Stateful Least Privilege Authorization for the Cloud

Leo Cao, Luoxi Meng, Deian Stefan, and Earlence Fernandes, *UC San Diego*

https://www.usenix.org/conference/usenixsecurity24/presentation/cao-leo

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Stateful Least Privilege Authorization for the Cloud

Leo Cao* *UC San Diego*
Luoxi Meng* *UC San Diego*
Deian Stefan *UC San Diego*
Earlence Fernandes *UC San Diego*

Abstract

Architecting an authorization protocol that enforces least privilege in the cloud is challenging. For example, when Zoom integrates with Google Calendar, Zoom obtains a bearer token — a credential that grants broad access to user data on the server. Widely-used authorization protocols like OAuth create overprivileged credentials because they do not provide developers of client apps and servers the tools to request and enforce minimal access. In the status quo, these overprivileged credentials are vulnerable to abuse when stolen or leaked. We introduce an authorization framework that enables creating and using bearer tokens that are least privileged. Our core insight is that the client app developer always knows their minimum privilege requirements when requesting access to user resources on a server. Our framework allows client app developers to write small programs in WebAssembly that customize and attenuate the privilege of OAuth-like bearer tokens. The server executes these programs to enforce that requests are least privileged. Building on this primary mechanism, we introduce a new class of stateful least privilege policies — authorization rules that can depend on a log of actions a client has taken on a server. We instantiate our authorization model for the popular OAuth protocol. Using open source client apps, we show how they can reduce their privilege using a variety of stateful policies enabled by our work.

1 Introduction

Secure and controlled sharing of data is fundamental to distributed systems like the Web and Cloud. However, this happens through rudimentary bearer credentials that cannot precisely govern how much sharing occurs. For example, let's say that you have integrated Zoom with your Google Calendar so that it can schedule video calls. As part of this process, Zoom obtains a bearer token that allows it access to your events on Google Calendar. As a user, you want the assurance that Zoom exercises minimal rights on your calendar events. The Zoom app developer wants the least possible rights over user resources because this limits the potential for abuse if Zoom's bearer token is stolen or accidentally leaked. The Google Calendar server developers want a method to give out minimum rights on user data to third parties. These properties are difficult to achieve in a uniform and flexible way. We contribute an authorization framework for controlled data sharing in distributed systems enables all these properties.

There are two fundamental issues with current widely-used authorization protocols like OAuth 2.0 [27] that make it difficult to achieve least privilege: (1) server-defined permissions and (2) statelessness. The first problem is that the API server pre-defines a list of permissions that a client app can request. For example, Calendar defines permissions that control access to various API endpoints. The Zoom developers have to determine which of these permissions "best match" their functionality requirements. Fundamentally, this "best match" is not a perfect fit. The client app will always obtain permissions that it does not need for its functionality because a server developer cannot anticipate the minimal permissions that clients need. The Travis Continuous Integration (CI) developers discuss that they obtain a token that can read *and* write code in user's public and private repositories because GitHub did not provide a finer-grained permission suited to Travis CI's needs [16]. Ideally, the client app should be able to communicate its access requirements to the server, who then enforces exactly that privilege (only accessing events it created in the case of Zoom; and only reading code in public/private repos in the case of Travis CI).

The second fundamental issue with current cloud authorization is that it is stateless. The server evaluates the authorization policy using data present only in the current HTTP request. This is insufficient because it leads the client to having more privilege than it needs. Consider Zoom again — it should only be able to access the events it has created; it does not need the ability to access other kinds of events in the calendar. This authorization policy involves *state* — knowledge about the actions that the client app (or, its bearer token) has taken on the server (i.e., which events it has created?). With currently deployed authorization protocols, writing and en-

*Equal Contribution.

forcing an authorization policy involving state is not possible.

Existing work has started to recognize these fundamental problems with cloud authorization. However, they only partially target the first problem of server-defined permissions [11,32]. For example, the Macaroons framework allows clients to include restrictions on how a bearer token may be used on the server [11]. The recent Rich Authorization Requests RFC for OAuth provides a similar mechanism [32]. In both frameworks, the server developer has to pre-define a set of conditions that the client developer can use to attenuate their tokens — this does not resolve the fundamental problem that the server developer cannot anticipate all possible ways in which a client may use a token. Furthermore, these frameworks only support attenuating the token's authority using stateless restrictions.

We construct an authorization framework that addresses the above fundamental issues using two mechanisms: (1) flexible client-defined attenuation of a bearer token's privilege; (2) flexible client-defined statefulness. Concretely, a client developer writes two programs that the API server runs each time the client app instance sends a request. The first program, called the *attenuation policy*, represents the minimum rights that the client app needs. The program's inputs are the current HTTP request and a state table — a log of operations that the client app has performed on the API server thus far. The second program, called the *state updater*, is also client developer-supplied and it captures the semantics of statefulness. Its inputs are the current HTTP request and the previous state information. It outputs an updated state table for use in future executions. This enables flexibility — the client app developer is free to define and use state that makes the most sense for their functionality.

Revisiting our running example, the Zoom developers will write the attenuation policy that any request should only access Calendar events created by Zoom. They will define the state as a table containing the API endpoints that were called for a particular event object. The state updater program contains logic that will add the current API call that the client has made to the state, if that call is successful. When the Zoom app makes an API call to the Calendar server to access an event, the API server will first execute the attenuation policy to ensure that the request is allowed (i.e., accessing an event that Zoom has created). Then, it will execute the actual API and finally execute the state updater to generate the new state that will be used in future executions.

The client developer is motivated to use this system because they want to limit abuse if the app's bearer token is stolen, a fairly common occurrence [26,28,30,35,37,40,43]. For example, Travis CI suffered a breach that allowed attackers to use stolen OAuth tokens to read private GitHub repositories [26].

Our design requires that the attenuation policy and state updater programs compile to WebAssembly (Wasm) [24]. We make this decision for the following reasons. First, it provides flexibility because developers can express arbitrary policies and state definitions that perfectly fit the privilege requirements of the client app with any language that compiles to Wasm. Second, it is portable because Wasm has emerged as a popular bytecode format and execution environment. Third, it protects the server because of its strong memory safety guarantees that enable in-process sandboxing [25].

State is abstractly a log of operations that a client app has executed on the server. However, it has many concrete forms. The simplest state definition is the list of distinct server API calls that a client makes and the number of times those API calls are made. Client developers can define and use whatever notion of state they want using the state updater program. A design question is what party should be responsible for storing state? Server-side storage is problematic for two reasons: (1) The state data could grow to be arbitrarily long because it is client-defined and the server operates at large scale with millions of users; (2) It would provide an arbitrary storage primitive for anyone on the Internet. For example, an attacker could abuse the state mechanism to store illegal material on the server.

Instead, we require the client app to store the state for all server-owned objects it accesses and only send a subset of state relevant to its API requests. This distributes state table storage into each client and avoids the problems mentioned above. However, it introduces a secondary challenge of state integrity and freshness — the server should be guaranteed that it always receives the most recent state data without alteration on every API call. Recall that the attacker can steal tokens, including the state information that is simply another credential. We address this challenge using cryptographic integrity protection where the server maintains a hash or HMAC of each object.

We structure our framework as a library compatible with OAuth 2.0, a widely-used authorization protocol for the cloud. The server developer must add a few lines of code to their route handlers that call into our library. We open source our prototype implementation for Authlib [33], a popular Python library for building OAuth clients and servers. We conduct case studies using a range of open-source client apps to show how our authorization framework can reduce token privilege while retaining functionality. Based on these studies, we develop one definition of state and implement it as a state updater program that client developers can reuse (Section 2.3). Our implementation is available at `https://github.com/earlence-security/stateful-auth`

Contributions.

- We design a model for stateful least privilege authorization for the cloud that allows a client app developer to attenuate and customize the app's authority to user resources on a server. This mechanism is flexible and introduces the primitive of statefulness that enables a new class of least-privilege authorization policies.
- We integrate our model with OAuth 2.0, the most widely-used authorization protocol in the cloud and port existing

open-source apps to our framework with minimal code changes. This allows the apps to lower the privilege of their bearer tokens while retaining functionality.

- We conduct performance tests and find that overhead is modest. Applying stateful authorization only introduces a 4.3% increase in latency for requests involving a single server-owned data object, compared with vanilla OAuth.

2 Stateful Least Privilege Authorization Model

We construct an authorization model that solves the two fundamental problems with current approaches — server-defined permissions and statelessness. This allows client apps and web services to achieve least privilege authorization and minimizes the abuse that can occur if a bearer token is stolen. At a high level, the model allows client apps to tell servers about their access requirements using a pair of programs that attenuate a bearer token's privilege using state information — a log of API calls that the client app has made to an API service. To demonstrate the security value of this model, we discuss four case studies of apps that can use our authorization framework to minimize their privilege while retaining functionality.

Threat Model. Our goal is to empower a client app developer (and their security team) to express the minimum privilege they need and to enable the service developer to enforce these requirements. The developers of the client app and the API server are motivated to use our framework because they want to limit abuse if tokens are stolen. Therefore, we assume that the client app developer and API server are trusted. This assumption is the same as what we currently have with OAuth systems and more fundamentally, any permission-based system. The client app developers are trusted to provide a useful service and want to obtain least privilege tokens. A malicious client developer is outside our threat model.

