Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud

Arnar Birgisson
Chalmers University of Technology
arnarbi@gmail.com

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Úlfar Erlingsson, Ankur Taly,
Michael Vrable, and Mark Lentczner
Google Inc
{ulfar,ataly,mvrable,mzero}@google.com

***Abstract*—Controlled sharing is fundamental to distributed systems; yet, on the Web, and in the Cloud, sharing is still based on rudimentary mechanisms. More flexible, decentralized cryptographic authorization credentials have not been adopted, largely because their mechanisms have not been incrementally deployable, simple enough, or efficient enough to implement across the relevant systems and devices.**

This paper introduces *macaroons*: flexible authorization credentials for Cloud services that support *decentralized delegation* between principals. Macaroons are based on a construction that uses nested, chained MACs (e.g., HMACs [43]) in a manner that is highly efficient, easy to deploy, and widely applicable.

Although macaroons are bearer credentials, like Web cookies, macaroons embed *caveats* that *attenuate* and *contextually confine* when, where, by who, and for what purpose a target service should authorize requests. This paper describes macaroons and motivates their design, compares them to other credential systems, such as cookies and SPKI/SDSI [14], evaluates and measures a prototype implementation, and discusses practical security and application considerations. In particular, it is considered how macaroons can enable more fine-grained authorization in the Cloud, e.g., by strengthening mechanisms like OAuth2 [17], and a formalization of macaroons is given in authorization logic.

I. Introduction

Macaroons are authorization credentials that provide flexible support for controlled sharing in decentralized, distributed systems. Macaroons are widely applicable since they are a form of bearer credentials—much like commonly-used cookies on the Web—and have an efficient construction based on keyed cryptographic message digests [43].

Macaroons are designed for the Web, mobile devices, and the related distributed systems collectively known as the *Cloud*. Such modern software is often constructed as a decentralized graph of collaborative, loosely-coupled services. Those services comprise different protection domains, communication channels, execution environments, and implementations—with each service reflecting the characteristics and interests of the different underlying stakeholders. Thus, security and access control are of critical concern, especially as the Cloud is commonly used for sharing private, sensitive end-user data, e.g., through email or social networking applications.

Unfortunately, controlled sharing in the Cloud is founded on basic, rudimentary authorization mechanisms, such HTTP cookies that carry pure bearer tokens [21, 54]. Thus, today, it is practically impossible for the owner of a private, sensitive image stored at one Cloud service to email a URL link to that image, safely—given the many opportunities for impersonation and eavesdropping—such that the image can be seen only by logged-in members of a group of users that the owner maintains at another, unrelated Cloud service. Currently, this use case is possible only if the image, access group, and users are all at a single service, or if two Cloud services keep special, pairwise ties using custom, proprietary mechanisms (e.g., as done by Dropbox and Facebook [55]).

Of course, the ubiquitous use of bearer tokens is due to advantages—such as simplicity and ease of adoption—that cannot be overlooked. For example, bearer tokens can easily authorize access for unregistered users (e.g., to the shopping cart of a first-time visitor to a Cloud service) or from unnamed, transient contexts (e.g., from a pop-up window shown during private, incognito Web browsing). Such dynamic and short-lived principals arise naturally in distributed systems, like the Cloud and the "Grid" [47]. In comparison, most authorization mechanisms based on public-key certificates are not directly suited to the Cloud, since they are based on more expensive primitives that can be difficult to deploy, and define long-lived, linkable identities, which may impact end-user privacy [21].

Even so, the inflexibility of current Cloud authorization is quite unsatisfactory. Most users will have first-hand experience of the resulting frustrations—for example, because they have clicked on a shared URL, only to be redirected to a page requesting account creation or sharing of their existing online identity. Similarly, many users will have uncomfortably surrendered their passwords to use some Cloud service functionality, such as to populate an address book (e.g., on LinkedIn.com) or to aggregate their financial data (e.g., on mint.com).

Macaroons aim to combine the best aspects of using bearer tokens and using flexible, public-key certificates for authorization, by providing (i) the wide applicability, ease-of-use, and privacy benefits of bearer credentials based on fast cryptographic primitives, (ii) the expressiveness of truly decentralized credentials based on authorization logic, like SPKI/SDSI [14], and (iii) general, precise restrictions on how, where, and when credentials may be used.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
 ISBN 1-891562-35-5
http://dx.doi.org/doi-info-to-be-provided-later

Macaroons allow authority to be *delegated* between protection domains with both *attenuation* and *contextual confinement*. For this, each macaroon embeds *caveats* which are predicates that restrict the macaroon's authority, as well as the context in which it may be successfully used. For example, such caveats may attenuate a macaroon by limiting what objects and what actions it permits, or contextually confine it by requiring additional evidence, such as third-party signatures, or by restricting when, from where, or in what other observable context it may be used. In particular, a macaroon that authorizes service requests may contain caveats that require proof that the requests have been audited and approved by an abuse-detection service, and come from a specific device with a particular authenticated user.

In short, Macaroons allow Cloud services to authorize resource access using efficient, restricted bearer tokens that can be delegated further, and, via embedded caveats, attenuated in scope, subjected to third-party inspection and approval, and confined to be used only in certain contexts. Fundamental to this flexibility of macaroons is a chained construction of nested HMAC values that simultaneously provides key distribution and protects integrity, generalizing earlier work dating back to the Amoeba operating system [3, 25, 51].

II. Motivation and Foundations

Conceptually, macaroons make an assertion about conditional access to a *target service*[1], along these lines: "The bearer may perform a request *as long as* predicates $C_1, \ldots, C_n$ hold true in the context of that request."

In distributed systems, the complexity of such assertions can grow surprisingly quickly, even for simple, practical authorization policies. For example, if, as in Section I, the owner of a private image at *TS* wants to safely email its URL link to the members of a group *G* kept at an unrelated service *A*, macaroons may establish an assertion like the following:

The bearer, client *C*, may perform the request at *TS*
as long as the operation is read,
as long as the object is `privateImage.jpg`,
as long as the user at *C* is logged into service *A*,
as long as that logged-in user is in group *G* at *A*,
as long as the above proofs are recent/fresh enough,
and, *as long as* the user at *C* didn't just log out of *A*.

Furthermore, the image may be subject to additional access controls at *TS* (e.g., to track changes of ownership), the request from client *C* may be performed from an unnamed context (e.g., a transient Web-page frame), and the logged-in user's privacy may need to be protected from *TS*. (Variants of this assertion are the basis for paper's examples and the end-to-end evaluation in Section VII.)

Due to their flexible, efficient HMAC-based construction, macaroons can enforce such complex assertions despite tight freshness constraints and the need to closely guard privacy.

Macaroons are an instance of what has been termed *claims-based*, *proof-carrying*, or simply *credentials-based authorization*, and is defined precisely in [48, Chapter 9]. Over the last two decades, several such credentials-based mechanisms have been developed using public-key certificates and the formalism of authorization logic [37, 53]. Of these, many support highly flexible, decentralized authorization for distributed systems, with notable examples including SPKI/SDSI [14], Trust Management [11], active certificates [12], and SecPAL [9]. However—whether due to the cost of their primitives, their need for federation, coordination, or public identities, or the complexity of their implementation—such flexible public-key-certificate mechanisms have seen little use in the Cloud [54].

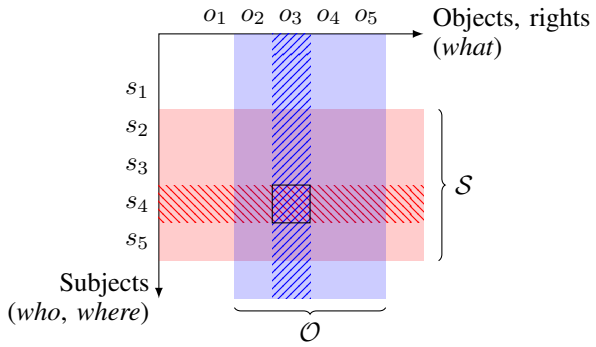


Fig. 1. Macaroons in the Access Control Matrix model [36]. By attenuating a macaroon authorizing $\mathcal{O}$, a derived macaroon may permit access only to o_3. Even when held by all subjects in $\mathcal{S}$, contextual caveats may confine the use of this derived macaroon to the observable context of principal s_4.

Instead, near exclusively, authorization in the Cloud is based on a pattern of pure *bearer tokens*: secrets that grant unconditional authority, and are sent directly between protection domains, even on unencrypted channels [21]. Examples of pure bearer tokens include the HTTP cookies sent to identify authorized sessions, the secrets for API keys, OAuth tokens, and SAML or OpenID assertions used at other protocol layers, as well as the "unlisted URLs" that are commonly created and shared via email or chat to provide limited access to private content [28, 50, 54].

Efficiency, wide applicability, and ease-of-adoption are three main reasons for bearer tokens' popularity in the Cloud. Macaroons share all three of these properties, in their HMAC-based construction. First, macaroons are simple enough to implement, and fast-enough to use in near any environment—even a severely-limited, out-of-date Web browser executing on slow, resource-constrained mobile hardware. Second, macaroons are *bearer credentials* and, so, principals may gain authority simply by gathering macaroons. Third, macaroons still provide new benefits even when unilaterally adopted, e.g., by allowing users to restrict when, and from where, credentials may be used [21].

By supporting *attenuation*, *delegation*, and *contextual confinement*, macaroons have much greater expressiveness than previously existing Cloud bearer tokens. Thus, as shown in Figure 1, a user that holds access rights to the objects in $\mathcal{O}$ can safely pass, via email, a macaroon for the right to access a single object o_4, limited to the specific recipient s_4, even if all the parties in $\mathcal{S}$ can read the email. Moreover, as shown in Figure 2, the user can do this without the operational difficulty or latency and bandwidth costs of running an intermediate proxy service, and even without contacting the target service to mint new, restricted credentials.

[1]This section shows key terms and new concepts in *italics* when introduced and informally defined; the next two sections and the appendix provide detailed definitions and describe concrete implementations.

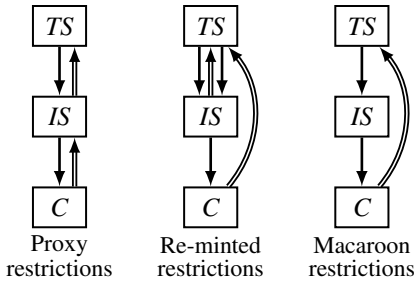


Fig. 2. Three ways for an intermediary service *IS* to give a client *C* limited access to a target service *TS*: by proxying requests, by having *TS* mint new, restricted credentials for *C*, or by *IS* deriving new macaroons for *C*. Arrows show credential passing and doubled arrows indicate access.

In their expressiveness, and in aspects of their construction, macaroons are related to cascaded proxy certificates [44], restricted delegation [33], active certificates [12], proof-carrying authentication [6], and Grid computing [22], as well as systems for limiting resource consumption [23] and access to networked storage services [3, 25]. Indeed, compared to systems based on authorization logic like SPKI/SDSI [14], it must be stated explicitly that macaroons have no real advantage other than their efficiency and ease of adoption. In particular, in terms of functionality and security, HMAC-based macaroons have the clear disadvantage of being verifiable only by the target service, and by revealing to it certain keys, which prevents useful types of key reuse. (This is detailed in Sections IV-D and V-B and Section V-H outlines a public-key variant of macaroons eliminating this disadvantage.) However, considering the macaroons' efficiency and flexibility, remarkably few such trade-offs must be made for their practical deployment.

