# Protocols and Formal Models for Delegated Authorisation with Server-Side Secrecy


## ABSTRACT

*OAuth 2.0* is a well-known protocol suite whereby customers of a web service can grant third-party applications access to their information (or *resources*) on said web service, all without handing over their long-term credentials. But what if the resources are encrypted? Should third parties get rights to decrypt them?

We propose *APEX*: an OAuth-grounded suite of protocols which systematically augment delegated authorisation to allow refined third-party access to encrypted resources, while maintaining OAuth's behaviour for any unencrypted resources. We also provide an implementation of APEX, showing its seamless integration with OAuth.

On the formal side, we propose a generalisation of APEX (and OAuth) into a paradigm which we call *restricted authorisation delegation (RAD)*. RAD is a model that lifts formal treatment from protocol to suites; and, it also stipulates the desirable requirements that delegated authorisation schemes should attain (including to enable access over encrypted resources). We also give a formal, cryptographic model that augments existing models in multi-party authorisation, authenticated key-exchange and access control.

Finally, we use this model to prove that APEX formally attains all the properties of a restricted authorisation delegation (RAD) scheme, and discuss that OAuth 2.0 does not.



**ACM Reference Format:**
. 2023. Protocols and Formal Models for Delegated Authorisation with Server-Side Secrecy. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX


*All our artefacts are at: https://apex.anon.science/*

## 1 INTRODUCTION

*OAuth 2.0* [4] is a ubiquitous suite of web-based protocols which allows one application to borrow the user's authorisation in another to gain access to its resources. This happens with the user's approval and at the user's discretion: we say that the user *delegates* their authorisation to the first application. Meanwhile, in this realm, modern protocols are multi-app, multi-device and multi-platform (e.g., the Apple [18] and Google [35] implementations of FIDO2 passkeys) and, as such, authorisation flows must also adapt to support cross-device scenarios [19, 23] where an action started on one device (e.g., accessing a document in a desktop web browser) is authorised by software on another (e.g., a mobile app).

Such user-friendly authorisation and wide interoperability has lead to the concentration of vast amounts of user data being entrusted to just a handful of multinational corporations. And, now, this data has become increasingly valuable with the advent of Large Language Models, such as GPT4 [43], as companies in the race for Artificial General Intelligence (AGI) have strong incentives to use it as part of the training of their models [12, 25]. Aside, there is already a history of provider abuses [14, 21, 33]. Thus, users may seek more and more to have agency over how their data is used and shared.

So, we are facing the following **research question (RQ)**:

> *How can we design cloud services such that only selected parties are given access to a user's data, while still preserving modern functionalities such as cross-app data access?*

*Challenges in Our RQ: Cross-Device, Selective-Access Cloud Storage.*
**Challenge 1: Data Portability.** One solution is that data is encrypted on the user's device before being sent to the storage server, and the key never leaves the user's device. Portability between a user's multiple devices (e.g., creating a file from a laptop and reading it later from a smartphone) can be handled by allowing end-to-end encryption between these devices, as with services such as Proton Drive [2] and Tresorit [3]. However, this solution does not permit the user to access their data using third-party applications. The user would be unable to open a document from their cloud storage account in a word processor such as Microsoft Word.

**Challenge 2: Scope of Trust.** To give a third-party application access to the encrypted data in their cloud storage, a user could share their encryption key with the third party. However, scope issues arise if the user only wishes to share a subset of the files, or share different (possibly overlapping) subsets of files with different third parties, or when a certain party's access needs to be revoked. While these may be addressed by using multiple keys, a protocol is clearly needed to handle the complexities of key management and rotation.

Such an authorisation management system could be built from scratch, but it would be ideal if we could base our solution on an established, battle-tested protocol like OAuth 2.0. Unfortunately, OAuth-like protocols are not designed to work on encrypted data and are built around the provider as a trusted party, so that highly non-trivial changes are needed, in particular when dealing with authorised key transfers where the provider could act as a man in the middle.

**Challenge 3: Compatibility.** The preceeding challenges seem to be naturally resolved through homomorphic encryption [26]



*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*



ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

which would allow the third party to work on the encrypted data directly. However, this would require significant engineering effort by developers to make their applications compatible with with homomorphic encryption, if possible at all. Many forms of data processing require access to the plaintext and for others, the performance penalty may be unacceptable. Furthermore, homomorphic primitives are not included in government standards like FIPS 140-2 [*], preventing their use by government entities or contractors.

*Our Contributions.*

(1) A construction, called *APEX* (§3), which extends OAuth in seamless way to allow delegated access to encrypted resource, but remains usable by real-world web developers and end users in the modern web ecosystem.
(2) An implementation of APEX (§4), built using off-the-shelf OAuth libraries, thus demonstrating how APEX can easily fit into existing OAuth deployments to add server-side secrecy while maintaining interoperability.
(3) The introduction of *restricted authorisation delegation (RAD)* schemes (§5), protocol suites which perform generically the enhanced delegated authorisation found in APEX: that is, when the data stored on the server is a ciphertext, the delegated authorisation intentionally applies to the underlying plaintext as well.
(4) A formal cryptographic model (§6), encoding a realistic threat model of delegated authorisation protocols, and a set of security definitions (§7) reflecting the guarantees we expect from RAD schemes, whilst also adhering to a modular build-up of security notions.
(5) Formal proofs (§8) that APEX realises all RAD properties in our security model.

## 2 THE OAUTH 2.0 STANDARD

Our work is centric to OAuth 2.0 [4], which we summarise next. We first introduce some OAuth-related terminology.

### 2.1 Authorisation Protocol Terminology

In authorisation standards like OAuth, it is said that a *resource owner* (typically an end user) delegates *access* to protected *resources*, hosted on a *resource server* (the API provider), to a *client* (the API consumer) which will make resource requests on behalf of the resource owner.

By "access" it is meant that the client can perform any action on the specific resource, as allowed by the resource server, e.g., create, read, update or delete. The "client" is often itself a web server. As this can be confusing, we generally prefer to use *consumer* in the rest of the paper.

An *authorisation server* (typically a service provided by the API provider) is responsible for authenticating, and obtaining approval from, the resource owner and issues to the consumer limited short-term credentials that grant appropriate access (in OAuth, this comes in the form of an *access token*).

In many real-world deployments, the duties of the two servers (resource server and authorisation server) are performed by the same party, which is allowable under the OAuth specification. And, even when distinct entities, both servers are considered trusted

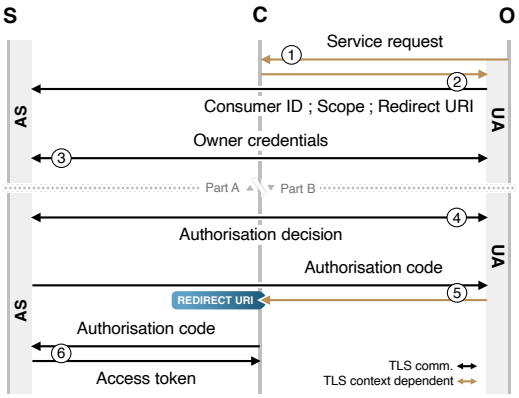


**Figure 1: OAuth 2.0 Authorisation Grant Protocol**

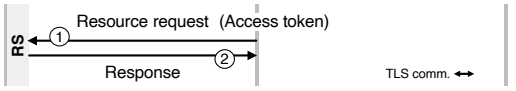


**Figure 2: OAuth 2.0 Resource Access Protocol**

at the same level. As such, we will often refer to both together as simply the *server*.

### 2.2 OAuth 2.0 Phases and Their Functions

OAuth 2.0 operates in two distinct, and somewhat independent, phases which we refer to as: (1) the *authorisation grant phase* and (2) the *resource access phase*.

In the authorisation grant phase, a consumer requests authorisation and obtains a secret from the authorisation server. In the resource access phase, the consumer uses the secret to authorise requests to the resource server. Resource access can be performed over and over, without repeating the authorisation grant phase, unless the consumer changes the scope of its access, or the authorisation server has revoked access.

The two phases are performed by the protocols detailed in the following two subsections.

### 2.3 OAuth 2.0 Authorisation Grant Protocol

The authorisation grant protocol proceeds according to the following steps, shown in Part A of Figure 1:

(1) An owner (O) with resources held at an OAuth resource server will request a service from an OAuth consumer (C) which requires access to resources held at the server.

    TLS usage: When the consumer is a remote server, it is assumed that this request is sent over TLS.

(2) The consumer will direct the owner's user agent (UA), i.e., web browser, to an authorisation endpoint on an authorisation server (AS) trusted by the resource server. With this message, the consumer also selects one of its preregistered redirect URIs, and includes the *scope* of the access requested (i.e., the actions it wishes to perform). Here the server also receives a consumer identifier, but is unable authenticate the consumer at this stage.

(3) The owner authenticates to the server by means of credentials over TLS, e.g., a password, session cookie, etc. The server is authenticated by way of X.509 certificate.

The protocol continues with Part B:

(4) If the owner has not previously authorised the consumer for the scope of access requested in (2) above, the authorisation server presents information about the authorisation request to the owner, and the owner is given the opportunity to approve it, deny it or limit its scope.

(5) The server returns a message to the owner's user agent to redirect to the redirect URI received in (2) above but modified to encode an *authorisation code* as a parameter, thereby passing this code along to the consumer.

Interprocess messaging: When the consumer is running on the resource owner's hardware (e.g., as a native application), the authorisation code is sent interprocess, typically without TLS. An extension to OAuth called *PKCE* must be used to protect the code [48].

(6) The consumer sends the received authorisation code to the server over TLS. The server replies with an *access token* and the scope, if changed from (2).

Consumer authentication: When running on a remote server, in this step, the consumer also authenticates to the server by providing pre-registered credentials.

## 2.4 OAuth 2.0 Resource Access Protocol

Execution of this phase proceeds as in Figure 2:

(1) Over TLS, the consumer (C) makes a request to the resource server (RS) to access a resource, presenting the access token previously obtained (in step 6 above).

(2) On valid requests, the server responds as follows: (a) for read access, with a message containing the resource; or (b) for create/update/delete access, it indicates whether the action has been performed (un)successfully.

# 3 THE APEX FRAMEWORK

We introduce the *Authorisation Protocol Encryption Extension (APEX)*, a framework layering end-to-end encryption (E2EE) on top of existing authorisation delegation protocols.

Note: APEX is suite of protocols that realise cryptographic aims (e.g., authorisation) as well as functional aims (e.g., seamless cross-device interoperability). To give this mixed domain, we choose not to present APEX subroutines in the common "Alice and Bob" notation, employing free-form textual explanations and accompanying diagrams instead.

## 3.1 Foundational Concepts

APEX re-imagines a number of authorisation concepts:

**Agents.** In APEX, resources are handled at the owner's end by software agents. These run solely on the resource owner's hardware and perform cryptographic operations on the owner's behalf. They are the only entities permitted to touch the plaintexts of encrypted resources.

There are two kinds of agents: a *provider agent* and a *consumer agent*, where the former is distributed by the API provider and the later by the API consumer. The provider agent has the additional responsibility of managing a set of encryption and signing keys belonging to the owner.

As the agents represent an extension of the owner and are trusted to handle sensitive data, in the next description of APEX, we treat them as part of the owner party.

**Consumers.** In OAuth, consumers communicate with the API provider and themselves access resources. But in APEX, making API calls is a separate role from accessing resources, as decryption of resources is not performed by the consumer itself but by the consumer agent.

The most versatile configuration of APEX is one in which the consumer is implemented as server-side software and this is the configuration we assume throughout the paper, although others are possible, as we discuss in Appendix C.

## 3.2 Further Design Considerations

*3.2.1 Functional requirements.* From APEX's operating context and envisioned use cases, a number of requirements arise:

a) **Mix of native and web apps:** provider and consumer agents may be implemented as browser-based applications, native applications, or as a mix of both types.
b) **Cross-device flows:** flows can start on one device and invoke a provider agent on another.
c) **Multiple agents:** a resource owner can have many multiple provider agents and, for each consumer, multiple consumer agents. Decryption and signing keys are syncable across agents (the specific mechanism is left unspecified).
d) **Agents with limited connectivity:** ideally agents should be reachable by the server and consumer at all times, however they can also be invoked by other means.
e) **Key and data recovery:** it is possible to implement solutions for key recovery if one, many or all provider agents and/or consumer agents are lost (the specific mechanism being left to the implementer).
f) **Data validation:** the API provider is able to validate data stored by the server, including data the server never sees as a result of being encrypted, by moving validation logic into the provider agent.

*3.2.2 Non-functional, non-security requirements.* Aspects related to developer experience were also considered:

a) **Ease of implementation:** the framework is straightforward for the average web developer to implement using the standard library of most languages.
b) **Practical to integrate:** easy to add to existing codebases which already use an authorisation framework.
c) **Minimised burden on consumers:** wherever possible, the burden associated with the implementation cost, required expertise and the consequences of negligence has been placed with the API provider over the API consumer.
d) **Lightweight:** the cryptography employed is commonplace for efficiency and ease of implementation

## 3.3 Trust Assumptions and Threat Model

*3.3.1 Trust assumptions.* APEX is mainly concerned with ensuring the secrecy of resources according to their owner's will; to this end, resource owners consider:

- servers: untrusted (assumed malicious)
- consumers: semi-trusted (assumed honest-but-curious)
- agents: trusted

However, the trust afforded is not absolute: consumers and consumer agents are only trusted for the resources and actions on those resources for which they have been granted access by the owner for so long as that access is valid.

*3.3.2 Informal threat model.* The outcome of these trust assumptions is a service and an API which protect the secrecy of resources from unintentional errors and even intentional subversion on the part of the API provider in the administration of their server infrastructure. Resource secrecy is also ensured against simple data leaks which affect the API consumer (e.g., SQL injection could cause exposure of a resource ciphertext persisted by the consumer but not the associated plaintext).

Note: Our threat model is as strong as is possible cryptographically without being at odds with our other requirements. Most notably, as the entire set of trusted consumer agents cannot be known by the API provider ahead of time, long-held keys are kept by the consumer which neccitates honest-but-curious corruption.

## 3.4 Construction

A developer implementing APEX may select the protocols (e.g., OAuth vs. other delegated-authorisation protocols) and cryptographic algorithms of their choice, specifically:

*3.4.1 Delegated authorisation protocols.* APEX works by enforcing access policies via the provider agents. However, most web services will also wish to enforce access policy on the server. As this is a solved problem, APEX can be composed with any delegated authorisation protocol suite which provides the following two protocols:[1]

a) **Server-mediated authorisation grant protocol:**
SMAG$(AS, C, O, scope)$ — a protocol for a consumer $C$ to request, from an authorisation server $AS$, permission to perform certain actions (indicated by $scope$) on resources of a resource owner $O$, kept on a resource server $RS$ that uses $AS$ as its authorisation server; $RS$ may equal $AS$.

Precondition: $O$ and $C$ are registered with $AS$.

Postcondition: $C$ has authorisation to perform the actions approved by the $AS$ and indicated by $scope^*$ which may or may not equal $scope$.

b) **Resource access protocol:**
RA$(RS, C, request)$ — a protocol for a consumer $C$ to request that a resource server $RS$ perform an action on a resource as encoded in $request$ using authorisation granted prior by an authorisation server $AS$.

Preconditions: (i) $AS$ is trusted by $RS$; (ii) an execution of SMAG$(AS, C, O, scope)$ has succeeded with $C$ approved for a level of access indicated by $scope^*$; (iii) $AS$ has not revoked the authorisation of $C$; and (iv) $C$ is authorised for the action indicated by $request$ on the resource as per $scope^*$.

Postcondition: $request$ is carried out by $RS$.

[1] For example, various instantiations of OAuth 2.0, GNAP [5], etc.

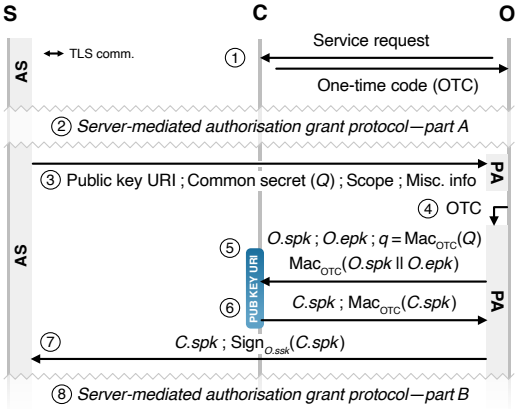


**Figure 3: APEX Dual-Mediated Authorisation Grant**

*3.4.2 Cryptographic algorithms.* APEX requires four different cryptographic schemes:

a) a **symmetric encryption scheme** Sym, consisting of three PPT algorithms (SGen, SEnc, SDec);

b) an **asymmetric encryption scheme** Asym, consisting of three PPT algorithms (AGen, AEnc, ADec);

c) a **digital signature scheme** Sig, consisting of three PPT algorithms (SigGen, Sign, SigVrfy); and

d) a **message authentication code scheme** MAC, consisting of three PPT algorithms (MacGen, Mac, MacVrfy).

