Cryptographic Security of Macaroon Authorization Credentials

Adriana López-Alt
New York University

December 6, 2013

Abstract

Macaroons, recently introduced by Birgisson et al. [BPUE+14], are authorization credentials that provide support for controlled sharing in decentralized systems. Macaroons are similar to cookies in that they are bearer credentials, but unlike cookies, macaroons include *caveats* that attenuate and contextually confine when, where, by who, and for what purpose authorization should be granted.

In this work, we formally study the cryptographic security of macaroons. We define *macaroon schemes*, introduce corresponding security definitions and provide several constructions. In particular, the MAC-based and certificate-based constructions outlined in [BPUE+14], can be seen as instantiations of our definitions. We also present a new construction that is privately-verifiable (similar to the MAC-based construction) but where the verifying party does not learn the intermediate keys of the macaroon, a problem already observed in [BPUE+14].

We also formalize the notion of a protocol for "discharging" third-party caveats and present a security definition for such a protocol. The encryption-based protocol outlined by Birgisson et al. [BPUE+14] can be seen as an instantiation of our definition, and we also present a new signature-based construction.

Finally, we formally prove the security of all constructions in the given security models.

1 Introduction

Macaroons, as introduced by Birgisson et al. [BPUE+14], are bearer tokens that act as authorization credentials. In its most basic form, a macaroon identifies a list of restrictions or *caveats* that specify when, where, by who, and for what purpose, access to a resource can be granted. For example, a caveat might specify that the macaroon is valid (and thus access should be granted) as long as the operation being performed is 'read'. Another caveat might specify that the macaroon is valid as long as the object being accessed is '`myimage.jpg`'. These examples are what Birgisson et al. call *first-party caveats*. The conditions they impose can be verified by the target service that manages the resource when the access request is made and the macaroon is presented. More interesting are *third-party caveats*, caveats that require a third-party to assert that a certain condition holds true. For example, a third-party caveat might require proof of membership in a group G at a third-party service A. A recipient of a macaroon holding a third-party caveat must first obtain such a proof or *discharge* from the corresponding third-party, and pass along the discharge with the macaroon when it accesses the intended resource at the target service.

Of course, care must be taken to ensure that caveats are 'bound together' so that a principal holding a macaroon cannot remove or modify caveats at his or her convenience. Therefore, macaroons include a cryptographic signature that guarantees the integrity of the list of caveats and the caveats themselves.

Macaroons also allow *delegation* and *attenuation* by permitting additional caveats to be added to a macaroon, further restricting the validity of the token. More importantly, this can be done using only the macaroon itself (and perhaps an additional *extension key* created at the same time as the macaroon), and does not require communication with other principals or the target service. It is this flexibility that makes macaroons a leading candidate for authorization in decentralized distributed systems, and in 'cloud' services in particular.

Our Results. In this work, we formally study the cryptographic security of macaroons. We divide this task into three parts. First, we study the security of *macaroon schemes*, a new primitive that captures the requirements of integrity and extension (for delegation or attenuation) of a macaroon. Second, we study *discharge protocols*, a new primitive that specifies the syntax of third-party caveats and how they can be used to obtain discharges from the corresponding third-party. Finally, we study *macaroon-tree protocols*, a new primitive that captures the full functionality of macaroons: issuing, extending with a first-party caveat, extending with a third-party caveat, discharging, and verifying.

We follow the formalization of Birgisson et al. and model a macaroon as follows. A macaroon includes a list of caveats, where each caveat is a bit-string $m_i \in \{0,1\}^*$. Additionally, a macaroon includes a key identifier id that opaquely encodes a root key k, and a signature σ that guarantees the integrity of the macaroon and can be verified using the key k.

Macaroon Schemes. We define a new primitive called a *macaroon scheme* that defines how macaroons can be issued, extended (for delegation or attenuation), finalized, and verified. A key generation algorithm generates root keys k, with which a macaroon can be issued. When a macaroon is issued, an *extension key* is also given. Using this extension key, the new macaroon can be extended, essentially adding more caveats to it, and a new extension key is issued so as to allow further extensions. A verification algorithm is also defined such that given the corresponding root key k, the validity of a macaroon μ can be verified. Finally, an algorithm $\mathsf{Finalize}$ is defined, that modifies a given macaroon so that it can be verified by the verification algorithm. This last algorithm can be seen as optional – indeed, only one of our constructions requires it.

