenger app.

Scenario	PC		Mobile	
	CPU	Memory	CPU	Memory
Google Calendar	< 1%	90.6 MiB	< 1%	181.5 MiB
Gmail	< 1%	92.5 MiB	< 5%	197.3 MiB
Google Drive	< 1%	91.6 MiB	< 3%	188.6 MiB

Table 4: Performance overhead of three use cases using OAuthHub when responding to 100 consecutive requests. CPU utilization rate represents the average percentage of active time consumed for 100 consecutive requests and memory indicates the peak memory usage within this process.

10 Related Work

Data Overaccess in Commericial Protocols. Many studies have studied the security and privacy of commercial protocols [4, 12, 16, 24, 47, 61, 64, 65, 71, 72]. The most relevant line of work is to



(a) PC (b) Mobile

Figure 9: Average relative latency overhead of three OAuth applications on PC and mobile settings.

Benchmark Scenario	Energy (Joule/Hour)	Δ energy (Joule/Hour)	Overhead (%)
Nothing running	3312	-	-
OAuthHub idle	3708	396	11.96
Google Calendar active	3816	504	15.23
Gmail active	4608	1296	39.13
Google Drive active	3960	648	19.56
A messenger app running	3852	540	16.30

Table 5: Energy consumption overhead of three use cases using OAuthHub.

study the overaccess problems in OAuth and IFTTT [13], where app developers request data access that they do not necessarily need [11, 13, 20, 23, 39, 46, 51]. For example, Chen et al. [13] identified two key design flaws in IFTTT that can cause data privacy problems: attribute level overprivilege and token-level overprivilege. In contrast to these works, OAuthHub focuses on the temporal domain and demonstrates that discretionary access to user data through OAuth is an avoidable privilege.

Several works have specifically analyzed privacy issues in OAuth and proposed tools to detect identity leaks [38, 65]. For example, SSO-Monitor provides large-scale monitoring of Single Sign-On implementations and their associated OAuth flows, collecting metrics on security and privacy practices across millions of websites to identify insecure or improper configurations [38]. In contrast to existing tools that monitor and analyze identity information leaks, our work enables developers to proactively limit data access.

Manifest & Operator-based API. Many studies use manifests to declare permissions and to create a sandboxed environment to contain untrusted applications (e.g., Janus [27], Android permissions [19], MUDs [14]). Conventional manifests often confine applications' binary access to system resources. For example, Android developers may declare the need for all-or-nothing access to users' contacts.

However, due to the vast number of data granularities and the nuanced contextual nature of privacy, this traditional attribute-based manifest representation is becoming increasingly insufficient. Researchers propose a few approaches to enhance the permission system. One relevant approach is through remote code execution [11, 17]. For example, Cao et al. [11] proposed allowing app developers to author data-filtering programs, which are then executed by service providers through in-process sandboxing. While the remote code execution offers great flexibility for adding new data accesses, it is hard to enforce (i.e., analyze and control) the data transformation behaviors of these arbitrary programs. In contrast, OAuthHub only allows developers to specify data collection behaviors using a set of high-level operators with well-defined semantics. So OAuthHub can easily infer the program behaviors by analyzing the manifest and enabling a centralized runtime permission system.

Local Hub. Centralizing security and privacy management through a local hub is a widely adoptedapproach [39, 73, 74]. For instance, Capture [74] uses a centralized hub to manage and deploy IoT device firmware. The hub design of OAuthHub is particularly inspired by Peekaboo [39], sharing several key elements, such as operator-based manifest design and centralized end-user management features. However, OAuthHub introduces several key differences. Unlike Peekaboo, which operates as an always-on hub that only transmits data and does not receive callbacks from servers, OAuthHub is designed to run on intermittently available personal devices and supports both read and write access. Further, OAuthHub explores a new runtime permission mechanism for OAuth apps.

11 Discussion

Benefits of OAuthHub. Having a local hub mediating the OAuth-based sharing has three advantages over the conventional OAuth implementation. First, OAuthHub enables fine-grained data access, even if the service providers do not offer such. Second, users have better control and transparency about how OAuth apps access their data. Third, OAuthHub helps developers reflect when their apps need to access user data.

Alternatives. We also consider alternative solutions to using the user's device as a local data hub, such as cloud-based approaches and user-managed serverless runtimes. For example, server-side filtering in a trusted execution environment prevents third parties from accessing raw data, which offers better security but increases computational overhead. A user-owned serverless runtime improves availability but requires technical expertise for setup.

The data minimization enabled by OAuthHub is complementary to these cloud-based privacy solutions. OAuth has become popular due to its lightweight design. Most OAuth developers may not have the expertise to utilize advanced privacy-enhancing technologies such as Google Confidential Computing [58].

OAuthHub adoption. In high-stakes domains (e.g., applications handling sensitive personal data), developers have strong incentives to earn and maintain user trust, since overly aggressive permissions can trigger denials and damage an app's reputation [66].

OAuthHub has three key ideas: (1) a data hub that mediates OAuth-based data sharing, (2) using users' personal devices as the hub, and (3) operator-based data access. We expect that the biggest adoption barrier of OAuthHub is requiring users to install the hub runtime (e.g., a Chrome extension or a mobile app) on their devices if OAuthHub is not a built-in service.

A promising approach to address this challenge is establishing a serverless data hub operating on a personal cloud [42, 54]. However, this solution introduces a privacy tradeoff: the service provider (e.g., AWS) will know what OAuth services the user has interacted with. An alternative is self-hosting the data hub. Self-hosting is increasingly popular among privacy-conscious users (e.g., Vaultwarden [26]), as it avoids reliance on third-party cloud providers

and gives users full control over their data. It can also provide additional benefits such as always-available OAuth processing, reduced latency within their local network, and centralized management across multiple personal devices. At the same time, self-hosting requires technical expertise and ongoing maintenance, which may limit its accessibility to non-expert users.

Integration of OAuthHub. To integrate OAuthHub, developers add a new "Sign in with OAuthHub" option and register an API endpoint for receiving tokens and data from the local hub, with rate limiting and robust DDoS protection. The integration mainly relies on the three library APIs shown in Figure 2: (1) `generateAuthUrl` to initiate authorization with a fine-grained manifest and declared access pattern, (2) `exchangeToken` to receive the access token via the app's callback endpoint, and (3) `query` to retrieve or write data through OAuthHub instead of directly calling service-provider APIs. Developers also implement server-side logic to verify payloads from the local hub and include error handling for offline devices. Appendix D illustrates code changes for the Zoom scenario.