An important nuance is that even though we trust the client app *developer*, the client app *itself* is vulnerable to security problems that can result in the bearer tokens being stolen or leaked [1, 14, 26, 28, 30, 35, 37, 40, 41, 43]. For example, we trust that the Travis CI developers want to use the bearer token for a user's GitHub repo in a least-privileged way, but the app instance can still be breached, resulting in token loss [26]. Broadly speaking, an attacker could steal tokens through infrastructure misconfigurations, phishing attacks and even through attacks on cloud-based credential management services that client apps might rely on. For example, an Okta breach allowed attackers to gain access to Cloudflare-owned tokens, ultimately allowing them to access internal systems [35, 43]. These tokens existed to enable various integrations such as Atlassian and BitBucket with the Cloudflare infrastructure. We generalize such attack incidents and assume that the attacker gains access to current tokens and also the OAuth `client_secret` value that will allow them to create new tokens at will.

Security Goals and Guarantees. We ensure that an authorization token is always bound to its client developer-supplied least privilege policy. This policy can be a combination of stateless and stateful checks. In other words, an attacker cannot use a token in ways that are inconsistent with the least privilege policy. We also ensure that any new tokens that an attacker might create (e.g., by stealing the OAuth `client_secret` value) is also bound by the developer-supplied policy. In summary, there is a window of vulnerability where bearer tokens are in the attacker's possession. Although the correct server behavior is to revoke the compromised tokens, there is always a delay in taking that action, during which time, damage is already done. By attenuating token privilege using stateless and stateful policies, our work limits the blast radius that results from stolen authorization tokens.

2.1 Fundamental Problems with Current Methods

Server-defined vs. Client-defined Permissions. Current mechanisms to control privilege in protocols like OAuth puts the onus on the API server developers. They have to create permissions (or "scopes" in OAuth parlance) that a client can request. The end-user is prompted to consent to these permission requests, that results in the client obtaining a bearer token — a credential that represents the client's privilege over user data on the server. The key issue is that because there are many possible ways in which a client can use an API, it is difficult for service developers to create a set of permissions that exactly match the client's requirements.

For example, fly.io is a cloud computing provider that hosts a server API to control virtual machines. The developers describe a problem in the design of their permission system — they cannot anticipate all client app uses of their APIs and are forced to make a static decision about what permissions to create [36]. Inevitably, this leads to some clients getting more permissions than they need. A similar example is Travis CI — the developers discuss that GitHub did not provide a read-only permission for private repos [16]. So they obtained a coarse-grained permission and had to wait until GitHub developed a completely different integration strategy that supported finer-grained permissions [18]. This cycle is tedious — the client app developers have to wait until the server developers create new permissions and then they have to refactor their codebase. In the meanwhile, the server developers have to support the older permissions and the newer ones as well, leading to long-term security issues.

Prior work has attempted to solve this problem by allowing the client to attenuate a token's power. For example, the Macaroons framework allows a client to add "caveats" that the server evaluates in the context of the current HTTP request [11]. For example, a caveat might enforce that the Zoom client is only editing events whose titles start with the substring "work-meeting." This is a stateless condition and the server checks it using data only present in the HTTP request (event title) and the object being manipulated (calendar event).

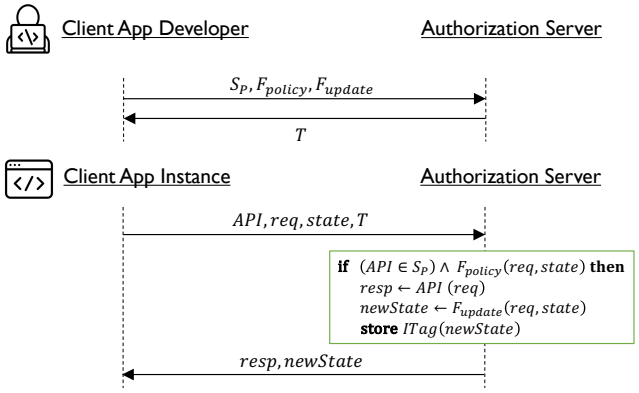
OAuth Rich Authorization Requests (RAR) specifies a JSON syntax that clients can use to limit the operations that can be performed using a token [32]. RAR is a simplified form of Macaroons. Both of these frameworks require the server developer to define a set of *stateless* attenuation conditions. We observe that this is the same as defining permissions and thus, it suffers from the same problem — the server developer cannot anticipate the varied uses of its APIs and thus, the caveat conditions (Macaroons) or JSON syntax (OAuth RAR) will always be incomplete.

Stateless vs. Stateful Authorization. The second problem is that no current authorization framework (vanilla OAuth, Macaroons, OAuth RAR) supports stateful attenuation of token privilege. Specifically, if a server has access to a log of API operations that a client has executed with a bearer token, then it can enforce a new class of *stateful* least privilege policies. For example, enforcing that Zoom may only access Calendar events it has created requires knowledge about what events it has created in the first place. There are several challenges in creating a mechanism that enables such policies. First, what data is recorded as state? Second, where is this state data stored and how is it recorded? Third, how is state made available to a client-defined authorization policy?

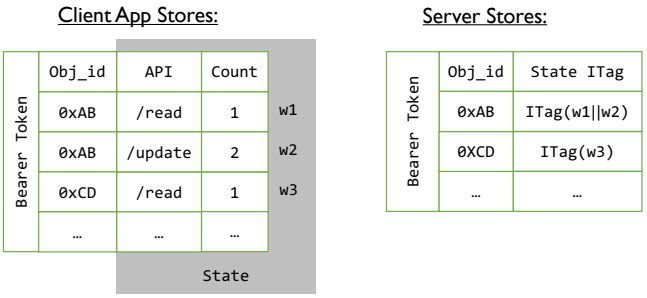
2.2 Stateful Model Overview

Hybrid Authorization. Fig. 1a shows the high-level workflow of our model. It combines server- and client-defined permissions and introduces statefulness. We trust the client app developers to: (1) Determine a set of server-defined permissions that "best matches" the app's functionality (S_P). Abstractly, we model the server as providing a set of API endpoints to the client. The server-defined permissions guard access to these endpoints and the client wants access to a subset of them. (2) The app developer writes an *attenuation policy* program (F_{policy}) that attenuates S_P to achieve a "perfect fit" in terms of the client's privilege requirements. The attenuation policy program's inputs are the client's request to the server, and state information — a log of API requests the client has made thus far. (3) The client app developer also writes a *state updater* program that defines state concretely and implements the update logic. The client developer transmits these three pieces of information (S_P, F_{policy}, F_{update}) to the server, and in return, it gets a token (T) that represents its privilege on the server. Overall, this is a hybrid system that combines the properties of server- and client-defined permissions.

When the client app instance makes a data access request, it specifies the API endpoint, arguments, the state information and the bearer token. The server first validates that the request complies with the server-defined permissions bound to the bearer token ($API \in S_P$) and then executes the attenuation policy ($F_{policy}(req, state)$). If these checks are true, the server will perform the API call and then execute the state updater ($F_{update}(req, state)$) to produce the new state information. It



(a) Stateful Authorization Workflow.



(b) Client app stores and supplies its own state. Note this figure visualizes one definition of state that we found to be useful for the case studies in Section 2.3. Our authorization framework supports any notion of state supplied by the client app developer. The server only stores an integrity tag (i.e., HMAC or hash) of the state data and uses it to validate the integrity and freshness of state supplied by a client app during an API request.

Figure 1: Stateful Authorization Model.

will store an integrity tag (i.e., HMAC or hash) of that new state and finally return the API call response and the updated state data to the client app. The client app then stores the state locally so that it is available for the next request.

This authorization model improves security in the following ways. First, by allowing the client app to attenuate its privilege, the bearer token has the privilege only needed for client functionality. Second, by including state information, we enable a broad class of authorization policies that help attenuate token privilege beyond what is possible using just stateless machinery. Third, the server-defined permissions serve as an upper-bound on the bearer token's authority. If one were to use a purely client-defined permission system, then the only thing that limits a token's authority is the client-supplied attenuation policy. If this program contains missing checks, then the token potentially has unbounded authority. In our model, if there is a missing check in the client-supplied policy, the token's authority is still upper-bounded by the server-defined permissions attached to it. That is, the security of the system defaults to the status quo. Finally, retaining server-defined permissions has the advantage of deployability

— the server doesn't have to replace its existing authorization system. Rather, our work adds to the existing security properties that the server provides.

We observe that the client developer is naturally trusted to write these attenuation policy and state updater programs because they are providing functionality to users and they want to minimize the potential for abuse if the bearer token is stolen. The server developer is also trusted but must run user-supplied programs on its infrastructure. An attacker could try to submit a malicious program that breaks out of the sandbox and takes over the server. The server must also allocate additional resources to run these programs at a large scale.

To tackle these two issues, our model requires the client app developers to compile policy programs to WebAssembly (Wasm). Wasm provides strong in-process memory isolation [25], so the server can run policy programs from multiple client apps in a single process without sacrificing security, but still gain performance benefits. An attacker has to compromise Wasm memory safety *and* the OS-enforced process isolation as well to take over the server.

Defining, Capturing and Storing State Information. A key contribution of our work is the notion of state — a log of API calls made by the client app. State information allows a client app to further attenuate the privilege of its bearer tokens. Rather than picking a specific definition of state, our framework allows client app developers to define their own concept of state and the rules governing its evolution. Fig. 1b shows one example definition of state — it has two pieces of information: (1) The called API endpoint; and (2) A counter that increments each time an API call is successful. This state gets updated every time the client app calls an API on the server. For each object that the client app accesses, the state updater program will create an entry in a table, keyed by that object's id. If an entry already exists, the state updater will increment the counter. This logic forms the evolution rules for the state under this definition.