We formalize a security definition for macaroon schemes that captures the desired guarantee that a principal cannot forge a macaroon. In other words, a principal cannot create a valid macaroon μ^* with respect to a root key k unless it obtained the macaroon via "valid means", i.e. μ^* was given to him, and μ^* was issued using k or is an extension of such a macaroon. Notice that this provides security against removing or modifying caveats from a macaroon obtained legally.

We provide three constructions of macaroon schemes. Two of these, a PRF-based construction and a certificate-based construction were already described in [BPUE+14]. We formalize these constructions and the properties required to prove their security. The third construction is new, and offers efficiency vs. security trade-offs when compared to the other two.

Finally, we make a distinction between a privately-verifiable macaroon scheme (as informally defined above), where a macaroon is issued and verified using the same root key k, and a publicly-verifiable macaroon scheme, where a macaroon is issued with a secret root key sk and verified with a corresponding (public) verification key vk. See Section 3.1 for a formal discussion of privately-verifiable macaroon schemes, and Section 3.2 for a formal discussion of publicly-verifiable macaroon schemes.

Discharge Protocols. We formally define *discharge protocols*, a new primitive that specifies the syntax of third-party caveats and their corresponding discharges, which are also macaroons. An initial setup creates a discharging key sk_D and an issuing key sk_I. We define a token-issuing algorithm, which takes as input the issuing key sk_I and outputs two tokens - a state token α_S and a discharge token α_D. The discharge token α_D, together with the discharging key sk_D, is used to obtain a discharge macaroon. The state token α_S is used to verify the validity of the discharge obtained.

A third-party caveat includes both the state token α_S, as well as the discharge token α_D. The discharge token α_D can be used by the holder of the macaroon to obtain the discharge from the third-party who holds the discharging key sk_D, and the state token α_S can be used by the target service to verify the validity of the discharge presented.

Note that in the use-case of discharge protocols, a principal will present a macaroon and the corresponding discharges to a target service. The target service will then use the state tokens that are part of the third-party caveats in the macaroon and use these to verify the discharges that are presented along with the macaroon. Therefore, our definition of security of a discharge protocol ensures that without knowing the issuing or discharge keys, a principal cannot forge a state token α^*_S and a discharge macaroon μ^* such that μ^* is a valid discharge macaroon for α^*_S, unless they were legally obtained.

We present two constructions of discharge protocols, one that uses encryption and a privately-verifiable macaroon scheme, and another that uses a message authentication code (MAC) and a publicly-verifiable

macaroon scheme. Though the first was already outlined in [BPUE+14], we formalize the properties required to prove its security. In particular, we show that the encryption used must be non-malleable.

Macaroon-Tree Protocols. Finally, we show how to combine macaroon schemes and discharge protocols to obtain *macaroon-tree protocols*, which have the full functionality of macaroons described in [BPUE+14]. In particular, macaroons can be issued, extended with a first-party caveat, extended with a third-party caveat, discharged, finalized, and verified. The security definition of macaroon-tree protocols follows directly from combining the security definitions of macaroon schemes and discharge protocols.

Our construction of a macaroon-tree protocol "glues" together a macaroon scheme and a discharge protocol using symmetric-key and public-key encryption, and a message authentication code. The symmetric-key encryption and the MAC are used to create the key identifier of a macaroon: the key identifier is a symmetric-key encryption of the root key k, together with a tag of this ciphertext. The public-key encryption is used to encrypt the token issuing key of the discharge protocol under the verifying party's public key. We note that this encryption can be removed in certain cases and discuss this optimization in Section 5. We further note that the construction is *generic*, so that *any* macaroon scheme and *any* discharge protocol can be combined. In particular, a privately verifiable macaroon scheme can be used with the discharge protocol that uses publicly-verifiable discharges, and vice versa.

Concrete vs. Asymptotic Security. The proofs of security in this work use asymptotics (e.g. $\mathsf{poly}(\cdot)$ and $\mathsf{negl}(\cdot)$ functions). Though they establish the *theoretical* security of the constructions, a more concrete analysis of probabilities that takes into account the computational power of the adversary and possible upper bounds for the number of queries, must be performed in specific use cases to guarantee their concrete security.

Organization. We begin by presenting a few preliminary definitions in Section 2. In Section 3, we present our definitions and constructions of macaroon schemes. In Section 4, we discuss discharge protocols, and finally, in Section 5, we discuss macaroon-tree protocols. For clarity of presentation, all proofs are presented in the Appendix.

2 Preliminaries

We assume basic knowledge of cryptographic primitives such as PRFs, MACs, signatures, and symmetric-key and public-key encryption. For completeness, we review their security definitions below.