In particular, revocation of authority is a long-standing challenge for credentials-based authorization systems, as detailed in [19, Section 6] and [48, Chapter 9.8]. Broadly, this challenge has been addressed by (i) using very short-lived credentials, (ii) allowing the addition of freshness constraints, (iii) relying on external, authoritative state (such as revocation lists, or epoch counters), and (iv) splitting credentials, and granting authority only to a freshly-gathered collection. Macaroons excel at all four of these revocation strategies: due to their efficiency, and flexibility, macaroons may live only for seconds, and may be split or constrained in any manner, as described in the following sections.

A. Concepts and Constructions

Macaroons operate from a service's point-of-view, in that they permit a given *target service*, such as a Cloud resource server[2], to mint a bearer credential that grants authority to access some aspects of the service. A distinguishing characteristic of macaroon credentials is that their bearer can delegate parts of their authority to other principals by deriving new macaroons that both *attenuate* the accessible aspects of the target service and also restrict, via *contextual confinement*, from where, by who, and with what extra evidence, the derived macaroon may be used.

To restrict the authority of a derived macaroon, *caveats* are added and embedded within it. Each macaroon caveat states a predicate that must hold true for the context of any requests that the derived macaroon is to authorize at the target service—with that predicate evaluated by the target service in the context of each particular request. Notably, as shown in Figure 2, by adding caveats, new, more restricted authorization credentials can be directly derived from macaroons, without proxying requests to the target service, or re-minting new credentials from it. Thus, macaroons can help avoid the latency and overhead, loss of transparency, and trust requirements imposed by the proxying and credential re-minting patterns commonly used in the today's Cloud.

For example, every macaroon is likely to contain one or more validity-period caveats whose predicates restrict the time (at the target service) during which the macaroon will authorize requests; similarly, each macaroon is likely to have caveats that restrict the arguments permitted in requests. A macaroon may have multiple caveats on the same attribute, such as time, in which case all the caveats' predicates must hold true in the context of requests.

It is straightforward to add and enforce such *first-party caveats* that confine the context observed at the target service when a macaroon is used. In the Cloud, such restrictions on bearer tokens are sometimes used to reduce the ease by which tokens can be stolen and abused by attackers, while retaining the flexibility and privacy benefits of token-based authorization [5, 13, 17, 31]. Because requests are sent directly from clients (as shown in Figure 2), target services need only check the predicates in such caveats against the context of incoming requests, for example to restrict the source IP address (or, better yet, ChannelID [21]).

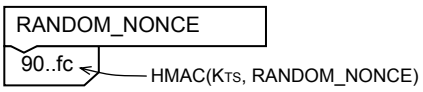
However, macaroons' flexibility stems mostly from *third-party caveats*, their main innovation, which allow a macaroon to specify and require any number of holder-of-key proofs[3] to be presented with authorized requests. For example, instead of using time-based validity-period caveats, macaroons might use caveats for a third-party revocation-checking service, and require authorized requests to present fresh proofs discharging those caveats. Such third-party caveats can ensure that each request authorized by a macaroon is subject to any number of extra steps for authentication, authorization, or audit—e.g., to name just a few, checks of network time, authentication at identity providers, auditing and provenance tracking at document repositories, scanning by anti-virus engines, and vetting by a DoS-prevention and risk-analysis service [29].

Even when macaroons are used to enforce complex authorization policies, there is no particular need for end users to be aware of macaroons. In any construction, client-side macaroons will be handled and processed not by users, but by software such as client-side JavaScript in Web browsers. Thus, users can remain equally unaware of macaroons as they are of current Cloud authorization tokens.

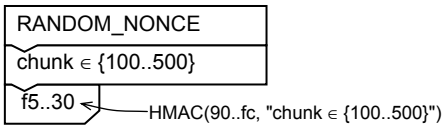
Revisiting the example at the start of this section, consider a construction where all macaroons are encoded into "unlisted URLs." Thus, instead of a HTTP session cookie, the owner may hold a macaroon URL granting them full access to their

[2]Protocols like OAuth2 [17] use the term *relying party*, not target service.

[3]In distributed authorization, the term-of-art *holder-of-key proof* is used when a party is authenticated by proving possession of a key, e.g., by signing a protocol message with a private key [30]. Macaroons use holder-of-key proofs extensively, in particular for third-party-caveat discharges.



Above is an example of the simplest possible macaroon credential. If it grants bearers access to a key-value storage service for chunks, it can be attenuated by adding a caveat whose predicate restricts accessible chunks, as below:



To check the macaroons' integrity, the storage service can compute and verify the terminal, derived HMAC value, even when the resulting macaroon is further attenuated with a caveat whose predicate permits only reading, as below:

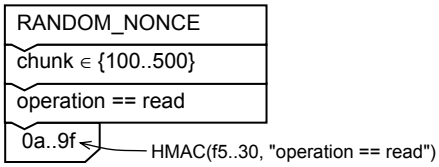


Fig. 3. Three macaroons, respectively giving full access to all chunks at a key-value storage service, permitting all requests to chunks in the range 100 to 500, or allowing only reading from that range. Starting at its key KTS, the service can verify the integrity of each macaroon by deriving its chain of nested HMAC values. Macaroons are presented with requests, and the service permits only requests that satisfy the macaroons' caveats.

private images at *TS*. From this URL, the owner can create a new, derived macaroon URL for read-only access to an image, embed within it a third-party caveat that requires proof of membership in a group *G* at service *A*, and share it via email. Any recipient of this derived URL would have to get a fresh discharge for the third-party caveat—which may require the user to log into *A*, via a redirection—and pass it along to *TS* when using the URL to view the image. All of the above could be accomplished using software, such that both parties remained unaware of the use of macaroons.

III. Concrete Mechanisms and Examples

Concretely, a macaroon granting authority to a Cloud server is constructed from a sequence of messages and a chain of keyed cryptographic digests derived from those messages. This chain of digest values is computed using HMAC functions with appropriate truncation (e.g., to 128 bits) [43]. Each macaroon contains, however, only the terminal, final HMAC value in the chain, with this value serving as the macaroon's *signature*: a generalized message authentication code that (i) allows target services to verify the integrity of macaroons, and (ii) also allows shared keys to be established with the bearers of any derived macaroons.

Fundamental to macaroons is how this chain of keyed HMAC values is constructed, in a nested fashion, along the sequence of messages in a macaroon—of which, all but the first are caveats. The first message in a macaroon is required to be a public, opaque *key identifier* that maps to a secret *root key* known only to the target service. (Such key identifiers can be implemented, e.g., using random nonces, indices into a database at the target service, keyed HMACs, or using public-key or secret-key encryption; see [37, Sections 4.2 and 4.3] for a discussion.) Starting with this root key, a macaroon's chain of HMAC values is derived by computing, for each message, a keyed HMAC value, nested so that each such value itself is used as the HMAC key for the next message. Thus, a macaroon with two messages *MsgId* and *MsgCav* and root key K_R could have the signature $\mathrm{HMAC}(K_{tmp}, \mathit{MsgCav})$, where K_{tmp} is $\mathrm{HMAC}(K_R, \mathit{MsgId})$.

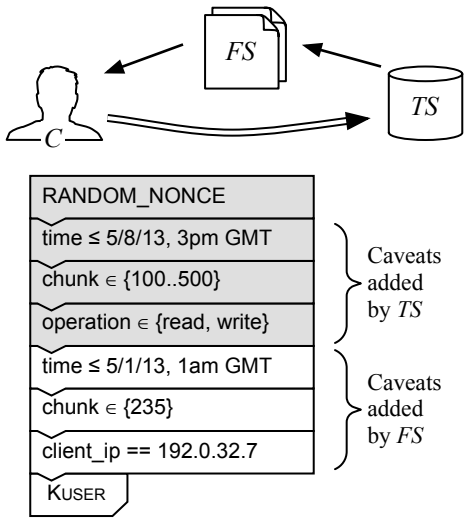


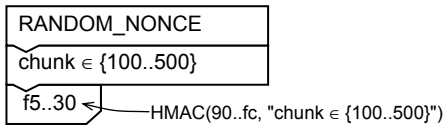
Fig. 4. A macaroon held by the user at client *C*, authorizing limited-time access to chunk 235 at a key-value storage service *TS* from one network IP address. This macaroon is derived from another (shown shaded in gray) that a social-networking "forum service" *FS* holds for a range of chunks at *TS*—delegating and attenuating the authority of that macaroon.

An example of this concrete macaroon construction is shown in Figure 3. (This example, like others in this paper, draws from the distributed systems literature where related chained-message-digest constructions have long been used for authorizing access to networked storage [3, 25, 51].)

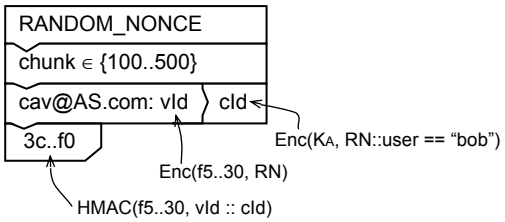
In Figure 3, starting from a macaroon with a random nonce key identifier, and a secret key KTS held at the target service, two macaroons are successively derived by adding caveat predicates and chaining the HMAC signatures. Each macaroon authorizes requests to a target service that offers key-value storage for *chunks*, with caveats in the derived macaroons restricting the authorized chunks and operations. To verify the integrity of these macaroons, the target service need only check their signature against the HMAC values 90..fc, f5..30, or 0a..9f, derived using its key KTS.

This construction makes it particularly simple to delegate and attenuate authorization credentials, since the chain of HMAC signatures establishes shared keys between the target service and the bearers of successive, derived macaroons.

For example, Figure 4 shows a macaroon authorizing a user at client *C* to access chunk 235 at a key-value storage service *TS*. This macaroon has been derived as a strict attenuation of another, more permissive macaroon—held by the social-networking "forum service" *FS*—by both eliding that macaroon's signature (called, say KFS), and by adding caveats whose predicates require a specific chunk and client IP address to be present in the request context. Even so, this macaroon's signature, KUSER, both allows the target storage service to verify its integrity, and also allows its bearer to



The above macaroon from the example in Figure 3 permits access to a range of chunks. By deriving a macaroon with an added third-party caveat, that authority can be delegated to a user "bob" at an authentication service, as below:



A third-party caveat message contains an root key and assertion encrypted for the discharging service (above, cId), a hint to its location (above, AS.com), and the same key encrypted for the target service to allow verification (above, vId). The signature of a discharge macaroon (below) is chained from this root key.

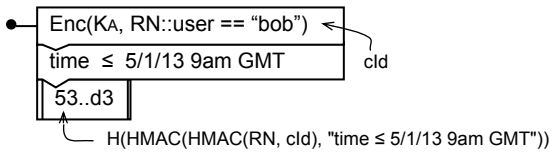


Fig. 5. A macaroon with a third-party caveat (in the middle), derived from another macaroon (at the top). At the bottom is a discharge macaroon for the caveat (whose signature is hashed to prevent further chaining), created by a third-party holder of the shared key KA after checking the assertion in the key identifier cId. By deriving the chain of nested HMAC values, the target service can verify the integrity of such discharge macaroons by decrypting the encrypted root key identifier vId.

derive new, further attenuated macaroons. In this manner, macaroon signatures both protect integrity and act as keys.

The third-party caveats in macaroons are requirements for holder-of-key proofs: assertions of one or more predicates signed by the holder of a key. The manner in which these requirements are embedded into a macaroon, and its chain of HMAC values, allows (i) the appropriate parties to create proofs asserting those predicates, (ii) those proofs, themselves, to embed additional caveats (even for new, additional third parties), and, (iii) the target service to verify that all required proofs have been discharged, recursively.