Consider our running example of Zoom and Google Calendar. Whenever Zoom makes an API request to access an event, the attenuation policy can consult the state table to determine whether there is an entry in it that marked the particular event as being created by Zoom in the past. If it is the case, then the Calendar server executes the API call and runs the state updater to add a new row to indicate that Zoom called the read API endpoint successfully. Thus, we can enforce a stateful policy of type *access-only-created*. Section 2.3 discusses other types of policies enabled by this particular state definition.

A key design question is where should this state be stored? One option is to allow the state updater program to access server storage, such as a key-value database. This is less desirable because state is client-defined and the server is supporting millions of users. State data could grow to be arbitrarily large and would require the server to allocate large amounts of low-latency memory. This demand is exacerbated by the fact that executing the attenuation policy with the state information is now part of the authorization check that is on the critical path of every API request. Allocating large amounts of low-latency memory can get expensive [8]. Furthermore, this design provides a storage primitive to anyone who signs up as a client. A malicious client could abuse the state updater program to store arbitrary (and perhaps illegal) data on the server for free.

A better design is to distribute state storage to the clients. Each client app stores its own state and then supplies it to the server along with the bearer token. This trades off server costs but introduces challenges of protecting the state and ensuring its consistency in the event of failures.

To address the issue of integrity protection, our model stores an integrity tag corresponding to each object's state data on the server (see Fig. 1b). The tag will be either a hash or an HMAC. We will discuss an HMAC-based design in Section 3 and compare hash- and HMAC-based integrity protection in Section 3.1.3. An attacker with a stolen bearer token could attempt three attacks involving the state information: (1) sending a request without state data; (2) replaying outdated state; (3) forging state data. Section 3.1 discusses how our framework prevents all these attacks by design. We discuss how we handle benign failures and state consistency in Section 3.2.

2.3 Case Studies

We discuss how client-defined privilege attenuation and statefulness allow various real apps to lower their privilege without affecting functionality. We also introduce how statefulness enables new classes of authorization policies. Table 1 provides a summary.

Only allow accessing objects that the token bearer has created. The *access-only-created* stateful policy formalizes the "ownership" between the object and the client app that created it. It helps to restrict the privilege when the client app instance does not require accessing all the objects, but only the objects it "owns" to achieve its functionality.

We have already discussed the Zoom and Google Calendar example. In that vein, another example is a note-taking application, like Joplin [2]. This app synchronizes notes with various cloud services such as Dropbox or Amazon S3. We observe that Joplin only needs to create backups and update existing ones. Based on the *access-only-created* policy, we translate the least privilege requirement of note-taking applications into a stateful policy that "only allows note-taking applications to access the files/folders it created" and define the state as the list of distinct API calls made on an object.

In the above examples, given a compromised bearer token, the attacker can only access the objects the token has created. Therefore, the *access-only-created* policy minimizes the vulnerability before lost tokens get revoked by the services.

Allow the token bearer to read an object at most N times. The *read-at-most-N-times* policy provides a type of "forward secrecy" in that if the bearer token is stolen, past-read objects

Table 1: Case Studies where Stateful Policies Help to Minimize Tokens' Privilege. We define a library of three stateful policy types based on the state definition from Fig. 1b, but the client developer can construct any stateful policy they wish.

Client App	Service API	Actions	Current Privilege	Stateful Policy Type	State Used	Least Privilege
Video-conferencing (e.g., Zoom)	Google Calendar	Zoom creates new events, modifies or deletes existing events on Google Calendar	View and edit all the events on all the calendars	Access-only-created	Whether Zoom has made a request to `Events:insert`	Only access the events created by Zoom
Note-taking (e.g., Joplin)	Dropbox	Joplin uploads files for backup and downloads the backup files from Dropbox	Download/upload all the files in /Apps/Joplin folder	Access-only-created	Whether Joplin has made a request to `Files:upload`	Only access the files uploaded by Joplin
Trip Planner (e.g., ChatGPT)	Gmail	Trip planner reads users' Gmail searching for booking information	View all the emails in users' mailbox at any time	Read-at-most-N-times	How many times Trip Planner has made requests to `Messages:get`	Read each email at most once
Continuous Integration (CI) Platform (e.g., Jenkins)	GitHub	CI Platform creates check runs for CI tests and updates check runs when CI tests are complete	Read and write the check runs at any time	Write-at-most-N-times	How many times CI Platform has made `PATCH` requests to `check-runs/CHECK-RUN-ID`	Update each check run at most once

will be unreadable in the future. This relies on a state definition that can count the number of times the client makes an API call on the server. Consider a trip planner client app that reads Gmail to look for new flight or hotel bookings. It needs to process each new email exactly once. However, with current authorization, the bearer token allows reading an email multiple times. If the token is stolen, then the attacker can read any email they wish. With a token bound to a *read-at-most-once* policy, the attacker can only read new emails, and not any emails that the client app might have read in the past, thus providing "forward secrecy." Furthermore, this can be combined with additional stateless conditions such as only allowing the token bearer to read emails from a set of pre-specified sender addresses (e.g., Expedia or Kayak). With the status quo, such a restriction can be enforced only if the server happens to define such permissions. With our work, the client has the flexibility to create these restrictions and bind them with the bearer token.

Allow the token bearer to write an object at most N times. The *write-at-most-N-times* policy is designed to express a sense of "forward integrity" that once data has been committed, it cannot be tampered with or altered in the future. This concept is particularly relevant in scenarios where data needs to be verified not only for its current state but also for its historical integrity. Like the *read-at-most-N-times* policy, it also depends on a state definition that can count API calls.

Consider a GitHub repository integrated with a Continuous Integration (CI) platform, like Jenkins or Travis CI. These CI platforms will receive webhook events when a new commit is pushed to the repository, create a check run for CI tests, and update the status of the check run when the test completes. The CI platform will request a bearer token with `checks:write` permission that allows the token bearer to create and update the status of any check runs associated with the repository at any time. We build a proof of concept GitHub App [19] and find that the results of a check run can be altered even after being marked as "completed". An attacker with this bearer token can alter the result of a check run from "failure" to "success" so that a buggy release will be posted and potentially impact the interests of the repository owner. In this case, once a check run has been marked as "completed", its status, conclusion, and other attributes are considered as final and should not be modified. Thus, we can express the requirement with a *write-at-most-N-times* stateful policy that "allows the CI platforms to update the results of a check run at most once", to effectively prevent results of CI tests from being tampered with.[1]

Flexibility and Extensibility. We have discussed three examples of stateful least privilege policies inspired by open source apps. Our framework provides client developers flexibility and extensibility — they could add any number of stateless checks to the attenuation program; they could design custom state definitions; and they could combine various types of stateful policies that we have already designed (e.g., access an object created by the client at most 3 times).

3 Design and OAuth Instantiation

We discuss the design and implementation of our authorization framework in the context of the OAuth 2.0 protocol, the most widely-used authorization system on the Internet. Recall that OAuth 2.0 involves a client app developer who signs up on the authorization server operated by the service provider. At runtime, the client app instance requests permissions (or scopes, denoted as S_P) to access various server resources. The end-user is prompted with a permission box, where they can choose whether to grant the requested permissions. At the end, the client instance obtains a token that authorizes it to access server resources on behalf of a user. Our primary security goal is to limit the privilege of bearer tokens to the

[1] The repository maintainers may request re-runs of CI tests, which will commonly trigger the creation of another check run instead of overwriting the existing one, to keep a record of completed check runs.

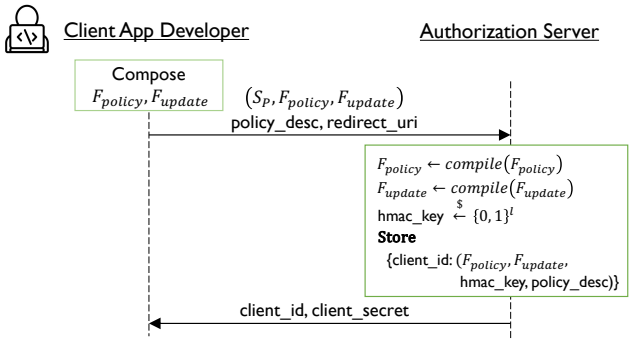


Figure 2: Client Registration Workflow. l is the key size and is set to 512 bits.

minimum privilege each client needs for its functionality. We use HMAC-based integrity protection of state data.

3.1 Workflow

We introduce our design in four steps that mirror the standard OAuth 2.0 phases: client registration, token request, token usage and token revocation/expiration. In each phase, we will use the running example of Zoom and Google Calendar to clarify the design.

3.1.1 Client Registration

In a standard OAuth flow, the client app developer will sign into a portal operated by the service provider. They will register their client app name, redirect URI and will request permissions (S_P) from a server-defined list. As discussed earlier, the server-defined list fundamentally cannot match the privilege requirements of the client exactly, so the developer selects a "best match" instead. We modify this registration step to include one more piece. The developer writes two programs (attenuation policy and state updater) in a language that compiles to WebAssembly and then submits the Wasm binaries to the service provider portal (see Fig. 2). We also require the developer to supply a string (`policy_desc`) that explains what the policy programs accomplish in layperson terms. This string will later be displayed to the end user while a bearer token is being requested. The server pre-compiles the Wasm binaries into modules for better performance, randomly generates a client-specific `hmac_key`, and returns the standard `client_id` and `client_secret` value to the client developer. All tokens that the server creates in the future for this client will be automatically bound to S_P, F_{policy}, and F_{update}.