Notation. For a randomized function f, we write $f(x;r)$ to denote the unique output of f on input x with random coins r. We write $f(x)$ to denote a random variable for the output of $f(x;r)$ over the random coins r. For a distribution or random variable X, we write $x \leftarrow X$ to denote the operation of sampling a random x according to X. For a set S, we overload notation and use $s \leftarrow S$ to denote sampling s from the uniform distribution over S. We use $y := f(x)$ to denote the deterministic evaluation of f on input x with output y, and in general, as the assignment operator. Finally, we use $a||b$ to denote the concatenation of bit strings a and b, and λ to denote the empty string.

2.1 Pseudorandom Functions (PRFs)

A *pseudorandom function* or *PRF* is a function that is indistinguishable from a truly random function. A more formal definition follows.

Definition 2.1. *A function* $F : \{0,1\}^n \times \{0,1\}^B \to \{0,1\}^n$ *is* pseudorandom *if for all PPT distinguishers* $\mathcal{A}$*,*

$$\left|\Pr[\mathcal{A}^{F_k(\cdot)}(1^n) = 1] - \Pr[\mathcal{A}^{f(\cdot)}(1^n) = 1]\right| \leq \mathsf{negl}(n),$$

where $k \leftarrow \{0,1\}^n$ *and* f *is chosen uniformly at random from the set of functions mapping* $\{0,1\}^B$ *to* $\{0,1\}^n$*.*

Note that the oracle $f(\cdot)$ can be simulated efficiently by answering every query with an independent and uniformly random value in $\{0,1\}^n$.

Definition 2.2. *Let $F : \{0,1\}^n \times \{0,1\}^B \to \{0,1\}^n$. Define the functions $F^{(1)}, F^{(2)}, \ldots$ inductively as follows:*

$$F^{(1)}(k, x_1) = F(k, x_1) \quad \text{and} \quad F^{(n)}(k, x_1, \ldots, x_n) := F(F^{(n-1)}(k, x_1, \ldots, x_{n-1}), x_n)$$

Define $F^*(k, x_1, \ldots, x_n) := F^{(n)}(k, x_1, \ldots, x_n)$.

The above construction is called the *cascade construction*. Bellare, Canetti, and Krawczyk [BCK96] showed that if F is a PRF, then F^* is a pseudorandom against prefix-free adversaries, that is, against adversaries whose query set is a prefix-free set.

Theorem 2.3 ([BCK96])**.** *If F is a PRF, then F^* described above is pseudorandom against distinguishers whose query set is prefix-free.*

2.2 Message Authentication Codes (MAC)

Definition 2.4. *A message authentication code (MAC) $\mathcal{M} = (\mathsf{KeyGen}, \mathsf{Mac}, \mathsf{Verify})$ is* existentially-unforgeable under a chosen message attack (EU-CMA-secure) *if for all PPT adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins the following game is negligible in κ.*

Setup: *The challenger generates a key $k \leftarrow \mathsf{KeyGen}(1^\kappa)$.*

Tag Queries: *The adversary $\mathcal{A}$ is given access to a tag oracle. On its ith query, the adversary sends a message $m^{(i)}$, the challenger computes $t^{(i)} \leftarrow \mathsf{Mac}(k, m^{(i)})$ and sends $t^{(i)}$ as response.*

Winning Condition: *The adversary outputs a message and forgery m^*, t^*, and he wins the game if t^* is a valid tag for m^* under key k, that is, if $\mathsf{Verify}(k, m^*, t^*) = 1$.*

2.3 Signatures

Definition 2.5. *A signature scheme $\mathcal{S} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is* existentially-unforgeable under a chosen message attack (EU-CMA-secure) *if for all PPT adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins the following game is negligible in κ.*

Setup: *The challenger generates a key pair $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$.*

Tag Queries: *The adversary $\mathcal{A}$ is given access to a signature oracle. On its ith query, the adversary sends a message $m^{(i)}$, the challenger computes $\sigma^{(i)} \leftarrow \mathsf{Sign}(\mathsf{sk}, m^{(i)})$ and sends $\sigma^{(i)}$ as response.*

Winning Condition: *The adversary outputs a message and forgery m^*, σ^*, and he wins the game if σ^* is a valid signature for m^* under key vk, that is, if $\mathsf{Verify}(\mathsf{vk}, m^*, t^*) = 1$.*

2.4 CPA-Secure Symmetric-Key Encryption

Definition 2.6. *A symmetric-key encryption scheme $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ has* indistinguishable ciphertexts under a chosen plaintext attack (is CPA-secure) *if for all PPT adversaries $\mathcal{A}$, the adversary's advantage in winning the following game is negligible in κ.*