A proof that discharges a macaroon's third-party caveat is itself a macaroon, as shown in Figure 5 (at the bottom). Such caveat discharge macaroons have a key identifier that encodes both the root key for deriving holder-of-key proofs and the predicates to be asserted by the third party (in Figure 5, it is denoted cId, and contains the key RN paired with the caveat predicate user == "bob" encrypted under the key KA). Conceptually, this key identifier can be thought of as a message encrypted with a public key for the third-party.

This same caveat key identifier is also contained in the message that a macaroon embeds for a third-party caveat, along with a hint to the location of caveat-discharging service (in Figure 5, that hint is @AS.com). This message also contains another, encrypted copy of the root key for holder-of-key proofs, encrypted with a key from the signature of

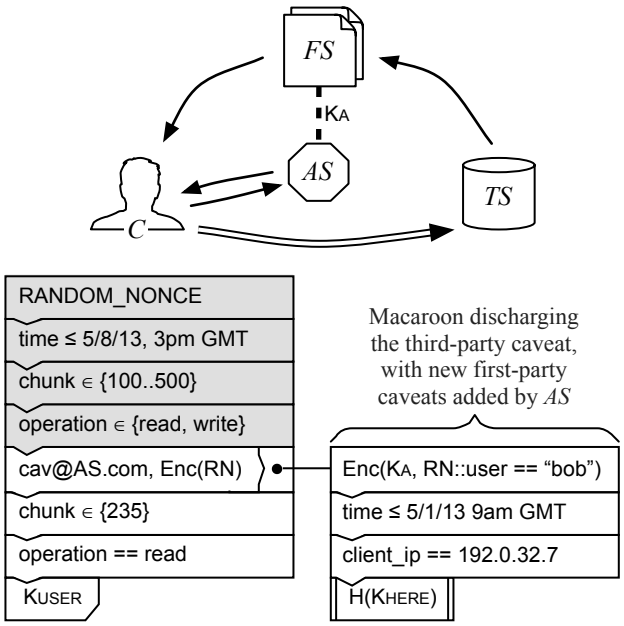


Fig. 6. Two macaroons that together give client *C* limited-time read access to chunk 235 at the key-value storage service *TS*. Using its macaroon from *TS*, the "forum service" *FS* derives for *C* a macaroon (on the left) that embeds a third-party caveat for an authentication service *AS*, confining access to clients *C* where a user "bob" is logged into *AS*. That caveat contains an encrypted root key and a key identifier. Using that key identifier, *AS* derives for *C* a macaroon (on the right) that asserts, for only a limited time, and for only one IP address, that *AS* has verified client *C*'s user.

the macaroon onto which the third-party caveat is added (in Figure 5, it is denoted vId, and is the root key RN, encrypted using the HMAC value f5..30). The target server can decrypt this second copy of the root key, and use it to verify the signatures of caveat discharge macaroons.

In general, to authorize a request, a target service may need to verify an entire set of macaroons presented with that request, and establish (i) that caveat discharge macaroons can be found for all third-party caveats, and (ii) that all first-party caveat predicates hold true in the context of the request. Target services can perform this verification by inducing a tree from the set of presented macaroons, and checking their macaroon signatures against the chain of HMAC values derived along each branch from the root to the leaves. To build this tree, the messages in the third-party caveats allow the target service to match the key identifiers of caveat discharge macaroons and decrypt their root keys—in Figure 5, respectively, the identifier Enc(KA, RN::user == "bob") and root key Enc(f5..30, RN).

The main advantages and innovative aspects of macaroons are illustrated by the example of Figure 6, in combination. From its existing macaroon for the service *TS*, the forum service *FS* can derive a macaroon for user "bob" that attenuates *what* aspects of *TS* are accessible; simultaneously, *FS* can delegate the authentication of users to a third-party service *AS*, using the key KA it holds for *AS*. (Of course, for this, *FS* must know *AS*, and trust it to perform authentication.) The *AS* service can evaluate the request from client *C* however it chooses, before returning a macaroon that discharges the user == "bob" caveat. Notably, this caveat discharge macaroon may be very short lived (e.g., be valid for minutes, or seconds), due to the efficiency and low overhead macaroons' HMAC-based

construction. Also, *AS* can also add arbitrary further caveats to that returned macaroon—even third-party caveats, although this is not shown in Figure 6—that restrict the context *where*, and when borne by *who*, the macaroon credential should grant any authority. In particular, those further caveats can ensure that the user is still logged in, by requiring a fresh holder-of-key proof based on a key held by the login agent at *C*. Finally, using macaroons, all of the above can be accomplished without the target service *TS* having any direct relationship with *AS*, and without *TS* knowing that the request from *C* is on behalf of a user "bob" (thus protecting the user's privacy).

This section has left many technical details of macaroons to be further considered. The following sections cover these aspects, with the next two sections and the appendix providing a precise definition of macaroons.

IV. Design, Operations, and Security

This section formally defines the notation, structure, and operations of macaroons, precisely defines macaroon-based authorization like that in the example of Figure 6, and also considers the security and expressiveness of decentralized authorization mechanisms based on macaroons.

Macaroons use primitives whose domain of values are $Keys$ for all cryptographic keys, $Locs$ for the locations of all services, and $BitStrs$ for all bit-string values. Keys and locations are all bit-strings; thus $Keys \cup Locs \subseteq BitStrs$. Macaroons are based on a secure, keyed cryptographic hash function MAC, and its output message authentication codes, as well as secure encryption and decryption functions Enc and Dec, with the following type signatures:

$$\begin{aligned} \mathsf{MAC} &: Keys \times BitStrs \to Keys \\ \mathsf{Enc} &: Keys \times BitStrs \to BitStrs \\ \mathsf{Dec} &: Keys \times BitStrs \to BitStrs \end{aligned}$$

It is assumed that each key output by a MAC operation can be used as a valid key for another MAC operation, or for symmetric-key-based Enc and Dec operations. Using this, the nested *chained*-MAC for a list of bit-strings $[b_1, \ldots, b_n]$, under a key k, is defined as as $\mathsf{MAC}(\ldots \mathsf{MAC}(\mathsf{MAC}(k, b_1), b_2) \ldots, b_n)$, where $\mathsf{MAC}(k, b_1)$ is used as the key for computing the MAC of b_2, which in turn is used as the key for computing the MAC of b_3, and so on.

Principals are each associated with a location, a set of keys and macaroons, and the ability to manage secrets; the universe of all principals is denoted $Prncs$. For each secret they hold, principals are assumed to be able to construct a public key identifier that confidentially, and appropriately conveys this secret. Such key identifiers have long been used in security protocols, with each acting as a certificate that reminds a principal P of a certain secret it holds; their specifics may vary across principals, with possible implementations including secret-key certificates based on symmetric, shared-key encryption as well as simple, ordinal identifiers that index into a principal's persistent, confidential database (see [18] and [37, Section 4.3] for a discussion).

Importantly for macaroons, key identifiers provide a means of authentication via holder-of-key proofs: a principal P may be presented with a key identifier and asked to prove knowledge of the secret, or some derivative of the secret.

Definition: A macaroon M is a tuple of the form

$$\mathsf{macaroon}_{@L}\langle id, \mathcal{C}, sig\rangle$$

where,

* $L \in Locs$ (optional) is a hint to the target's location.
* $id \in BitStrs$ is the macaroon identifier.
* $\mathcal{C}$ is a list of caveats of the form $\mathsf{cav}_{@cL}\langle cId, vId\rangle$, where
  - $cL \in Locs$ (optional) is a hint to a discharge location.
  - $cId \in BitStrs$ is the caveat identifier.
  - $vId \in BitStrs$ is the verification-key identifier.
* $sig \in Keys$ is a chained-MAC signature over the macaroon identifier id, as well as each of the caveats in $\mathcal{C}$, in linear sequence.

Fig. 7. Definitions of the components of macaroons and their caveats.

A. Structure of Macaroons and Caveats

Every macaroon is associated with a root key, and is minted by a target service principal, henceforth called the *target*. Macaroons begin with a key identifier for the target, followed by a list of caveats and a *macaroon signature*, computed using the root key. The key identifier, or *macaroon identifier*, conveys the secret root key and (optionally) a list of predicates to the target service. The macaroon signature is the chained-MAC of the identifier and caveats, in sequence, under the root key.

Both first-party and third-party caveats are defined using the same structure that consists of two identifiers: a *caveat identifier* meant for the caveat's discharging principal, and a *verification-key identifier* meant for the embedding macaroon's target service, with the embedding macaroon's signature ensuring the integrity of both identifiers.

For third-party caveats, the caveat identifier encodes within it one or more predicates and a root key, henceforth called the *caveat root key*; its construction is held abstract and may vary across caveat-discharging principals. The verification-key identifier encodes within it only the caveat root key, encrypted using the current signature from the embedding macaroon; the target can decrypt this identifier to obtain the root key, since it can always recover all intermediate signatures in a macaroon. On the other hand, for first-party caveats the caveat identifier is simply the authorization predicate in the clear, and the constant 0 is used as the verification-key identifier. Caveats also have an optional *location*, not covered by the signature of the embedding macaroon, which provides a hint for where the principal that can discharge a caveat may be contacted.

The target considers a macaroon's third-party caveat to be discharged if it finds a *caveat discharge macaroon* that begins with the caveat identifier and has a chained-MAC signature that is correctly constructed using the caveat root key that the target decodes from the verification-key identifier. Such a caveat discharge macaroon may have other third-party caveats, for which the target must recursively seek discharges. On the other hand, a first-party caveat is discharged by checking that its authorization predicate holds true, when evaluated in the context of the request.

Figure 7 provides a definition of macaroons and their caveats. Macaroon and caveat locations are left optional, and without integrity, both out of practical concerns (address and

$\mathsf{CreateMacaroon}(k, id, L)$
  $sig := \mathsf{MAC}(k, id)$
  return $\mathsf{macaroon}_{@L}\langle id, [\,], sig\rangle$

$M.\mathsf{addCaveatHelper}(cId, vId, cL)$
  $\mathsf{macaroon}_{@L}\langle id, \mathcal{C}, sig\rangle \leftarrow M$ *// $\leftarrow$ is pattern matching*
  $C := \mathsf{cav}_{@cL}\langle cId, vId\rangle$
  $sig' := \mathsf{MAC}(sig, vId :: cId)$ *// :: is pair concatenation*
  return $\mathsf{macaroon}_{@L}\langle id, \mathcal{C} \triangleright C, sig'\rangle$ *// $\triangleright$ is list append*

$M.\mathsf{AddThirdPartyCaveat}(cK, cId, cL)$
  $vId := \mathsf{Enc}(M.sig, cK)$
  return $M.\mathsf{addCaveatHelper}(cId, vId, cL)$

$M.\mathsf{AddFirstPartyCaveat}(a)$
  return $M.\mathsf{addCaveatHelper}(a, 0, \top)$

$M.\mathsf{PrepareForRequest}(\mathcal{M})$
  $\mathcal{M}^{sealed} := \emptyset$
  for $M' \in \mathcal{M}$
    $\mathsf{macaroon}_{@L'}\langle id', \mathcal{C}', sig'\rangle \leftarrow M'$
    $sig'' := M.\mathsf{bindForRequest}(sig')$
    $\mathcal{M}^{sealed} := \mathcal{M}^{sealed} \cup \{\mathsf{macaroon}_{@L'}\langle id', \mathcal{C}', sig''\rangle\}$
  return $\mathcal{M}^{sealed}$

$M.\mathsf{Verify}(TM, k, \mathcal{A}, \mathcal{M})$
  $cSig := \mathsf{MAC}(k, M.id)$
  for $i := 0$ **to** $|M.\mathcal{C}| - 1$
    $\mathsf{cav}_{@L}\langle cId, vId\rangle \leftarrow M.\mathcal{C}[i]$
    $cK := \mathsf{Dec}(cSig, vId)$
    if $(vId = 0)$
      assert $(\exists a \in \mathcal{A} : a = cId)$
    else
      assert $\left(\exists M' \in \mathcal{M} : \begin{array}{l} M'.id = cId \\ \wedge\ M'.\mathsf{Verify}(TM, cK, \mathcal{A}, \mathcal{M}) \end{array}\right)$
    $cSig := \mathsf{MAC}(cSig, vId :: cId)$
  assert $(M.sig = TM.\mathsf{bindForRequest}(cSig))$
  return `true`

Fig. 8. The operations essential to authorization using macaroon credentials.

name redirection is common, in the Cloud) and also to honor the maxim that, in distributed systems, only keys speak [53].