Security Analysis. We trust the client app developer and their development environment. We also assume that the client developer writes the pair of programs that capture their least privilege requirements. Per our threat model, our goal is to empower developers to access minimum privilege and reduce abuse if bearer tokens are stolen. It is in the client app developer's best interest to protect their users' data. The server stores the programs, along with any requested OAuth permissions. This freezes the privilege level of any bearer tokens

```
1  def zoom_access_only_created(req, state):
2    if req.path.startswith('/events'):
3      if (req.path, req.method) == ('/events', 'POST'):
4        # Allow creating a new event.
5        return True
6      else:
7        # Iterate through the state - history of API calls,
8        # to see if the user has created this event.
9        for e in state:
10         if (e.path, e.method) == ('/events', 'POST'):
11           # Allow access if the state indicates that
12           # the user has created this event.
13           return True
14       # Deny if no creation in the state.
15       return False
16   # Allow accessing other endpoints in the scope.
17   return True
```

(a) *Access-only-created* Attenuation Policy for Zoom integrating with Google Calendar. Note that the client app supplies the state data and it is specific to the event object that the client is trying to access.

```
1  def state_updater(req, state):
2    for e in state:
3      if (e.path, e.method) == (req.path, req.method):
4        return state
5    newEntry = StateEntry(path=req.path, method=req.method)
6    state.append(newEntry)
7    return state
```

(b) Example State Updater Program. The state is defined as a list of distinct API calls made in the past.

Figure 3: Policy programs that compile to WebAssembly.

given by the server to a client app. The client app developers should follow all best-practices related to securing their credentials that allow them access to the server-operated registration portal, such as using strong passwords and two-factor authentication. Furthermore, there is no technical reason for developer credentials to appear in the client app's code. Thus, any compromise of the app instance does not give the attacker the ability to access the server-operated registration portal. This ensures that for a specific client, the binding between all its tokens (current and future), the Wasm policy file, and OAuth scopes remains static and can only change through explicit action of a physical developer with the right set of credentials.

Example. We revisit our running example of Zoom integrated with Google Calendar. Zoom only wants to either create new events or read/modify events it has created in the past. Google Calendar provides the `calendar.events` [22] scope that allows the token bearer to create/read/modify all events, including ones that the bearer did not create. So, the developer will first select this scope as it is the "best match". To attenuate this privilege further, Zoom developers will write the policy in Fig. 3a. It analyzes the request data and if there's a matching entry in the state information (Line 10), it means that Zoom has accessed/created this particular event in the past. If no matching entry is present in the state information, then the request is denied (Line 15). The Zoom developer defines state

as a list of distinct API calls it has made, keyed by event id. Every time the server successfully processes an API call on behalf of Zoom, the state updater program (See Fig. 3b) will be executed. If the program finds the request API call does not exist in the state, it will return a new state by appending a new state entry to the old state.

3.1.2 Token Request

A client app instance requests a bearer token before it can provide service to users. Without loss of generality, we will describe how our work modifies the standard OAuth authorization code grant flow (Fig. 4). Per this flow, the client app instance initiates the token request by directing the user to an authorization endpoint on the server. As part of this request, the app instance transmits the `client_id`. This allows the server to look up the previously stored OAuth permission, Wasm binaries and end-user description information. This information is displayed to the user in a standard OAuth permission prompt. Upon user consent, the server mints a new bearer token (T), associates it with the pre-defined OAuth permissions and Wasm programs ($S_p, F_{policy}, F_{update}$), and returns it to the client app instance.

Security Analysis. Per our threat model, an attacker could breach a client app instance to leak tokens and/or the `client_secret` value. In standard OAuth, this allows them to negotiate new tokens, often with more privilege than what the app had at the time of the attack. In our framework, the client app developer has already frozen the privilege of all current and future tokens during the trusted registration phase. We require the authorization server to ignore any scope requests outside the predefined permissions bound by S_p and F_{policy}. Thus, the attacker cannot change token privilege — any tokens they create will always be bound by the client developer's stateful least privilege policy. We also note that there is no mechanism through which the attacker can upload their own Wasm policy files during a token request. This step can only occur through the server-operated developer-only portal (see the client registration step above), and we assume that the client *developer's credentials* for this portal are secure.

End-User Experience. The end-user sees a standard OAuth permission prompt with the addition of a string description that explains how the Wasm policies further attenuate the token's permissions. In our example, the user will see that Zoom is asking for OAuth scope `calendar.events` and a policy description: *Zoom can only access the events it creates*.

3.1.3 Token Usage

The client app instance will transmit the bearer token with every request it sends to the API server. Our framework requires the client to also transmit the state information corresponding to the object(s) it wants to access (as an HTTP request header). This implies that the client knows ahead-of-time about the object identifiers involved in a request.

When the server's route handler receives a request, it will perform the following steps sequentially:

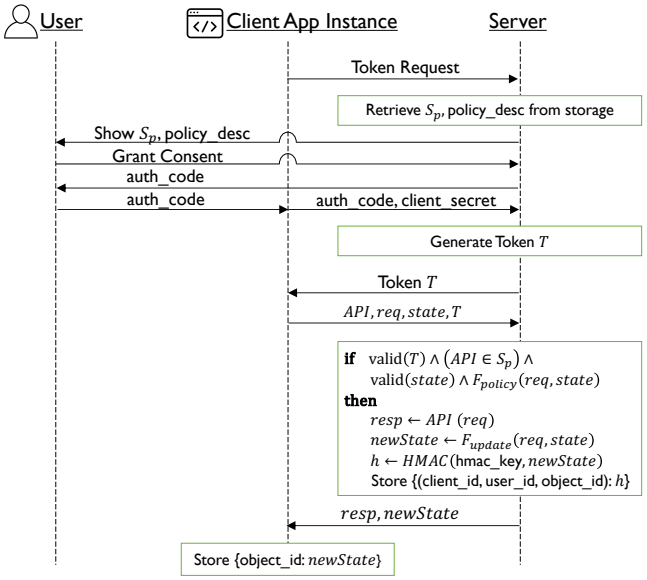


Figure 4: Token Request and Usage Workflow.

- Verify that the token T is valid (unexpired and not revoked) and that the API request is within the set of OAuth permissions granted to this token ($API \in S_P$).
- If the above step is successful, the server will verify integrity of the client-supplied state table by computing its HMAC with the client-specific `hmac_key` and comparing it with the server-stored value.
- The server runs the attenuation policy program (F_{policy}) with the request data and state to produce a final authorization decision. We use `wasmtime` [6] to execute the sandboxed Wasm policies. Note that policies do not need to interact with the external OS — they are pure functions. This provides strong memory isolation guarantees and allows the server to be efficient by running policies from different clients in the same OS-level process.
- If F_{policy} is true, then the server will allow the resource call to run as requested by the client. If the resource call is successful, the server will execute the state updater (F_{update}) to produce the new state and store the HMAC of the new state ($HMAC$(`hmac_key`, $newState$)). Then the server returns the API call response and new state to the client. Finally, the server stores the mapping of the triple (`client_id`, `user_id`, `object_id`) to the HMAC.
- Upon receiving the response, the client app instance will synchronize its local state storage with the new data (i.e., updating the map from `object_id` to state tables).

Security Analysis. Three attacks are possible with a compromised token.

- *Transmit a request without state information*. The server stores the latest state integrity tag for each object of the client. When it receives a request without any state attached, it will deny the request because it is expecting state data for integrity verification. The corner case is when the

state information is empty (i.e., the first time a client accesses an object). Our framework handles this naturally because there won't be an entry in the server's state storage for the client. However, the moment an entry is created, the server will always expect the correct state in the request.

- *Replay out-of-date state information*. The server maintains the invariant that it always stores the latest state integrity tag corresponding to an object. It achieves this by always first updating the state integrity tag on its end and then returning the updated state data to the client. Therefore, it will simply deny a request that has out-of-date state information. Note that a client can send out-of-date state in a request due to benign failures. We discuss this in Section 3.2.
- *Transmit a request with modified state information or state information from unrelated clients*. The attacker can tamper with state data. We trivially prevent this due to the integrity protection with either HMAC or hash.

Hash- vs. HMAC-based Integrity Protection. While we have discussed an HMAC-based protocol, our protocol is agnostic to using either hashing or HMAC. Both ensure state integrity: hashing relies on a collision-resistant hash function, whereas HMAC requires a client-specific secret key known only to the server. HMAC provides a stronger integrity guarantee compared to hashing, since collision vulnerabilities are mitigated by the HMAC construction [10, 29]. Furthermore, HMAC offers authenticity, ensuring that only states generated by the server are accepted. The server should select the appropriate mechanism based on its requirements for authenticity and key management capabilities.

Wasm security configuration. An attacker could write Wasm designed to affect the server's operation. We use `wasmtime` as our execution engine and our policies do not need to interface with the external OS because they are pure functions. This limits the available attack surface. Wasm guarantees memory safety — a module cannot interfere with the memory of other modules within/outside the Wasm process. An attacker could try a denial of service attack by writing a policy that doesn't terminate, in the hope of crashing the server. `wasmtime` provides a watchdog timer-like abstraction called "fuel" that limits the runtime of a piece of code. We use this feature to ensure that a misbehaving policy cannot hang the server.

Example. Following the example in Section 3.1.1, assume that Zoom wants to use its token to make a GET request on a calendar event it created earlier. The state is defined as a list of distinct past API calls made by Zoom on an object and we assume that the following state is associated with the token:

```
# The current state stored on Zoom instance.
object_id_1: [(/events, POST)]
object_id_2: [(/events, POST)]
```

Zoom wants to read the event with `object_id_1`, the event it created via the POST `/events` API. Zoom does this by sending a GET request to the `/events/object_id_1` API endpoint. Along with the request, Zoom will send only the state associated with `object_id_1`, since it is the only object the current request is accessing. Therefore, inside the request header, Zoom will include the state:

```
# The state of object_id_1 will be sent along with
# the request that accesses it.
object_id_1: [(/events, POST)]
```

After Calendar receives the request, it will validate Zoom's token, make sure the token is not expired or revoked, and the requested resource is in the scope of the token. Then, it validates the received state by calculating its HMAC, and checking the result against the correct HMAC in its database:

```
# Server-side storage of state HMACs.
object_id_1: hmac_1
object_id_2: hmac_2
```

Assume the state we present above is the latest, i.e., sent with the last response from the server. The HMAC stored on the server's state table (`hmac_1`) should be the same as the latest state HMAC on `object_id_1`. Thus, the state is valid.