Setup: *The challenger generates a key $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$.*

Encryption Queries: *The adversary $\mathcal{A}$ is given access to an encryption oracle. On its ith query, the adversary sends a message $m^{(i)}$, the challenger computes $c^{(i)} \leftarrow \mathsf{Enc}(\mathsf{sk}, m^{(i)})$ and sends $c^{(i)}$ as response.*

Challenge: *The adversary picks two challenge plaintexts, m_0, m_1, and sends them to the challenger. The challenger picks a bit at random, $b \leftarrow \{0,1\}$, computes $c^* \leftarrow \mathsf{Enc}(\mathsf{sk}, m_b)$, and sends c^* to $\mathcal{A}$.*

Encryption Queries: *The adversary can make more encryption queries.*

Winning Condition: *The adversary outputs a bit b^* and wins if $b^* = b$. We define the advantage of $\mathcal{A}$ in this game to be: $|\Pr[\mathcal{A} \text{ wins}] - 1/2|$.*

2.5 CCA2-Secure Public-Key Encryption

Definition 2.7. *A public-key encryption scheme* $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable ciphertexts under a chosen ciphertext attack (is CCA2-secure) *if for all PPT adversaries* $\mathcal{A}$*, the adversary's advantage in winning the following game is negligible in* κ*.*

Setup: *The challenger generates a key pair* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$*.*

Decryption Queries: *The adversary* $\mathcal{A}$ *is given access to a decryption oracle. On its* i*th query, the adversary sends a ciphetext* $c^{(i)}$*, the challenger computes* $m^{(i)} \leftarrow \mathsf{Dec}(\mathsf{sk}, c^{(i)})$ *and sends* $m^{(i)}$ *as response.*

Challenge: *The adversary picks two challenge plaintexts,* m_0, m_1*, and sends them to the challenger. The challenger picks a bit at random,* $b \leftarrow \{0,1\}$*, computes* $c^* \leftarrow \mathsf{Enc}(\mathsf{sk}, m_b)$*, and sends* c^* *to* $\mathcal{A}$*.*

Decryption Queries: *The adversary can make more decryption queries, as long as the query ciphertext* $c^{(i)}$ *is not the challenge ciphertext* c^**, i.e. as long as* $c^{(i)} \neq c^*$*.*

Winning Condition: *The adversary outputs a bit* b^**, and wins if* $b^* = b$*. We define the advantage of* $\mathcal{A}$ *in this game to be:* $|\Pr[\mathcal{A} \text{ wins}] - 1/2|$*.*

3 Macaroons

A *macaroon* is a bearer token that acts as an authorization credential and includes *caveats* that attenuate and contextually confine when, where, by who, and for what purpose authorization should be granted. We model each caveat by a bit-string $m_i \in \{0,1\}^*$. Additionally, a macaroon includes a key identifier id that opaquely encodes a root key k, and a signature σ that guarantees the integrity of the macaroon and can be verified using the key k.

Definition 3.1. *A* macaroon *consists of a key identifier* id*, an optional tuple of messages or* caveats $(m_1, \ldots, m_n)$ *for some* $n \geq 1$*, and a signature* σ*. We use the notation* $\mu := (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$ *to denote a macaroon* μ*, or* $\mu := (\mathsf{id} \mid \ \mid \sigma)$ *if the macaroon does not have any caveats.*

3.1 Macaroon Schemes

We define a new primitive called a *macaroon scheme* that defines how macaroons can be issued, extended (for delegation or attenuation), finalized, and verified. A key generation algorithm generates root keys k, with which a macaroon can be issued. When a macaroon is issued, an *extension key* is also given. Using this extension key, the new macaroon can be extended or delegated, essentially adding more caveats to it, and a new extension key is issued so as to allow further delegation. A verification algorithm is also defined such that given the corresponding root key k, the validity of a macaroon μ can be verified. Finally, an algorithm $\mathsf{Finalize}$ is defined, that modifies a given macaroon so that it can be verified by the verification algorithm. This last algorithm can be seen as optional – indeed, only one of our constructions requires it.

We make a distinction between a privately-verifiable macaroon scheme (as informally defined above), where a macaroon is issued and verified using the same root key k, and a publicly-verifiable macaroon scheme, where a macaroon is issued with a secret root key sk and verified with a corresponding verification key vk. We formally define privately-verifiable macaroon schemes below, present constructions, and discuss their security in this section. See Section 3.2 for a formal discussion of publicly-verifiable macaroon schemes.