Decentralized web. Today, service providers like Google manage users' data on their servers. This paradigm minimizes the burden for users but introduces a few key privacy concerns. More recently, web pioneers are advocating the idea of Web5, where users' devices (nodes) store and process data locally, with the server only caching data for service availability when the local device is unavailable [40]. However, most users do not have the expertise to maintain the availability of their nodes, and most Web5 services are incompatible with existing services. OAuthHub represents a balanced approach to the Web5 vision, offering a practical middle ground where the local hub delivers a substantial level of privacy protection and user control while remaining feasible for implementation.

Cross-device integration. The current implementation of OAuthHub operates local runtimes independently on individual devices, typically a PC and a smartphone for most users. Users have to grant access to the same service separately across devices, the same as the current OAuth implementation. A potential advantage of the previously discussed personal cloud version of OAuthHub is its ability to unify management across multiple devices, where users only need to authorize each app once across devices.

Service reliability. Each device runs its own OAuthHub instance if the user logs in through OAuthHub on multiple devices. Losing a phone does not affect service on other devices. The insight behind OAuthHub is that most OAuth apps can wait to fetch information until users actually need to use the device, when the device becomes online. Even after a prolonged offline period, the device will fetch data when the user opens the service on that device. For certain special cases that involve time-sensitive scheduled tasks across devices, such as when a user schedules a Drive backup on their phone but later needs to access the file from their laptop while the phone is offline, OAuthHub does not support this scenario.

12 Limitations

OAuthHub is not designed to support **data-intensive OAuth access**. For instance, transferring multi-gigabyte files through the local data hub can be slow and is better handled via conventional OAuth protocols. Similarly, an app like draw.io, which may want to save drawing files to Google Drive after each minor edit, may lead to significant power consumption for the personal device. Fortunately, our empirical analysis of real-world OAuth applications has not revealed these data-intensive OAuth usages.

Data schema. Service providers often implement diverse data schemas, requiring manual efforts to define and unify them within OAuthHub. Despite these differences, our findings indicate that most providers share common semantic structures. However, achieving full interoperability might be a significant engineering challenge. Future work could explore leveraging generative AI techniques to automate schema unification [68].

Maintenance efforts. OAuthHub supports updates from third-party applications and service providers. To change the use case or requested data (e.g., adding new providers or supporting multiple languages), developers can submit a new manifest for new providers or update the existing manifest (e.g., adding keywords). Any manifest update must go through the authorization process again. Additionally, if OAuth endpoints change their data schema, OAuthHub needs to update the code that maps input data to the predefined schema. We expect OAuthHub to require only periodic updates rather than frequent fixes for breaking API changes, as major OAuth service providers maintain backward compatibility with long deprecation periods to minimize disruptions.

Measuring user preferences. We used surveys to understand how likely users will deny an OAuth request. However, these surveys may introduce biases, as previous research found that survey participants often align their answers with social expectations to some extent [5, 43].Despite these limitations, survey responses still provide meaningful insights into user expectations [59], particularly their desire for more fine-grained data access controls.

However, the expected behavior studied may differ from real-world use. In real-world scenarios, users may act differently, either due to a lack of attention to privacy controls or because they perceive the benefits of a service outweigh its potential risks. We plan to monitor the open-source adoption, and interview users in the wild to understand their perspectives in future work.

Sampling bias. The OAuth apps evaluated were primarily sourced from workplace-oriented marketplaces, which may be biased towards corporate environments. We plan to extend our evaluation to additional domains in future work (e.g., social networks).

13 Conclusion

This paper introduces OAuthHub, a development framework that uses personal devices as the intermediary controller for data sharing between cloud services. A key challenge is that these devices are not always online and have no static IP. To address this, OAuthHub leverages a new insight that most apps need data access only at install time, user-driven events, or scheduled times. OAuthHub requires developers to declare when they will request data explicitly, and mediates these data access using not-always-on local devices. The introduction of local data hubs enables a centralized runtime permission model for managing OAuth access across services. We developed OAuthHub as an open-source library for OAuth app developers to integrate. Our evaluation with real-world apps shows that OAuthHub requires minimal changes to the application code, and is easy for developers to use compared to the conventional OAuth APIs.

Acknowledgments

We thank the anonymous reviewers and the revision editor for their invaluable feedback and the participants for their kind involvement in our studies. This research is in part supported by the Society of Hellman fellowship and a gift from Solix Technologies.

References

[1] 2022. https://community.hubspot.com/t5/APIs-Integrations/Does-the-OAuth-refresh-token-expire/m-p/335543

[2] 2024. App manifest overview | Android Developers — developer.android.com. https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed 06-06-2024].

[3] 2025. Jenkins plugins index. https://plugins.jenkins.io/. (Accessed on 01/20/2025).

[4] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2019. Oauthlint: An empirical study on oauth bugs in android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.

[5] Ashwaq Alsoubai, Reza Ghaiumy Anaraky, Yao Li, Xinru Page, Bart Knijnenburg, and Pamela J Wisniewski. 2022. Permission vs. app limiters: profiling smartphone users to understand differing strategies for mobile privacy management. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–18.

[6] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. 2016. Permissions snapshots: Assessing users' adaptation to the Android runtime permission model. In *2016 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 1–6.

[7] Android. 2024. Android Keystore system. https://developer.android.com/privacy-and-security/keystore. (Accessed on 01/06/2024).

[8] AppAuth. 2021. AppAuth–Native App SDK for OAuth 2.0 and OpenID Connect implementing modern best practices. https://appauth.io/. (Accessed on 09/04/2025).

[9] David G Balash, Xiaoyuan Wu, Miles Grant, Irwin Reyes, and Adam J Aviv. 2022. Security and Privacy Perceptions of {Third-Party} Application Access for Google Accounts. In *31st USENIX Security Symposium (USENIX Security 22)*. 3397–3414.

[10] Joshua Bloch. 2006. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 506–507.

[11] Leo Cao, Luoxi Meng, Deian Stefan, and Earlence Fernandes. 2024. Stateful least privilege authorization for the cloud. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3477–3494.