B. Macaroon Operations

Figure 8 shows the operations to create and extend macaroons, prepare them for use in a request, and verify them at a target service. The operations are written as pseudocode, with semantic clarity in mind. For a macaroon M, the notations $M.id$, $M.cavs$ and $M.sig$ are used to refer to its identifier, list of caveats, and signature, respectively.

Creating macaroons: Given a high-entropy root key k and an identifier id, the function $\mathsf{CreateMacaroon}(k, id)$ returns a macaroon that has the identifier id, an empty caveat list, and a valid signature $sig = \mathsf{MAC}(k, id)$.

Adding caveats: Third-party and first-party caveats can be added to a macaroon M using the methods $M.\mathsf{AddThirdPartyCaveat}$ and $M.\mathsf{AddFirstPartyCaveat}$ respectively. $M.\mathsf{AddThirdPartyCaveat}$ takes as input a caveat root key cK, a caveat identifier cId, and a location cL of the caveat's discharging principal. It first computes a verification-key identifier vId by encrypting the key cK with the macaroon signature of M as the encryption key. Next, using the method $M.\mathsf{addCaveatHelper}$, it adds the caveat $\mathsf{cav}_{@cL}\langle cId, vId\rangle$ to the caveat of M, and updates the signature to the MAC of the pair $vId :: cId$, using the existing signature as the MAC key.

The $M.\mathsf{AddFirstPartyCaveat}$ operation takes as input an authorization predicate a, and adds it using the $M.\mathsf{addCaveatHelper}$ method, as the caveat $\mathsf{cav}_{@\top}\langle a, 0\rangle$ to the caveat list of M. Here, $\top$ is a well-known location constant that refers to the target service.

Preparing requests: While making an access request, a client is required to provide an authorizing macaroon along with discharge macaroons for the various embedded third-party caveats. These discharge macaroons are powerful holder-of-key proofs that may be used to satisfy obligations in any and all macaroons that embed the corresponding third-party caveat identifier that uses the same root key.

In particular, an attack is possible if the client accidentally makes a request to a principal other than the original target (perhaps by falling prey to phishing). In this case, this other principal can maliciously re-use any discharge macaroons it receives to discharge third-party caveats embedded in macaroons for itself, thereby authorizing itself using contextual caveat discharges from the client. To prevent such malicious re-use, all discharge macaroons are required to be bound to the authorizing macaroon before being sent along with a request to the target. This binding is carried out by the method $M.\mathsf{PrepareForRequest}(\mathcal{M})$, which binds the authorizing macaroon M to each discharge macaroon M' in the list $\mathcal{M}$ by modifying their signature to $M.\mathsf{bindForRequest}(M')$. Here, $M.\mathsf{bindForRequest}(M')$ is kept abstract, but one possible implementation would be to hash together the signatures of the authorizing and discharging macaroons, so that $M.\mathsf{bindForRequest}(M') = \mathsf{H}(M'.sig :: M.sig)$.

Verifying macaroons: In order to verify an incoming access request consisting of an authorizing macaroon TM and a set of discharge macaroons $\mathcal{M}$, a target service must ensure that all first-party and third-party caveats embedded in either TM or any macaroon in $\mathcal{M}$ are satisfied. For the purpose of formalization, this task is simplified by assuming that the target service first generates a set $\mathcal{A}$ of all embedded first-party caveat predicates that hold true in the context of the request whose macaroon is to be verified.

To authorize the request, the target service invokes the method $TM.\mathsf{Verify}(TM, k, \mathcal{A}, \mathcal{M})$ where k is the root key of macaroon TM. The method iterates over the list of caveats in TM and checks each of them. For each embedded first-party caveat $\mathsf{cav}_{@\top}\langle a, 0\rangle$, it checks if the predicate a appears in $\mathcal{A}$. For each embedded third-party caveat $\mathsf{cav}_{@L_c}\langle cId, vId\rangle$ in TM, it extracts the root key cK from vId, and then checks if there exists a macaroon $M' \in \mathcal{M}$ such that (i) M' has cId as its macaroon identifier and (ii) M' can be recursively verified by invoking $M'.\mathsf{Verify}(TM, cK, \mathcal{A}, \mathcal{M})$. Finally it checks that the signature of the current macaroon is a proper chained-MAC signature bound to the authorization macaroon TM.

C. Example Authorization Using Macaroons

This section outlines in detail the macaroon-based authorization flow for the paper's running example, previously described in Sections II and III and illustrated in Figure 6.

Minting the macaroon at the target service: The flow begins with the target service (the storage service) constructing a macaroon M_{TS} which grants time-limited read/write access to a subset of the storage chunks. To do so it generates a fresh root key k and a public identifier $\{k\}_{TS}$ (which, recall, allows TS to retrieve k).

$M_0 := \mathsf{CreateMacaroon}(k, \{k\}_{TS})$
$M_1 := M_0.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{chunk} \in \{100 \ldots 500\}\text{”})$
$M_2 := M_1.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}\text{”})$
$M_{TS} := M_2.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{time} < 5/1/13\ 3\mathsf{pm}\text{”})$

The macaroon M_{TS} is handed to the forum service.

Attenuating the macaroon at the forum service: The forum service adds caveats to the macaroon before passing it on, in particular, it adds a third-party caveat requiring the user authenticate as "*bob*" at AS.com. For this, the forum service must trust and have a relationship with AS.com, and there are many options for how they interact. For example, the forum service can mint a fresh caveat root key cK and communicate with AS.com to obtain a caveat key identifier for it:

$FS \longrightarrow AS : cK, \text{“}\mathsf{user} = bob\text{”}$
$AS \longrightarrow FS : \{(cK, \text{“}\mathsf{user} = bob\text{”})\}_{AS} = cId$
$M_3 := M_{TS}.\mathsf{AddThirdPartyCaveat}$
$\qquad (cK, \{(cK, \text{“}\mathsf{user} = bob\text{”})\}_{AS}, \mathsf{AS.com})$

Alternately, if AS publishes a public key, or if AS and the forum service have previously shared a symmetric key, the forum service can construct the caveat key identifier itself by encrypting cK and the predicate. All parties in the message exchanges above are are assumed to always perform proper authentication, and establish confidentiality and integrity, using an out-of-band mechanism like TLS [20].

The forum service further attenuates the macaroon so that it only grants read-only access to a single chunk:

$M_4 := M_3.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{chunk} = 235\text{”})$
$M_{FS} := M_4.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{operation} = \mathsf{read}\text{”})$

Subsequently, the macaroon M_{FS} is passed on to the user.

Acquiring discharges from the authentication service: The user scans the macaroon M_{FS} for third-party caveats to to discharge. Finding one, the user sends cId to AS.com, where AS extracts cK and the predicate from cId and, if the predicate is true, creates a discharge macaroon:

$M'_0 := \mathsf{CreateMacaroon}(cK, cId)$
$M'_1 := M'_0.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{time} < 5/1/13, 9\mathsf{am}\text{”})$
$M'_{AS} := M'_1.\mathsf{AddFirstPartyCaveat}(\text{“}\mathsf{ip} = 192.0.32.7\text{”})$
$AS \longrightarrow U : M'_{AS}$

To check predicates, AS may perform extra work, e.g., to redirect the user to authenticate as "*bob*" using a password. After receiving M'_{AS}, the user adds it to their set of discharges $\mathcal{M}_U$, and if it contains any new third-party caveats, the user also collects discharges for those, recursively.

Making an access request using macaroons: The user first binds all discharge macaroons in $\mathcal{M}_U$ to the authorizing macaroon M_{FS}:

$\mathcal{M}_U^{sealed} := M_{FS}.\mathsf{PrepareForRequest}(\mathcal{M}_U)$

It then makes an access request to the target service by sending it the request parameters, the authorizing macaroon M_{FS} and the set of discharge macaroons $\mathcal{M}_U^{sealed}$:

$U \longrightarrow TS : \mathsf{chunk} = 235,\ \mathsf{operation} = \mathsf{read};\ M_{FS};\ \mathcal{M}_U$

Handling the request at the target service: The target service handles access requests using a three stage process:

(1) *First-party caveat discharge.* Scan all the provided macaroons for first-party caveats and check the associated predicates in the context of the request. Let $\mathcal{A}$ be the set of validated predicates.
(2) *Macaroon verification.* Extract the root key k from the identifier of the macaroon M_{FS} and invoke $M_{FS}.\mathsf{Verify}(M_{FS}, k, \mathcal{A}, \mathcal{M}_{user})$. If Verify fails (i.e., an assertion fails), the request is forbidden.
(3) *Dispatch.* Provide services according to the now-authorized request parameters.

D. Security and Expressiveness

Macaroons use cryptographic means to provide the symbolic security properties of verifiable integrity, secrecy of intermediate keys, and the inability for adversaries to remove caveats. Even when Enc and Dec can be assumed to be secure, the cryptographic security of the chained MAC construction must still be considered (e.g., as previously done in [3, 25]). Fortunately, its security is straightforward when MAC is an HMAC function [43]. The signature of a macaroon whose starting identifier id conveys the root key K_0 can be written as $\mathrm{HMAC}(\mathrm{KDF}(K_0, [id, m_1, \ldots, m_{n-1}]), m_n)$, where KDF is a key derivation function, defined by $\mathrm{HMAC}(K_0, id)$ for the base case, or by using the successive HMAC outputs as the HMAC keys for a list of messages. Then, the required properties follow from HMAC unforgeability and the confidentiality of the HMAC key parameter due to HMACs being pseudo-random functions [43].

On top of the above sketch, Adriana López-Alt has created complete proofs of security for three different macaroon constructions [41]. Usefully, these proofs have explicitly identified the need to use an authenticated encryption function Enc for caveat identifiers in HMAC-based macaroons.

It is relatively straightforward to establish the flexibility of macaroon authorization, by comparing them to well-known, highly-expressive decentralized authorization credentials. It can be proven formally that macaroons are at least as expressive as SPKI/SDSI [16], and we have constructed such proofs (in particular, based on Li and Mitchell's formal semantics [39]), where SPKI/SDSI delegation is reduced to third-party caveats for conjunctive holder-of-key assertions. Intuitively, such proofs of macaroons' expressiveness are possible because, public-key certificates can be emulated using opaque key identifiers and additional network requests in online systems (see [18] and [37, Section 4.3]).

However, unfortunately, our proofs have offered few insights, except to show that symmetric cryptography and

enough extra messages can emulate public-key-based mechanisms; therefore, these proofs are omitted here. Meanwhile, the formalization of macaroons in authorization logic has been more clarifying, and is presented in the appendix.