## 3.5 APEX Authorisation Grant Protocol

The OAuth 2.0 authorisation grant protocol (§2.3) allows a resource owner to authorise a consumer of their choice to access a subset of their resources. In APEX, that authorisation also needs to include encryption and decryption of those resources via a provider agent trusted by the owner. However, without a central trusted authority, the consumer cannot itself discover which provider agents belong to the owner.

APEX instead uses the untrusted server to establish the initial contact with an agent. To enable the provider agent receiving the request and the consumer initiating the flow to authenticate one another, the identity of each gets bound to a *one-time code (OTC)* for the duration of the session.

This is the basis of the *APEX dual-mediated authorisation grant protocol*, shown in Figure 3. The protocol proceeds according to the following steps in which all network communication is over TLS:

(1) A resource owner (O) requests a service from a consumer (C) which requires access to the owner's resources held by a server (S). The consumer selects a *one-time code (OTC)* chosen at random.

A *request ID* ($q$) is derived from the OTC by calculating a message authenticate code (MAC) of a *common value* ($Q$)

pre-shared between the server and consumer. This is used to store the OTC and the future time at which it expires under a known identifier for later retrieval.

The consumer displays the OTC to the owner.

(2) The owner, consumer and server initiate a run of the server-mediated authorisation grant protocol (SMAG) to request access to a certain *scope*, pausing on the achievement of owner–server authentication.

(3) The server contacts one of the owner's provider agents (PA) and sends the consumer's *public key URI*, which the consumer has previously registered with the server.

If the owner has not previously authorised the consumer for the scope of access requested in (2) above, the provider agent presents information about the authorisation request to the owner, including the domain name of the consumer extracted from the public key URI. The owner approves, denies or limits the scope of the authorisation request.

(4) The agent prompts the owner for the OTC from (1).

(5) The provider agent derives the request ID ($q$) from the OTC in the same way as in (1). It then calculates a MAC of the owners's public signature verification and encryption keys ($O.spk$ and $O.epk$), using the OTC.

The agent sends all of this to the consumer's public key URI together with the aforementioned public keys.

(6) The consumer looks up the OTC from the request ID and uses this to verify the owner's public keys against the MAC. If verification succeeds, the consumer returns its public signature verification key ($C.spk$) and a new MAC calculated over the key.

(7) The provider agent verifies the received public key against the MAC and OTC. If successful, the agent signs the public key with the owner's secret signing key ($O.ssk$) and sends it to the server. The server persists the consumer's public key and the signature so that other provider agents can later obtain it if needed.

(8) The provider agent passes back control to the server and the run of the server-mediated authorisation grant protocol (SMAG) is resumed and continued until completion, ending with the owner redirected to the consumer.

On successful conclusion of the protocol, the consumer is authorised to invoke the owner's provider agents to perform certain tasks. The consumer and set of agents also now have an established relationship for future secure communication.

*On Usability of One-Time Codes.* In the above procedure, the resource owner is expected to ferry across the one-time code (OTC) from the consumer to the provider agent. If the page or window in which the OTC is first displayed to the owner closes, the owner will need to write the code down or momentarily commit it to memory. So, it recommended that the consumer initiate the server-mediated authorisation grant protocol in step 2 in a separate window. This permits visibility of the OTC while the owner interacts with the provider agent, even when both consumer and agent are being accessed on a single device. For flows initiated from a non-windowed device such as a mobile phone, the consumer may deliver the OTC in a domain-bound SMS text message [10, 34], to allow system-autofill into the provider agent on user consent.

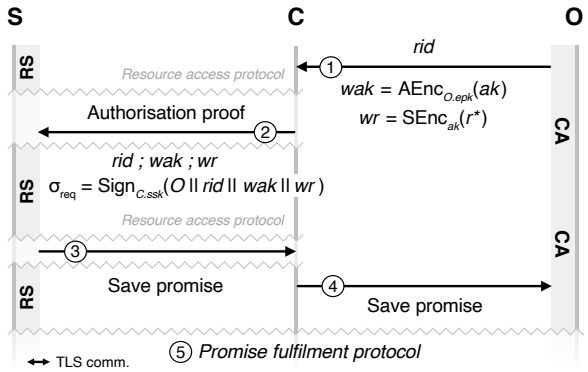


**Figure 4: APEX Resource Registration**

*Cross-Device Flows.* On redirect to the server (step 2), the server may elect to transfer control to a provider agent on a second device. Once this happens, the server may redirect back to the consumer immediately, performing step 8 without waiting for approval of the authorisation request (steps 3-7). Any tokens received (like the OAuth authorisation code) should be invalid until the server receives the consumer key in step 7. For a better experience during cross-device flows, a QR code embedding the OTP may be shown alongside the textual version. When the flow starts on a mobile device, the role of consumer and provider agent in generating the OTP may be reversed to utilise the initiating device's camera.

*Safeguards Against Consumer Impersonation.* The OTC ensures that the owner and agent approve the expected consumer and not another substituted by the server in step 3. However, this does not prevent consumer impersonation attacks in which the owner is tricked into contacting the wrong consumer from the start (in step 1). As a partial mitigation, the domain name of the consumer is displayed to the owner in step 3 and the consumer public key is obtained strictly from that domain.

## 3.6 APEX Resource Registration Protocol

The *APEX resource registration protocol* allows a consumer to use the authorisation it has been granted to create new resources or update existing resources on behalf of a resource owner. It is initiated by a consumer agent, authenticated to the consumer by some application-specific means, which has a resource plaintext it wishes to store on the server. The protocol proceeds as follows according to Figure 4, wherein all communication is over TLS:

(1) The consumer agent (CA) generates a random *agent key* ($ak$) and encrypts the resource using this key to obtain a wrapped resource ($wr$). The agent key itself is encrypted with the resource owner's public key, the result of which is referred

to as the *wrapped agent key* (*wak*). The two wrapped keys are sent to the consumer (C).

If the consumer agent wishes to update an existing resource, it also sends the identifier of the resource (*rid*).

(2) The consumer signs the received data with its secret key and submits the result ($\sigma_{\text{req}}$) along with the wrapped resource and agent key to the server (S), providing the proof of authorisation required by the resource access protocol RA (when OAuth 2.0 is used, an access token is provided as this proof).

(3) The server checks the proof of authorisation and responds with a unique ID representing a *save promise*, i.e., the server has not yet saved the resource, but will do so if certain requirements are met.

(4) The consumer delivers the save promise ID to the CA.

(5) The consumer agent fulfils the save promise with the help of the server and a provider agent. Refer to §3.8.

In a typical web service without encrypted resources, you might expect the protocol to end at step 2 with a reply indicating that the resource was saved successfully. However, the APEX encrypted resource first needs to be decrypted, checked for validity and signed by a provider agent. This is achieved by performing the *promise fulfilment* process (§3.8).

At the conclusion of promise fulfilment, the server will have stored ciphertexts $R$ and $RK$ where the former is the *encrypted resource* and the later is its *wrapped resource key*. Alongside these, signatures $\sigma_R$ and $\sigma_{RK}$ are also stored.

## 3.7 APEX Resource Retrieval Protocol

The *APEX resource retrieval protocol* allows a consumer to use the authorisation it has been granted to read resource plaintexts. It is initiated by a consumer that wishes to retrieve a resource from the server and, if encrypted, to employ a consumer agent and provider agent to decrypt it. The protocol proceeds as follows according to Figure 5, wherein all communication is over TLS:

(1) The consumer (C) requests an encrypted resource identified by *rid* from the server (S) using the resource access protocol RA.

(2) The server checks the proof of authorisation and replies with the *encrypted resource* ($R$), its *wrapped resource key* ($RK$) and signature on both of these ($\sigma_R$ and $\sigma_{RK}$).

(3) The consumer submits these to a consumer agent (CA), running on resource owner hardware and authenticated to the consumer, to retrieve a new *wrapped agent key*.

(4) The consumer agent generates a *wrapped agent key* (*wak*) by choosing an *agent key* (*ak*) at random and encrypting it with the resource owner's public key. The wrapped agent key is returned to the consumer.

(5) To decrypt the resource, the consumer initiates the *key rewrapping* process. The consumer signs the wrapped resource key together with the wrapped agent key and submits

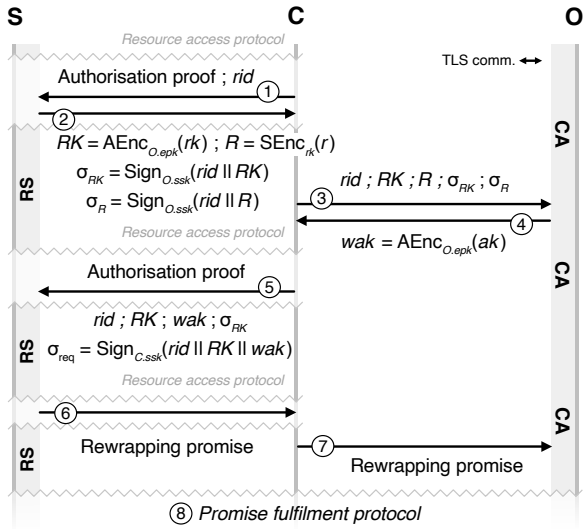


**Figure 5: APEX Resource Retrieval**

everything to the server's *key rewrapping endpoint* providing the authorisation proof required by RA.

(6) The server checks the proof of authorisation and responds with a unique ID representing a *rewrapping promise*. The consumer agent can subsequently use this to have the resource key rewrapped with the agent key it chose previously.

(7) The consumer delivers the rewrapping promise ID to the consumer agent.

(8) The consumer agent fulfils the promise with the help of the server and a provider agent. Refer to §3.8.

If regular OAuth were being used, the consumer would receive the plaintext resource immediately in step 2. In APEX, the server responds instead with an encrypted resource and key. As the key is itself encrypted with the owner's private key, the consumer can submit that to be decrypted by a provider agent. The process by which this happens (*promise fulfilment*) supports sending the key in a URI query string which may be length constrained and could be delivered over an insecure channel. It is for this reason that the resource itself is not sent alongside the key and why an ephemeral key (*ak*) is chosen by the agent: it protects the newly-decrypted key on its way back from the provider agent.

At the conclusion of promise fulfilment, the consumer agent receives a *rewrapped resource key* (*rwrk*) which it can decrypt using the *ak* chosen in step 4. The result is the *resource key* (*rk*) which can in turn be used to decrypt $R$ and obtain the plaintext resource.

## 3.8 APEX Promise Fulfilment & Resource Deletion Protocols

The APEX registration, retrieval and deletion protocols all result in, or can result in, the creation of a *promise* stored on the resource server. A promise represents a request that has been received by the server but can only be processed with the assistance of the provider agent and has yet to actually be processed, akin to promises in programming languages such as JavaScript [1]. Promises are

uniquely identifiable with a cryptographically random string ID which can later be submitted to obtain the results of the processing. This is referred to as *fulfilling the promise*.

Due to space constraints and lesser significance, we present this in Appendix B. Similarly for resource deletion, which is found in Appendix A.

## 4 IMPLEMENTATION

To validate APEX as practical for use in the real world, we have developed a demo implementation. Our aims are:

(1) show the protocols are implementable in the web context, with all its unique runtime and security characteristics;
(2) demonstrate the framework's flexibility for use with native or web apps and in single- and cross-device scenarios;
(3) establish that the framework can be applied in a typical programming environment with off-the-shelf libraries;
(4) prove that APEX can be layered overtop of an existing OAuth implementation; and
(5) obtain empirical measurements of APEX's performance characteristics relative to OAuth.

The demo implementation and its source code is available at https://apex.anon.science/.

### 4.1 OAuth 2.0 Implementation Assumptions

When we refer to "OAuth 2.0" in this paper, we assume the following implementation decisions:

- The *authorisation code grant* is used over the less secure *implicit grant*, to be deprecated in [40] and removed in the draft 2.1 version of the framework [6].
- The authorisation code is transmitted over TLS unless the consumer runs on resource owner hardware (as required by [6, 40]), in which case it may not be so long as PKCE [48] is used.
- All other communication between the owner's user agent and the consumer takes place over TLS.
- All communication between the consumer and server takes place over TLS.
- If the consumer does not run on resource owner hardware and instead runs on a remote server, the consumer is considered to be of type "confidential" and therefore the server will authenticate the consumer when an authorisation code is exchanged for an access token (per §3.2.1 of [4]).
- Whenever a redirect is performed, either HTTP code 303 is used or the navigation is achieved by user agent script, thereby eliminating Fett et al.'s [24] "307 redirect" attack.
- For consumers which interact with more than one OAuth server, when the server redirects back to the consumer, the consumer verifies that this redirect originated from the expected server tied to the current session (per [6, 40]), eliminating Fett et al.'s [24] "naive RP session integrity" attack.
- In general, all mitigations required by [40] are implemented.

**Demonstrated Use Case.** Our demonstration implements an integration between a cloud storage service (called *Cloud Drive*) and a web-based notes application (*Cloud Notes*). These communicate via REST API, with OAuth used for server-mediated authorisation. This enables notes to be saved in a user's cloud storage account.

Optionally, the user may elect to protect a note with end-to-end encryption (E2EE). In this case, APEX is used to add server-side secrecy and JavaScript-based consumer and provider agents handle the encryption/decryption in browser on behalf of the user.

We also provide a mobile application (*Cloud Drive Mobile*) which can be used in place of the in-browser provider agent. This allows the user to keep their encryption keys on their Android device, protected by the hardware-backed Android Keystore [*]. After logging into their Cloud Drive account on the mobile app, it becomes the default provider agent. Subsequent promises are sent to the app for handling via push notification.

**Technologies Used.** The web service backends have been written in Python and use Flask [7] as the web framework. Front-end UI is built with JavaScript, HTML and CSS using the Materialize [52] and Bulma [51] CSS frameworks. The mobile app is rendered in a WebView [*] and makes calls to Android system APIs using the Capacitor cross-platform development framework [*].

The Cloud Drive service acts as an OAuth 2.0 authorisation/resource server by way of the Authlib [53] Python library, and the Requests-OAuthlib [8] library is used by the Cloud Notes app to interface with the server.

**Limitations.** The user's keys are stored in the browser's localStorage [*] but isolated to the provider agent by way of the same origin policy [9] (the agent is served from a separate origin than the rest of the Cloud Drive service). The mobile implementation of the provider agent has its own set of keys. This means that notes created with one agent cannot be accessed by another. In a true implementation, the API provider would need to provide a mechanism for syncing the keys between agents, or derive the keys from the user's password using a key derivation function.

Additionally, we did not utilise the APEX feature which allows the provider agent to implement custom logic to validate a resource before it is accepted for storage, as this is of limited benefit to a simple file store.

**Performance.** To compare the performance of APEX against plain OAuth, we measured the duration to complete several register and retrieve operations[2]. For this, we repeatedly saved and opened an encrypted and unencrypted note. In the former case, we did this first using only the browser-based agents and then again using the mobile app.

We performed this test twice: over a fibre-to-the-premises (FTTP) connection (900 mbps, symmetric) and over a 5G connection in an area with poor cellular connectivity. The Cloud Drive and Cloud Notes applications were hosted on the same sever in the "Europe (London)" AWS region. Each operation was performed 10 times per connection.

The results of these tests are given below in Table 1, including combined averages across both connection types. We found a 9.03:1 APEX:OAuth ratio across all registration operations and a 10.77:1 ratio across all retrieval operations.

[2]We give no figures for the initial authorisation as the authorisation grant phase depends on user interaction and is generally undertaken once per user.