[12] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 892–903.

[13] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2022. Practical data access minimization in {Trigger-Action} platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2929–2945.

[14] Cisco. 2021. What is MUD? - Manufacturer Usage Description - Document - Cisco DevNet. https://developer.cisco.com/docs/mud/#!what-is-mud. (Accessed on 02/05/2021).

[15] Citizens Information. 2023. How to access your personal data under the GDPR. https://www.citizensinformation.ie/en/government-in-ireland/data-protection/rights-under-general-data-protection-regulation/. (Accessed on 11/30/2023).

[16] Kevin Corre, Olivier Barais, Gerson Sunyé, Vincent Frey, and Jean-Michel Crom. 2017. Why can't users choose their identity providers on the web? *Proceedings on Privacy Enhancing Technologies* 2017, 3 (2017), 72–86.

[17] Yves-Alexandre De Montjoye, Erez Shmueli, Samuel S Wang, and Alex Sandy Pentland. 2014. openpds: Protecting the privacy of metadata through safeanswers. *PloS one* 9, 7 (2014), e98790.

[18] W. Denniss, Google, J. Bradley, and Ping Identity. 2017. *OAuth 2.0 for Native Apps*. RFC 8252. RFC Editor. https://datatracker.ietf.org/doc/html/rfc8252

[19] Android Developer. 2020. Network security configuration | Android Developers. https://developer.android.com/training/articles/security-config. (Accessed on 09/07/2020).

[20] Yana Dimova, Tom Van Goethem, and Wouter Joosen. 2023. Everybody's Looking for SSOmething: A large-scale evaluation on the privacy of OAuth authentication on the web. *Proceedings on Privacy Enhancing Technologies* (2023).

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. 627–638.

[22] Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. The effectiveness of application permissions. In *2nd USENIX Conference on Web Application Development (WebApps 11)*.

[23] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action IoT platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*.

[24] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1204–1215.

[25] Marios Fokaefs, Rimon Mikhaiel, Nikolaos Tsantalis, Eleni Stroulia, and Alex Lau. 2011. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*. IEEE, 49–56.

[26] Daniel García. 2022. Unofficial Bitwarden compatible server written in Rust, formerly known as bitwarden_rs. https://github.com/dani-garcia/vaultwarden. [Accessed 02-15-2026].

[27] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, Vol. 6. 1–1.

[28] Google. 2023. Choose Google Calendar API scopes. https://developers.google.com/calendar/api/auth?hl=en. (Accessed on 14/09/2023).

[29] Google. 2023. Using OAuth 2.0 to Access Google APIs. https://developers.google.com/identity/protocols/oauth2. (Accessed on 11/30/2023).

[30] Google. 2024. Using OAuth 2.0 to Access Google APIs. https://developers.google.com/identity/protocols/oauth2#expiration. (Accessed on 06/04/2024).

[31] Google. 2025. Permission Denials | App quality | Android Developers. https://developer.android.com/topic/performance/vitals/permissions [Online; accessed 2025-04-10].

[32] Eran Hammer-Lahav. 2010. *The OAuth 1.0 Protocol*. RFC 5849. RFC Editor. https://www.rfc-editor.org/rfc/rfc5849

[33] Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor. https://www.rfc-editor.org/rfc/rfc6749

[34] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage publications Sage CA: Los Angeles, CA, 904–908.

[35] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*. 1155–1164.

[36] Vincent C Hu, D Richard Kuhn, David F Ferraiolo, and Jeffrey Voas. 2015. Attribute-based access control. *Computer* 48, 2 (2015), 85–88.

[37] Shubham Jain, Janne Lindqvist, et al. 2014. Should I protect you? Understanding developers' behavior to privacy-preserving APIs. In *Workshop on Usable Security (USEC'14)*.

[38] Louis Jannett, Maximilian Westers, Tobias Wich, Christian Mainka, Andreas Mayer, and Vladislav Mladenov. 2024. Sok: Sso-monitor-the current state and future research directions in single sign-on security measurements. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 173–192.

[39] Haojian Jin, Gram Liu, David Hwang, Swarun Kumar, Yuvraj Agarwal, and Jason I Hong. 2022. Peekaboo: A hub-based approach to enable transparency in data processing within smart homes. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 303–320.

[40] Angie Jones. 2022. What is Web5? https://dev.to/tbdevs/what-is-web5-233o. (Accessed on 11/01/2024).

[41] Rishabh Khandelwal, Thomas Linden, Hamza Harkous, and Kassem Fawaz. 2021. {PriSEC}: a privacy settings enforcement controller. In *30th USENIX Security Symposium (USENIX Security 21)*. 465–482.

[42] Dohyun Kim, Prasoon Patidar, Han Zhang, Abhijith Anilkumar, and Yuvraj Agarwal. 2022. Self-serviced iot: Practical and private iot computation offloading with full user control. *arXiv preprint arXiv:2205.04405* (2022).

[43] Spyros Kokolakis. 2017. Privacy attitudes and privacy behaviour: A review of current research on the privacy paradox phenomenon. *Computers & security* 64 (2017), 122–134.

[44] Maximilian Kroschewski and Anja Lehmann. 2023. Save the implicit flow? Enabling privacy-preserving RP authentication in OpenID connect. *Proceedings on Privacy Enhancing Technologies* (2023).

[45] Barry Leiba. 2012. Oauth web authorization protocol. *IEEE Internet Computing* 16, 1 (2012), 74–77.

[46] Wanpeng Li and Chris J Mitchell. 2020. User access privacy in OAuth 2.0 and OpenID connect. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 664–6732.

[47] Wanpeng Li, Chris J Mitchell, and Thomas Chen. 2019. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In *Proceedings of the 5th ACM workshop on security standardisation research workshop*. 35–44.