HMAC-based macaroons do suffer in one important way compared to public-key-based credentials: only the target service that originally mints a macaroon can check its validity. Specifically, the verification of an HMAC-based macaroon requires knowledge of its root key—and since this key confers the ability to arbitrarily modify the macaroon and its caveats, it cannot be widely shared. On the other hand, third-party verification is possible when using asymmetric signing keys and public verification keys, as in SPKI/SDSI, or the public-key-based macaroons described later in Section V-H. However, the HMAC-based macaroons are orders-of-magnitude more efficient (as shown in Section VII), which, in many cases, is a worthwhile trade-off.

Macaroons' flexibility does not obviate the need to analyze the security of any concrete authorization protocols using them—just as with any other type of authorization credential. In particular, such protocols must account for how target services learn, during verification, the intermediate signatures of macaroons, as well as the root keys of all caveat discharge macaroons, and consider how—in an adversarial model—other parties may also learn those keys, e.g., due to flaws in target services. Similarly, the concrete means for establishing macaroon key identifiers must be analyzed, especially if shared symmetric keys are used to add third-party caveats. Fortunately, many frameworks exist for the analysis of authorization protocols that use public-key, shared-key, and MAC-based constructions [45, 10, 4]. Also, such analysis can be greatly simplified if fresh, pairwise symmetric keys establish trust between principals and if macaroons use caveats with fresh, high-entropy root keys that are unique and distinct from root keys in other macaroons. For this purpose, protocols may choose to derive root keys using the entropy of intermediate macaroon signatures.

V. Implementation Concerns

This section considers how distributed systems may implement macaroon-based authorization. The choice of topics is based on experiences building prototype macaroon mechanisms for Cloud clients, servers, and mobile devices, and using these to implement both new authorization protocols and to improve existing ones, like those based on OAuth2 [17].

Empirically, in distributed systems, macaroons are simple to implement, easy to integrate with existing code, and incur little overhead. In particular, it simplifies the task that much of macaroons' flexibility stems from the semantics of caveats being left up to application services. However, macaroon-based systems still need to support a common library of well-known caveats—such as for specifying expiration time, as well as predicates about client context such as IP addresses, ChannelID, etc.—leaving service- or application-specific caveat predicates to be used in specific protocols.

A. Encoding Key Identifiers

The key identifiers for both macaroons and caveats may be constructed using service-specific means, as discussed in Section IV. For example, root keys, lists of predicates, or other secrets may be stored in a database at the service and identified by an index key (such as a random nonce, like in Section III). Such identifiers have the nice property of being completely opaque and allowing revocation to be simply done by deleting a database row; however, the downside is that the target must maintain a database, either centralized or replicated across the servers operating the service.

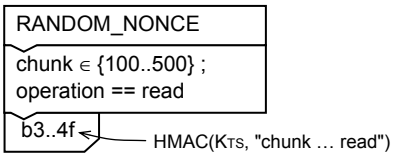


Fig. 9. Multiple caveats under one signature, as an optimization.

Alternatively, one may minimize state kept at the principal's server nodes by encrypting the keys and other secrets to be identified. For this, shared, symmetric encryption keys may be used, known to and trusted by each principal pair; alternatively, public-key encryption may be used, which can allow the same key to be used by several principals to create caveats for a single service that holds the private key.

B. Minting Caveat Root Keys

The root keys for caveats and caveat discharge macaroons are shared across principals; in particular, the root keys of all verified macaroons become known to the target service for a request. Thus, such root keys should not be reused—e.g., by embedding the same caveat identifier in multiple macaroons—unless this is strictly necessary (e.g., for performance reasons), and it is done in a known-to-be secure fashion (e.g., verified by a separate protocol analysis). In particular, while such reuse may be safe as long as all macaroons are for the same target service, it's most likely not safe when multiple target services are involved.

Therefore, to add a third-party caveat to a macaroon, an unpredictable caveat root key must be chosen—otherwise malicious parties might guess the root key and forge a discharge macaroon—even though some principals (e.g., Web browser frames, or embedded devices), may not have a good source of entropy for creating truly random keys.

To address this concern, caveat root keys may be derived from the signature of the embedding macaroon. Concretely, for adding a caveat to a macaroon with signature sig, its bearer may to generate a strong root key using $\mathsf{HMAC}(sig, n)$—where n is an arbitrary nonce—because HMAC is a pseudo-random function [43]. Such a key will be unique, due to the second pre-image resistance of HMAC, and will therefore never be repeated across different caveats.

C. Optimizations and Privacy

Macaroons permit many optimizations, in both their primitives and applications. For example, a reduced number of operations can be used for chained-MACs, if caveats are combined as in Figure 9.

Both as an optimization, and also to protect privacy, applications and services may choose to omit certain information from caveat discharge macaroons. In particular, for nested

third-party caveats, an inner, nested discharge macaroon need not be included in the discharge for the outer caveat. Thus, for example, if a third party caveat may restrict access to users within a certain age range, the caveat discharging service may add a caveat that requires a third-party authority to assert the bearer's date of birth; subsequently, the same discharging service may remove this birthdate-assertion caveat when it issues a discharge macaroon, to not reveal the bearer's age, thereby protecting their privacy.

With regards to privacy, it should be noted that attackers may try to add third-party caveats to probe for private information—e.g., by tricking users to reveal membership in some group. However, for this, attackers must be able to both trigger users to make requests with certain parameters, and observe the results. Under these conditions, similar attacks are equally possible in many existing authorization frameworks (e.g., SPKI/SDSI), and not specific to macaroons.

D. Local Discharging of Third-party Caveats

Third-party caveats can be used to implement decentralized authorization using holder-of-key proofs from authentication servers, as explained in earlier examples. However, third-party caveats may be discharged not just by networked servers, but by any isolated protection domain capable of holding secrets—such as, for example, a Web browser frame or extension, a mobile application, or a hardware token that is available locally to a user at a client.

The delegation of third-party caveat discharging to such local principals can improve both performance (by reducing network messages), as well as the precision of authorization policy enforcement. For example, a collection of unrelated Web services may use the same federated login system; if each service maintains separate sessions, logging out of one service may leave the user, unknowingly, still logged in and authorized in other services, and thereby put them at risk. Instead, using macaroons, each request, on all sessions, may derive a fresh, or very short-lived, caveat discharge from the third-party federated login service (e.g., via a hidden `iframe` that holds the caveat root key to derive discharge macaroons and is accessible from each service's Web page). Thus, a federated login service may hold the authoritative credentials, and services may be decoupled, yet the user need log out only once to immediately de-authorize all services.

E. Revocation and Versioning

As previously mentioned in Section II, macaroons excel at the common revocation strategies for credential-based authorization: using short-lived, fresh credentials, that can be subject to state-based checking and can be split between principals [48, Chapter 9.8]. Indeed, the local discharging of caveats of the previous subsection is just one example of split credentials, subject to state-based checks. For other workloads or desired policies, revocation may be done via a database "whitelist" (resp. "blacklist") of valid macaroons (resp. invalid), via embedding expiry times or epoch counters in macaroons, or via some combination of the above.

Commonly, state-based revocation may be via a third-party caveat to a service like a network time authority. To avoid the need for redundant discharging, such caveats might—when they encode a monotonic property—use the same caveat identifier (and, hence, the same root key), for multiple macaroons, without risk. Thus, for example, a Web browser might hold hundreds of macaroons for images at the same Cloud service, such that all macaroons share the same third-party revocation-check caveat, and a single macaroon can discharge that caveat for any image. Of course, this discharge macaroon can, and should, be very short lived.

Revocation may also be required when target services are updated. For example, the operation delete may be added to a target service that offers previously-immutable, read-only storage, and the existing macaroons for this target may not constrain the operations in requests. In this case, the service may implement a default-deny policy for the delete operation, and authorizing its use only in future macaroons. As a more principled approach, targets may *version* aspects of their service, and embed first-party predicates in macaroons that constrain the authorized versions.

F. Disjunctive and Negated Caveats

When adding third-party caveats to a macaroon, it may be desirable to add a set of them where only one needs to be discharged. An example use case would be to allow a client to choose which authentication provider to use. One way of achieving this would be to augment macaroons to also include a conjunction or disjunction operator at the beginning of every caveat list. The Verify method would then accordingly conjunct or disjunct the proof obligations specified by the caveats. This leads to a notion of caveats on a macaroon as a logical formula with both disjunction and conjunction.

One may ask if it makes sense to add negation as well, where a macaroon can only be invoked if a certain caveat is *not* discharged. However, the semantics of negation become tricky when the negated caveat discharge macaroon contains further nested third-party caveats. Thus, negation should only be supported at the level of predicate checking, e.g., along the lines of $\mathsf{op} \notin \{\mathsf{write}, \mathsf{delete}\}$, if needed.

G. Caveats on the Structure of Macaroons

In some applications, there may be a need for caveat predicates to make assertions about the structure and values in the set of authorizing macaroons. In this case, one macaroon's caveats could restrict the shape of the tree of macaroons that is induced during verification at the target service, or the values permitted in another macaroon in that tree.

Such a *structural caveat* may, for example, be used to enforce notions of well-formed transactions, by ensuring that a certain order of nested third-party caveat discharges are present [15]. Structural caveats may also name values in other caveats or discharge macaroons. For example, the BrowserID protocol behind Mozilla Persona [42] states that identity assertions (minted by a Web browser and verified by a trusted service) must successfully verify only if they state the domain name of the relying party they are meant for. Recasting BrowserID in terms of macaroons, this check can be enforced in the form of a structural caveat.

Structural caveats can also be used to make assertions about public-key signatures. For example, a service may mint

a macaroon that requires bearers to prove possession of a certain private key. For this, the service can add a first-party structural caveat whose predicate states "the set of macaroons must contain a digitally signed message m, verifiable by the public key P." The bearer would then be required to present such a signed message along with each request to the target service, if it is to be authorized by a macaroon with this caveat.

H. Public-key-based Macaroon Constructions

Many existing authorization systems are based on public-key certificates, including SPKI/SDSI [16], KeyNote trust management [11], etc. Several of these systems are not truly decentralized, but instead require a globally-accessible infrastructure, such as a revocation list, or a repository of certificates. Further, upon an authorization request, some of these systems carry out a certificate-chain discovery process that can be costly; e.g., the worst-case complexity for SPKI/SDSI being cubic in the size of the repository, as shown in [16].

Instead, authorization based on public-key certificates can benefit from following the patterns of macaroon constructions, as an instance of proof-carrying authorization [6]. Already, some public-key-based applications use a delegation pattern where authority is sent between principals by chaining an existing certificate to a newly-minted public/private key pair—handed off to a new principal—such that the recipient holds a private key whose public key is certified to wield a part of the sender's authority.

By having each principal, recursively, apply the above pattern to all authorizations, public-key certificates can effect the same type of delegation, attenuation, and key distribution as macaroons. While increasing the number of online messages, following such a macaroon-based pattern eliminates the need to maintain a global certificate repository, since each principal will individually manage the set of certificates relevant to it, and each certificate is unique.

Furthermore, by appropriately choosing the certificate validity times, and the predicates in their assertions, other benefits of macaroons may be replicated. In particular, such a certificate-based construction can provide the same privacy benefits as macaroons, in preventing linkability. Unfortunately, unlike the MAC in macaroons, the cryptographic primitives of public-key certificates can incur significant overhead—as shown in the measurements of Section VII—especially for the minting of the new public/private key pairs essential to the above construction. Therefore, to be more widely applicable, public-key-based macaroon constructions might also make use of HMAC-based signatures, e.g., as in [41].