From these numbers, we can see that accessing resources using OAuth 2.0 extended with APEX takes longer than using OAuth alone, but that is to be expected given the cryptographic overheads. In the demo, the agents are visually dominant for clarity, but they could operate in the background, which would reduce the duration of these operations.

| | OAuth 2.0 | OAuth 2.0 + APEX | |
|---|---|---|---|
| | | Fully in browser | Cross device |
| **Registration** | | | |
| Fibre (a) | 34.41ms [0.46] | 235.23ms [9.24] | 287.14ms [16.47] |
| 5G (a) | 91.77ms [22.48] | 980.54ms [53.99] | 775.20ms [73.21] |
| Combined (b) | 63.09ms [32.79] | 607.89ms [374.66] | 531.17ms [249.73] |
| **Retrieval** | | | |
| Fibre (a) | 33.86ms [1.67] | 282.05ms [13.13] | 359.05ms [32.49] |
| 5G (a) | 84.78ms [22.47] | 1,012.82ms [102.89] | 901.30ms [197.18] |
| Combined (b) | 59.32ms [30.03] | 647.44ms [372.67] | 630.18ms [305.74] |

Results shown are the mean across several operations ($n$) with the standard deviation given in brackets. a. $n = 10$ b. $n = 20$

**Table 1: Performance of APEX vs. OAuth 2.0 in Our Proof of Concept Implementation**

## 5 GENERALISING APEX: THE RESTRICTED AUTHORISATION DELEGATION (RAD) SCHEME

One can generalise the refined delegated authentication in APEX, and in so doing generalise OAuth as well. We title such generic delegated authentication mechanism *restricted authorisation delegation (RAD)* scheme. Its purpose is to offer delegated authorisation whereby a resource owner can further restrict access to its resources (even to the server itself), while also allowing them to delegate limited authorisation to certain parties at their discretion. RAD-instantiating protocols no longer elicit authorisations which are fully server-reliant/mediated, but enforce that the resource owner (or a software agent acting on behalf of the owner) be part of the authorisation process. We formalise this below.

*RAD scheme.* A *restricted authorisation delegation (RAD) scheme* consists of four probabilistic polynomial time (PPT) algorithms (DMAG, RReg, RRet, RDel), each of which executes a protocol with a specific function. The algorithms in the RAD scheme are run by interactive Turing machines (ITM) [42], which we call *parties*: an authorisation server *AS*, a resource server *RS* (where *AS* and *RS* may be the same party), a resource owner *O*, and an API consumer (which may be a server or an application running on an end-user device). The RAD algorithms[3], run by these parties, are as follows:

(1) **Dual-mediated authorisation grant algorithm:**

DMAG$(AS, C, O, scope)$ — implements a request to a registered resource owner $O$ to delegate its authorisation to create, read, update and/or delete resources indicated by *scope* and stored at a server *RS* to consumer $C$; the resource owner $O$ may approve/deny/limit the request.

Postcondition: $C$ can authenticate and authorise requests to *RS* and $O$ which are allowable by the approved $scope^*$ which may or may not equal the requested *scope*.

(2) **Resource registration algorithm:**

RReg$(RS, C, O, r, rid)$ — implements a request to a server *RS* to register a resource on behalf of a consumer $C$. It requires prior approval from the owner $O$, or from the software acting on behalf of $O$, for the specific resource where $r$ is the plaintext of the resource. If a resource identifier *rid* is given, an update of the existing resource stored at $S$ under *rid* will be attempted. If not, *RS* and $O$ will (try to) create a new resource.

Precondition: An execution of DMAG$(AS, C, O, scope)$ has completed successfully with the authorisation server *AS* used by *RS*, resulting in $C$ being approved for a $scope^*$ which includes the resource $r$ identified by *rid*.

Postcondition: $r$, or a value derived from $r$ by way of a reversible operation, is registered at *RS* as belonging to $O$ under $rid^*$ which may or may not equal *rid*.

(3) **Resource retrieval algorithm:**

RRet$(RS, C, O, rid)$ — implements a request to a server *RS* to return a resource $r$ identified by *rid* to a consumer $C$. There may be intermediate processing by owner $O$, or by a software acting on behalf of $O$, if required for the specific resource.

Precondition: Same as above.

Postcondition: $C$, or software deployed by $C$ on hardware under the control of $O$, has obtained $r$.

(4) **Resource deletion algorithm:**

RDel$(RS, C, O, rid)$ — implements a request to a server *RS* on behalf of a consumer $C$ to delete a resource $r$ identified by *rid*. Approval from owner $O$ may be required.

Precondition: Same as above.

Postcondition: $r$ is no longer registered at *RS*.

## 6 A FORMAL MODEL FOR RAD

Our formalism for RAD is given in the style of Bellare-Rogaway formalisms [13] (i.e., session-based, with notions of partnering and freshness). On top, we give game-based security definitions in line with the latest trend for OAuth [39].

Note: As per usual, all our measures of probability and time complexity are asymptotic in a security parameter $s$.

***Parties.*** The set of *RAD-suite parties* $\mathcal{P}$, or simply *parties*, is partitioned into three subsets: servers $\mathcal{S}$, consumers $\mathcal{C}$ and resource owners $\mathcal{O}$. Variables $n_{\mathcal{P}}$, $n_{\mathcal{S}}$, $n_{\mathcal{C}}$ and $n_{\mathcal{O}}$ represent the number of parties, servers, consumers and owners.

Each owner is linked to a server: i.e., each $O \in \mathcal{O}$ is in fact $O_S$ where $S \in \mathcal{S}$. This follows real life: one user may have accounts with different storage providers (e.g., Google Drive and Dropbox) but would have different credentials for each, and so would appear to the providers as separate owners.

Consumers do not operate in the same way: one consumer can potentially communicate with multiple servers and might share authentication credentials between them, or not.

[3]We describe these algorithms just in intuitive terminology, without formalising matters irrelevant to our formal model, e.g., resources' storage.

**Attributes common to all parties.** Each party $P \in \mathcal{P}$ is a stateful PPT algorithm, with the following attributes:

- the party's **resource encryption secret and public keys**, $P$.esk and $P$.epk; set to $\perp$ if $P \in \mathcal{S}$;
- a map of **other parties' public encryption keys**, $P$.epk′, which the party has knowledge of and which contains pairs $(Q,\ epk_Q)$ where $Q \in \mathcal{P}$ and $epk_Q$ is the public key for $Q$ obtained by $P$;
- the party's **signing secret and public keys**, $P$.ssk and $P$.spk; set to $\perp$ if $P \in \mathcal{S}$;
- a list of **credentials**, $P$.authz, containing tuples $(A, B, cred, from, until, scope, aux)$, as follows:

  (1) $A$ is the party at which actions are authorised; (2) $B$ is the party for which actions are authorised on behalf of, which may be set to $\perp$; (3) *cred* is the credential used to prove authorisation and could be a shared secret, a secret signing key, or something else; (4) *from* is an immutable timestamp representing the moment the credential was first usable by $P$; (5) *until* is a timestamp representing the moment the credential is no longer valid and can only be set to a future time, the current time to expire it right away or $\perp$ to make it valid indefinitely; (6) *scope* is a value which encodes what actions the credential is valid for; (7) *aux* is any auxiliary data about the credential.
- a list of **verifiers**, $P$.authz′, containing tuples $(A, B, vrfr, from, until, scope, aux)$, as follows:

  (1) $A$ is the party whose actions are authorised at $P$; (2) $B$, *from*, *until*, *scope* and *aux* are the same as for $P$.authz above; (3) *vrfr* is the verifier confirming a valid proof of authorisation was presented, i.e., a shared secret, a signature verification key, or something else.

If $P$.authz or $P$.authz′ has an entry where $scope = \perp$, that credential/verifier does not authorise actions on resources. Instead, it is used for authentication only.

**Server attributes.** Each server party $S \in \mathcal{S}$ also keeps:

- a list of **registered resources**, $S$.res, containing tuples $(O, rid, R)$ where $O \in \mathcal{O}$, $rid$ is a unique resource identifier, and $R$ is the raw resource data stored by $S$.

For encrypted resources, $R$ is the ciphertext associated with a plaintext $r$ plus any auxiliary data required by the protocol suite. For unencrypted resources, $R$ is the resource plaintext.

**Owner attributes.** Each owner party $O \in \mathcal{O}$ sets:

(1) a list of **cached resources**, $O$.cache, for which $O$ has knowledge of the plaintext, containing tuples $(R, r)$ where $R$ is the encrypted resource and $r$ is its plaintext; (2) the **server**, $O$.srv $\in \mathcal{S}$, at which the owner is registered.

***Executing Parties (over Phases): Instances.*** The executions of the algorithms in a RAD scheme form the *phases*: authorisation grant, resource registration, resource retrieval and resource deletion. Each RAD party may be running one or several phases, at one given point. An execution of one phase by a party is referred to as a *phase instance* (or simply *instance*), dubbed $\pi_P^i$: $P$ is the executing party and $i$ refers to the $i$th execution over all phase instances of the party $P$. We use $P.n_\pi$ for the total number of instances for $P$ and, generally, we write $i \in [1, P.n_\pi]$ for an instance $i$.

**Instance attributes** are:

- the **phase type**, $\pi_P^i$.type, fixed to one of {"auth", "reg", "ret", "del"} indicating the specific phase the instance has been executing;
- the **status**, $\pi_P^i$.status, set to "active" if the instance is still executing, "success" or "fail" after it terminated in a success/error state;
- the **start time**, $\pi_P^i$.start, at which $\pi_P^i$'s execution began;
- the **end time**, $\pi_P^i$.end, at which $\pi_P^i$'s execution stopped;
- the **credential index**, $\pi_P^i$.authzid, set to the index of the list item added to $P$.authz by the phase instance, if any;
- the **verifier index**, $\pi_P^i$.authzid′, set to the index of the list item added to $P$.authz′ by the phase instance, if any;
- a list of **authorisation proofs**, $\pi_P^i$.proof$^{\leftrightarrow}$, sent or received in the phase instance to either authorise a request or allow verification of a request's authorisation, where each proof could be a shared secret, a signature, etc.
- the list of **resources**, $\pi_P^i$.R$^{\leftrightarrow}$, either encrypted or unencrypted and sent or received in the phase instance when $\pi_P^i$.type $\in$ {"reg", "ret", "del"}; and
- the list of **resource identifiers**, $\pi_P^i$.rid$^{\leftrightarrow}$, sent or received in the instance when $\pi_P^i$.type $\in$ {"reg", "ret", "del"}.

When $\pi_P^i$.type = "reg" and a resource is being created rather than updated, the first resource identifier sent/received in the phase instance is empty such that $\pi_P^i.\text{rid}_1^{\leftrightarrow} = \perp$.

***Channels.*** In a RAD scheme, we use *channels* to model instances sending and receiving messages to one another: $\pi_P^i.\Pi_j$ denotes the $j$-th channel for instance $\pi_P^i$ with $j \in [1, \pi_P^i.n_\Pi]$, where $\pi_P^i.n_\Pi$ is the total channel count for $\pi_P^i$.

We consider cryptographically insecure channels, as well as channels secured with an *authenticated and confidential channel establishment (ACCE)* [31] protocol (such as TLS). In RAD schemes, ACCE channels can use either unilateral or mutual authentication. This is detailed in Appendix D.

***Utility Functions.*** To describe the model, we use a set of helper or *utility* functions associated to instances, and their attributes. These are generally self-explained, e.g., : ProtectRes$(epk, ssk, r)$ will encrypt a resource $r$ with public key $epk$ and sign it using secret key $ssk$. Inversely, ExtractRes$(esk, spk, R)$ decrypts resource $R$ with secret key $esk$ and verifies it using public key $spk$. A complete and descriptive list of utility functions is given in in Appendix D.

***Adversarial Model.*** As per the usual, the execution of (instances of) parties are simulated by the interactions between an unbounded algorithm called *challenger* and probabilistic polynomial time (PPT)

algorithm called *adversary*. The adversary can corrupt up to a polynomial number of parties, act as a legitimate party in the communication flows, and interfere in all communications; as per the normal, these abilities is modelled via *oracles*.

The oracles[4] most pertinent to the notions and security definitions discussed subsequently are given below:

- CommCorrupt($P$):
  Returns $P$.csk used to secure ACCE communications.
- ResCorrupt($P$):
  Returns ($P$.esk, $P$.ssk) used to secure resources.
- DataReveal($P$):
  Returns the values of the party's attributes, including data for all phase instances of the party, and decrypts any encrypted data using any and all keys in the party's possession.

  Excludes any secrets (including secret keys), all credentials in $P$.authz, any verifiers in $P$.authz′ which are shared secrets with another party, and proofs which are also a credential or verifier for the party such that $\pi_P^i.\text{proof}_j^{\leftrightarrow} = P.\text{authz}_m$ or $\pi_P^i.\text{proof}_j^{\leftrightarrow} = P.\text{authz}'_n$ for any $i, j, m, n \in \mathbb{N}$.
- GetData($P$):
  Returns all data for the party like DataReveal($P$) but does not perform any decryption of data.

  If $P \in \mathcal{O}$ then this query excludes $O$.cache.

The full list of oracles can be found in Appendix D.

To give our notions of partnering (in §6) and freshness (in §6), we first introduce the supporting notions of "message Authentication", "authorisation origination" and "authorisation credential and verifier matching", below.

**Message Authentication**. For an authenticator $t$ to *authenticate a message* $m$ to a party $P$ as being authored by another party $Q$, party $P$ must have knowledge of a public/secret key used to verify $t$ against $m$ (e.g., if a signing/MAC-ing key is present in $P$.authz′). This is formally defined in Appendix D.

**Authorisation Origination**. Since a party receives authorisation to perform actions at another, we keep record of the credential and verifier which facilitate that authorisation get as a result of two *different* phase instances (of different parties). We say that a credential/verifier *originates* in the phase instance that causes it to be added to the authz or authz′ lists. This is formally defined in Appendix D.

**Authorisation Credential and Verifier Matching**.If a credential can be used to produce a proof which is verifiable by a particular verifier, we say that the credential *matches* the verifier. This is formally defined in Appendix D.

***Instance Partnering***. Different phase instances from distinct parties execute in concert with one another to form a complete phase execution across parties. These instances are considered *partnered* with one another. Because the instance-partnering conditions differs based on phases which makes the formal definition somewhat convoluted and deferred to t Appendix E. The intuition of instances' partnering is: (a) During the authorisation grant phase, two instances are partnered when there is a credential originating in one which has a matching verifier that originates in the other; (b) During the resource registration, retrieval and deletion phases, instances are partnered when they act on the same resource.

***Freshness***. In line with our trust assumptions in §3.3, we exclude attacks which require the corruption of an owner or a consumer but still permit attacks in which a consumer leaks all its data to the attacker. To put this formally, we define *fresh* instances.

**Definition 1.** For some $P \in \mathcal{P}$ and $i \in [1, P.n_\pi]$, an instance $\pi_P^i$ is considered *fresh* iff:

(1) the instance has terminated successfully, i.e., $\pi_P^i.\text{status}$ = "success";
(2) there exists at least one $\pi_Q^j$ for some $Q \in \mathcal{P}$, $j \in [1, Q.n_\pi]$ such that $\pi_P^i$ and $\pi_Q^j$ are partnered;
(3) no ResCorrupt($X$) or CommCorrupt($X$) query has been made for any party $X \in \mathcal{C} \cup \mathcal{O}$, where $X = P$ or where $X$ has an instance $\pi_X^m$ partnered with $\pi_P^i$, for some $m \in [1, X.n_\pi]$; and
(4) no DataReveal($O$) query has been made for any $O \in \mathcal{O}$, where $O = P$ or where $O$ has an instance $\pi_O^n$ partnered with $\pi_P^i$, for some $n \in [1, O.n_\pi]$.

The implication of this definition is that, for any $a \in [1, S.n_\pi]$, $b \in [1, C.n_\pi]$, $c \in [1, O.n_\pi]$, instances $\pi_S^a$, $\pi_C^b$ or $\pi_O^c$ can only be considered fresh if the adversary $\mathcal{A}$ limits itself to queries on $S$, $C$ and $O$ as given in Table 3 in Appendix D.