Once Google Calendar confirms that the state is valid, it will proceed to run the policy program registered for Zoom, specified in Fig. 3a. This program evaluates to true because Zoom did create the event object. Next, Calendar performs the resource call and generates a response. If the resource call returns successfully, the server will execute the state updater program registered for Zoom. Based on the definition that the state is the list of distinct API calls Zoom made in the past, this state updater program will generate a new state for `object_id_1` by adding the URL path and HTTP method of the current request to the state.

```
# New state returned by the state updater program.
object_id_1: [(/events, POST),
              (/events/object_id_1, GET)]
```

The server (Calendar) then calculates the HMAC of the new state, and updates the old HMAC value in its database. Finally, along with the API response, the new state for `object_id_1` will be sent to the client. Zoom receives the response and stores the new state in its database.

3.2 Miscellaneous Operations

Token Revocation and Expiration. When the server revokes or expires a bearer token that belongs to a client app instance, our framework ensures that the newly-created token will automatically and seamlessly inherit the latest state information. This process is enabled by the server-side storage that maps `(user_id, client_id, object_id)` to state HMAC $HMAC(\texttt{hmac\_key}, state)$. When the server creates a new token for a client (using the procedure outlined in Section 3.1.2), the newly-created token is naturally associated with the existing `client_id` and `user_id` pair for that client app instance. Once the client receives the new token, it will simply associate that token with the existing state data. When it makes an API request using the new token and the `object_id`, the server

will look up the corresponding `client_id` and `user_id` pair to locate the state HMAC and will seamlessly find a match.

Benign Failures. Network- and system-level failures can result in loss of state data. We assume that TCP handles all network-related packet drops. We expect that the client app and authorization server use crash-consistent storage techniques for backing up state data and its HMAC respectively. The server maintains a time-limited cache of previously-sent state data, using a system such as DynamoDB configured with a Time-to-Live for key-value entries [15]. If the client goes offline while it is receiving a response, it can contact the server once its back up to retrieve the most recently cached state data. We envision that more advanced techniques are possible that allow a client to communicate its desired time-to-live for cached state to the server based on the client's failure rate, but these are orthogonal and beyond the scope of this work.

Synchronizing State Between Client Instances. There can be multiple instances of the same client app for a single user at the same time. For example, a user can be logged into Zoom on their smartphone and their browser. Both of these client app instances need the same state information. We require that the client app instances for the same user utilize existing replication techniques to ensure that the state table is consistent across them. A simple method to achieve this is for each client app instance to always upload the latest state table to a cloud storage service that other instances can pull from.

Synchronization Attacks. An attacker can de-synchronize the client-stored state table and the server-stored state HMACs, causing requests to be rejected in state validation. Upon the detection of token leakage, the server will revoke all the leaked tokens. The client app developer will pull the state tables from the affected app instances and send them to the server via a portal accessible only to the client app developers. Observe that these tables do not contain the updates caused by the attacker. The server will compute and store new state HMACs, thus achieving re-synchronization. If the `client_id` and `client_secret` are leaked, the attacker could obtain new tokens. Thus, besides the operations above, the server will revoke `client_id`/`client_secret` pair and all the client tokens. The client app developer should derive a new `client_id`/`client_secret` pair from the server portal.

State Lifecycle and Time-based Limits. State data comes into existence when an object is created on the server and is deleted when the object is removed from the server. Concretely, the server will remove the state hash value when the corresponding object is deleted. To manage storage independently of object lifetime, the client developer could set a Time-to-Live (TTL) for how long state is available. For example, it is unlikely that the user wants Zoom to manipulate events that have occurred in the distant past and thus, it is unlikely that Zoom will make an API request to access those events. The attenuation policy could include a time interval condition that only allows access to Calendar events newer than a configurable value. Beyond that, the server deletes state hashes whose TTL has expired and the client deletes the actual state data. Similarly, for the Trip planner app, it is unlikely that it needs access to emails older than a configurable threshold (say, 6 months). Thus, the client and server can co-operate to reduce their state storage independently of the actual object lifetime. Finally, these time intervals can be exposed to the user, providing them with a knob to tune the level of access they are willing to give out.

Privilege Changes. An app developer can add or remove features over time, and this impacts the amount of privilege it needs. Currently, OAuth handles this by requiring the developer to change their permissions on a portal that the server provider operates. We integrate with this mechanism — the client developer will modify the attenuation policy and optionally the state updater program based on new requirements using the same portal from the client registration phase discussed above.

Changes to the state updater might change the schema of state tables. Thus, both the client-side state table and server-side state HMACs need to be updated accordingly. We require the client app to undergo a maintenance period and perform migration following the new schema. Specifically, the client app developer will write a program that updates a state table following the new state definition and distributes it to all the app instances. Each app instance updates its state table locally. Then, the client app developer will pull and submit the updated state table to the authorization server. The authorization server will recompute and store the new HMACs.

4 Evaluation

We integrate with OAuth 2.0 protocol and provide a collection of APIs for developers to apply stateful authorization as a "drop-in" solution. We evaluate the performance of our framework based on the development effortsstorage overhead and end-to-end latency. We refer to our authorization framework as StatefulAuth in the following discussion.

4.1 Porting Real Client Apps and API Servers

We design APIs for both server- and client-side developers based on Authlib [33], a Python library of generic OAuth implementations, outlined in Table 3 in Appendix A. With these APIs, the transition to stateful authorization only requires minor changes to client- and server-side codebase.

Fig. 5 reports the example programming interface for the client and server. On the server side (Fig. 5a), the developers will create a Python decorator [38] `@require_oauth_stateful` with the `ResourceProtectorStateful` class (Line 7) and register a stateful token validator for it (Line 8, 9). Then, the developer will wrap each protected resource API endpoint with this decorator (Line 14). At runtime, when a new request to the protected resource arrives, it has to pass the following checks inside `@require_oauth_stateful`: validating that

```python
from flask import Flask, Response, make_response
from statefulauth.models import OAuth2Token
from statefulauth.serverlib import
    ResourceProtectorStateful, update_state,
    create_bearer_token_validator_stateful

from models import Event

require_oauth_stateful = ResourceProtectorStateful()
validator_stateful =
    create_bearer_token_validator_stateful(OAuth2Token)
require_oauth_stateful.register_token_validator(
    validator_stateful())

app = Flask(__name__)

@app.route('/api/events/<event_id>')
@require_oauth_stateful('events')
@update_state()
def get_event(event_id: str) -> tuple[Response, str]:
  event = Event.query.get(event_id)
  return make_response(event.as_dict), event_id
```

(a) Server-Side

```python
from requests import get, Request
from statefulauth.clientlib import attach_state,
    store_state

def send(req: Request, obj_id: str):
  req = attach_state(req, obj_id)
  resp = get(req)
  store_state(resp, obj_id)
```

(b) Client-Side

Figure 5: Code Snippets of Client- and Server-Side APIs.

the bearer token is valid and that the request is within the pre-approved OAuth scope of that token, validating state integrity and freshness, and executing the policy program. After a request passes all the checks and the resource query succeeds, another decorator `@update_state` (Line 15) will execute the client-supplied state updater program to generate a new state, send it back to the client, and store its hash value on the server.

On the client side (Fig. 5b), we provide `attach_state` to load the state and attach it to a request's header before sending the request to the server (Line 5). When the client receives a response from the server, `store_state` will fetch the new state in the server's response header and update the client's state storage accordingly (Line 7).

Deploying to Real Applications. We deploy our authorization framework to protect two apps from the case studies in Table 1: Joplin (a note-taking app that backs up to Dropbox) and a ChatGPT-based trip planner (that reads Gmail). We only port these two apps because they can be customized to meet the deployment requirements of StatefulAuth. The client apps in other case studies are proprietary in nature. We present the lines of code (LoC) of attenuation policy and state updater programs of all the case studies in Table 2.

We integrate our library into the Joplin terminal app written in JavaScript by implementing a JavaScript client library that helps Joplin load and store state. The code changes to

Table 2: Worst-Case Storage and LoC of F_{policy} and F_{update}.

Client App	Service API	F_{policy} LoC	F_{update} LoC	Client-Side Storage Per Object	Server-Side Storage Per Object
Zoom	Google Calendar	14	8	1.02 KB	256 bits
Joplin	Dropbox	18	9	0.42 KB	256 bits
Trip Planner	Gmail	18	9	0.13 KB	256 bits

Joplin are similar to those illustrated in Fig. 5b, involving the insertion of only two lines of code before and after sending the request. These modifications are made within the File API Driver package of Joplin. During the synchronization process, Joplin will create a folder called "/Apps/Joplin", create a backup file for each note, and create metadata files — this process involves 9 different Dropbox API endpoints. On the server side, we replace the token validator with a stateful validator (Fig. 5a Line 8, 9) and decorate these endpoints with `@require_oauth_stateful` and `@update_state`. For `@update_state` to work properly, each endpoint will return a tuple of `Response` and `object_id`. All the above requires 30 lines of code changes on the server side.