VI. Applications and Use Patterns

Macaroons may be used not only to reimplement existing authorization mechanisms—to state them using a single, unified model, rather than using specific, bespoke protocols—but also to enable new types of authorization for Cloud applications. For example, as in Section V-D, macaroon credentials may be split, so that a separate party holds a necessary caveat discharge, thereby enabling a form of privilege amplification similar to that in [52]. In addition to building such new protocols, macaroons can also be used to enhance several existing authorization mechanisms.

A. Macaroons and End Users

Macaroons are useful for many existing end-user Cloud application scenarios. For example, a macaroon can be minted for sharing an image on Google Docs or Dropbox, with third-party caveats limiting its use to a particular set of users. In such scenarios, when making use of a macaroon, end users need not experience any significant change from the workflow they use today. The client interface of the Cloud service could allow the user to request an "unlisted URL", with options for selecting the allowed recipients, and provide means for that URL to be shared with those parties, e.g., over email.

Sometimes, a user's management of macaroon credentials may involve more than one application. For example, when sharing a Dropbox image with a limited group, a user might copy its URL from Dropbox to a service such as Google+ or Facebook, and use that service's client interface to pick the recipient group and attenuate the authority of the URL's macaroon credentials. Similarly, when visiting the link for this shared URL, its recipients might be redirected to a login screen, to discharge third-party caveats, before being able to access the image at the URL.

In general, since macaroons are easily serialized, the current mechanisms for sharing and storing access tokens over email, social postings, and chat may continue to work as before. However, instead of copying and pasting URLs, the user interfaces for macaroons may be more user-friendly, e.g., taking the form of explicit, custom "Share" buttons, similar to those used for minting OAuth2 tokens today.

B. Existing Use of Contextual Caveats

Current bearer token authorization relies heavily on implementation guidelines, which puts the security of the mechanisms in the hands of implementors. For example, Mozilla's BrowserID recommends constraining messages by checking the domain of the relying party [42], and many similar checks apply to OAuth2 [40]. By using macaroons, such guidelines can be made into explicit, mandatory contextual caveats.

As another example, ChannelID binds bearer tokens to a channel-specific public key [7]. ChannelID is well suited for use in first-party caveat predicates that aim to contextually confine the use of macaroons to specific (recipient) principals.

Such caveats are likely to use application-specific predicate languages, similar to how Bucket Policies in the Amazon S3 service define a rich policy language for contextual authorization of clients accessing S3 storage [5]. Of course, using macaroons, such policies could be set per issued token, instead of per storage bucket, efficiently and with very little effort.

C. Cookies as Macaroons

Cookies are widely used for authorization bearer tokens, such as HTTP session identifiers. With macaroon-based cookies, immediate benefits may be gained: macaroons would ensure integrity despite not maintaining state at the server, and, through the use of caveats, allow the cookies to be bound to specific, observable client context, such as IP address, user agent strings, referrers etc. An interesting benefit is also that clients could add caveats, for example to limit the usefulness of stolen cookies. Even simple attenuation might offer clients

significant protection, e.g., by limiting any outbound cookies to a single client IP address, and a validity period of only a few seconds. Cookies may also benefit from macaroon delegation, for example to implement light-weight, precise revocation via session liveness checking—like described in Section V-D, and used by the second end-to-end application in Section VII-B.

D. OAuth2 and Macaroons

The OAuth 2.0 Authorization Framework (or, simply, OAuth2) defines a popular protocol for allowing Web services to grant their users certain access to other services, referred to as relying parties. OAuth2 authorization relies on several kinds of tokens. Most important of these are *access tokens*, which are short-lived tokens that act as capabilities, e.g., to login sessions, and *refresh tokens*, which are long-lived tokens meant to be exchanged for access tokens at relying parties. Tokens are transferred between protocol participants through one of several *flows* defined by the OAuth2 RFC [17].

In the context of OAuth2, one use of macaroons might be to structure each concrete OAuth token as a macaroon, thereby allowing caveats to be embedded into the token. Such caveats may be used to either attenuate the authority granted by the token (similar to the examples in Section II), or to explicitly encode security guidelines prescribed for agents participating in OAuth flows (see [8, 17, 40, 50]). As suggested earlier in this section, the latter is particularly useful, since implementations frequently ignore these security guidelines [50].

As an example, consider the OAuth Implicit-Grant flow where an access token is handed to a client-side application by placing it in the fragment identifier of the application's URI. Since an attacker can easily replace a token in the fragment identifier with a token meant for another application, a "caveat" associated with this flow is that all received access tokens must be validated at the resource server by an application to ensure that they are indeed meant for it (see Section 10.16 in [17]). This validation step can be mandated by simply having a third-party caveat for the token-validation endpoint on all issued access tokens. More generally, most security recommendations on tightly confining tokens to particular clients and limiting their validity, to prevent token theft, session swapping and impersonation attacks—e.g., as listed in [50]—can be mandated by specifying them as caveats embedded into OAuth tokens that are re-cast as macaroon credentials.

OAuth2 also suffers from problems due to the nature in which relationships are formed. Relying parties must have a pre-existing relation to the Web service that they (or their user) want to use. For example, a website wishing to allow its users to login with an OAuth2 identity provider must first register itself as a client application with that provider. In practice, this limits the user's choice of identity providers, requires the Web service to maintain centralized state, and forces relying parties to keep and protect a number of client tokens. There are several means by which macaroons can address these issues, such as by safely storing client identities and redirection URLs directly in the macaroons issued by relying parties, thus avoiding the need for a central registry.

VII. Measurements and Evaluation

This section considers the performance of macaroon primitives, both through microbenchmark measurements of prototypes implemented in Python and JavaScript, and also through measurement of an end-to-end Web application that implements an image-sharing scenario using macaroons credentials for authorization, as in this paper's running example.

SHA-1	0.36 μs	RSA sign	458 μs
SHA-256	0.63 μs	RSA verify	27 μs
HMAC-SHA-1	1.7 μs	RSA encrypt	30 μs
HMAC-SHA-256	2.8 μs	RSA decrypt	461 μs
AES-128	0.35 μs	RSA keygen	54000 μs

TABLE I. Overhead of cryptographic primitives, in OpenSSL.

	Python	**JS/Chrome**	**JS/Node.js**
HMAC-SHA-256	13.7 μs	26.9 μs	11.4 μs
AES-128	5.4 μs	41.3 μs	18.7 μs
Mint macaroon	16.0 μs	27.2 μs	19.4 μs
Add caveat	54.9 μs	294.8 μs	56.3 μs
Verify	96.3 μs	358.5 μs	68.9 μs
Marshal as JSON	15.2 μs	3.6 μs	3.0 μs
Parse from JSON	35.0 μs	3.3 μs	5.0 μs

TABLE II. Overhead of macaroon primitives, in practice.

A. Cost of Authorization Primitives

To compare the performance overhead of different primitives for authorization credentials, Table I lists the cost of some cryptographic operations, in microseconds, using microbenchmarks of the implementations in OpenSSL 1.0.1c. The measurements were run on a 2.33 GHz Intel Core2 server.

In Table I, hash algorithms are benchmarked using short messages, such as would be common in the use of macaroons, and, in this case, HMAC setup overhead dominates. Even so, computing an HMAC for short messages (up to a few hundred bytes) imposes a less than $2\times$ slowdown compared to the processing of messages without integrity codes. To reduce MAC overhead further, an implementation might choose to define $\mathsf{MAC}(k, msg) = \mathsf{SHA\text{-}256}(k \,\|\, msg)$ truncated to 128 bits. Since the key to MAC is always a fixed length, there is no confusion over where the key ends and the message begins, and truncating the hash prevents extension attacks.

Public key operations are more expensive. Using 1024-bit RSA, operations with the public key (encrypt/verify) take 27–30 μs, and private-key operations (decrypt/sign) take 460 μs. Minting fresh keys is particularly expensive: a single CPU core can mint at most 18 key pairs per second, in our measurements. Thus the public-key alternative to macaroons discussed in Section V-H may have prohibitive overhead, since it is fundamentally based on the minting of new keys. Relying on symmetric cryptography, or a combination of the two, greatly improves the performance—by as much as two to four orders of magnitude over a public-key construction. This makes macaroons fast enough to be applicable nearly anywhere, even when implemented in JavaScript [49].

Table II shows the performance overhead of the primitive operations for macaroon authorization, based on measurements of two prototype implementations of macaroon credentials in Python 2.7.2 and JavaScript. For the JavaScript implementation, the measurements are from a Web browser (Google Chrome v27.0) using the Stanford JS Crypto library [49,

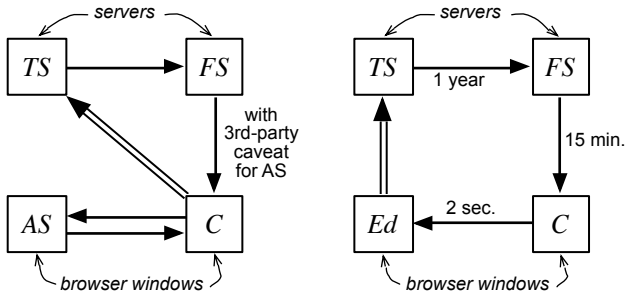


Fig. 10. *Left:* To access a service *TS*, a third-party caveat is discharged by *AS* to authenticate a user of the *FS* service at client *C*, just as in Figure 6. *Right:* Using macaroons, authority is delegated for different lengths of time between principals, finally passing to an image editor Web browser frame *Ed*.

Aug. 2013], as well as from Node.js (v0.10.3) and its standard crypto module wrapper of OpenSSL (v1.0.1c). As can be seen in the table, the minting of macaroons requires little more than an HMAC signature, which is fast (although slower than when performed natively, as in Table I). In general, the measurements show data structure manipulation to be the slowest aspect of the operations in Table II, with serialization from JSON being especially costly in Python.

B. End-to-end Application Performance

To model real-world performance of macaroons, we constructed an end-to-end Web application based on this paper's running example. In this application, a file-sharing website *FS* handles user accounts and the data for images to be shared, while the actual image files are in a Cloud storage service *TS*. As in Section II, the storage service *TS* issues a long-lived macaroon to *FS*, which grants *FS* full access to certain files. When a user connects to *FS* from its Web browser client *C*, their browser receives URLs to the user's images on *TS* along with macaroon credentials. Each of these macaroons is an attenuated, derived version of *TS*'s macaroon, of shorter duration, tied to a specific filename, and confined to the user's client *C*. Then, authorized by these macaroons, the user's Web browser can fetch the image files directly from *TS*.

Measuring this Web application on a local network, an average of 470 μs was added to the end-to-end request processing latency—compared to file serving without access checks—with authorization in Node.js verifying a macaroon with four caveats on the expiration time and request arguments

As another end-to-end scenario, we enhanced our Web application to perform user authentication across Web browser frames, as shown on the left in Figure 10. For this, *FS* adds a third-party caveat to the macaroon it provides to the client *C*, where those caveats need discharging by a separate authentication service *AS*. The *AS* service has delegated the discharging of its caveats to an isolated Web browser frame for the user's login session, so that authentication caveats can be discharged without network access. The *TS* service verifies that all caveats are discharged, but does not get to know the identity of *AS*, or the user. In our experiments, the measured total overhead of discharging these third-party caveats was on average only 3 milliseconds for each image access.

As a final end-to-end scenario, we had our Web application share images with a third-party, Web-based image editor *Ed*, as shown on the right in Figure 10. For an image dragged-and-dropped by the user on client *C*, the Web browser window for *Ed* gets a URL and derived macaroon for direct access to an image on *TS*. This macaroon is valid only for a few seconds and could be further confined (e.g., to *C*'s IP address). Measured from *Ed*'s Web browser window, it takes only an average of 5.0 milliseconds to request and obtain such a URL and an authorizing macaroon. Bringing the total access time to 12.5 milliseconds, it took on average 7.5 milliseconds to download a 40 kB image from a locally-hosted Web server, with Node.js verifying the macaroon and checking its caveats.