# 7 SECURITY IN THE RAD MODEL

## 7.1 RAD Game Setup

All our security definitions rely on the challenger *Ch* first setting up the environment by running Algorithm 1:

**Algorithm 1** Setup$_{\mathcal{A},\gamma}$ for RAD games

1: Create parties $\mathcal{P}$ ▷ *creates a polynomial number $n_\mathcal{S}$, $n_\mathcal{C}$, $n_\mathcal{O}$ of server, consumer and owner parties*
2: **for each** $P \in \mathcal{P}$ **do**
3: Initialise attributes on $P$
▷ *generates keys $P.ssk$, $P.spk$, $P.esk$, $P.epk$, $P.csk$ and $P.cpk$*
4: **if** $\gamma$ is a resource access protocol **then**
5: *2**if** $P \in \mathcal{C}$ **then** Set values for $P$.authz
6: *2**if** $P \in \mathcal{S} \cup \mathcal{O}$ **then** Set values for $P$.authz′
▷ *lines 5-6 cause consumers to be pre-authorised to access the resources of a subset of owners on a subset of servers*
7: **if** $\gamma$ is a retrieval or deletion protocol **and** $S \in \mathcal{S}$ **then**
8: Create a no. of resources and add them to $S$.res
▷ *such that some owners $O \in \mathcal{O}$ have encrypted resources pre-registered on some servers*
9: Set constant $\epsilon$ to contain a list of all resources
▷ *$\epsilon$ contains triples $(S, O, rid, R)$ where $R$ is a resource for a particular owner $O$ stored at the given server $S$ and identified by $rid$*
10: Give $\mathcal{A}$ a handle to $\epsilon$ and access to all oracles

[4]In the oracles given here, for ease, the bookkeeping by the challenger in the attributes of the various parties, instances and channels is implicit.

## 7.2 Authorisation Grant (AG) Security

In protocol suites like OAuth, the server is trusted to authenticate the consumers and enforce the correct authorisation policy. In RAD schemes, this trust is removed, and the owners enforce the authorisation policy, so they need to perform mutual authentication directly with the consumers. Crucially, if a malicious server were to subvert this mutual authentication, it could impersonate the consumer and access plaintexts, violating the secrecy requirements of RAD schemes. We introduce *authorisation grant (AG) security* to formalise these.

Our AG security notion, to follow, means to say this: an adversary should not be able, after manipulating runs of a dual-mediated authorisation grant protocol, to cause an owner to accept authentication presented by the adversary as if it were a legitimate consumer, or vice versa. This is formalised, based on the AG game between the adversary $\mathcal{A}$ and the challenger *Ch* given in Algorithm 2, via the following definitions:

**Definition 2 (AG game's winning condition).** The adversary $\mathcal{A}$ *wins the AG game* $\mathrm{G}^{\mathrm{AG}}_{\mathcal{A},\gamma}$ when:

(1) there exist two partnered phase instances $\pi^i_C$ and $\pi^j_O$ of some consumer $C \in \mathcal{C}$ and owner $O \in \mathcal{O}$ such that $i \in [1, C.n_\pi]$ and $j \in [1, O.n_\pi]$;
(2) the authenticator $t^*$ authenticates the message $m^*$ to party $P$ as originating with party $Q$, where $P = C$, $Q = O$ or $P = O$, $Q = C$; and
(3) $\pi^i_C$ and $\pi^j_O$ are fresh.

**Definition 3 (AG security).** A RAD dual-mediated authorisation grant protocol $\gamma$ is secure w.r.t. *authorisation grant security* (*AG secure*) iff, for all PPT adversaries $\mathcal{A}$:

$$\Pr\left[\mathcal{A} \text{ wins } \mathrm{G}^{\mathrm{AG}}_{\mathcal{A},\gamma}\right] \leq negl(s)$$

where *negl* is a negligible function in our security parameter $s$ (i.e., the inverse of an exponential in $n_\mathcal{S}, n_\mathcal{C}, n_\mathcal{O}$ and the number of queries by the adversary) and the probability is considered over all random coins in the game $\mathrm{G}^{\mathrm{AG}}_{\mathcal{A},\gamma}$.

## 7.3 Encrypted Resource Authorisation (ERA) Security

An adversary should not be able to exceed its authorisation to read/modify a resource plaintext. This is formalised by the ERA game given in Algorithm 3 and the definitions below:

**Definition 4 (ERA game's winning condition).** The adversary $\mathcal{A}$ *wins the ERA game* $\mathrm{G}^{\mathrm{ERA}}_{\mathcal{A},\gamma}$ when:

(1) there exists an entry $(O, rid, R)$ in $S$.res for some $S \in \mathcal{S}$ and $O \in \mathcal{O}$ such that $r^* = \mathsf{ExtractRes}(O.\mathsf{esk}, \perp, R)$;
(2) at least one instance of $S$ has sent/received $R$ such that a set $\Phi$ can be constructed as follows:
$\Phi = \{\ \pi^x_S \mid x \in [1, S.n_\pi] \wedge \pi^x_S.\mathrm{res}^{\leftrightarrow}_y = R \text{ for all } y \in \mathbb{N}\}$; and
(3) all $\pi \in \Phi$ are fresh.

**Definition 5 (ERA security).** A RAD protocol or suite $\gamma$ is secure w.r.t. *encrypted resource authorisation* (ERA secure) iff, for all PPT adversaries $\mathcal{A}$:

$$\Pr\left[\mathcal{A} \text{ wins } \mathrm{G}^{\mathrm{ERA}}_{\mathcal{A},\gamma}\right] \leq negl(s)$$

where *negl* is a negligible function in $s$ and the probability is considered over all random coins in the game $\mathrm{G}^{\mathrm{ERA}}_{\mathcal{A},\gamma}$.

We note that ERA also (non-obviously) covers attacks in which a legitimate party is able to exceed their authorisation. If a server or consumer can obtain a resource plaintext, the adversary can access it without violating the freshness definition. If an owner instance were to obtain a resource plaintext, the instance would not be considered partnered with the instances of the server where it is stored (because the soundness principle would be violated) and again the adversary would be able to use an oracle to access it and win the game.

Finally, unauthorised deletion is not a violation of ERA security. This is because deletion cannot be limited by cryptographic means and so the untrusted server is ultimately in control of what resources get deleted. Corruption of a resource by the adversary is considered a form of deletion.

**Algorithm 2** Authorisation Grant (AG) Game $\mathrm{G}^{\mathrm{AG}}_{\mathcal{A},\gamma}$
for a RAD Dual-Mediated Authorisation Grant Protocol $\gamma$

1: The challenger *Ch* runs $\mathsf{Setup}_{\mathcal{A},\gamma}$
2: $\mathcal{A}$ makes a polynomial number of oracle queries
3: $\mathcal{A}$ outputs a tuple $(m^*, t^*)$ containing a message and an authenticator

**Algorithm 3** Enc. Resource Authorisation (ERA) Game $\mathrm{G}^{\mathrm{ERA}}_{\mathcal{A},\gamma}$
for a RAD resource registration, retrieval or deletion protocol $\gamma$, or an entire RAD scheme $\gamma$

1: The challenger *Ch* runs $\mathsf{Setup}_{\mathcal{A},\gamma}$
2: $\mathcal{A}$ makes a polynomial number of oracle queries
3: $\mathcal{A}$ outputs a resource $r^*$

# 8 FORMAL SECURITY ANALYSIS OF APEX

*Instantiating RAD as APEX.* Consider the APEX protocol suite $\alpha$:

$$\alpha = (\mathsf{SMAG}, \mathsf{RA}, \mathsf{Sym}, \mathsf{Asym}, \mathsf{Sig}, \mathsf{MAC}, \mathsf{OTCGen})$$

where SMAG is the server-mediated authorisation grant protocol, RA is the resource access protocol, and Sym, Asym, Sig and MAC are the encryption, signature and MAC schemes. OTCGen is the algorithm used to generate one-time codes uniformly at random. APEX allows SMAG and RA to be fulfilled by any authorisation protocol suite but, for our proofs, we will fix them to be the OAuth 2.0 protocols from §2. No specific cryptographic schemes are assumed. So, the APEX protocol suite $\alpha$ is an instance of RAD scheme $\mathsf{RAD}_\alpha$:

$$RAD_\alpha = (\mathsf{DMAG}_\alpha, \mathsf{RReg}_\alpha, \mathsf{RRet}_\alpha, \mathsf{RDel}_\alpha)$$

where $\mathsf{DMAG}_\alpha$, $\mathsf{RReg}_\alpha$, $\mathsf{RRet}_\alpha$ and $\mathsf{RDel}_\alpha$ are the protocols described in §3 and instantiated with configuration $\alpha$.

*RAD Security of APEX.* The security of APEX is established by proving the security of each RAD phase and composing the result, as below:

**Theorem 1.** If an APEX protcol suite $\alpha$ uses:

a) EUF-CMA secure MAC and signature schemes;
b) IND-EAV secure symmetric encryption;
c) IND-CPA secure asymmetric encryption; and
d) one-time codes (OTCs) in an exponential space in the security parameter

then the APEX protocol suite $RAD_\alpha$ is ERA secure.

The above theorem is proven via the following lemmas:

**Lemma 1 (Authorisation grant security).** The dual-mediated authorisation grant protocol $\mathsf{DMAG}_\alpha$ in an APEX suite $RAD_\alpha$ is AG secure, if it uses:

a) an EUF-CMA secure MAC scheme; and
b) one-time codes (OTCs) in an exponential space in the security parameter.

**Lemma 2 (Resource access security).** The resource registration protocol $\mathsf{RReg}_\alpha$, resource retrieval protocol $\mathsf{RRet}_\alpha$ and resource deletion protocol $\mathsf{RDel}_\alpha$ in an APEX suite $RAD_\alpha$ are each independently ERA secure if each uses:

a) an EUF-CMA secure signature scheme;
b) an IND-EAV secure symmetric encryption; and
c) an IND-CPA secure asymmetric encryption.

All proofs can be found in Appendix G.

*APEX Security Compared to OAuth.* OAuth 2.0 has previously been analysed in a computational model by authors Li *et al.* [39]. RAD authorisation grant (AG) security is roughly comparable to their notion of "3P-ASD" security. Similarly, ERA security and their "authorisation" security are analogous.

Our models are different enough that these properties do not lend themselves to easy translation from one to the other. However, our RAD properties are defined in the presence of a considerably stronger adversary. This is summarised in Table 2. We also show, in Appendix F, how OAuth falls short of meeting the RAD security properties.

| Security Property | OAuth | OAuth + APEX |
|---|---|---|
| Li *et al.*'s "3P-ASD" security | ✓ | |
| Li *et al.*'s "authorisation" security | ✓ | |
| RAD authorisation grant (AG) security | | ✓✓ |
| RAD encrypted res. access (ERA) security | | ✓✓ |

**Table 2: Security Properties of OAuth vs. APEX**

*Complexity of the RAD Formal Model & Implications of the Security Results.* Due to space constraints, only in Appendix **??**, we discuss in laymen terms the intricacies of our model and the meaning of our formal security results.

## 9 RELATED WORK

**Server-Side Secrecy in OAuth-like Systems.** Two systems currently limit the sharing of end-to-end encrypted data: AAuth [50], an authorisation protocol drawing inspiration from OAuth, and Droplet [49], an authorisation scheme for IoT devices. Relying on attribute-based encryption and a blockchain respectively, neither of these proposals meet our requirements of ease of implementation and simple integration into existing codebases, which we provably realise. Other attempts to improve the security of cloud-stored data include: Sporc [22], CryptDB [46], SUNDR [38] and CloudProof [45]. None of these handle sharing of data to entities unconnected to the service provider, as we do.

**Formal Treatments of Authorisation Protocols.** Li *et al.* [39] presented a cryptographic, security model for OAuth 2.0. We give a substantially more complex model, catering for multi-phase suite that generalises OAuth in several ways, intricate session interleaving, including cross-phase notions and attacks. Our security properties are also more involved that in [39]. In more detail, Li *et al.* only consider authorisation pertaining to consumers; the server and owner are deemed to have full authorisation to all resources. Meanwhile, APEX/RAD needs to capture the following: (1) servers are no longer afforded this access, (2) consumers are given only limited access, and (3) consumers can have different authorisation levels and exceeding this level constitutes a valid attack. Because Li *et al.* [39] ignore these, they coarsely substitute authorisation with authentication, whereas we model authorisation directly. So, Li *et al.*'s models are not easily adaptable to RAD proofs.

Simulation-based models for OAuth, based on UC [15], also exist: [16] for OAuth Core 1.0 and [17] for OAuth 2.0. Similarly, Dolev-Yao [20] verification for OAuth Core, OAuth 1.0 & 2.0 exists [11, 24, 28–30, 32, 36, 37, 44], but these models are not of direct interest.

The Grant Negotiation and Authorisation Protocol (GNAP) [5] is a new authorisation framework, consolidating various OAuth standards [41, 47, 48]. GNAP was recently analysed formally in [27].

## 10 CONCLUSIONS

We extended the concept of *delegated authorisation* from OAuth and related protocols, to include resistance to malicious servers, resulting in a formalism called *restricted authorisation delegation (RAD)* and instantiated in a protocol suite called APEX. APEX builds on the principles of OAuth 2.0 to allow a resource owner (user) to keep resources (files or other data) encrypted on an untrusted server (e.g., a cloud storage service), whereby only the owner and the API consumers they authorise (e.g., a word processor) can decrypt and act on them. Our implementation of APEX leverages the same web-based environments, infrastructures and libraries as OAuth and exhibits APEX's new features, whilst maintaining interoperability with existing systems and having performance comparable to that of plain OAuth. We also have proven APEX secure in the RAD model. All this is in line with our engagements with the broader community working to improve the security of RAD-like schemes, e.g., OAuth, OpenID Connect, GNAP, etc.

*To see the artefacts which are already available at submission time, please visit*
*https://apex.anon.science/*

# REFERENCES


[1] [n. d.]. Promise - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[2] [n. d.]. Proton Drive: Free Encrypted Cloud File Storage & Sharing. https://proton.me/drive.

[3] [n. d.]. Tresorit: Secure Cloud Storage. https://tresorit.com/individuals.

[4] 2012. *The OAuth 2.0 Authorization Framework*. RFC 6749. IETF. https://datatracker.ietf.org/doc/html/rfc6749.

[5] 2022. *Grant Negotiation and Authorization Protocol*. Internet-Draft draft-ietf-gnap-core-protocol-10. IETF. https://datatracker.ietf.org/doc/html/draft-ietf-gnap-core-protocol-10.

[6] 2022. *The OAuth 2.1 Authorization Framework*. Internet-Draft draft-ietf-oauth-v2-1-06. IETF. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-06.

[7] 2023. Flask. Pallets. https://github.com/pallets/flask.

[8] 2023. Requests-OAuthlib. requests. https://github.com/requests/requests-oauthlib.

[9] 2023. Same-Origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[10] Apple Developer. [n. d.]. *Enabling AutoFill for domain-bound SMS codes*. https://developer.apple.com/documentation/security/one-time_codes/enabling_autofill_for_domain-bound_sms_codes (https://archive.today/ZbcGZ).

[11] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Discovering Concrete Attacks on Website Authorization by Formal Analysis1. *Journal of Computer Security* 22, 4 (April 2014), 601–657. https://doi.org/10.3233/JCS-140503 https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/JCS-140503.

[12] Lejla Batina, Shivam Bhasin, Jakub Breier, Xiaolu Hou, and Dirmanto Jap. 2022. On implementation-level security of edge-based machine learning models. In *Security and Artificial Intelligence: A Crossdisciplinary Approach*. Springer, 335–359.

[13] Mihir Bellare and Phillip Rogaway. 1994. Entity Authentication and Key Distribution. In *Advances in Cryptology — CRYPTO' 93*, Douglas R. Stinson (Ed.). Vol. 773. Springer Berlin Heidelberg, Berlin, Heidelberg, 232–249. https://doi.org/10.1007/3-540-48329-2_21 http://link.springer.com/10.1007/3-540-48329-2_21.

[14] Rich Brown. 2012. SkyDrive Content Restrictions among the Toughest in the Cloud. *CNET* (Aug. 2012). https://www.cnet.com/tech/services-and-software/skydrive-content-restrictions-among-the-toughest-in-the-cloud/.

[15] R. Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, Newport Beach, CA, USA, 136–145. https://doi.org/10.1109/SFCS.2001.959888 https://ieeexplore.ieee.org/document/959888/.

[16] Suresh Chari and Charanjit Jutla. 2009. *Universally Composable Web Security Protocols for Delegation*. Research Report RC24856. IBM. https://dominoweb.draco.res.ibm.com/b0d33665257dd3a0852576410043bcdd.html.

[17] Suresh Chari, Charanjit Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. https://eprint.iacr.org/2011/526.

[18] Garrett Davidson. 2022. Meet Passkeys. https://developer.apple.com/videos/play/wwdc2022/10092/.

[19] William Denniss, John Bradley, Michael B. Jones, and Hannes Tschofenig. 2019. *OAuth 2.0 Device Authorization Grant*. RFC 8628. IETF. https://www.rfc-editor.org/info/rfc8628.