Our second implemented case study is a ChatGPT-based trip planner built using a custom GPT [34]. The custom GPT feature allows ChatGPT to call cloud service APIs using OAuth. We built our trip planner GPT to mirror the functionality of commercial trip planners that scan the users' emails to recommend trip itineraries, such as TripIt [3] and Wanderlog [4]. We define custom "actions" that make calls to an email server's API available to ChatGPT. The email server is modified similarly to the Joplin integration, we add the decorators `@require_oauth_stateful` and `@update_state` to the server's API endpoints, and we include the `object_id` of the current request in the API response. Since we cannot modify any client-side code, all state storage is managed on the email server for demonstration purposes.

Deploying stateful authorization requires changes to both the client and server's codebase. Due to the lack of open-source service APIs, we implement a proxy server as an extra layer between the client and the server. The proxy is registered as a client of the real service. When receiving a request from the client, the proxy will perform the validation process (i.e., token authentication, state validation, and policy execution) and forward the request to the real service if validation succeeds. When receiving a response from the server, the proxy will update the state if the request succeeds and forwards the response to the client. We note here that the real implementation is to modify both the client- and server-side codebase. Our implementation with the proxy is to show that the system works with real client apps and API servers.

4.2 Storage and Memory Overhead

Storing State. The server-side state storage scales with the product of the number of client-user pairs and the number of

objects (Section 3.1.3). We use HMAC-SHA256; each entry in the server-side state storage is a 256-bit value.

The client-side state storage only scales with the number of objects. The client will store the complete state for each object. Fig. 1b demonstrates an example where the state is defined as a list of distinct API calls made by the client and their respective counters. With this state definition, we can compute the state size of a single object on the client side as:

$$\begin{aligned} \texttt{StateSize} &= \texttt{MAX\_STATE\_ENTRY\_SIZE} \times |S_u| \\ &\leq \texttt{MAX\_STATE\_ENTRY\_SIZE} \times |S_p| \end{aligned} \quad (1)$$

Here $|S_u|$ denotes the number of API endpoints used by the client app and $|S_p|$ the number of API endpoints in the set of server-defined permissions granted to the token. Note that $S_u \subseteq S_p$ and $|S_u| \leq |S_p|$, i.e., the used API endpoints must be a subset of those permitted by the server-defined scope. `MAX_STATE_ENTRY_SIZE` is the maximum size of an entry in the state table, a constant based on the definition and design of the state data structure. We measure the client- and server-side state storage size per object for a subset of our case studies and present the result in Table 2.

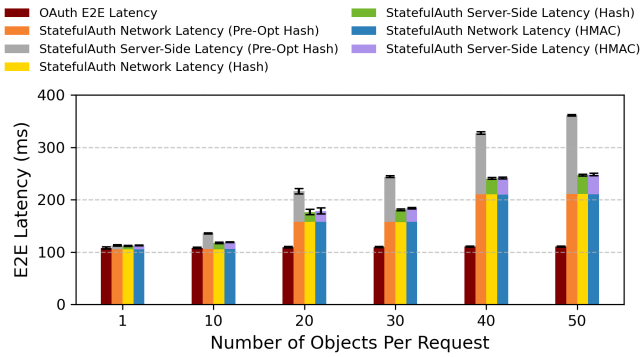
Storing Policies. The server will store two programs for each client. To estimate the size of Wasm binaries, we write a collection of policies for our case studies in Table 1, including stateless policies, stateful policies, and combined policies. Stateful policies include the *access-only-created* and *read/write-at-most-N-times* policies described in Section 2.3. Examples of stateless policies include scrutinizing attributes in the request body (e.g., only permitting the creation of calendar events if the specified time falls beyond February 15th). We also write more complicated policies by combining the above policies. We observe that the size of each Wasm binary is 120-200 KB. The size of *access-only-created* Wasm binary we use in the integration with Joplin is 124 KB, and the size of the *read-at-most-once* Wasm binary used in ChatGPT trip planner is 121 KB. The size of the state updater Wasm binary for both Joplin and trip planner is 115 KB.

Memory Overhead of Policy Execution. We measure the amount of memory required to execute Wasm programs. Each of the Wasm programs (including both the policy and state updater programs) in the three examples we describe in Table 2 takes 128 KB (i.e., two WebAssembly pages).

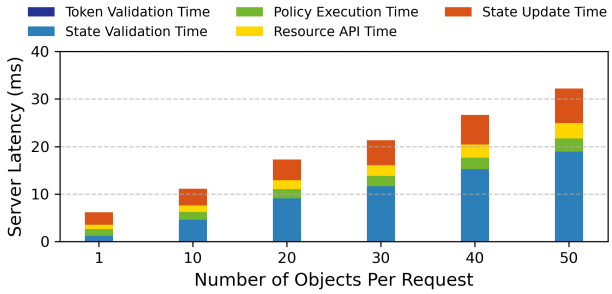
4.3 Latency Overhead

We measure latency of authorization in the baseline case (vanilla OAuth 2.0) and our case (StatefulAuth).

Experiment Setup. We implement a client-server framework to measure end-to-end latency. We host our server on a c5d.2xlarge instance with 8 vCPUs and 16 GiB memory on Amazon Web Services (AWS) in the US East (Ohio) region, and our client app on another instance with the same configuration in the US West (North California) region. To simulate the overhead of a real system, we mock up an implementation of Google Calendar APIs. Specifically, we mirror



(a) End-to-end latency of OAuth and StatefulAuth (Hash- and HMAC-based integrity.



(b) StatefulAuth (HMAC-based) Server-side Latency Breakdown.

Figure 6: End-to-end and server-side latency breakdown.

the Events API [21]. We define an `Event` table in the server-side database. Each API endpoint performs the corresponding operations on the `Event` table. We use an in-memory SQLite database [39] for the above operations.

We use an *access-only-created* policy written for Events API in our evaluation of StatefulAuth. The client generates a sequence of requests. Each request is randomly selected from the set of Google Calender Events API endpoints. If the protocol is OAuth, the client will send a bare request; if the protocol is StatefulAuth, the client will attach the state associated with a request to its header. To estimate the worst-case latency, we generate requests that incorporate the maximum state length for each object ($|S_p|$ in Eq. 1). Each state table is configured to include 10 records, representing the entirety of API endpoints in Google Calendar `calendar.events` scope. We also assume all the requests will be completed successfully to estimate an upper bound of latency.

In StatefulAuth, the client selectively transmits only the state information of the objects associated with a particular request. Our objective is to analyze the end-to-end latency in relation to the quantity of objects included in a single request. We simulate batch requests where N objects are processed in a single request. For example, when $N = 10$, the client will send 10 `object_ids` and their associated states, and the resource API endpoint will sequentially process each object. We set the upper limit for batch size at 50, aligning with the

guidelines in Google's API documentation. Google advises against exceeding this threshold, as larger batch sizes are susceptible to triggering rate limiting [20]. Hence, we adhere to this recommendation and restrict our analysis to batch sizes not surpassing 50 objects.

Server-side Latency Optimization. We optimize the server-side latency of batch requests. Instead of executing a separate Wasm program for each object, we offload the iteration over objects to the Wasm module. This allows us to execute a single Wasm program for all objects in a request, eliminating the overhead of repeatedly instantiating and calling Wasm modules. We also optimize the latency of database queries by performing a bulk save of all the state integrity tags in a request. We achieved an improvement in server-side latency by over 2.3× when $N \geq 10$, and by over 4× when $N = 50$. We improved the end-to-end latency of StatefulAuth by over 20% when $N \geq 30$, and by over 30% when $N = 50$, as shown in Fig. 6a. We tried parallelizing operations on different objects, but it didn't yield performance benefits due to the Python Global Interpreter Lock and the overhead of parallelism. Since the optimized latency is low and mostly compute-bound, the added complexity of parallelism worsened performance.

End-to-End Latency. Fig. 6a presents the end-to-end latency when applying OAuth and StatefulAuth (hash- and HMAC-based) to our client and server framework. We see that StatefulAuth (HMAC-based) introduces a 4.3% increase to end-to-end latency when processing a single object in each request ($N = 1$) compared with OAuth, and 9.5% when $N = 10$. Although most requests in a real-world setting only operate on a single object, we also present the case of $N = 50$ to represent the theoretical upper bound of the end-to-end latency, which we observe as 2.2× of OAuth. We conduct an analysis of the end-to-end latency of StatefulAuth and break it into network latency and server-side latency. As per the comparison of network latency between OAuth and StatefulAuth, the transmission of additional state data incurs linear growth relative to the number of objects, consequently introducing supplementary network latency. We observe that when $N \geq 20$, the network latency accounts for above 68% of increase in end-to-end latency compared with OAuth.

Server-Side Latency Breakdown. The server-side latency of StatefulAuth can be further divided into five parts, shown in Fig. 6b. We see that the server-side latency increases proportionally with the number of objects because the server will sequentially iterate through each object for each operation. State validation is the predominant contributor to server-side latency, which takes around 36-53% of total server-side latency. Upon closer inspection, we have identified the primary source of latency in state validation as the sequential reads of the database for HMAC values.

Comparison of Stateless Policies with Macaroons. We compare the end-to-end latency of stateless-only policies in our work with the existing Macaroons framework. Note that we only compare with Macaroons because they functionally subsume OAuth RAR (Section 2.1). We use pymacaroons and create caveats (i.e., stateless authorization policies) that involve string comparisons on HTTP POST request data. We implement the same policy in Wasm for our work. We measure end-to-end latency using the same setup as before. Fig. 7 shows that our work only adds 1–2 ms of latency compared to Macaroons on average. We conclude that this overhead is a small price to pay in exchange for the richer set of authorization policies that we enable.