In comparison, it is enlightening to consider OAuth2-based constructions for the above scenarios. To be as fine grained as a macaroon-based approach, while strictly following the flows specified by OAuth2, separate access tokens would have to be issued for each image access. This might well be impractical, e.g., since the latency of a sequence of HTTP redirects would then be imposed upon each access to *TS*. Instead, in practice, it is common to sacrifice fidelity to the OAuth2 flows, e.g., by granting longer-lived, more broadly-scoped access tokens. In this case, this might mean that the *Ed* image editor would receive a hard-to-revoke credential for read access to all of the user's photos *FS*, which might last for an hour, or more.

The light-weight nature of macaroons allows them to be used liberally. On the client, fast delegation between protection domains, such as Web browser frames, means that separate macaroons, with precise contextual confinement, can be issued for each access—thus adhering tightly to the principle of least privilege. Notably, in the above scenario, no pre-existing relationship is required between *Ed* and *FS*, except an agreement on the use of macaroons, and on a common protocol for accessing image files.

VIII. Discussion

As flexible authorization credentials for distributed systems, macaroons build upon a wealth of prior results, developed over nearly half a century. Thus, for macaroons, the closely related work covers entire fields of study—including access control [32], authorization logic [37], trust management [11], proof-carrying authorization [6], extensibility of mechanisms [38] and more—which cannot be fully represented here, although the most closely related lines of work warrant a discussion, as below.

Traditional capability systems do not directly support confinement or revocation of access. Over the years, using all of the three patterns illustrated in Figure 3, capability systems have been modified to support controlled delegation [24]. As in the first pattern, in [46] a proxy-based mechanism is described for revoking and delegating the access provided by a capability. As in the second pattern, in [26] an identity-based capability system is described where delegation involves services re-minting new capabilities after checking applicable security policies. As in the third pattern, in [44], a macaroon-like mechanism is proposed for chaining restrictions upon an authorization credential using symmetric-key operations, thereby confining its context of use—although this work provided no provision for third-party caveats, like those in macaroons.

Macaroons are credentials, not capabilities, because the possession of a macaroon is a necessary, but not a sufficient

condition for granting authority. As bearer credentials, the contextual caveats in macaroons help them avoid the confinement problem, which arises for unmodified capabilities where "the right to exercise access carries with it the right to grant access" [26]. Furthermore, upon use of a macaroon, target services need not only check that all caveats are discharged, but may also perform additional access control, for example, to check continued ownership of the accessed resource.

Over the last two decades, several public-key certificate mechanisms [9, 11, 12, 14, 35, 37, 53] have been developed to provide highly flexible, decentralized authorization for distributed systems, while avoiding the problems of pure bearer tokens. The general idea is that principals possess unique public/private key pairs, and delegate access to other principals by issuing certificates signed using their private key. SPKI certificates [14] specify the delegate's public key, a validity period, and an S-expression describing a set of rights; whereas Active certificates [12] specify mobile code that is executed by the target service at the time of a request and mediates all access between the requestor and the service.

While such public-key delegation certificates are highly expressive and truly decentralized (since they can be minted locally and verified by anyone), they have been harder to adopt in the Cloud than schemes founded on bearer tokens. In particular, adoption has been hampered by the need for certificate-revocation infrastructure, the use of cryptographic primitives with significant per-request overhead, and the use of long-lived, linkable public identities for principals.

Macaroons are only verifiable by their target service, but for that service they offer flexible, efficient *credentials-based authorization* [48]. Notably, macaroons permit bearers of credentials to delegate authority, in a decentralized, general manner, using an efficient chained-HMAC construction. The authority of these derived credentials can be subject to attenuation and contextual confinement, based on first-party and third-party caveats that restrict both the permitted functionality (by limiting the set of allowed operations and objects) as well as the authorized principals (by confining the environment and context in which a macaroon may be successfully wielded).

Macaroons are an improvement upon cookies. They are especially well-suited to the Cloud, where the notion of a central authority is often lacking, and where the notion of principals is highly dynamic and may involve unnamed, local entities such as `iframe`s in Web browsers. As bearer credentials that use HMAC signatures for both integrity and for key distribution, macaroons are highly efficient, widely applicable, and compatible with existing Cloud software. Even so, compared to previous mechanisms, macaroons can support equally expressive authorization policies, even more precisely, with fresh, short-lived credentials for each access request.

ACKNOWLEDGMENT

Macaroons originated in the Belay research project at Google [27], and, thus, benefitted from the work of Arjun Guha, Iain McGinniss, Ben Laurie, and Mark Miller. Our NDSS reviewers and shepherd, as well as Martín Abadi, Adriana López-Alt, Domagoj Babic, Mike Burrows, Michael R. Clarkson, Sergio Maffeis, Robbert van Renesse, and Ben Sittler, all gave feedback that improved this paper's content.

REFERENCES

[1] M. Abadi, "Variations in access control logic," in *Intl. Conf. on Deontic Logic in Computer Science*, 2008.

[2] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin, "A calculus for access control in distributed systems," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 4, 1993.

[3] M. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. Thekkath, "Block-level security for network-attached disks," in *USENIX Conf. on File and Storage Technologies, (FAST)*, 2003.

[4] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of Web security," in *IEEE Computer Security Foundations (CSF)*, 2010.

[5] Amazon Inc., "Example cases for Amazon S3 Bucket Policies," 2013, http://docs.aws.amazon.com/AmazonS3/latest/dev/AccessPolicyLanguage_UseCases_s3_a.html.

[6] A. W. Appel and E. W. Felten, "Proof-carrying authentication," in *ACM Computer and Communications Security (CCS))*. ACM, 1999.

[7] D. Balfanz and R. Hamilton, "Transport layer security (TLS) Channel IDs," IETF Draft, 2013, http://tools.ietf.org/html/draft-balfanz-tls-channelid.

[8] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," in *IEEE Computer Security Foundations (CSF)*, 2012.

[9] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, 2010.

[10] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, 2009.

[11] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *IEEE Symp. on Security & Privacy*, 1996.

[12] N. Borisov and E. A. Brewer, "Active certificates: A framework for delegation," in *Network and Distributed Systems Security Symp. (NDSS)*, 2002.

[13] J. Bradley, P. Hunt, T. Nadalin, and H. Tschofenig, "The OAuth 2.0 authorization framework: Holder-of-the-key token usage," IETF Draft, http://tools.ietf.org/html/draft-tschofenig-oauth-hotk.

[14] E. Carl Ellison, "SPKI certificate theory," IETF RFC 2693 (Experimental), 1999, http://www.ietf.org/rfc/rfc2693.txt.

[15] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies." in *IEEE Symp. on Security & Privacy*, 1987.

[16] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest, "Certificate chain discovery in SPKI/SDSI," *Journal of Computer Security*, vol. 9, no. 4, 2002.

[17] E. D. Hardt, "The OAuth 2.0 Authorization Framework," IETF RFC 6749 (Informational), 2012, http://tools.ietf.org/html/rfc6749.

[18] D. Davis and R. R. Swick, "Network security via private-key certificates," *Operating Systems Review*, vol. 24, no. 4, 1990.

[19] J. DeTreville, "Binder, a logic-based security language," in *IEEE Symp. on Security & Privacy*, 2002.

[20] T. Dierks and E. Rescola, "The Transport Layer Security (TLS) Protocol," IETF RFC 5246 (Standards track), 2008, http://www.ietf.org/rfc/rfc5246.txt.

[21] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach, "Origin-bound certificates: A fresh approach to strong client authentication for the web," in *Proc. USENIX Security*, 2012.

[22] I. T. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A security architecture for computational grids," in *ACM Computer and Communications Security (CCS))*, 1998.

[23] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, "Sharp: An architecture for secure resource peering," in *Symp. on Operating Systems Principles (SOSP)*, 2003.

[24] M. Gasser and E. McDermott, "An architecture for practical delegation in a distributed system," in *IEEE Symp. on Security & Privacy*, 1990.

[25] H. Gobioff, "Security for a high performance commodity storage subsystem," Ph.D. dissertation, Carnegie Mellon University, 1999.

[26] L. Gong, "A secure identity-based capability system," in *IEEE Symp. on Security & Privacy*, 1989.

[27] Google Inc., "Belay research project," 2012, https://code.google.com/p/google-belay/.

[28] ——, "YouTube video privacy settings," 2013, http://support.google.com/youtube/bin/answer.py?hl=en&answer=157177.

[29] E. Grosse and M. Upadhyay, "Authentication at scale," *IEEE Security & Privacy Magazine*, vol. 11, no. 1, 2013.

[30] P. Hallam-Baker, C. Kaler, R. Monzillo, and A. Nadalin, "Web Services Security: SAML Token Profile," OASIS, 2004, http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf.

[31] E. Hammer-Lahav, A. Barth, and B. Adida, "HTTP authentication: MAC access authentication," IETF Draft, http://tools.ietf.org/html/draft-hammer-oauth-v2-mac-token.

[32] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Comm. of the ACM (CACM)*, vol. 19, no. 8, 1976.

[33] J. Howell and D. Kotz, "An access-control calculus for spanning administrative domains," Dartmouth College, Tech. Rep., 1999.

[34] ——, "A formal semantics for SPKI," in *European Symp. on Research in Computer Security*, 2000.

[35] T. Jim, "SD3: A trust management system with certified evaluation," in *IEEE Symp. on Security & Privacy*, 2001.

[36] B. W. Lampson, "Protection," *Operating Systems Review*, vol. 8, no. 1, 1974.

[37] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. on Computer Systems*, vol. 10, no. 4, 1992.

[38] C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek, "Alpaca: Extensible authorization for distributed services," in *ACM Computer and Communications Security (CCS))*, 2007.

[39] N. Li and C. Mitchell, "Understanding SPKI/SDSI using first-order logic," *Intl. Journal of Inf. Security*, vol. 5, no. 1, 2006.

[40] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," IETF RFC 6819 (Informational), 2013, http://www.ietf.org/rfc/rfc6819.txt.

[41] A. López-Alt, "Cryptographic security of macaroon authorization credentials," New York University, Tech. Rep. TR2013-962, 2013, http://cs.nyu.edu/web/Research/TechReports/TR2013-962/TR2013-962.pdf.

[42] Mozilla, "BrowserID specification," https://github.com/mozilla/id-specs/blob/prod/browserid/index.md.

[43] National Institute of Standards and Technology, "FIPS PUB 198-1: The keyed-hash message authentication code (HMAC)," 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

[44] B. C. Neuman, "Proxy-based authorization and accounting for distributed systems," in *Conf. on Distributed Computing Systems*, 1993.

[45] A. Project, "Automated validation of Internet security protocols and applications," http://avispa-project.org/.

[46] D. D. Redell, "Naming and protection in extendable operating systems," Ph.D. dissertation, Massachusetts Institute of Technology, 1974.

[47] E. S. Tuecke, "Internet X.509 public key infrastructure (PKI) proxy certificate profile," IETF RFC 3820 (Standards track), 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3820.txt

[48] F. B. Schneider, "*Untitled Textbook on Cybersecurity.* Chapter 9: Credentials-based authorization," 2013, http://www.cs.cornell.edu/fbs/publications/chptr.CredsBased.pdf.

[49] E. Stark, M. Hamburg, and D. Boneh, "Stanford JavaScript crypto library," 2013, http://crypto.stanford.edu/sjcl/.