[20] D. Dolev and A. Yao. 1983. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory* 29, 2 (March 1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650 http://ieeexplore.ieee.org/document/1056650/.

[21] Emily Dreyfuss. 2018. Was It Ethical for Dropbox to Share Customer Data with Scientists? *Wired* (July 2018). https://www.wired.com/story/dropbox-sharing-data-study-ethics/.

[22] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. 2010. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. USENIX, Vancouver, BC, Canada. https://www.usenix.org/conference/osdi10/sporc-group-collaboration-using-untrusted-cloud-resources.

[23] G. Fernandez, F. Walter, A. Nennker, D. Tonge, and B. Campbell. 2021. *OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0*. Technical Report. OpenID Foundation. https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html.

[24] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna Austria, 1204–1215. https://doi.org/10.1145/2976749.2978385 https://dl.acm.org/doi/10.1145/2976749.2978385.

[25] Niv Haim, Gal Vardi, Gilad Yehudai, Ohad Shamir, and Michal Irani. 2022. Reconstructing training data from trained neural networks. *Advances in Neural Information Processing Systems* 35 (2022), 22911–22924.

[26] Shai Halevi. 2017. *Homomorphic Encryption*. Springer International Publishing, Cham, 219–276. https://doi.org/10.1007/978-3-319-57048-8_5

[27] Florian Helmschmidt. 2022. *Security Analysis of the Grant Negotiation and Authorization Protocol*. Ph. D. Dissertation. University of Stuttgart, Stuttgart, Germany. https://elib.uni-stuttgart.de/bitstream/11682/12220/1/Security_Analysis_GNAP.pdf.

[28] Yating Hsu and David Lee. 2010. Authentication and Authorization Protocol Security Property Analysis with Trace Inclusion Transformation and Online Minimization. In *The 18th IEEE International Conference on Network Protocols*. IEEE, Kyoto, Japan, 164–173. https://doi.org/10.1109/ICNP.2010.5762765 http://ieeexplore.ieee.org/document/5762765/.

[29] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11, 2 (April 2002), 256–290. https://doi.org/10.1145/505145.505149 https://dl.acm.org/doi/10.1145/505145.505149.

[30] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. 2000. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the International Conference on Software Engineering*. Limerick, Ireland.

[31] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. 2017. Authenticated Confidential Channel Establishment and the Security of TLS-DHE. *Journal of Cryptology* 30, 4 (Oct. 2017), 1276–1324. https://doi.org/10.1007/s00145-016-9248-2 http://link.springer.com/10.1007/s00145-016-9248-2.

[32] K. S. Jayasri, K. P. Jevitha, and B. Jayaraman. 2018. Verification of OAuth 2.0 Using UPPAAL. In *Social Transformation – Digital Way*, Jyotsna Kumar Mandal and Devadatta Sinha (Eds.). Vol. 836. Springer Singapore, Singapore, 58–67. https://doi.org/10.1007/978-981-13-1343-1_7 http://link.springer.com/10.1007/978-981-13-1343-1_7.

[33] Joel Khalili. 2022. Google Drive Is Locking Some People's Files for No Reason. *TechRadar* (Jan. 2022). https://www.techradar.com/news/google-drive-is-locking-some-peoples-files-for-no-reason.

[34] Eiji Kitamura. 2019. *Verify phone numbers on the web with the WebOTP API*. Chrome for Developers. https://developer.chrome.com/docs/identity/web-apis/web-otp (https://archive.today/mN9Wo).

[35] Eiji Kitamura. 2022. A Path to a World without Passwords. https://io.google/2022/program/e3bb37a4-2723-4d72-a5b3-1a23abb94ac0/.

[36] Apurva Kumar. 2011. Model Driven Security Analysis of IDaaS Protocols. In *Service-Oriented Computing*, Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato (Eds.), Vol. 6470. Springer Berlin Heidelberg, Berlin, Heidelberg, 312–327. https://doi.org/10.1007/978-3-642-25535-9_21 http://link.springer.com/10.1007/978-3-642-25535-9_21.

[37] Apurva Kumar. 2012. Using Automated Model Analysis for Reasoning about Security of Web Protocols. In *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*. ACM Press, Orlando, Florida, 289. https://doi.org/10.1145/2420950.2420993 http://dl.acm.org/citation.cfm?doid=2420950.2420993.

[38] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. 2003. Secure Untrusted Data Repository (SUNDR):. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI '04)*. USENIX, San Francisco, CA, 121–135. https://doi.org/10.21236/ADA445862 https://www.usenix.org/conference/osdi-04/secure-untrusted-data-repository-sundr.

[39] Xinyu Li, Jing Xu, Zhenfeng Zhang, Xiao Lan, and Yuchen Wang. 2020. Modular Security Analysis of OAuth 2.0 in the Three-Party Setting. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 276–293. https://doi.org/10.1109/EuroSP48549.2020.00025 https://ieeexplore.ieee.org/document/9230361/.

[40] Torsten Lodderstedt, John Bradley, Aney Labunets, and Daniel Fett. 2022. *OAuth 2.0 Security Best Current Practice*. Internet Draft draft-ietf-oauth-security-topics-21. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics-21.

[41] Torsten Lodderstedt, Brian Campbell, Nat Sakimura, Dave Tonge, and Filip Skokan. 2021. *OAuth 2.0 Pushed Authorization Requests*. Request for Comments RFC 9126. Internet Engineering Task Force. https://doi.org/10.17487/RFC9126 https://datatracker.ietf.org/doc/rfc9126.

[42] Goldreich Oded. 2009. *Foundations of Cryptography: Volume 2, Basic Applications* (1st ed.). Cambridge University Press, USA.

[43] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://api.semanticscholar.org/CorpusID:257532815

[44] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. 2011. Formal Verification of OAuth 2.0 Using Alloy Framework. In *2011 International Conference on Communication Systems and Network Technologies*. IEEE, Katra, Jammu, India, 655–659. https://doi.org/10.1109/CSNT.2011.141 http://ieeexplore.ieee.org/document/5966531/.

[45] Raluca Ada Popa, Jacob R Lorch, David Molnar, Helen J Wang, and Li Zhuang. 2011. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Annual Technical Conference*. USENIX, Portland, OR. https://www.usenix.org/conference/usenixatc11/enabling-security-cloud-storage-slas-cloudproof.

[46] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating*

*Systems Principles - SOSP '11*. ACM Press, Cascais, Portugal, 85. https://doi.org/10.1145/2043556.2043566 http://dl.acm.org/citation.cfm?doid=2043556.2043566.
[47] Justin Richer, Michael Jones, John Bradley, Maciej Machulak, and Phil Hunt. 2015. *OAuth 2.0 Dynamic Client Registration Protocol*. Request for Comments RFC 7591. Internet Engineering Task Force. https://doi.org/10.17487/RFC7591 https://datatracker.ietf.org/doc/rfc7591.
[48] Nat Sakimura, John Bradley, and Naveen Agarwal. 2015. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. IETF. https://datatracker.ietf.org/doc/html/rfc7636.
[49] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. 2020. Droplet: Decentralized Authorization and Access Control for Encrypted Data Streams. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. USENIX, 2469–2486. https://www.usenix.org/conference/usenixsecurity20/presentation/shafagh.
[50] Anuchart Tassanaviboon and Guang Gong. 2011. OAuth and ABE Based Authorization in Semi-Trusted Cloud Computing: Aauth. In *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds - DataCloud-SC '11*. ACM Press, Seattle, Washington, USA, 41. https://doi.org/10.1145/2087522.2087531 http://dl.acm.org/citation.cfm?doid=2087522.2087531.
[51] Jeremy Thomas. 2023. Bulma. https://bulma.io/.
[52] Alvin Wang. 2023. Dogfalo/Materialize. https://github.com/Dogfalo/materialize.
[53] Hsiaoming Yang. 2023. Authlib. https://github.com/lepture/authlib.

# A APEX RESOURCE DELETION PROTOCOL

The *APEX resource deletion protocol* performs the function of the RRet protocol required by RAD (§5) and begins with a consumer that wishes to delete a resource from the server. The protocol proceeds as follows with all communication taking place over TLS:

(1) A consumer uses the resource access protocol RA to request that a resource identified by an $rid$ is deleted.
The consumer also provides a list of supported promise fulfilment methods and a signature on the $rid$.
(2) The server can: (a) delete the resource and return a success status; or (b) respond with an ID representing a *deletion promise* and a chosen promise fulfilment method.
The server might select the second option if the API provider only wishes to allow deletion of a resource under certain circumstances, based on the contents of the resource.
In the case of the first option, the run of the deletion protocol ends here. Otherwise, the subsequent steps are performed.
(3) The consumer communicates with a consumer agent and provides the deletion promise.
(4) The consumer agent fulfils the deletion promise with the help of the server and a provider agent, using the promise fulfilment method negotiated in the preceding steps. Promise fulfilment is described in §3.8 and §3.8.

# B APEX PROMISE FULFILMENT PROTOCOL

*Asynchronous vs synchronous fulfilment.* If one or more provider agents are reachable by the server (e.g., the agent has a persistent connection to the server or is registered to receive push notifications) at the time it receives the request to register, retrieve or delete a resource, it may notify one or more of these agents on the spot, starting the promise fulfilment process straight away. In this case, the consumer agent receives the promise asynchronously (while the provider agent processes the promise) and must periodically query the server to obtain the status of the processing.

Alternatively, upon receiving a promise, the consumer agent may submit it to a provider agent directly. This causes the provider agent to begin processing of the promise and return to the consumer agent upon completion. It can be said that the promise is fulfilled synchronously. This only works if both the consumer and provider agent are running on the same device and are able to be invoked by navigating to a URI (e.g., the agent is implemented as browser script or has registered a custom URI scheme).

Even if fulfilment of a promise begins in an asynchronous fashion, the consumer agent may still proceed with synchronous fulfilment. This is particularly useful, e.g., when a push notification is sent to a provider agent but that agent is on a device which is momentarily inaccessible to the resource owner (perhaps the device stays at home during the workday).

*The promise fulfilment procedure.* Fulfilment begins when a promise is received by a provider agent and proceeds according to the following steps (illustrated in Figure 6):

(1) The provider agent (PA) receives the promise from the server (S) in the case of asynchronous fulfilment.
   - ♦ In the synchronous case, the promise is instead received from the consumer agent (CA) along with a redirect URI. The provider agent will later use this redirect URI to return execution to the consumer agent.
(2) The provider agent requests the data associated with the promise by submitting it to the server.
   - ♦ For most protocol runs, execution continues to the next step. However, if the promise has already been fulfilled by another run, execution proceeds from (6).
(3) The server responds with any information that has been associated with the promise, which differs depending on the type of promise:
   - $rid$: the identifier at which the resource is currently saved which may be null (all promises)
   - $rid^*$: the resource identifier, chosen by the server, under which to save the resource, which may or may not differ from $rid$ (save promise)
   - $wr$: the resource data to save, wrapped (encrypted) using the agent key (save promise)
   - $wak$: the agent key wrapped using the owner's public key (save or rewrapping promise)
   - $\sigma_{\text{req}}$: a signature on the request data submitted by the consumer (save or rewrapping promise)
   - $R, \sigma_R$: the encrypted resource currently stored by the server, with signature (save or deletion promise)
   - $RK, \sigma_{RK}$: the wrapped resource key currently stored by the server or provided by the consumer for rewrapping, together with signature (all promises)
(4) The provider agent verifies the signatures received and, if successful, unwraps (decrypts) all keys and resources received. Resource data to be saved is validated against application-specific rules and rewrapped (encrypted) with a newly-generated resource key $rk^*$. Resource keys to be rewrapped are re-encrypted with the agent key from the consumer agent.
(5) The provider agent returns certain data to the server, depending on the promise type:

- $R^*, RK^*, \sigma_{R^*}, \sigma_{RK^*}$: the resource data to save (re-encrypted with the new resource key), the newly wrapped resource key, and signatures (save promise)
- $rwrk$: the supplied resource key rewrapped with the supplied agent key, which is also known as the *rewrapped resource key* (rewrapping promise)
- $\sigma_{\text{res}}$: a signature over the request data received by the provider agent and the response data returned (save or rewrapping promise)
- whether the data received by the provider agent was well formed and treated as valid (all promises)
- whether the provider agent made any changes to the resource beyond those requested (all promises)

Provided the promise has not been fulfilled by a different protocol run, if the information provided is valid, the server commits the changes to the stored resources.

(6) The server replies with a message indicating whether it has committed changes for this promise. This happens even if the promise happened to be fulfilled by a different protocol run while the current run was in progress.

(7) If in (1) the provider agent received a redirect URI (because synchronous fulfilment is being performed), the provider agent navigates to that URI and includes the promise as a URI parameter.

♦ In asynchronous fulfilment, this step is skipped.

(8) The consumer agent submits the promise to the server in order to retrieve the results. This may happen directly after (7) or, if asynchronous fulfilment is being performed, whenever the consumer agent wishes to try.

(9) The server returns info about the promise's status (i.e., has processing completed and was it successful?) and various resource attributes, depending on the promise type:

- $rwrk$: rewrapped resource key (rewrapping promise)
- $rid^*$: the new resource identifier (save promise)
- $\sigma_{\text{res}}$: a signature over the request and response data to be checked against the consumer agent's view of the execution (save or rewrapping promise)

(10) The consumer agent performs any processing as necessary. For a rewrapping promise, it verifies the signature, unwraps the resource key and decrypts the resource.

♦ If the server indicated in (9) that processing of the promise has not yet been completed, the consumer agent instead waits a period and then retries from (8).

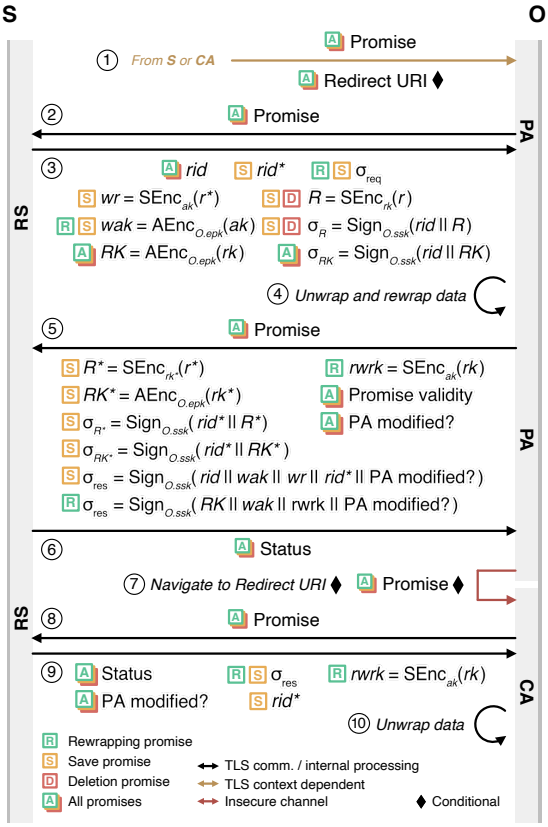


**Figure 6: APEX Promise Fulfilment**

# C ALTERNATIVE MODES FOR APEX

## C.1 Default Mode: Consumer Agents Independent of Consumer

Throughout this paper, we have assumed (§3.3) that the APEX parties are so configured:

- the APEX consumer which talks to the APEX server is itself running on a remote server; and
- consumer agents are running on devices belonging to the resource owner.

This has a couple of benefits:

(1) When the API provider provides access to encrypted and unencrypted resources side by side, processing of any unencrypted resources can be performed by the consumer server-side without involvement from the resource owner or its devices (after an initial authorisation grant phase has been completed) just as is the case with OAuth.
(2) When APEX is composed with OAuth 2.0, the exchange of authorisation code for access token is performed by a entity capable of authenticating to the APEX server with an API secret which has been kept confidential (as it does not need to be distributed to end user devices). This means there is no need to employ PKCE [48] to prevent authorisation codes from being intercepted and exchanged by an unauthenticated malicious party.

However, there are other possible modes of operations which may be more appropriate for certain use cases.

## C.2 Other Modes: Combining Consumer and Agent

Instead of keeping the consumer and consumer agent as separate entities, it is possible that both these roles are fulfilled by the same party. This changes our trust assumptions such that the same level of trust extended to the consumer agent must also be extended to the consumer. So the combined consumer+agent is treated as trusted, rather than semi-trusted.