[48] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 76 (Sept. 2017), 26 pages. https://doi.org/10.1145/3130941

[49] T. Lodderstedt, J. Richer, and B. Campbell. 2023. *RFC 9396 - OAuth 2.0 Rich Authorization Requests*. RFC 5849. RFC Editor. https://datatracker.ietf.org/doc/html/rfc9396

[50] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. 2017. User interactions and permission use on android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing*

Systems. 362–373.
[51] Srivathsan G Morkonda, Sonia Chiasson, and Paul C van Oorschot. 2021. Empirical analysis and privacy implications in OAuth-based single sign-on systems. In *Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society*. 195–208.
[52] Ed. N. Sakimura, Nomura Research Institute, J. Bradley, Ping Identity, N. Agarwal, and Google. 2015. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. RFC Editor. https://www.rfc-editor.org/rfc/rfc7636
[53] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*. 328–332.
[54] Shoumik Palkar and Matei Zaharia. 2017. Diy hosting for online privacy. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 1–7.
[55] Anthony Peruma, Jeffrey Palmerino, and Daniel E Krutz. 2018. Investigating user perception and comprehension of android permission models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 56–66.
[56] Jan Philipps and Bernhard Rumpe. 1999. Refinement of pipe-and-filter architectures. In *FM'99—Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*. Springer, 96–115.
[57] Pi-hole. 2025. Pi-hole – Network-wide Ad Blocking. https://pi-hole.net/ [Online; accessed 2025-11-30].
[58] Nelly Porter and Sam Lugani. 2020. Introducing Google Cloud Confidential Computing with Confidential VMs. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms. [Accessed 02-15-2026].
[59] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. 2019. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1326–1343.
[60] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. 2012. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 224–238.
[61] Yassine Sadqi, Yousra Belfaik, and Said Safi. 2020. Web oauth-based sso systems security. In *Proceedings of the 3rd International Conference on Networking, Information Systems & Security*. 1–7.
[62] Bingyu Shen, Lili Wei, Chengcheng Xiang, Yudong Wu, Mingyao Shen, Yuanyuan Zhou, and Xinxin Jin. 2021. Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model. In *30th USENIX Security Symposium (USENIX Security 21)*. 751–768.
[63] Yizhe Shi, Guangliang Yang, Zhemin Yang, Yifan Yang, Min Yang, Kangwei Zhong, and Xiaohan Zhang. 2025. The Skeleton Keys: A Large Scale Analysis of Credential Leakage in Mini-apps. (2025).
[64] Jaimandeep Singh and Naveen Kumar Chaudhary. 2022. OAuth 2.0: Architectural design augmentation for mitigation of common security vulnerabilities. *Journal of Information Security and Applications* 65 (2022), 103091.
[65] San-Tsai Sun and Konstantin Beznosov. 2012. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 378–390.
[66] Mohammad Tahaei, Ruba Abu-Salma, and Awais Rashid. 2023. Stuck in the permissions with you: Developer & end-user perspectives on app permissions & their privacy ramifications. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–24.
[67] Mohammad Tahaei and Kami Vaniea. 2022. Recruiting participants with programming skills: A comparison of four crowdsourcing platforms and a CS student mailing list. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–15.
[68] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938* (2023).
[69] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. {SmartAuth}:{User-Centered} authorization for the internet of things. In *26th USENIX Security Symposium (USENIX Security 17)*. 361–378.
[70] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 31–40.
[71] Chamila Wijayarathna and Nalin AG Arachchilage. 2019. An empirical usability analysis of the google authentication api. In *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*. 268–274.
[72] Feng Yang and Sathiamoorthy Manoharan. 2013. A security analysis of the OAuth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 271–276.
[73] Igor Zavalyshyn, Axel Legay, Annanda Rath, and Etienne Rivière. 2022. SoK: Privacy-enhancing Smart Home Hubs. *Proceedings on Privacy Enhancing Technologies* 4 (2022), 24–43.
[74] Han Zhang, Abhijith Anilkumar, Matt Fredrikson, and Yuvraj Agarwal. 2021. Capture: Centralized library management for heterogeneous {IoT} devices. In *30th USENIX Security Symposium (USENIX Security 21)*. 4187–4204.
[75] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2411–2425.
[76] Noé Zufferey, Kavous Salehzadeh Niksirat, Mathias Humbert, and Kévin Huguenin. 2023. "Revoked just now!" Users' Behaviors toward Fitness-Data Sharing with Third-Party Applications. *Proceedings on Privacy Enhancing Technologies* 1 (2023), 47–67.

A Analysis of Data Access in OAuth Apps

We analyzed 11 randomly selected OAuth apps using browser inspection and dynamic instrumentation to validate our assumptions about their data access (see Table 6).

Method. We randomly selected 11 apps from the set analyzed in Section 3 using a computer program and manually installed each of them. These apps include both open-source and closed-source applications. For every app, we created dedicated test accounts, completed the OAuth authorization process, and systematically interacted with the application's core features to trigger representative OAuth data access.

Because OAuth data transfers between an app's back-end server and Google's resource server are not observable from the client side, we employed three complementary analysis techniques based on each app's platform and source availability. For open-source applications (Zotero, Canvas, and WordPress), we conducted direct source code analysis, examining the server-side code that interfaces with Google's resource APIs to identify the exact endpoints invoked and the data fields accessed. For closed-source web-based applications, we used Chrome Developer Tools to inspect network traffic after OAuth authorization. We monitored fetch requests and corresponding responses to identify the invoked API endpoints, examine response payloads, and determine which user attributes were delivered to the client. For mobile applications, we instrumented the apps at runtime using Frida to intercept network API requests and relevant client-side data-processing routines, allowing us to determine which user attributes were accessed and how they were utilized locally.

Results. Our findings confirm our analysis results in Section 3. Source code analysis of three open-source applications directly confirms that actual API usage aligns with our least-privilege characterization. For example, Zotero's Google Docs integration asks for full CRUD access to all Drive files, even though it only invokes endpoints to read and edit citations within specific documents. Similarly, Canvas requests full Google Drive permissions, but only performs read operations for file selection during assignment submission.