[50] S.-T. Sun and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems," in *ACM Computer and Communications Security (CCS))*, 2012.

[51] A. Tanenbaum, M. Kaashoek, R. van Renesse, and H. Bal, "The amoeba distributed operating system–a status report," *Computer communications*, vol. 14, no. 6, 1991.

[52] R. van Renesse, H. D. Johansen, N. Naigaonkar, and D. Johansen, "Secure abstraction with code capabilities," in *Intl. Conf. on Parallel, Distributed, and Network-Based Processing*, 2013.

[53] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the Taos operating system," *ACM Trans. on Computer Systems*, vol. 12, no. 1, 1994.

[54] M. Zalewski, *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2011.

[55] ZDNet: Between the Lines, "Dropbox adds Facebook sharing," 2012, http://www.zdnet.com/facebook-gets-involved-with-cloud-storage-via-dropbox-integration-7000004861/.

APPENDIX

This appendix presents a formalization of macaroons using a variant of Abadi's authorization logic from [1], originally in [2, 37]. A macaroon is seen as an assertion made by a target service, saying that the holder of certain keys can *speak for* the target service regarding all access requests, as long as all relevant predicates for the macaroon's embedded caveats can be seen to be valid.

The effects of caveat predicates cannot be directly captured in standard authorization logic and, therefore, the logic used in this appendix is extended with two new types of principals—*predicate principals* and *request principals*—and a special axiom that characterizes their behavior. Using these new principals and regular principal compounding, this extended logic can, in a simple manner, express the speaks-for restrictions imposed by macaroons' caveats and their predicates. (In particular, this logic is considerably simpler than the authorization logic extended with new *speaks-for* primitives, previously used to express similarly-predicated authority in [34].)

An Extended Authorization Logic for Macaroons: Here, an extended authorization logic is defined for the purpose of formalizing the semantics of macaroons and the assertions made by the principals relevant to a target-service request that is authorized using macaroons.

First, the set $PropCavs$ is assumed to contain all atomic propositions relevant to target service requests, including request parameters, such as chunk and operation, and contextual attributes, such as time. Examples of such propositions might be "chunk is in [100, 500]", "action is read", etc. These propositions serve as the building blocks for the first-party caveat predicates that restrict macaroons' authority.

The syntax, axioms and deduction rules for the authorization logic are defined in Figure 11. Principals $A, B, \ldots$ can be atomic principals from $P \in Prncs$, keys $k \in Keys$, compound principals of the form $A \wedge B$, or special predicate or request principals (discussed later). Here, p ranges over the set of propositions in $PropCavs$, and X ranges over the set of propositional variables. The $says$ modality for principals is standard from [37], with $A\ says\ \phi$ meaning that principal A asserts ϕ. While $\supset$ is used for logical implication, the statement $A \Rightarrow B$ expands to A *speaks-for* B, for principals A and B, meaning that B also makes all assertions that A makes. A compound principal $A \wedge B$ makes an assertion if, and only if, it is also made by both A and B. To establish that a request is authorized, a proposition valid is included, with $A\ says$ valid meaning that principal A considers the request to be valid.

This logic is a normal modal logic that includes the [DISTRIBUTION] axiom, the [NECESSISTION] rule, the [MODUS-PONENS] rule, and all standard axioms of propositional constructive logic, as used in [1]. Also included, as in [37], are the standard [SPEAKS-FOR] axiom, the [HANDOFF] axiom, and the expected axioms for characteristics of principal conjunction. From these, the monotonicity of $\wedge$ over $\Rightarrow$ and the transitivity of $\Rightarrow$, follow:

$$\vdash (A \Rightarrow B) \quad \supset \quad (A \wedge C) \Rightarrow (B \wedge C),$$
$$\vdash (A \Rightarrow B) \wedge (B \Rightarrow C) \quad \supset \quad (A \Rightarrow C).$$

The logic also includes a special axiom [PPRIN] relating request and predicate principals, which is discussed next.

Syntax:

$$\begin{array}{rrl} \text{Compound principals } A, B & ::= & P \mid k \mid A \wedge B \mid \widehat{\phi} \\ \text{Formulas } \psi, \phi & ::= & \mathsf{valid} \mid \mathsf{true} \\ & & \mid p \mid X \mid \forall X.\phi \\ & & \mid \psi \wedge \phi \mid \psi \vee \phi \mid \psi \supset \phi \\ & & \mid A \ says\ \phi \mid A \Rightarrow B \end{array}$$

Axioms:

All the axioms of propositional constructive logic [PROP]

$\vdash (A\ says\ (\psi \supset \phi)) \supset (A\ says\ \psi \supset A\ says\ \phi)$ [DISTRIBUTION]

$\vdash (A\ says\ B \Rightarrow A) \supset B \Rightarrow A$ [HANDOFF]

$\vdash A \wedge A \equiv A$ [CONJ1]

$\vdash A \wedge B \equiv B \wedge A$ [CONJ2]

$\vdash (A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ [CONJ3]

$\vdash (A \wedge B\ says\ \phi) \equiv (A\ says\ \phi) \wedge (B\ says\ \phi)$ [CONJ4]

$\vdash A \Rightarrow B \equiv (A \wedge B = A)$ [SPEAKS-FOR]

$\vdash (TR\ says\ \phi) \supset (\widehat{\phi}\ says\ \mathsf{valid})$ [PPRIN]

Rules:

$$\frac{\vdash \psi \quad \vdash \psi \supset \phi}{\vdash \phi}$$ [MODUS-PONENS]

$$\frac{\vdash \phi}{\vdash A\ says\ \phi}$$ [NECESSITATION]

Fig. 11. The syntax, axioms and rules for an authorization logic suitable for macaroons. Here, TR and $\widehat{\phi}$ are request and predicate principals, respectively, $P \in Prncs$, $k \in Keys$, and X is a variable in a proposition $p \in PropCavs$.

Request principals and predicate principals: A *request principal* TR is a special principal that makes the strongest possible assertion—using propositions from $PropCavs$—that a target service can see to be true for the context of a request. Such principals are completely controlled by the target service. For example,

$$TR\ says \quad \begin{array}{l} \text{“chunk is 250”} \ \wedge \ \text{“operation is read”} \\ \wedge \ \text{“IP is 172.12.34.4”} \end{array}$$

indicates that the target service is seeing a request to read chunk 250 being made from the IP address 172.12.3.44.

A *predicate principal* $\widehat{\phi}$ is a special principal introduced to model the effect of a first-party caveat with predicate ϕ. Informally, the principal $\widehat{\phi}$ says that a request is valid, and the caveat is satisfied, if the request principal TR asserts ϕ. This characteristic is formalized by the axiom [PPRIN].

By combining the axiom [PPRIN], the [DISTRIBUTION] axiom and the [NECESSITATION] rule, the following rule can be derived:

$$\frac{\vdash \psi \supset \phi \quad \vdash TR\ says\ \psi}{\vdash \widehat{\phi}\ says\ \mathsf{valid}}$$ [FCAVDISCHARGE].

This rule means that if TR asserts a predicate for a request, and this predicate logically implies a weaker predicate in a first-party caveat, then that caveat is satisfied for this request.

Macaroon formulas: In this logic, macaroons are defined by formulas that describe restricted speaks-for delegations from the target service to certain caveat principals (corresponding to embedded first-party caveats), and certain keys (corresponding to the root keys of embedded third-party caveats). A request made using macaroon credentials is authorized if it can be proven—from the macaroon's formulas, and the request principal TR's assertion about the request context—that the root key of the macaroon says valid, which implies that the macaroon signatures verify using this root key, and that all embedded caveat predicates are satisfied. Before formally defining macaroon formulas, it's worth giving some examples.

Consider a macaroon $M_1 := \mathsf{macaroon}_{@L}\langle kId, [\,], k_1\rangle$ that is minted from root key k_0 without any caveats. This macaroon M_1 represents a complete delegation from the key k_0 to the empty caveat principal $\widehat{\mathsf{true}}$, which considers all requests valid. Thus, M_1 is modeled by the formula $k_0\ says\ \widehat{\mathsf{true}} \Rightarrow k_0$.

A macaroon $M_2 := \mathsf{macaroon}_{@L}\langle kId, [\mathsf{cav}_{@\top}\langle \phi, 0\rangle], k_2\rangle$ can be obtained by extending M_1 with a first-party caveat whose predicate is ϕ, to represent a delegation from the root key k_0 to the caveat principal $\widehat{\phi}$. This macaroon M_2 is modeled by the formula $k_0\ says\ \widehat{\phi} \Rightarrow k_0$.

Finally, the macaroon M_2 can be extended with a third-party caveat whose root key is cK, to obtain the macaroon

$$M_3 := \mathsf{macaroon}_{@L}\langle kId, [\mathsf{cav}_{@\top}\langle \phi, 0\rangle, \mathsf{cav}_{@l}\langle cId, vId\rangle], k_3\rangle.$$

This macaroon M_3 represents a delegation from the root key k_0 to the conjunction of the cK and $\widehat{\phi}$ principals—i.e., that k_0 says that a request is valid only if both cK and $\widehat{\phi}$ say so. Thus, M_3 is modeled by the formula $k_0\ says\ cK \wedge \widehat{\phi} \Rightarrow k_0$. The formula for an arbitrary macaroon M can now be defined.

Definition 1 (Macaroon formulas): A macaroon M whose root key is k_0, embedding first-party caveats whose predicates are $\phi_1, \ldots, \phi_m$, and embedding third-party caveats whose root keys are $cK_1, \ldots, cK_n$, is modeled using the formula

$$\alpha(M) := k_0\ says\ (\widehat{\phi_1} \wedge \ldots \wedge \widehat{\phi_m} \wedge cK_1 \wedge \ldots \wedge cK_n) \Rightarrow k_0.$$

For a set $\mathcal{M}$, the formulas are $\alpha(\mathcal{M}) := \{\alpha(M) \mid M \in \mathcal{M}\}$.

Macaroon verification: To verify that a request is authorized by a macaroon M, the target service must verify—using a root key k_0 for M, known only to the target service—the set $\mathcal{M}$ of macaroons presented with the request, including M and all third-party discharges, and establish that all their embedded first-party caveats are satisfied. The verification of a first-party caveat with a predicate ϕ is modeled by having the request principal TR assert the strongest possible formula ψ_{req} about the request context, and, if $\psi_{req} \supset \phi$, apply the derived rule [FCAVDISCHARGE] to show that the principal $\widehat{\phi}$ considers the request to be valid. In general, a request accompanied by such a macaroon set $\mathcal{M}$ is authorized if the formulas $\alpha(\mathcal{M})$ together with the formula $TR\ says\ \psi_{req}$ imply that $k_0\ says$ valid for the root key k_0. Recursively, this requires that $\mathcal{M}$ contain discharge macaroons for any third-party caveats involved, and that $TR\ says\ \psi_{req}$ allows $cK\ says$ valid to be established for the root key cK of each of those discharge macaroons.

Definition 2 (Macaroon verification): A set of macaroons $\mathcal{M}$, whose distinguished authorizing macaroon M has the root key k_0, authorizes a request whose target-service context is described by the propositional assertion ψ_{req}, if, and only if

$$\vdash (TR\ says\ \psi_{req} \ \wedge \bigwedge_{M \in \mathcal{M}} \alpha(M)) \ \supset \ (k_0\ says\ \mathsf{valid}).$$

In addition to being a basis for the formal analysis of macaroon-based protocols, the above definition makes it clear how macaroons only grant authority for a target-service request, as long as all caveats have been discharged in its context.