When considering this variant of APEX in the context of the RAD model, one must treat the consumer+agent party as both owner and consumer such that a consumer+agent $x$ is simultaneously an element of the set of owners $\mathcal{O}$ and an element of the set of consumers $\mathcal{C}$. Any query restrictions placed on owners at the invocation of the freshness definition (§6) also therefore apply to consumer+agents.

*Consumer on resource owner device.* When the combined consumer+agent is implemented as an software running on the resource owner's device, much of the characteristics related to the secrecy of data are retained. However, there some difference in this and other areas:

(1) The resource owner no longer has to place trust in a remote server (the software of which can more easily be changed without the owner's knowledge) to faithful execute the protocol according to specification.
(2) Without a server to hold or securely sync $C.ssk$, each instance of the consumer+agent will need to obtain authorisation independently.
(3) If APEX is used with OAuth, PKCE must be employed to protect access tokens.

*Consumer agent on server.* Another possibility is that the consumer+agent runs on a remote server, thereby moving the responsibility of performing encryption/decryption operations server-side. This has the obvious drawback that the entity controlling the API consumer is now able to read the owner's resources. However, it opens up some new possibilities, e.g., consumers which function as a traditional web application based on the stateless request–response model and which may not have any programs/scripts running on the resource owner's device. The costs and benefits should be carefully considered and made known to the resource owner.

# D FURTHER FORMALISATIONS IN THE RAD MODEL

## D.1 Channels Formalisation

In addition to those given in §6, each party $P \in \mathcal{P}$ is also assigned attributes for:

- the party's **channel secret and public keys**, $P$.csk and $P$.cpk, used for secure channel establishment, where $P$.cpk is known by all other parties.

Each channel $\pi_P^i.\Pi_j$, where $\pi_P^i$ is an instance of $P$, has the following attributes set, whether ACCE is used or not:

- the **secure flag**, $\pi_P^i.\Pi_j$.secure, set to 1 if the channel is an ACCE channel or 0 otherwise;
- the **role**, $\pi_P^i.\Pi_j.\rho$, set to "init" if $P$ sends the first message on channel $j$ or "resp" otherwise; and
- the **transcript**, $\pi_P^i.\Pi_j.\tau$, a list of messages communicated on the channel (in an encrypted form, if ACCE is used).

The channels that are ACCE secure additionally employ the next attributes (which are set to $\perp$ when a channel is insecure):

- the **session identifier**, $\pi_P^i.\Pi_j$.sid, set to $\perp$ until this has been negotiated;
- the **communication peer**, $\pi_P^i.\Pi_j$.peer, set to some party $Q \in \mathcal{P}$ to indicate the intended communication partner of the party in this channel or $\perp$ if unilateral authentication is used and the peer is an ACCE consumer;
- the **channel acceptance flag**, $\pi_P^i.\Pi_j.\alpha$, which is set to $\perp$ while the session is being established, 1 if the authentication of the communication peer has been accepted, or 0 if the authentication has been rejected; and
- the **channel key**, $\pi_P^i.\Pi_j$.ck, used to authenticate and encrypt messages, once it has been negotiated between the two party instances (before then, set to $\perp$).

## D.2 Utility Functions' Details

The following utility functions can be called by any instance during phase execution:

- InScope($scope$, $action$, $rid$, $r$):
  Returns 1 when $action$ is allowed to be performed on a resource according to the given authorisation $scope$ and 0 otherwise. $rid$ is any valid resource identifier and $r$ is any resource plaintext.

  When $action$ = "reg", $r$ must be set to a non-$\perp$ value whereas $rid$ is only set to indicate a resource update.

  When $action \in$ {"ret","del"}, $rid$ must be set to a non-$\perp$ value. The value of $r$ is ignored.
- ProduceProof($cred$, $m$, $aux$):
  Generates a proof from the credential $cred$ and message $m$, e.g., a signature on $m$ using $cred$ as the signing key. $aux$ models extra information about $cred$ and may affect the type of proof produced. When $cred$ is a shared secret, $m$ must be set to $\perp$ and $cred$ returned as the proof.
- VerifyProof($vrfr$, $m$, $p$, $aux$):
  Returns 1 if proof $p$ matches message $m$ and verifies using verifier $vrfr$ with metadata $aux$ or 0 otherwise. When $vrfr$ is a shared secret, $m$ must be set to $\perp$.
- ProtectRes($epk$, $ssk$, $r$):
  Converts a plaintext resource $r$ to a form that can only be read given knowledge of the secret key corresponding to public encryption key $epk$. A signing key $ssk$ can be provided to sign the resource.
- ExtractRes($esk$, $spk$, $R$):
  Decodes and decrypts a raw resource $R$ using secret decryption key $esk$ to return plaintext resource $r$. Verification of $R$ against public verification key $spk$ will be attempted, if provided, and $\perp$ returned if unsuccessful.

## D.3 Additional Oracles

In addition to the oracles given in §6 for corrupting parties to various degrees, the adversary $\mathcal{A}$ is also given access to following oracles to instantiate new phase executions and facilitate communication between them:

- NewInstance($P$, $type$):
  Creates a new phase instance $\pi_P^i$ of party $P$ and $type \in$ {"auth", "reg", "ret", "del"}, setting the instance's status to "active" and recording the start time. $Ch$ returns to $\mathcal{A}$ the instance identifier $i \in P.n_\pi$.
- NewChannel($\pi_P^i$, $\rho$, $peer$):
  Creates a new channel for the instance $\pi_P^i$ with role $\rho \in$ {"init", "resp"} and its intended communication partner $peer$ if $peer$ ought to authenticate to $P$ via some ACCE protocol.
- Send($\pi_P^i.\Pi_j$, $m$):
  Sends a message $m$ to the instance $\pi_P^i$ over channel $\pi_P^i.\Pi_j$. A reply is generated according to the protocol specification and returned to $\mathcal{A}$.
- BeginPhase($\pi_P^i$, $Q$, $rid$):
  Causes $\pi_P^i$ to generate the first message required to start the phase, assuming that $Q \in \mathcal{P}$ is the recipient. Returns the generated message unless, according to the protocol, either $P$ or $Q$ is a party of a type not permitted to initiate the phase, or the current state of $\pi_P^i$ or $P$ is not such that beginning the phase is a valid action.

  If $\pi_P^i$.type $\in$ {"reg", "ret", "del"}, then $rid$ can optionally be given to perform an action on a specific resource, otherwise the affected resource is chosen at random.

$\mathcal{A}$ also receives oracle access to the utility functions defined previously in §D.2.

The calls to various oracle impacts our freshness definition, as per Table 3.

| | $S$ | $C$ | $O$ |
|---|---|---|---|
| CommCorrupt | ✓ | ✗ | ✗ |
| ResCorrupt | ✓ | ✗ | ✗ |
| DataReveal | ✓ | ✓ | ✗ |
| GetData | ✓ | ✓ | ✓ |

**Table 3: The Impact of Oracles on the Freshness of a Party's Instances**

## D.4 Message Authentication

An authenticator $t$ is said to *authenticate* a message $m$ to a party $P \in \mathcal{P}$ as originating at a party $Q \in \mathcal{P}$ iff **any** of the following statements are true:

(1) $P.\text{authz}'_i = (Q, B_i, vrfr_i, from_i, until_i, scope_i, aux_i)$ for some $i \in \mathbb{N}$ such that VerifyProof($vrfr_i$, $m$, $t$, $aux_i$) = 1 and $until_i$ is some future time or $\perp$;
(2) VerifyProof($Q$.cpk, $m$, $t$, "cpk") = 1; or
(3) there exists an instance $\pi_P^j$ for some $j \in P.n_\pi$ such that $\pi_P^j$.peer = $Q$ and VerifyProof($\pi_P^j$.ck, $m$, $t$, "ck") = 1.

## D.5 Authorisation Origination

An authorisation credential or verifier $x$ is said to have *originated* within a phase instance $\pi_P^i$ for some $P \in \mathcal{P}$ and $i \in [1, P.n_\pi]$ iff:

(1) the instance recorded a new credential/verifier such that $\pi_P^i.\text{authzid} = j$ or $\pi_P^i.\text{authzid}' = k$ for any $j, k \in \mathbb{N}$; and
(2) the credential/verifier $x$ is present at index $j$ or $k$ such that $P.\text{authz}_j$ or $P.\text{authz}'_k$ is equal to ($A$, $B$, $X$, $from$, $until$, $scope$, $aux$) where $X = x$.

## D.6 Authorisation Credential and Verifier Matching

A credential $x$ and verifier $y$ are said to *match* iff:

(1) $x$ originates within a phase instance $\pi_P^a$ where $\pi_P^a.\text{authzid} = j$ for some $P \in \mathcal{P}$, $a \in [1, P.n_\pi]$ and $j \in \mathbb{N}$;
(2) $y$ originates within a phase instance $\pi_Q^b$ where $\pi_P^a.\text{authzid}' = k$ for some $Q \in \mathcal{P}$, $b \in [1, Q.n_\pi]$ and $k \in \mathbb{N}$;
(3) $P.\text{authz}_j = (Q, B_j, x, from_j, until_j, scope_j, aux_j)$;
(4) $Q.\text{authz}'_k = (P, B_k, y, from_k, until_k, scope_k, aux_k)$;
(5) $P \neq Q$; and
(6) from $x$ it is possible to produce a proof of authorisation which can verified as valid using $y$, i.e., if $p$ = ProduceProof($x$, $m$, $aux_j$) for a randomly chosen $m$, then VerifyProof($y$, $m$, $p$, $aux_k$) = 1.

## D.7 Instance Partnering

In RAD protocols, there are different ways in which instances can be partnered, depending on the protocol phase and the parties involved. This is formalised below.

*Partnering in the authorisation grant phase.* For any $S \in \mathcal{S}, C \in \mathcal{C}$, $O \in \mathcal{O}$ and $a \in [1, S.n_\pi]$, $b \in [1, C.n_\pi]$, $c \in [1, O.n_\pi]$, a server, consumer and owner instance, $\pi_S^a$, $\pi_C^b$ and $\pi_O^c$, are partnered iff:

(1) all three have successfully completed the authorisation grant phase such that:
  $\pi_S^a.\text{type} = \pi_C^b.\text{type} = \pi_O^c.\text{type}$ = "auth" and
  $\pi_S^a.\text{status} = \pi_C^b.\text{status} = \pi_O^c.\text{status}$ = "success";
(2) an authorisation credential which has originated within the consumer instance $\pi_C^b$ matches an authorisation verifier which has originated within the server instance $\pi_S^a$ such that $C$ is authorised to perform actions on $S$ at the conclusion of the phase execution;
(3) an authorisation credential which has originated within the consumer instance $\pi_C^b$ matches an authorisation verifier which has originated within the owner instance $\pi_O^c$ such that $C$ is authorised to perform actions on $O$ at the conclusion of the phase execution;
(4) the authorisation credential from condition 2 above is only valid for actions performed on behalf of $O$; and
(5) the credentials and verifiers from conditions 2 and 3 above all share the same scope.

*Partnering in resource access phases.* For any $S \in \mathcal{S}, C \in \mathcal{C}$ and $a \in [1, S.n_\pi]$, $b \in [1, C.n_\pi]$, a server instance $\pi_S^a$ and consumer instance $\pi_C^b$ are considered partnered iff:

(1) both have successfully completed the same resource access phase $ph$ such that:
$ph = \pi_S^a.\text{type} = \pi_C^b.\text{type} \in \{\text{“reg”}, \text{“ret”}, \text{“del”}\}$ and
$\pi_S^a.\text{status} = \pi_C^b.\text{status} = \text{“success”}$;
(2) both are acting on the same resource $R$ with identifier $rid$ for the full duration of phase execution, i.e.,:
$R = \pi_S^a.\mathsf{R}_d^{\leftrightarrow} = \pi_C^b.\mathsf{R}_e^{\leftrightarrow}$ for all $d, e \in \mathbb{N}$ and
$rid = \pi_S^a.\text{rid}_f^{\leftrightarrow} = \pi_C^b.\text{rid}_g^{\leftrightarrow}$ for all $f, g \in \mathbb{N}$
where $\pi_S^a.\text{rid}_f^{\leftrightarrow} \neq \perp$ and $\pi_C^b.\text{rid}_g^{\leftrightarrow} \neq \perp$;
(3) the resource $R$ is registered with $S$ under identifier $rid$ for an owner $O \in \mathcal{O}$, i.e.,:
$S.res_h = (O, rid, R)$ for some $h \in \mathbb{N}$;
(4) the instances have exchanged the same authorisation proof $p_1$, i.e., $p_1 = \pi_S^a.\text{proof}_i^{\leftrightarrow} = \pi_C^b.\text{proof}_j^{\leftrightarrow}$ for some $i, j \in \mathbb{N}$;
(5) the proof $p_1$ verifies using a verifier $v_1$ stored by $S$ which is valid at the start of $\pi_S^a$'s execution; and
(6) the resource is within the scope $sc$ of the authorisation verifier $v_1$, i.e., :
$\mathsf{InScope}(sc,\ ph,\ \pi_C^b.\text{rid}_1^{\leftrightarrow},\ r) = 1$
where $r = \mathsf{ExtractRes}(O.esk,\ O.spk,\ R)$.

An instance $\pi_O^c$ of owner $O$ is additionally partnered with $\pi_S^a$ and $\pi_C^b$ for some $c \in [1, O.n_\pi]$ iff:

(1) $\pi_O^c$ has successfully completed the same resource access phase as the other two instances, i.e., $ph = \pi_O^c.\text{type}$ and $\pi_O^c.\text{status} = \text{“success”}$;
(2) $\pi_O^c$ is acting on the same resource $R$ with the same identifier $rid$ as the other two instances, i.e.,:
$R = \pi_O^c.\mathsf{R}_m^{\leftrightarrow}$ for all $m \in \mathbb{N}$ and
$rid = \pi_O^c.\text{rid}_n^{\leftrightarrow}$ for all $n \in \mathbb{N}$ where $\pi_O^c.\text{rid}_n^{\leftrightarrow} \neq \perp$;
(3) $O$ has a sound understanding of the contents of $R$, i.e., there is no $O.\text{cache}_q = (R_q,\ r_q)$ for any $q \in \mathbb{N}$ where $r_q \neq \mathsf{ExtractRes}(O.esk,\ O.spk,\ R_q)$ and $R_q = R$;
(4) $\pi_O^c$ has received an authorisation proof $p_2$ from $\pi_C^b$, i.e., $p_2 = \pi_O^c.\text{proof}_s^{\leftrightarrow} = \pi_C^b.\text{proof}_t^{\leftrightarrow}$ for some $s, t \in \mathbb{N}$;
(5) the proof $p_2$ verifies using a verifier $v_2$ stored by $O$ which is valid at the start of $\pi_O^c$'s execution; and
(6) the scope of verifier $v_2$ is equal to $sc$.

# E INSTANCE PARTNERING

In RAD protocols, there are different ways in which instances can be partnered, depending on the protocol phase and the parties involved. This is formalised below.

*Partnering in the authorisation grant phase.* For any $S \in \mathcal{S}, C \in \mathcal{C}$, $O \in \mathcal{O}$ and $a \in [1, S.n_\pi]$, $b \in [1, C.n_\pi]$, $c \in [1, O.n_\pi]$, a server, consumer and owner instance, $\pi_S^a$, $\pi_C^b$ and $\pi_O^c$, are partnered iff:

(1) all three have successfully completed the authorisation grant phase such that:
$\pi_S^a.\text{type} = \pi_C^b.\text{type} = \pi_O^c.\text{type} = \text{“auth”}$ and
$\pi_S^a.\text{status} = \pi_C^b.\text{status} = \pi_O^c.\text{status} = \text{“success”}$;
(2) an authorisation credential which has originated within the consumer instance $\pi_C^b$ matches an authorisation verifier which has originated within the server instance $\pi_S^a$ such that $C$ is authorised to perform actions on $S$ at the conclusion of the phase execution;
(3) an authorisation credential which has originated within the consumer instance $\pi_C^b$ matches an authorisation verifier which has originated within the owner instance $\pi_O^c$ such that $C$ is authorised to perform actions on $O$ at the conclusion of the phase execution;
(4) the authorisation credential from condition 2 above is only valid for actions performed on behalf of $O$; and
(5) the credentials and verifiers from conditions 2 and 3 above all share the same scope.