For closed-source apps, since we cannot access their server, we can only observe partial evidence of OAuth data access from the client side. This partial evidence also confirms our assumptions. In Doodle, the website retrieves calendar event objects via its API, whose structure closely mirrors raw Google Calendar API responses (including titles, descriptions, and timestamps), then processes them locally to display only time-availability blocks. This suggests that

Example App	App Functionality	Current Privilege	Least Privilege
Uber Travel	Scan email for flights to book ride-share trips on arrival	All emails; Email settings (filters, labels)	Flight itinerary details
WellyBox	Scan email for financial receipts for reimbursement filing	All emails; Email settings (filters, labels)	Receipt attachments in emails
TripIt	Organize travel itinerary and information	All emails; Email settings (filters, labels)	Transportation and accommodation information in emails
Zoom	View upcoming video conferencing meetings; Create video conferencing meetings; See contacts on Zoom.	View, edit, and delete events on all calendars; See, edit, and delete all accessible calendars; See, edit, and permanently delete contacts	Title, time, and video conferencing links of events; Create events, not delete; See or add contacts, not modify
Doodle	Scheduling meetings with video conferencing and support for event invites on Google Calendar.	See and download any calendar you can access using your Google Calendar; View and edit events on all your calendars	Time availability blocks; Create events, not delete or edit
Notability	Note taking app with backups on Google Drive	See, edit, create, and delete all of your Google Drive files	See, edit, create, and delete Notability files on Google Drive
Slack Google Drive Add-on	Slack integration for Google Drive actions like file sharing and creation	See, edit, create, and delete all of your Google Drive files	Create files or modify file sharing permissions
Zotero GoogleDocs Integration	Edit citations in a specific file in Google Docs	See, edit, create, and delete all of your Google Drive files	Edit files; See the citations used in the files
Canvas	Attach Google Docs in assignment submissions	See, edit, create, and delete all of your Google Drive files	See files in Google Drive
Wordpress	Access to Google Photos to attach to posts	View your Google Photos library	View specific photos or album
Lenovo Smart Frame	Access to Google Photos to display on the frame	View your Google Photos library	View specific photos or album

Table 6: We analyzed 62 OAuth apps from Google Marketplace. Most apps request more privileges than they need to perform their tasks, and the data minimization needs are diverse and app-dependent.

Google's current OAuth scopes lack sufficient data granularity, requiring apps to fetch raw data and filter it on the client side. For WellyBox, the client app receives pre-processed data from its backend containing only receipt-relevant information, indicating server-side filtering prior to transmission.

B Manifest in Our Case Study

Zoom accesses Google Calendar.

```
TITLE: Zoom
DESCRIPTION: Get all upcoming Zoom meetings
PIPELINE: PullCalendarEvents -> SelectEvents -> FilterTime
                             -> FilterZoom -> PostToZoom

PullCalendarEvents(type: "Pull", resourceType: "google_calendar",
                   query: "{ events(calendarId) {...EventDetails} }")
SelectEvents(type: "Select", field: "events")
FilterTime(type: "Filter", operation: ">",
           field: "start.dateTime", targetValue: NOW)
FilterZoom(type: "Filter", operation: "match",
           field: ["location", "description"],
           pattern: "zoom\.us", requirement: "any")
PostToZoom(type: "Post", destination: "www.zoom.us")
```

Notability write Google Drive

```
TITLE: Notability
DESCRIPTION: Backup notes to Google Drive
PIPELINE: ReceiveRequest -> FilterPath -> Upload

ReceiveRequest(type: "Receive", source: "www.notability.com")
FilterPath(type: "Filter", operation: "match", field: ["parents"],
           targetValue: "folderId")
Upload(type: "Write", action: "create", resourceType: "google_drive")
```

C Selected Applications For Analysis

The applications below are used in the OAuth overaccess study. These applications were sampled from authors' Google's management interface and Google Workplace Marketplace's recommendation page, covering common scenarios of OAuth access to various Google services.

C.1 Google OAuth Apps

(1) Viber (2) WhatsApp Messenger (3) Figma (4) Internet Archive (5) Quora (6) Typing.io (7) penpot (8) Uber Travel (9) WellyBox (10) TripIt (11) Zoom (12) Doodle (13) Notability (14) Slack Google Drive Add-on (15) Zotero Google Docs Integration (16) Canvas (17) Wordpress (18) Draw.io (19) Hypatia Create (20) Pokemon Go (21) EasyBib Bibliography Creator (22) Lumio (23) Flubaroo (24) Form Ranger (25) Colaboratory (26) Pear Deck (27) CloudConvert (28) MathType (29) Lucidchart (30) DocHub (31) ZIP Extractor (32) Autocrat (33) SketchUp for Schools (34) CoRubrics (35) OrangeSlice (36) Auto-LaTeX Equations (37) Grade Reports (38) Highlight Tool (39) CLOZEit (40) Easy Accents (41) Slides Translator (42) Automagical Forms (43) Kaizena (44) Lenovo Smart Frame (45) Spark Email Client (46) Form Mule (47) Doctopus (48) Quilgo (49) Kami (50) Nearpod (51) formLimiter (52) Fluency Tutor for Google (53) ClassReporter (54) Slides Randomizer (55) Classroom Attendance Tracker (56) Timer for Google Forms (57) Copy Down (58) AutoProctor (59) Doc Appender (60) Classroom Manager (61) Form Notifications (62) Form Presenter+Timer

C.2 Trello OAuth Apps

(1) Analytics & Reports by Screenful (2) Card Priority Badge (3) API Developer ID Helper (4) Binotel (5) Hipporello Apps (6) ScaledbyScreenful (7) Habit Tracker by UpgradeYourBrain (8) BigPicture (9) ShowAttachments (10) SmartDeadlines (11) WhatsApp Notifications (12) RewriteText (13) SumUp (14) Notes & Docs (15) Statuses Workflow (16) TeamsHub (17) Durations (18) Google Sheets + Trello Two-Way Sync (19) ConfluenceTrello (20) Office File Viewer (21) Wordefy (22) Miro (23) CustomFields (24) Localize Borads (25) 3DViewer (26) Notion (27) ExportsbyBlueCat (28) BulkActions (29) Contalist (30) Hourly (31) Timeline, Calendar, Time Tracking by Planyway (32) OKRStudio (33) Litmus (34) OfficeHub (35) Crmble (36) Costello (37) Jira + Trello 2-Way Sync (38) Attachments Archive (39) BoardExport (40) Tables & Spreadsheet (41) PipeDrive (42) join.me (43) GitLab (44) Countdown (45) Amazing Fields (46) TrelloTimeTracking (47) FormsbyBlueCat (48) EmailforTrello (49) TimeFlowTracker (50) Notejoy (51) Onedrive (52) Marker.io