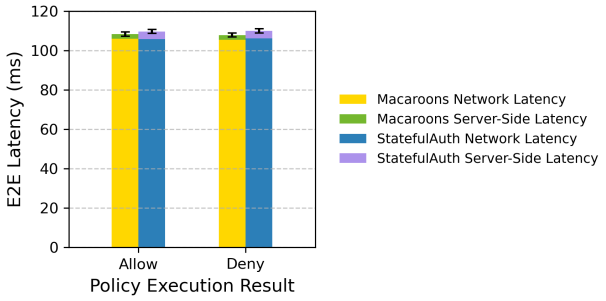


Figure 7: Stateless-only Policy Comparison with Macaroons.

5 Related Work

Credentials-based Authorization. Our authorization framework is an instance of credentials-based authorization that uses credentials and guards to enforce policies [17]. Credentials define beliefs about principals and the state of a distributed system. Guards use logical formulas and inference to determine whether a given access is allowed. We define principals as the client app and the API server. We allow the client app developers to define and manage state. However, we do not use a logical formula to express the authorization policy. Rather, we use a Wasm program that offers flexibility and practicality.

There is a long line of work in this general area that uses public key certificates and formal authorization logic to support a range of flexible and decentralized authorization policies for distributed systems. SPKI/SDSI [31], Trust Management [12], Active Certificates [13] are a few notable examples. However, such mechanisms have not been widely adopted in the cloud [42] for a variety of reasons such as their implementation complexity or the need for long-term identities.

Currently, authorization in the cloud is based on bearer tokens — secrets that grant unconditional access to resources. These tokens are exchanged between protection domains and are vulnerable to a range of token stealing attacks [26, 28, 30, 37, 40]. Our contributions exist in this practical space of trying to enforce the least privilege principle on bearer tokens. Macaroons was a first effort at improving the security of OAuth-like bearer tokens [11]. They provide a method for first and third parties to attenuate the power of bearer tokens by adding caveats — conditions that limit the

token's authority. Similarly, the OAuth Rich Authorization Requests RFC attempts to limit token privilege by defining a JSON-based standard to express caveats [32]. As we explained in Section 2.1, these existing efforts still suffer from fundamental problems associated with their use of server-defined permissions and stateless policies. Concretely, Macaroons still require the server developers to define the caveat language. This results in a lack of extensibility — clients may want to enforce policies that fall outside the server-defined caveat language. Furthermore, there is an entire class of authorization policies that rely on state information, as we explained earlier. These stateful classes can limit a token's authority even further compared to what is achievable by just using stateless machinery. Our work provides a flexible WebAssembly-based substrate to execute client-defined stateful authorization policies.

We take inspiration from Active Certificates [13]. Originally invented in the context of delegating rights to a remote resource, they introduce the idea that a small piece of code can represent the rights a principal has to a server-owned object. When that principal makes a request to the server, it presents the active certificate that is signed by the object owner. The server executes the code inside the active certificate to produce an authorization decision. Our work builds on this idea and allows client app developers to write Wasm policies that represent their intended access to the user's data. We offer additional improvements to this basic idea: (1) We retain the server-defined permissions and use them as an upper bound on token authority in the event that the Wasm policy has a developer error in its checks. In their original form, active certificates are vulnerable to coding mistakes and can result in a principal getting unbounded access to server resources. (2) We define and use the notion of state that serves as an additional input to the Wasm policy and enables new classes of least privilege policies.

WAVE is a recent delegated authorization framework designed to enforce server-defined permissions even when the authorization server is compromised [9]. It is orthogonal to our work. We trust the authorization server and our goal is to empower client app developers to request minimum privilege.

WebAssembly Security. Wasm has emerged as a portable bytecode format and execution environment for the web [24]. Multiple languages offer Wasm as a compilation target. Writing policies in any source language developers prefer and compiling to Wasm gives us portability and flexibility. More importantly, it also offers strong in-process memory isolation. The authorization server can execute Wasm policies from multiple different clients in the same OS-level process and be guaranteed that the policies cannot interfere with each other. Our particular usage of Wasm is to run policies that are pure functions. Thus, we avoid the security issues of integrating with an external environment by design [5].

6 Discussion and Limitations

Time-limited OAuth Tokens and Fine-grained Scopes. It is possible to reduce the damage that can occur from stolen OAuth tokens by limiting their lifetime. Unfortunately, this implies that the server operator has to detect an attack in a very timely manner to stop honoring refreshing requests. Experience has shown that this rarely happens. More fundamentally, the tokens are still overprivileged and there will always be a mismatch because a server developer cannot always anticipate the permissions a client might actually need [16, 36]. Additionally, as we have shown with our case studies, client apps can benefit from stateful policies as well.

Encrypting State. Our implementation only guarantees state integrity. A client app breach could result in state getting leaked. Depending on the nature of the API endpoints, this could potentially be a privacy problem because the attacker can obtain coarse information on what API endpoints are being used. A simple extension is to also encrypt the state table when it exists on the client using a key known to the server. We do not implement this, but leave it to future work.

Automatic Least Privilege Policies. Our current system requires developers to manually write attenuation policies. As future work, we envision two possibilities: (1) The client app could profile its usage of the server APIs and use that data to automatically synthesize a least privilege policy. (2) The server developer could write a set of well-known/commonly-used policies (e.g., access-only-created, read-at-most-once) and provide it as an option for the client developer to select during registration. A related line of work is to verify that a client application is indeed obtaining tokens that are least privileged. The server operator could keep a log of the APIs that a client actually uses and then compare those APIs with the set of APIs that are callable under the least privilege policy that the client has created.

7 Conclusion

Current cloud authorization uses bearer tokens whose authority is limited only by a set of permissions that API server developers create. In practice, client app developers are forced to request tokens whose privilege is more than what is necessary for functionality. Existing efforts have recognized this problem and attempted solutions. The most notable is the Macaroons framework that allows server developers to define caveats that can be used to limit a token's authority. We contributed to this line of work by introducing the notion of purely client-defined authorization policies that can operate on state — a log of actions that a client has taken on API endpoints. When combined with server-defined permissions and Wasm as a vehicle for expressing the policies, we obtain a flexible authorization system that can enforce a variety of stateful least privilege policies. Using case studies of real apps, we showed how app developers can restrict the authority of their tokens while retaining functionality.

Acknowledgements

We thank the anonymous reviewers and our shepherd for valuable feedback. We also thank Arnar Birgisson, Amir Rahmati, Andrei Sabelfeld, Stefan Savage and Geoff Voelker for insightful comments. This work is partly supported by NSF grants CNS-2312119, CNS-2048262 and by gifts from Amazon and Google.

References

[1] Cve-2023-30527. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-30527. Available from MITRE, CVE-ID CVE-2023-30527.

[2] Joplin. https://joplinapp.org/, 2024.

[3] Tripit. https://www.tripit.com/, 2024.

[4] Wanderlog. https://wanderlog.com/, 2024.

[5] WaVe: a verifiably secure WebAssembly sandboxing runtime. Evan johnson and evan laufer and zijie zhao and dan gohman and shravan narayan and stefan savage and deian stefan and fraser brown. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2023.

[6] Bytecode Alliance. Wasmtime. https://wasmtime.dev, 2024.

[7] Amazon Web Service Inc. DynamoDB encryption at rest. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html, 2024.

[8] Amazon Web Services, Inc. Amazon S3 pricing. https://aws.amazon.com/s3/pricing, 2023.

[9] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1375–1392, Santa Clara, CA, August 2019. USENIX Association.

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 1–15. Springer, 1996.

[11] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*, 2014.

[12] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[13] Nikita Borisov and Eric A Brewer. Active Certificates: A Framework for Delegation. In *NDSS*. Citeseer, 2002.

[14] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 892–903, 2014.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Travis Developers. Travis CI's use of GitHub API Scopes. https://docs.travis-ci.com/user/github-oauth-scopes/#repositories-on-httpstravis-cicom-private-and-public.

[17] Fred Schneider. Credentials-based authorization. https://www.cs.cornell.edu/fbs/publications/chptr.CredsBased.pdf, 2015.

[18] GitHub. Differences between GitHub Apps and OAuth apps. https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/differences-between-github-apps-and-oauth-apps, 2024.

[19] GitHub. GitHub App. https://docs.github.com/en/apps, 2024.

[20] Google. Batching requests. https://developers.google.com/gmail/api/guides/batch, 2023.

[21] Google. Events | Google Calendar. https://developers.google.com/calendar/api/v3/reference/events, 2023.

[22] Google. Choose Google Calendar API Scopes. https://developers.google.com/calendar/api/auth?hl=en, 2024.

[23] Google Cloud. Default encryption at rest. https://cloud.google.com/docs/security/encryption/default-encryption, 2024.

[24] W3C WebAssembly Working Group and Community Group. Webassembly. https://webassembly.org, 2024.

[25] W3C WebAssembly Working Group and Community Group. Webassembly security. https://webassembly.org/docs/security, 2024.

[26] Mike Hanley. Security alert: Attack campaign involving stolen OAuth user tokens issued to two third-party integrators. https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens, Apr 2022.

[27] Dick Hardt. The OAuth 2.0 Authorization Framework. https://www.rfc-editor.org/info/rfc6749, October 2012.

[28] Tommaso Innocenti, Matteo Golinelli, Kaan Onarlioglu, Ali Mirheidari, Bruno Crispo, and Engin Kirda. OAuth 2.0 Redirect URI Validation Falls Short, Literally. In *Proceedings of the 39th Annual Computer Security Applications Conference*, ACSAC '23, page 256–267, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[30] David Krispin and Nir Swartz. Microsoft and GitHub OAuth Implementation Vulnerabilities Lead to Redirection Attacks. https://www.proofpoint.com/us/blog/cloud-security/microsoft-and-github-oauth-implementation-vulnerabilities-lead-redirection, Mar 2022.