*Partnering in resource access phases.* For any $S \in \mathcal{S}, C \in \mathcal{C}$ and $a \in [1, S.n_\pi]$, $b \in [1, C.n_\pi]$, a server instance $\pi_S^a$ and consumer instance $\pi_C^b$ are considered partnered iff:

(1) both have successfully completed the same resource access phase $ph$ such that:
$ph = \pi_S^a.\text{type} = \pi_C^b.\text{type} \in \{\text{“reg”}, \text{“ret”}, \text{“del”}\}$ and
$\pi_S^a.\text{status} = \pi_C^b.\text{status} = \text{“success”}$;
(2) both are acting on the same resource $R$ with identifier $rid$ for the full duration of phase execution, i.e.,:
$R = \pi_S^a.\mathsf{R}_d^{\leftrightarrow} = \pi_C^b.\mathsf{R}_e^{\leftrightarrow}$ for all $d, e \in \mathbb{N}$ and
$rid = \pi_S^a.\text{rid}_f^{\leftrightarrow} = \pi_C^b.\text{rid}_g^{\leftrightarrow}$ for all $f, g \in \mathbb{N}$
where $\pi_S^a.\text{rid}_f^{\leftrightarrow} \neq \perp$ and $\pi_C^b.\text{rid}_g^{\leftrightarrow} \neq \perp$;
(3) the resource $R$ is registered with $S$ under identifier $rid$ for an owner $O \in \mathcal{O}$, i.e.,:
$S.res_h = (O, rid, R)$ for some $h \in \mathbb{N}$;
(4) the instances have exchanged the same authorisation proof $p_1$, i.e., $p_1 = \pi_S^a.\text{proof}_i^{\leftrightarrow} = \pi_C^b.\text{proof}_j^{\leftrightarrow}$ for some $i, j \in \mathbb{N}$;
(5) the proof $p_1$ verifies using a verifier $v_1$ stored by $S$ which is valid at the start of $\pi_S^a$'s execution; and
(6) the resource is within the scope $sc$ of the authorisation verifier $v_1$, i.e., :
$\mathsf{InScope}(sc,\ ph,\ \pi_C^b.\text{rid}_1^{\leftrightarrow},\ r) = 1$
where $r = \mathsf{ExtractRes}(O.esk,\ O.spk,\ R)$.

An instance $\pi_O^c$ of owner $O$ is additionally partnered with $\pi_S^a$ and $\pi_C^b$ for some $c \in [1, O.n_\pi]$ iff:

(1) $\pi_O^c$ has successfully completed the same resource access phase as the other two instances, i.e., $ph = \pi_O^c.\text{type}$ and $\pi_O^c.\text{status} = \text{“success”}$;
(2) $\pi_O^c$ is acting on the same resource $R$ with the same identifier $rid$ as the other two instances, i.e.,:
$R = \pi_O^c.\mathsf{R}_m^{\leftrightarrow}$ for all $m \in \mathbb{N}$ and
$rid = \pi_O^c.\text{rid}_n^{\leftrightarrow}$ for all $n \in \mathbb{N}$ where $\pi_O^c.\text{rid}_n^{\leftrightarrow} \neq \perp$;
(3) $O$ has a sound understanding of the contents of $R$, i.e., there is no $O.\text{cache}_q = (R_q,\ r_q)$ for any $q \in \mathbb{N}$ where $r_q \neq \mathsf{ExtractRes}(O.esk,\ O.spk,\ R_q)$ and $R_q = R$;
(4) $\pi_O^c$ has received an authorisation proof $p_2$ from $\pi_C^b$, i.e., $p_2 = \pi_O^c.\text{proof}_s^{\leftrightarrow} = \pi_C^b.\text{proof}_t^{\leftrightarrow}$ for some $s, t \in \mathbb{N}$;
(5) the proof $p_2$ verifies using a verifier $v_2$ stored by $O$ which is valid at the start of $\pi_O^c$'s execution; and
(6) the scope of verifier $v_2$ is equal to $sc$.

# F EVALUATION OF OAUTH 2.0 AS A RAD SCHEME

As a point of comparison with APEX, consider OAuth 2.0 (as described in §2) instantiated as a RAD scheme (§5):

$$suite = (\text{DMAG, RReg, RRet, RDel})$$
$$\text{OAuth} = (\text{SMAG, RA, RA, RA})$$

Here, the dual-mediated authorisation grant protocol required for a RAD scheme (DMAG) is fulfilled by OAuth's server-mediated protocol (SMAG). Similarly, the expected phase-specific resource protocols (RReg, RRet and RDel) are all fulfilled by OAuth's generic resource access protocol (RA).

Note that since OAuth does not perform encryption of resources, $\text{ProtectRes}(epk, ssk, r) = r$ and $\text{ExtractRes}(epk, ssk, R) = R$.

Let $\mathcal{A}$ be a PPT algorithm which plays the ERA security game against OAuth. It can perform a polynomial number of runs of the SMAG and RA protocols by issuing queries to the challenger.

As the server in a RAD scheme is considered adversarial, $\mathcal{A}$ had direct access to all data stored on the server and can even act as the server, unlike in the standard OAuth model. It also has access to data stored by the consumer since the consumer is considered only semi-trusted. These capabilities are modelled through the GetData and CommCorrupt oracles which can be used as in the following examples to win the game $\text{G}^{\text{ERA}}_{\mathcal{A},\text{OAuth}}$ :

- $\mathcal{A}$ issues the $\text{GetData}(S)$ query to get the contents of $S$.res for any server $S$ with one or more registered resources;
- $\mathcal{A}$ issues the $\text{GetData}(C)$ query to get the contents of $\pi^i_C.\text{R}^{\leftrightarrow}$ for any instance of any consumer $C$ where $\pi^i_C.\text{status}$ = "success" and $\pi^i_C.\text{type} \in \{\text{"reg"}, \text{"ret"}, \text{"del"}\}$;
- $\mathcal{A}$ uses the $\text{CommCorrupt}(S)$ query to interleave resources requested by two different consumers $C_1, C_2$ such that a consumer receives a resource it was not authorised to receive; or
- $\mathcal{A}$ uses the $\text{CommCorrupt}(S)$ query and forges a message from some server $S$ in reply to a resource retrieval request from some consumer $C$ who later updates the resource with minimal modifications, in effect allowing $\mathcal{A}$ to register resource data of its choice.

If $\mathcal{A}$ were to simply perform one run of RA to register a single resource before attempting the first attack above, its success probability would be 1. As such:

$$\text{Adv}^{\text{ERA}}_{\mathcal{A},\text{OAuth}} > negl(s)$$

where *negl* is any negligible function in the security parameter $s$.

Therefore, OAuth is not secure w.r.t. ERA, our security property for RAD schemes.

# G PROOFS

## G.1 Proof of Theorem 1: Relationship Between the Security of APEX and Its Sub-Protocols

As a reminder, Theorem 1 states:

> *If an APEX protcol suite $\alpha$ uses:*
> *a) EUF-CMA secure MAC and signature schemes;*
> *b) IND-EAV secure symmetric encryption;*
> *c) IND-CPA secure asymmetric encryption; and*
> *d) one-time codes (OTCs) in an exponential space in the security parameter*
> *then the APEX protocol suite $RAD_\alpha$ is ERA secure.*

Let APEX represent any instantiation of APEX consisting of instantiations of the APEX component protocols (DMAG, RReg, RRet, RDel) configured to use cryptographic primitives (Sym, Asym, Sig, MAC).

Let $\mathbb{A}$ represent the set of all possible PPT algorithms.

Notice that any $\mathcal{A} \in \mathbb{A}$ which wins in the ERA game against an arbitrary RAD suite or sub-protocol, does so either by:

(1) obtaining the plaintext of a legitimately registered resource $r'$, or
(2) obtaining sufficient information to allow it to impersonate and use the authorisation of a legitimate party.

To break ERA security of the suite/protocol it must be able to do so with a non-negligible probability.

Assume: $\nexists\ \mathcal{A}'_1 \in \mathbb{A}$ s.t. $\text{Adv}^{\text{AG}}_{\mathcal{A}'_1,\text{DMAG}} > negl(s)$
$\nexists\ \mathcal{A}'_2 \in \mathbb{A}$ s.t. $\text{Adv}^{\text{ERA}}_{\mathcal{A}'_2,\text{RReg}} > negl(s)$
$\nexists\ \mathcal{A}'_3 \in \mathbb{A}$ s.t. $\text{Adv}^{\text{ERA}}_{\mathcal{A}'_3,\text{RRet}} > negl(s)$
$\nexists\ \mathcal{A}'_4 \in \mathbb{A}$ s.t. $\text{Adv}^{\text{ERA}}_{\mathcal{A}'_4,\text{RDel}} > negl(s)$
for any negligible function $negl(s)$.

Suppose: $\exists\ \mathcal{A} \in \mathbb{A}$ s.t. $\text{Adv}^{\text{ERA}}_{\mathcal{A},\text{APEX}} > negl(s)$

$\therefore\ \mathcal{A} \notin \{\mathcal{A}'_1, \mathcal{A}'_2, \mathcal{A}'_3, \mathcal{A}'_4\}$

$\therefore\ \mathcal{A}$ cannot, by limiting its interactions to one of the four phases:

(1) obtain a resource plaintext $r'$ with a non-negligible probability; or
(2) obtain an authorising credential $cred'$ for a consumer which has a matching verifier $vrfr'$ at an owner, or vice versa, with a non-negligible probability.

$\therefore\ \mathcal{A}$ must break the ERA security of APEX by either:

(1) obtaining partial information about $r'$ in two or more of the phases to allow it to deduce the full $r'$ with a non-negligible probability; or
(2) obtaining partial information about $cred'$ in two or more of the phases to allow it to deduce the full $cred'$ with a non-negligible probability.

The proof proceeds by contradiction, showing that it is not possible to obtain even one bit of $r'$ or $cred'$ with a non-negligible probability in any of the phases.

For all instances of the ERA game in which $\mathcal{A}$ is playing against APEX:

> Let $\Gamma$ be the set of resource ciphertexts that correspond to $r'$ s.t. $\exists$ an entry $(O, rid, R)$ in $S.res$ for some $S \in \mathcal{S}$ and $O \in \mathcal{O}$ where $r' = \text{ExtractRes}(O.esk, \perp, R)$.

Let $\Phi$ be the set of all server instances which have sent/received an $R \in \Gamma$.

Let $\mathbb{S}$ be the set of servers which have an instance in $\Phi$.

Let $\mathbb{C}$ be the set of consumers which have an instance in $\Phi$.

Let $\mathbb{O}$ be the set of owners which have an instance in $\Phi$.

Because $\mathcal{A}$ satisfies the winning condition of the ERA game, all $S \in \mathbb{S}$ are fresh.

$\therefore$ No CommCorrupt($C$) or ResCorrupt($C$) has been called on any $C \in \mathbb{C}$.

No CommCorrupt($O$) or ResCorrupt($O$) has been called on any $O \in \mathbb{O}$.

No DataReveal($O$) has been called on any $O \in \mathbb{O}$.

All other queries are permitted.

Note. It is not possible for $\mathcal{A}$ to corrupt a phase instance of a party $P \in \mathcal{P}$ without also corrupting all other instances of $P$ whether executing the same type of phase or not.
Let $\mathsf{V}_P^i$ be a set representing the view of instance $\pi_P^i$ for some party $P \in \mathcal{P}$ and $i \in [1, P.n_\pi]$ w.r.t. data which has a relation to the plaintext of any resource acted on in $\pi_P^i$.

For all $O \in \mathcal{O}, C \in \mathcal{C}$ and $S \in \mathcal{S}$:

$$\mathsf{V}_O^a = \{\ wak_a = \mathsf{AEnc}_{O_a.epk}(ak_a),\ wr_a = \mathsf{SEnc}_{ak_a}(r_a^*),\ RK_a = \mathsf{AEnc}_{O_a.epk}(rk_a),\ R_a = \mathsf{SEnc}_{rk_a}(r_a),\ RK_a^* = \mathsf{AEnc}_{O_a.epk}(rk_a^*),\ R_a^* = \mathsf{SEnc}_{rk_a^*}(r_a^*)\ \} \quad \text{for all } a \in [1, O.n_\pi] \text{ s.t. } \pi_O^a.\text{type} = \text{“}reg\text{”}$$

$$\mathsf{V}_C^b = \{\ wak_b = \mathsf{AEnc}_{O_b.epk}(ak_b),\ wr_b = \mathsf{SEnc}_{ak_b}(r_b^*)\ \} \quad \text{for all } b \in [1, C.n_\pi] \text{ s.t. } \pi_C^b.\text{type} = \text{“}reg\text{”}$$

$$\mathsf{V}_S^c = \{\ wak_c = \mathsf{AEnc}_{O_c.epk}(ak_c),\ wr_c = \mathsf{SEnc}_{ak_c}(r_c^*),\ RK_c = \mathsf{AEnc}_{O_c.epk}(rk_c),\ R_c = \mathsf{SEnc}_{rk_c}(r_c),\ RK_c^* = \mathsf{AEnc}_{O_c.epk}(rk_c^*),\ R_c^* = \mathsf{SEnc}_{rk_c^*}(r_c^*)\ \} \quad \text{for all } c \in [1, S.n_\pi] \text{ s.t. } \pi_S^c.\text{type} = \text{“}reg\text{”}$$

For all $a \in [1, O.n_\pi]$, $b \in [1, C.n_\pi]$, and $c \in [1, S.n_\pi]$ such that $r_a^* = r'$, $r_b^* = r'$, $r_c^* = r'$ or $r_c = r'$:

- $O_a.epk, O_b.epk$ and $O_c.epk$ are not known to the $\mathcal{A}$ by virtue of freshness but can be guessed with probability = $3 \cdot |\mathcal{K}_{\text{Sig}}|^{-1}$
- $wak_a$, $wak_b$ and $wak_c$ reveal nothing about $ak_a$, $ak_b$ and $ak_c$ respectively unless $\mathcal{A}$ is able to win in the IND-CPA game against Asym. As such, the probability of $wak_a$, $wak_b$ and $wak_c$ leaking any information about $ak_a$, $ak_b$ and $ak_c$ is = $3 \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- Similarly, $wr_a, wr_b, wr_c$ reveal nothing about $r_a^* = r_b^* = r_c^* = r'$ except with probability = $3 \cdot \mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$
- $RK_a, RK_c$ reveal nothing about $rk_a^*, rk_c^*$ except with probability = $2 \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- $R_a, R_c$ reveal nothing about $r_a = r_c = r'$ except with probability = $2 \cdot \mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$
- $RK_a^*, RK_c^*$ reveal nothing about $rk_a^*, rk_c^*$ except with probability = $2 \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- $R_a^*, R_c^*$ reveal nothing about $r_a^* = r_c^* = r'$ except with probability = $2 \cdot \mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$

$\therefore \Pr[\mathcal{A} \text{ learns a bit of } r' \text{ in RReg}] \leq negl(s)$

For all $O \in \mathcal{O}, C \in \mathcal{C}$ and $S \in \mathcal{S}$:

$$\mathsf{V}_O^d = \{\ wak_d = \mathsf{AEnc}_{O_d.epk}(ak_d),\ rwrk_d = \mathsf{SEnc}_{ak_d}(rk_d),\ RK_d = \mathsf{AEnc}_{O_d.epk}(rk_d),\ R_d = \mathsf{SEnc}_{rk_d}(r_d)\ \} \quad \text{for all } d \text{ s.t. } \pi_O^d.\text{type} = \text{“}ret\text{”}$$

$$\mathsf{V}_C^e = \{\ wak_e = \mathsf{AEnc}_{O_e.epk}(ak_e),\ RK_e = \mathsf{AEnc}_{O_e.epk}(rk_e),\ R_e = \mathsf{SEnc}_{rk_e}(r_e)\ \} \quad \text{for all } e \text{ s.t. } \pi_C^e.\text{type} = \text{“}ret\text{”}$$

$$\mathsf{V}_S^f = \{\ wak_f = \mathsf{AEnc}_{O_f.epk}(ak_f),\ rwrk_f = \mathsf{SEnc}_{ak_f}(rk_f),\ RK_f = \mathsf{AEnc}_{O_f.epk}(rk_f),\ R_f = \mathsf{SEnc}_{rk_f}(r_f)\ \} \quad \text{for all } f \text{ s.t. } \pi_S^f.\text{type} = \text{“}ret\text{”}$$

For all $d \in [1, O.n_\pi]$, $e \in [1, C.n_\pi]$, and $f \in [1, S.n_\pi]$ such that $r_d = r'$, $r_e = r'$, $r_f = r'$ or $r_c = r'$:

- $O_d.epk, O_e.epk$ and $O_f.epk$ can only be obtained with probability = $3 \cdot |\mathcal{K}_{\text{Sig}}|^{-1}$
- $wak_d$, $wak_e$ and $wak_f$ reveal nothing about $ak_d$, $ak_e$ and $ak_f$ respectively except for probability = $3 \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- $rwrk_a$, $rwrk_c$ reveal nothing about $rk_d$, $rk_f$ except with probability = $2 \cdot \mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$
- $RK_d, RK_e, RK_f$ reveal nothing about $rk_d$, $rk_e$, $rk_f$ except with probability = $3 \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- $R_d, R_e, R_f$ reveal nothing about $r_d = r_e = r_f = r'$ except with probability = $3 \cdot \mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$

$\therefore \Pr[\mathcal{A} \text{ learns a bit of } r' \text{ in RRet}] \leq negl(s)$

For all $O \in \mathcal{O}, C \in \mathcal{C}$ and $S \in \mathcal{S}$:

$$\mathsf{V}_O^g = \{\ \} \quad \text{for all } g \text{ s.t. } \pi_O^g.\text{type} = \text{“}del\text{”}$$

$$\mathsf{V}_C^h = \{\ \} \quad \text{for all } h \text{ s.t. } \pi_C^h.\text{type} = \text{“}del\text{”}$$

$$\mathsf{V}_S^i = \{\ RK_i = \mathsf{AEnc}_{O_i.epk}(rk_i),\ R_i = \mathsf{SEnc}_{rk_i}(r_i)\ \} \quad \text{for all } i \text{ s.t. } \pi_S^i.\text{type} = \text{“}del\text{”}$$

For all $i \in [1, S.n_\pi]$ such that $r_i = r'$:

- $O_i.epk$ can only be obtained with probability = $|\mathcal{K}_{\text{Sig}}|^{-1}$
- $RK_i$ reveals nothing about $rk_i$ except with probability = $\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\text{Asym}}$
- $R_i$ reveal nothing about $r_i = r'$ except with probability = $\mathsf{Adv}^{\text{IND-EAV}}_{\mathcal{A},\text{Sym}}$

$\therefore \Pr[\mathcal{A} \text{ learns a bit of } r' \text{ in RDel}] \leq negl(s)$

For all $O \in \mathcal{O}, C \in \mathcal{C}$ and $S \in \mathcal{S}$:

$$\mathsf{V}_O^j = \{\ \} \qquad \text{for all } g \text{ s.t. } \pi_O^j.\text{type} = \text{“}auth\text{”}$$
$$\mathsf{V}_C^k = \{\ \} \qquad \text{for all } h \text{ s.t. } \pi_C^k.\text{type} = \text{“}auth\text{”}$$
$$\mathsf{V}_S^l = \{\ \} \qquad \text{for all } i \text{ s.t. } \pi_S^l.\text{type} = \text{“}auth\text{”}$$

$\therefore \Pr[\mathcal{A} \text{ learns a bit of } r' \text{ in DMAG}] \leq negl(s)$

In the resource access phases, APEX authenticates consumers and owners to one another by signing the messages which appear in $\mathsf{V}_O^a$, $\mathsf{V}_C^b$, $\mathsf{V}_S^c$, $\mathsf{V}_O^d$, $\mathsf{V}_C^e$, $\mathsf{V}_S^f$,$\mathsf{V}_O^g$, $\mathsf{V}_C^h$ and $\mathsf{V}_S^i$.

In the authorisation grant phase, APEX authenticates consumer and owners by calculating a MAC on a one-time code (OTC).

The keys $O.ssk$ and $C.ssk$ are unknown to $\mathcal{A}$ by virtue of the freshness definition.

As such, so long as the signature scheme Sig and MAC scheme MAC are EUF-CMA secure and the search space for the OTC is exponential in the security parameter:

$\Pr[\mathcal{A} \text{ learns a bit of } cred' \text{ in AG}] \leq negl(s)$

$\Pr[\mathcal{A} \text{ learns a bit of } cred' \text{ in RReg}] \leq negl(s)$

$\Pr[\mathcal{A} \text{ learns a bit of } cred' \text{ in RRet}] \leq negl(s)$

$\Pr[\mathcal{A} \text{ learns a bit of } cred' \text{ in RDel}] \leq negl(s)$

$\therefore \nexists\, \mathcal{A} \in \mathbb{A} \text{ s.t. } \mathsf{Adv}^{\mathsf{ERA}}_{\mathcal{A},\mathsf{APEX}} > negl(s)$

## G.2 Proof of Lemma 1: Security of the APEX Authorisation Grant Protocol

As a refresher, Lemma 1 states:

> *The dual-mediated authorisation grant protocol* $\mathsf{DMAG}_\alpha$ *in an APEX suite* $RAD_\alpha$ *is AG secure, if it uses:*
>
> *a) an EUF-CMA secure MAC scheme; and*
>
> *b) one-time codes (OTCs) in an exponential space in the security parameter.*

The proof proceeds as a series of reductions, but first:

- Let $\gamma = \mathsf{DMAG}_\alpha$.
- Let $S_0, \ldots, S_6$ denote the events that $\mathcal{A}$ has won in games $\mathrm{G}_0, \ldots, \mathrm{G}_6$ that follow.

*Game 0.* This game is the authorisation grant (AG) security game as stated in §7.2 with no modifications. As per the definitions, $\mathcal{A}$ makes queries to the challenger as it chooses including, at most:

(1) fully corrupting all servers, consumers and owners but leaving at least one consumer and owner uncorrupted;
(2) with the exception of at least one owner, obtaining the data (except keys) stored by all parties and decrypted by each parties' keys; and
(3) decrypting all ACCE channels, all such that the freshness requirement (§6) is satisfiable in at least one run of $\mathsf{DMAG}_\alpha$.

As these actions alone afford the adversary no quantifiable advantage, the most that can be concluded at this point is that $\mathsf{Adv}^{\mathrm{G}_0}_{\mathcal{A},\gamma} = \mathsf{Adv}^{\mathsf{AG}}_{\mathcal{A},\gamma}$.

*Game 1.* This game handles the event in which a key used to authenticate data repeats for two different consumers or owners. If this condition were to occur, the adversary could, e.g., forge a MAC from one consumer by corrupting another. This could constitute a valid attack in the previous game as the freshness of one consumer will not always impact the freshness of another or the instances with which the second is partnered (according to the definition of instance partnering in §6).

The game therefore proceeds as in the preceding one except that $Ch$ aborts the game if:

(1) any owner $O_1$ generates a $O_1.ssk$ which is the same as that of another owner $O_2$ at any point (let $E_1$ denote this event);
(2) any consumer $C_1$ generates a $C_1.ssk$ which is the same as that of another consumer $C_2$ at any point (let $E_2$ denote this event); or
(3) any consumer $C_3$ generates a one-time code (OTC) which is the same as that of another $C_4$ at any point (let $E_3$ denote this event).

The probability of $E_1$ occurring can be bound using the likelihood of a cross-owner collision:

$$\Pr[E_1] \leq (n_{\mathcal{O}} - 1) \cdot \left(\frac{1}{|\mathcal{K}_{\mathsf{Sig}}|}\right)$$

Recall from §6 that $n_{\mathcal{O}}$ represents the number of owners. One (1) is subtracted from this number because a cross-owner collision can only occur if there is a minimum of two owners. $\mathcal{K}_{\mathsf{Sig}}$ represents the key space for the secret keys generated by signature scheme Sig.

The probabilities of $E_2$ and $E_3$ can be bound using the likelihood of a cross-consumer collision:

$$\Pr[E_2 \vee E_3] \leq (n_{\mathcal{C}} - 1) \cdot \left(\frac{1}{|\mathcal{K}_{\mathsf{Sig}}|} + \frac{1}{|\mathcal{K}_{\mathsf{OTC}}|}\right)$$

In a similar fashion to be previous bound, $n_{\mathcal{C}}$ represents the number of consumers and $\mathcal{K}_{\mathsf{Sig}}$ represents the key space for the secret keys generated by Sig. $\mathcal{K}_{\mathsf{OTC}}$ represents the key space from which OTCs are chosen.

It is worth noting that the size of $\mathcal{K}_{\mathsf{Sig}}$ and $\mathcal{K}_{\mathsf{OTC}}$ are assumed exponential in the security parameter $s$.

From this, it follows that:

$$\Pr[S_0] - \Pr[S_1] \leq \Pr[E_1 \vee E_2 \vee E_3]$$
$$\mathsf{Adv}^{\mathrm{G}_0}_{\mathcal{A},\gamma} - \mathsf{Adv}^{\mathrm{G}_1}_{\mathcal{A},\gamma} \leq \Pr[E_1 \vee E_2 \vee E_3]$$
$$\mathsf{Adv}^{\mathrm{G}_0}_{\mathcal{A},\gamma} \leq \mathsf{Adv}^{\mathrm{G}_1}_{\mathcal{A},\gamma} + (n_{\mathcal{O}} - 1) \cdot \left(\frac{1}{|\mathcal{K}_{\mathsf{Sig}}|}\right) + (n_{\mathcal{C}} - 1) \cdot \left(\frac{1}{|\mathcal{K}_{\mathsf{Sig}}|} + \frac{1}{|\mathcal{K}_{\mathsf{OTC}}|}\right)$$

*Game 2.* This game captures the chance of guessing a one-time code (OTC). If the adversary is able to do so, it can impersonate a consumer to an owner or vice versa.

The game therefore proceeds as in the preceding one except that $Ch$ additionally aborts if $\mathcal{A}$ correctly guesses an OTC generated by any $C \in \mathcal{C}$. Let $E_4$ denote this event.

The probability of $E_4$ can be bound as follows:

$$\Pr[E_4] \leq g \cdot n_{otc} \cdot \frac{1}{|\mathcal{K}_{\mathsf{OTC}}|}$$

Here $g$ is used to represent the polynomial maximum number of guesses that $\mathcal{A}$ will perform and $n_{otc}$ is the number of OTCs generated across all runs of $\gamma$.

From this, it follows:

$$\begin{aligned}\Pr[S_1] - \Pr[S_2] &\leq \Pr[E_4] \\ \mathrm{Adv}^{\mathrm{G}_1}_{\mathcal{A},\gamma} - \mathrm{Adv}^{\mathrm{G}_2}_{\mathcal{A},\gamma} &\leq \Pr[E_4] \\ \mathrm{Adv}^{\mathrm{G}_1}_{\mathcal{A},\gamma} &\leq \mathrm{Adv}^{\mathrm{G}_2}_{\mathcal{A},\gamma} + g \cdot n_{otc} \cdot \frac{1}{|\mathcal{K}_{\mathrm{OTC}}|}\end{aligned}$$

*Game 3.* This game captures the chance of forging a MAC calculated by an owner without the OTC (as the chance to obtain it has been eliminated in previous games). If an adversary were to succeed at this, it could impersonate a legitimate owner to a consumer.

The game therefore proceeds as in the preceding one except that $Ch$ additionally aborts if $\mathcal{A}$ is able to break the EUF-CMA security of $\mathsf{Mac}_{otc}(O.spk)$ in message 5 of Figure 3 to produce a valid MAC not present in the transcript of any fresh owner instance but verifiable using one-time code $otc$. This is once again only possible if $\mathcal{A}$ can win in the EUF-CMA game against MAC, yielding:

$$\mathrm{Adv}^{\mathrm{G}_2}_{\mathcal{A},\gamma} \leq \mathrm{Adv}^{\mathrm{G}_3}_{\mathcal{A},\gamma} + n_{otc} \cdot \mathrm{Adv}^{\text{EUF-CMA}}_{\mathcal{A},\mathsf{MAC}}$$

*Game 4.* This game, like the previous one, captures the chance of forging a MAC without the OTC, but this time a MAC calculated by a consumer rather than the owner. If an adversary were to succeed at this, it could impersonate a legitimate consumer to an owner.

The game therefore proceeds as in the preceding one except that $Ch$ additionally aborts if $\mathcal{A}$ is able to break the EUF-CMA security of $\mathsf{Mac}_{otc}(C.ppk)$ in message 6 of Figure 3 to produce a valid MAC not present in the transcript of any fresh consumer instance but verifiable using one-time code $otc$. This is only possible if $\mathcal{A}$ can win in the EUF-CMA game against the MAC scheme, yielding:

$$\mathrm{Adv}^{\mathrm{G}_3}_{\mathcal{A},\gamma} \leq \mathrm{Adv}^{\mathrm{G}_4}_{\mathcal{A},\gamma} + n_{otc} \cdot \mathrm{Adv}^{\text{EUF-CMA}}_{\mathcal{A},\mathsf{MAC}}$$

*Game 5.* This game captures the chance of guessing an owner's secret signing key directly. Succeeding at this would allow the adversary to trivially win in the previous game by signing a message of its choice with the guessed key.

The game therefore proceeds as in the preceding one except that $Ch$ additionally aborts if $\mathcal{A}$ correctly guesses $O.ssk$ for any $O \in \mathcal{O}$ which has participated in a run of $\gamma$. The probability of this can be bound as:

$$\mathrm{Adv}^{\mathrm{G}_4}_{\mathcal{A},\gamma} \leq \mathrm{Adv}^{\mathrm{G}_5}_{\mathcal{A},\gamma} + g \cdot n_{\mathcal{O}} \cdot \frac{1}{|\mathcal{K}_{\mathrm{Sig}}|}$$

*Game 6.* This game captures the chance of guessing a consumer's secret signing key directly. Succeeding at this would once again allow the attacker to win trivially.

The game therefore proceeds as in the preceding one except that $Ch$ additionally aborts if $\mathcal{A}$ correctly guesses $C.ssk$ for any $C \in \mathcal{C}$ which has participated in a run of $\gamma$. The probability of this can be bound as:

$$\mathrm{Adv}^{\mathrm{G}_5}_{\mathcal{A},\gamma} \leq \mathrm{Adv}^{\mathrm{G}_6}_{\mathcal{A},\gamma} + g \cdot n_{\mathcal{C}} \cdot \frac{1}{|\mathcal{K}_{\mathrm{Sig}}|}$$

*Conclusion for Lemma 1.* To successfully impersonate a consumer or owner at the conclusion of the authorisation grant phase, $\mathcal{A}$ must either obtain that party's private signing key or cause a consumer or owner to accept a fraudulent public key as belonging to a legitimate party.

The former cannot be achieved without guessing a signing key or exploiting a signing key which repeats. And the latter cannot be achieved without forging a MAC, guessing the OTC or exploiting a repeating OTC. Note that $\mathcal{A}$ cannot obtain the OTC by breaking channel security because it is never sent over the wire. And it cannot obtain it by corrupting the consumer or owner because that would violate the freshness requirement.

As such, in game 6, we have removed all appreciable options available to $\mathcal{A}$ for winning in the original AG game. As such, we can say that:

$$\mathrm{Adv}^{\mathrm{G}_5}_{\mathcal{A},\gamma} \leq negl(s)$$

where $negl$ is any negligible function and $s$ is the security parameter. From this, we are able to combine the above inequalities to yield the bound:

$$\begin{aligned}\mathrm{Adv}^{\mathrm{AG}}_{\mathcal{A},\gamma} \;\leq\; & 2n_{otc} \cdot \mathrm{Adv}^{\text{EUF-CMA}}_{\mathcal{A},\mathsf{Mac}} \;+\; (n_{\mathcal{O}} - 1) \cdot \left(|\mathcal{K}_{\mathrm{Sig}}|^{-1}\right) \\ & +\; (n_{\mathcal{C}} - 1) \cdot \left(|\mathcal{K}_{\mathrm{Sig}}|^{-1} + |\mathcal{K}_{\mathrm{OTC}}|^{-1}\right) \\ & +\; g \cdot \left(\frac{n_{otc}}{|\mathcal{K}_{\mathrm{OTC}}|} + \frac{n_{\mathcal{O}} + n_{\mathcal{C}}}{|\mathcal{K}_{\mathrm{Sig}}|}\right) \;+\; negl(s)\end{aligned}$$

Since $g$, $n_{otc}$, $n_{\mathcal{O}}$ and $n_{\mathcal{C}}$ are all polynomials in the security parameter, as long as Sig and MAC each have the properties required in **??** and the search space of the OTC is exponential, the probability that $\mathcal{A}$ wins in the AG game is negligible and therefore $\gamma$ is AG secure (§7.2). □

At this point, we are reaching the page-limit for ASIACCS. So, we politely send the reader to https://apex.anon.science/? for the last three proofs (Lemmas 3–5), which follow the same style as the above.