C.3 Slack OAuth Apps

(1) BambooHR (2) Stack Overflow for Teams (3) G2 for Slack (4) Trello (5) Hootsuite (6) ZoomInfo (7) HubSpot (8) Miro (9) Live Chat (10) Qualtrics (11) Zapier (12) TheNewYorkTimes (13) JiraCloud (14) Workday (15) Polly (16) Freshdesk (17) EddyTravels (18) Evergreen (19) Lattice (20) AWSChatbot (21) Bitbucket Cloud (22) Box (23) OneDrive and SharePoint (24) GitLab (25) Typeform (26) Front (27) Donut (28) monday.com (29) Asana (30) Dropbox (31) Drift (32) ActiveCampaign Bot (33) SurveySparrow (34) ConfluenceCloud (35) Airtable (36) PivotalTracker (37) Sentry (38) Intercom (39) Zoom (40) Notion (41) ClickUp (42) Frame (43) Todoist (44) GitHub (45) Coffice (46) PagerDuty (47) Favro (48) Harvest (49) SimplePoll (50) SalesforceforSlack (51) Linear (52) Crashlytics (53) latexbot

C.4 Github OAuth Apps

(1) Jenkins (2) Slack (3) Choreo.dev (4) Swarmia (5) DeepSource (6) CodeReviewBot.AI (7) Backup Github (Backrightup) (8) giscus (9) Pullflow (10) Sourcery (11) SonarCloud (12) CR.GPT (13) Ally (14) Stacked Pull Requests (15) codebeat (16) GraphQL Hive (17) Apollo Studio (18) Report.Ci (19) ReadME API (20) Dpulls (21) autofix.ci (22) SyncDepot (23) Honeycomb.io (24) Test Check Publisher (25) PeerGraph (26) Google Cloud Build (27) Bytesafe Integration (28) Faable Deploy (29) gitpod.io (30) XRPL Bot (31) DevHub (32) Octobox (33) GitGuardian (34) Auto Docs (35) AppSweep (36) QuickFastChange (37) Localazy (38) Sadaiv CI (39) Cloudback: GitHub Backup & Restore (40) Back Git Up (41) Telebirr-cloud-bot (42) webapp.io (43) harpoon.io (44) Devbox Cloud (45) Depfu (46) Phylum (47) PDDON (48) GitBook (49) StarChart Labs | Chronicler (50) Payvost (51) rootly

D Annotated Code Changes for OAuthHub Integration

```
// src/pages/index.tsx
  ......
  // Include the manifest pipeline
+ const manifest = "<Zoom Manifest Text>`;

  const getAuthUrl = async () => {
-   const res = await fetch("/api/auth");
-   const { authUri } = await res.json() as { authUri
      ?: string };

    // Authorization
+   const OAuthHubClient = require("~/lib/oauthub-
     client");
+   return OAuthHubClient.generateAuthUrl({
+       provider: "google_calendar",
+       manifest: manifest,
+       redirect: "zoom.us/callback",
+       accessType: "user_driven",
+   }) as Promise<string>;
  };

  const requestData = async () => {
-   await fetch("/api/data", {method:"POST"});
+   const OAuthHubClient = require("oauthub-client");
+   const authRes = await fetch("/api/oauthub/auth");
+   const { token } = await authRes.json() as { token:
      string | null };
+   const result = await OAuthHubClient.query({ token,
      manifest: manifest });
+   await fetch("/api/oauthub/auth", {
+     method: "POST",
+     headers: { "Content-Type": "application/json" },
+     body: JSON.stringify({token: result.token }),
+   });
  };
  ......

  // UI Changes
+ <OAuthHubIcon />
+ Sign in with OAuthHub
  ......
```

```
// src/pages/api/callback.tsx
  ......
  // Extract hub identity params
+ let publicKey = JSON.parse(params.get("
    oauthub_public_key");

  // Exchange auth code for access token
+ const OAuthHubClient = require("~/lib/oauthub-
    client");
+ const { access_token } = await OAuthHubClient.
    exchangeToken({ code, state });

- await fetch("/api/auth", {
-   body: JSON.stringify({ code }),

+ await fetch("/api/oauthub/auth", {
+   method: "POST",
+   headers: { "Content-Type": "application/json" },
+   body: JSON.stringify({ publicKey, userSub, token:
     access_token }),
  });
  ......
```

```
// src/pages/api/data.ts
  ......
+ const OAuthHubClient = require("oauthub-client");

  export default async function handler(req, res) {
-   const tokens = await db.getData("/google_tokens");
-   oauth2Client.setCredentials(tokens);
-   const calendar = google.calendar({ version: "v3",
     auth: oauth2Client});
-   const { data } = await calendar.events.list({
-     calendarId: "primary",
-     timeMin: new Date().toISOString(),
-     maxResults: 10,
-     singleEvents: true,
-     orderBy: "startTime",
-   });
-   const events = (data.items.map((event) => ({
-     summary: event.summary",
-     start: event.start.dateTime,
-        end: event.end.dateTime,
-   }));
```

```
21
22 +   const signature = req.headers["x-oauthub-signature
       "];
23 +   const body = req.body;
24 +   const rawBody = JSON.stringify(body);
25
26     // Verify payload signature
27 +   let publicKeyJwk = await db.getData("/oauthub").
       publicKey;
28 +   if (signature && publicKeyJwk) {
29 +     const valid = await OAuthHubClient.verifyJWT({
       jwt: signature, publicKeyJwk, body: rawBody });
30 +     if (!valid) return res.status(401).json({ error
       : "Invalid signature" });
31 +   }
32 +   const events = body.data;
33    ......
```

E Survey Questions for the User Study

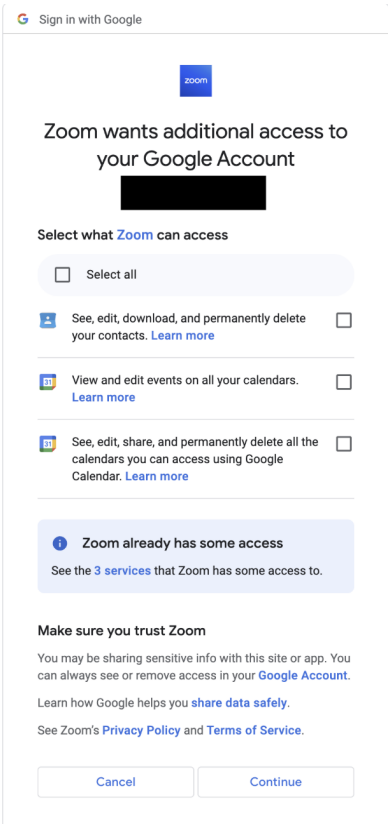
E.1 Zoom

Zoom is a video conferencing app that allows users to connect for meetings, webinars, and virtual events. It supports high-quality video and audio, screen sharing, and collaboration tools like chat and file sharing. Zoom is widely used for remote work, education, and social interactions, providing an easy-to-use platform for connecting with others online.