[31] Ninghui Li and C. Mitchell. Understanding spki/sdsi using first-order logic. *Int. J. Inf. Secur.*, 5(1):48–64, jan 2006.

[32] Torsten Lodderstedt, Justin Richer, and Brian Campbell. OAuth 2.0 Rich Authorization Requests. https://www.rfc-editor.org/info/rfc9396, May 2023.

[33] Hsiaoming Ltd. Authlib. https://authlib.org, 2017.

[34] OpenAI. Introduing GPTs. https://openai.com/blog/introducing-gpts, 2023.

[35] Matthew Prince, John Graham-Cumming, and Grant Bourzikas. Thanksgiving 2023 security incident. https://blog.cloudflare.com/thanksgiving-2023-security-incident, 2023.

[36] Thomas Ptacek. Fly.io server-defined permission issues. https://fly.io/blog/macaroons-escalated-quickly/, 2024.

[37] Jody Serrano. Hacker Group Reportedly Leaks Sensitive Data of 2.28 Million People Registered on Dating Site MeetMindful. https://gizmodo.com/hacker-group-reportedly-leaks-sensitive-data-of-2-28-mi-1846122878, Jan 2021.

[38] Kevin D. Smith, Jim J. Jewett, Skip Montanaro, and Anthony Baxter. Pep 318 – decorators for functions and methods. https://peps.python.org/pep-0318, 2003.

[39] SQLite. In-Memory Databases. https://www.sqlite.org/inmemorydb.

[40] San-Tsai Sun and Konstantin Beznosov. 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*, pages 378–390, 2012.

[41] Phil Hunt Torsten Lodderstedt, Mark McGloin. OAuth 2.0 Threat Model and Security Considerations. https://datatracker.ietf.org/doc/html/rfc6819#section-4.1.1, 2013.

[42] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, USA, 1st edition, 2011.

[43] Sourov Zaman, Lucas Ferreira, Kimberly Hall, and Grant Bourzikas. How Cloudflare mitigated yet another Okta compromise. https://blog.cloudflare.com/how-cloudflare-mitigated-yet-another-okta-compromise, 2024.

Table 3: API Overview of StatefulAuth Integrated with OAuth 2.0

Server-Side APIs	**Semantics**
`class ResourceProtectorStateful`	A protecting method for resource server with StatefulAuth enabled. This class provides a wrapper function that validates any incoming request.
`create_bearer_token_validator_stateful (token_model)`	Create a bearer token validator class with StatefulAuth enabled. This validator will authenticate the bearer token, validate history and run the policy program.
`update_state()`	A method that calls the client-supplied state updater to update state after the successful completion of a request and send it back to the client; besides, update state HMAC on server-side storage.
Client-Side APIs	**Semantics**
`attach_state(request, object_id)`	Load state indexed by `object_id` and attach it to the request header field "Authorization-State".
`store_state(response, object_id)`	Update client-side state storage of an object based on the response header "Set-Authorization-State".

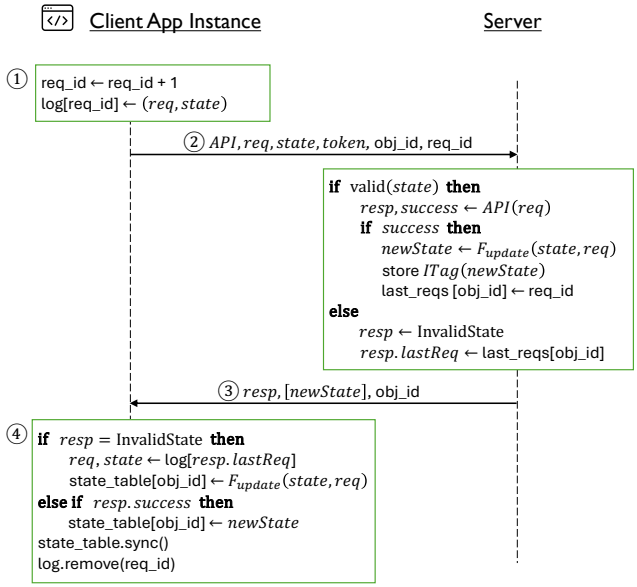


Figure 8: Token Usage Workflow with Fault Tolerance. For simplicity, we assume here that all the token validation and policy execution are passed successfully.

A API Overview of StatefulAuth Integrated with OAuth 2.0

We integrate with the widely-used OAuth 2.0 protocol and present the APIs we provide to help enable stateful authorization in Table 3.

B Fault Tolerance

We assume that the API server is consistently online, while the client app instance running on user devices may experience intermittent offline periods. Our goal is to facilitate seamless recovery of the client app' most recent state despite offline conditions. As presented in Fig. 1, F_{update} generates the new state based on the current state and the request being processed. A naive idea is to back up state or recent requests on the server side and recover the most recent state by replaying state updates. However, both state and request data are user-controlled. Storing them on the authorization server will have a potential security problem: a malicious client could leverage this arbitrary storage primitive to store illegal material on the authorization server. Instead, we propose an approach based on client- and server-side logging, as presented in Fig. 8. For simplicity, we assume that token validation and policy execution succeed, and focus on state validation and recovery. The client will assign an ID for each request (`req_id`). This request ID will serve as a client-side global counter, unique for this client-server pair. The client app will also create a cloud-based encrypted log file to keep track of all the past requests and states, ordered and indexed by the request ID. Before making a new request, the client app will log the request and state information it will supply to the server. Then, the client app will make requests as usual, with the request ID attached. Upon receiving the request, the server will verify the state. If the state is valid and the request is processed successfully, the server will store a mapping (i.e., `last_reqs` in Fig. 8) from the object ID to the request ID and perform the state update as usual; otherwise, the server will return an error message. If the state is invalid, the server will send a response containing a special error message `InvalidState` and attach the last request ID performed on the object to the response. Once the client receives the `InvalidState` error, it can recover its latest state by looking up the log with the request ID provided by the server and perform state updates. After state recovery, the client will synchronize the state table with other instances and remove the request already used for updates from the log file. Then, the client will check the log file for request ID counter, increment it, and send its next request.

Table 4 discusses how our system could be recovered when the client goes offline at different points in the workflow. The circled number refers to the steps in Figure 8. Our design distributes the responsibility of logging to both the client and server: the client stores the request information and the server only stores the request ID of the last request performed on each object. This enables the client to recover the most recent state table without compromising server storage to arbitrary client data.

We discuss the storage overhead introduced by this fault tolerance mechanism. Due to the sequential state updates on a single object, the client app can only send one request for an object at a time. Thus, for each client-user pair, the server will

Table 4: Client Failures and Recoveries. The circled number refers to the steps in Figure 8.

Client goes offline	**Request in Log**	**State Updated on Server**	**Recovery**
Before/during logging the request ①	False	False	No state update on the server side. The client-side state table remains up-to-date. No recovery needed.
After logging the request ①, before sending the request ②	True	False	
After sending the request ②, before receiving the response ③ (request **failed** on server)	True	False	
After sending the request ②, before receiving the response ③ (request **succeeded** on server)	True	True	When the client comes back, it will send a request with the outdated state. The server will find the state is invalid, reply `InvalidState`, and supply the `last_reqs` mapping. The client will then look up its log file indexed on the request ID, find the request and state information, and perform the state update for each object if needed.
After receiving the response ③, i.e., during updating the state table ④	True	True	If the client goes offline while updating/recovering the state table, the state table will be corrupted. But the log file and server-side `last_reqs` mapping stay unchanged. Thus, the same updates will be performed on the state table when the client sends a new request.

store one entry for each object in the `last_reqs` mapping. This server-side storage overhead will grow linearly with the product of the number of client-user pairs and objects, i.e., the same as the server-side state integrity tag storage. The size of the cloud-based log files maintained by the client app will grow linearly with the number of objects this client has accessed.

In Section 3.2, we discuss the state synchronization between multiple client app instances. We require the client to keep a cloud-based log file to enable all its instances to recover the state table through offline periods. As we have described, this log file serves as a key-value store that allows the client to look up the request information it needs for state recovery: the key is the request ID and the value is the request information, including HTTP method, URI endpoints, headers, and body, along with the state table when making the request.

We require the log file to be fully client-owned and managed, because similar to the state table, the request header and body could be arbitrary data from the client. The log file itself has no security implication on the existing system design. If the log file is compromised and an unacknowledged client request `req` is replaced with a fake request `req′` crafted by the attack, the client will end up updating its state by $newState' \leftarrow F_{update}(state, req')$. However, The latest state on the server side is derived by $newState \leftarrow F_{update}(state, req)$. There will be a mismatch between the state computed by the client and the server. The client will then be notified of this error and request the server to revoke that token. The recovery against synchronization attacks we describe in Section 3.2 will handle this case.

As the log file contains information on the recent requests made by the client, it should be kept confidential. Thus, we require the log file to be encrypted. The client should implement client- or server-side encryption based on its capability of storage, key management, and file encryption. For example, if the client has its own backend storage server, it can perform either client- or server-side encryption. On the other hand, if the client app does not have its own storage server, it needs to either encrypt the log file before storing to cloud services or rely on the server-side encryption provided by major cloud storage platforms, like Google Cloud [23] or DynamoDB [7]. Besides encryption, information stored on this log file should also be minimized in two ways. First, the client should store the minimum data in its log file based on its state definition. If the state is defined as the historical APIs made by the client, the log file should only contain the HTTP methods and URI endpoints of each request. Second, the client app should remove a request after using it for state updates (Figure 8 ④).