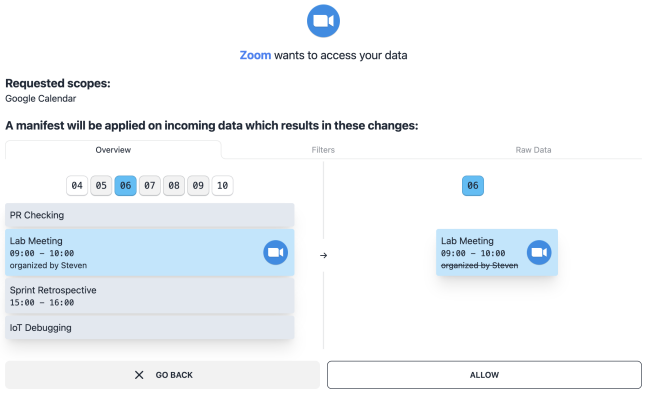
Zoom can be integrated with Google Calendar so that users can schedule and join Zoom meetings directly from the calendar.

[Displayed one of the following authorization interface conditions]

Condition A. When linking with Google Calendar, you'll be presented with an authorization page as shown below.



Condition B. When linking with Google Calendar through OAuthHub, you'll see an interactive authorization page below. If you grant the permission, OAuthHub will processes your personal data locally on your device and sends only the processed results to the app. The "Overview" provides a general illustration of the process. "Filters" gives a detailed step-by-step description. "Raw data" shows the data before and after processing.



[Present the following questions after the interface exposure]

1. If you grant Zoom the requested permission, what data will Zoom be able to access?
   - Events you explicitly share with others
   - All of your calendar events
   - All Zoom meeting events
   - Zoom meeting topics, time, and links
2. How likely would you grant the app the requested permission? (Choose from a 5-point Likert scale)
   - Extremely unlikely
   - Somewhat unlikely
   - Neither likely nor unlikely
   - Somewhat likely
   - Extremely likely
3. Please share your thoughts on why you'd choose to grant or not grant Zoom these permissions. Do you have any unaddressed concerns? (Open-ended question)

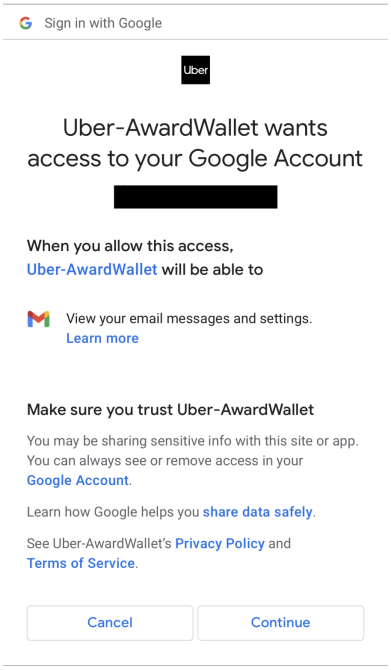
E.2 Uber

Uber is a ride-hailing platform that connects passengers with drivers for transportation services. Uber links with Gmail to automatically import and organize your travel plans, such as flight, hotel, and restaurant reservations, directly into the Uber app. By accessing your Gmail account, Uber can scan for travel-related emails and create a unified itinerary within the app.

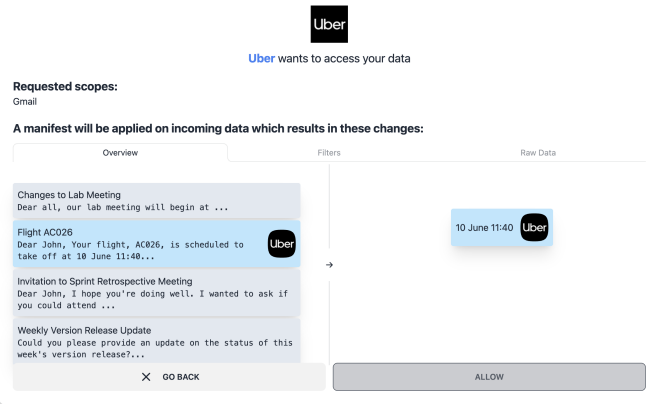
[Displayed one of the following authorization interface conditions]

Condition A. When linking with Gmail, you'll be presented with an authorization interface as shown below.

Condition B. When linking with Google Drive through OAuthHub, you'll see an interactive authorization page below. If you grant the permission, OAuthHub will process your personal data locally on your device and send only the processed results to the app. The



"Overview" provides a general illustration of the process. "Filters" gives a detailed step-by-step description. "Raw data" shows the processed data.



[Present the following questions after the interface exposure]

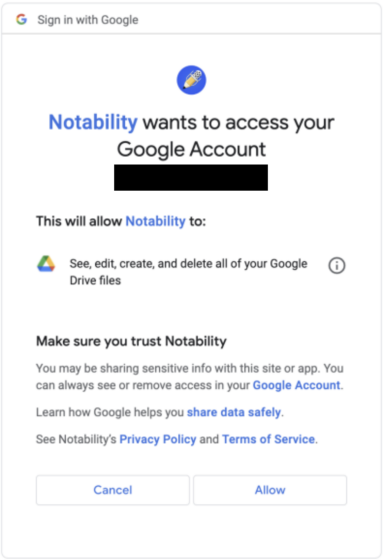
1. If you grant Uber the requested permission, what data will Uber be able to access?
   - Only emails with travel information
   - All of your emails
   - Only flight information extracted from emails
   - All emails received from Uber
2. How likely would you grant the app the requested permission? (Choose from a 5-point Likert scale)
   - Extremely unlikely
   - Somewhat unlikely
   - Neither likely nor unlikely
   - Somewhat likely
   - Extremely likely
3. Why would you choose to grant or not grant these permissions to Uber? Do you have any unaddressed concerns? (Open-ended question)

E.3 Notability

Notability is a note-taking app designed for iOS and macOS that combines handwriting, typing, and audio recording into a single platform. The app supports multimedia, allowing users to add images, web clips, and more to their notes. Notability links with Google Drive to provide users with additional storage and backup options for their notes.

[Displayed one of the following authorization interface conditions]

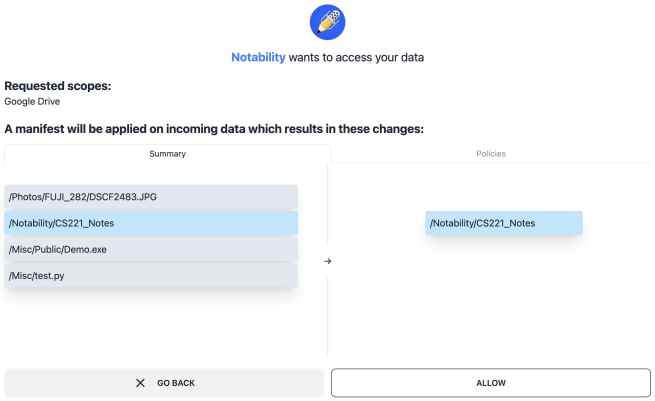
Condition A. When linking with Google Drive, you'll be presented with an authorization page as shown below.



Condition B. When linking with Google Drive through OAuthHub, you'll see an interactive authorization page below. If you grant the permission, OAuthHub will processes your personal data locally on your device and sends only the processed results to the app. The "Overview" provides a general illustration of the process. "Filters" gives a detailed step-by-step description. "Raw data" shows the processed data.

[Present the following questions after the interface exposure]

1. If you grant Notability the requested permission, what data will Notability be able to access?
   - Files created by Notability
   - All Images and documents
   - Files under the Notability folder
   - All Google Drive files
2. How likely would you grant the app the requested permission? (Choose from a 5-point Likert scale)
   - Extremely unlikely
   - Somewhat unlikely
   - Neither likely nor unlikely
   - Somewhat likely
   - Extremely likely



3. Why would you choose to grant or not grant these permissions to Notability? Do you have any unaddressed concerns? (Open-ended question)

E.4 Demographics and Background

1. What is your gender?
   - Male
   - Female
   - Non-binary
   - Prefer to self-describe: ____
   - Prefer not to answer
2. What is your age?
   - 18–24
   - 25–34
   - 35–44
   - 45–54
   - 55–64
   - 65+
3. What is the highest level of education you have completed?
   - High school or below
   - Some college
   - Associate degree
   - Bachelor degree
   - Master degree
   - Doctoral degree
   - Prefer not to answer
4. Please select the statement that best describes your comfort level with computing technology.
   - I can build my own computers, run my own servers, code my own apps, etc.
   - I know my way around computers and mobile/IoT devices pretty well; I am the person who helps friends and family with technical problems.
   - I know how to use computers and mobile/IoT devices to perform my job and life responsibilities; I often need technical help from others.
   - Technology usually scares me. I only use it when I have to.
5. Indicate the extent to which you agree or disagree with the following statements. *[For each statement below, participants were asked to choose from a 5-point likert scale "Strongly disagree" "Somewhat disagree" "Neither agree or disagree" "Somewhat agree" or "Strongly agree." Statements were presented in randomized order.]*
   - In general, I believe privacy is important
   - In today's world, privacy does not exist anymore.
   - I'm concerned that "smart" devices collect too much personal information.
   - I'm confident in my ability to control how much personal information I share online and through my phone.

F Example OAuth Screenshots

See Figure 10.

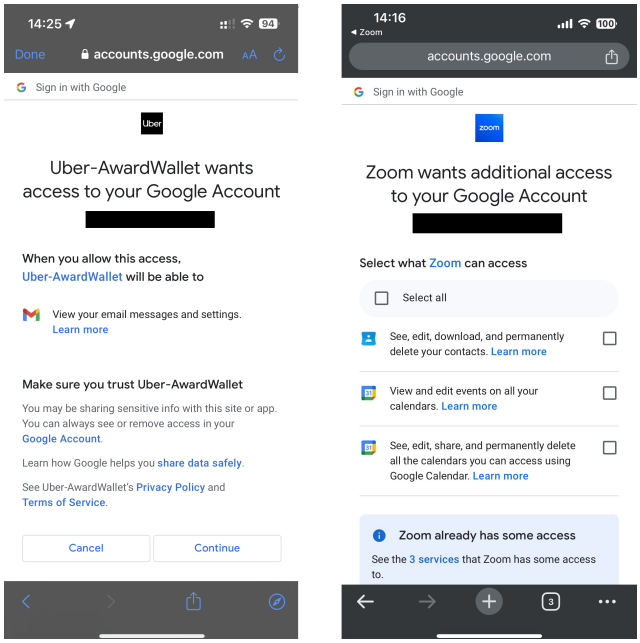


Figure 10: OAuth applications often request more data access than they need. Uber AwardWallet requires access to all users' emails, although it only needs to access flight confirmation emails to extract users' itineraries (Left). Zoom requests both read and write access to users' calendars and contacts (Right).

G Operator Design

See Table 7.

Category	Operator	Parameters	Description
Provider	Pull	resourceType	Get data from an OAuth resource
	Receive	source	Accept action requests from OAuth apps
Reduction	Filter	operation, requirement, field, targetValue	Filter data based on specific conditions on certain field(s)
	Select	field	Select specific field(s) in the data
	Extract	operation, field, pattern	Extract data in a defined pattern from a certain field
	Limit	count	Limit the number of results returned
Transform	Aggregate	operation, field, groupKey	Compute aggregate data for specific field
	Map	operation, field	Apply a mapping operation to a specific field
	Anonymize	dataType, field, method	Anonymize data in a specific field using a defined method
Network	Post	destination	Send the processed results to OAuth apps
	Write	action, resourceType	Perform actions on a specified OAuth resource
Utility	Inject	repeatNum, interval	Trigger the execution periodically
	Debug		Print output data for debugging

Table 7: The taxonomy of OAuthHub operators. Each operator corresponds to a "verb" statement relative to its semantics.