Definition 3.2. *A* privately-verifiable macaroon scheme *is a tuple of algorithms* $\mathcal{M} = (\mathsf{Setup}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Finalize}, \mathsf{Verify})$ *with the following syntax:*

- $k \leftarrow \mathsf{KeyGen}(1^\kappa)$: *Given a security parameter* κ*, outputs a root key* k*.*
- $(\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(k, \mathsf{id})$: *Given a root key* k *and an identifier* id*, outputs a macaroon* $\mu = (\mathsf{id} \mid \ \mid \sigma)$ *with no caveats, and an extension key* ek*.*
- $(\mu', \mathsf{ek}) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$: *Given an extension key* ek*, a macaroon* $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$ *and a message* $m \in \{0,1\}^*$*, outputs a new macaroon* $\mu' := (\mathsf{id}' \mid m_1, \ldots, m_n, m \mid \sigma')$ *with key identifier* id' *such that* id *is a prefix of* id'*, and whose caveat list is the same as that of* μ*, with* m *appended at the end. In particular, it should be easily determined if a macaroon* μ' *is an extension of a macaroon* μ*.*

- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: *Given an extension key* ek *and a macaroon* $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$*, outputs a new macaroon* $\mu' = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma')$ *with the same key identifier and the same caveat list as* μ*.*
- $b := \mathsf{Verify}(k, \mu)$: *Given a root key* k *and a macaroon* μ*, outputs a bit* b*.*

We require that these algorithms satisfy correctness: for all $\mathsf{id} \in \{0,1\}^*$*, for all* $n \geq 1$ *and all* $m_1, \ldots, m_n \in \{0,1\}^*$*, if* $k \leftarrow \mathsf{KeyGen}(1^\kappa)$*,* $(\mu_0, \mathsf{ek}_0) \leftarrow \mathsf{Issue}(k, \mathsf{id})$*,* $(\mu_i, \mathsf{ek}_i) \leftarrow \mathsf{Extend}(\mathsf{ek}_{i-1}, \mu_{i-1}, m_i)$ *for* $i \in [n]$*,* $\mu_{\mathsf{final}} \leftarrow \mathsf{Finalize}(\mathsf{ek}_n, \mu_n)$*, then* $\mathsf{Verify}(k, \mu_{\mathsf{final}}) = 1$ *with probability 1.*

The above definition outlines the syntax of a macaroon scheme; we now turn to defining its security. The desired security guarantee for a macaroon scheme is that a principal cannot *forge* a macaroon. In other words, a principal cannot create a valid macaroon $\mu*$ with respect to a root key k unless it obtained the macaroon via "valid means", i.e. μ^* was given to him, and μ^* was issued using k or is an extension of such a macaroon. Notice that this includes removing caveats from a macaroon obtained legally.

We wish to guarantee security even against colluding principals and as such define a global "adversary" that might obtain valid macaroons and whose goal is to forge a *new* macaroon. This has a similar flavor to the security definition of MACs or signatures, and so we follow the same structure, allowing the macaroon equivalent of "signature queries". In our case, however, macaroons can be obtained from issuing a new macaroon or extending an existing one. Therefore, we allow the adversary to make both *issuing queries* and *extension queries*.

Finally, we must define a "valid forgery". For signatures, a valid forgery means a signature of message that was not given as part of a signature query. For macaroons, we must extend this to mean a macaroon that was not a result of a query, or an extension of such a macaroon. This is because the extension key can be either implicit in the macaroon (as in our first construction) or empty (as in our second construction). Therefore, obtaining the macaroon itself is enough to extend or delegate it. To this end, we have the challenger keep track of the macaroons that have been issued and extended in queries, and have the adversary explicitly ask to receive these macaroons. This allows a valid forgery to include "parents" of macaroons that have been revealed to the adversary. In other words, this guarantees a breach of security if an adversary successfully removes caveats in macaroon that it obtained legally.

Definition 3.3. *We say that a macaroon scheme* $\mathcal{M} = (\mathsf{KeyGen}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Finalize}, \mathsf{Verify})$ *is* secure *if for all PPT adversaries* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins in the following game is negligible in* κ*.*

Setup: *The challenger generates a key* $k \leftarrow \mathsf{KeyGen}(1^\kappa)$*. The challenger keeps a tree of macaroons* $\mathcal{T}$*, which has macaroon/extension key pairs as nodes and ids/caveats as edges. As a general rule, a macaroon in* $\mathcal{T}$ *whose path from the root is* $(\mathsf{id}, \vec{m})$ *has the form* $(\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma)$ *for some signature* σ *and key identifier* $\widetilde{\mathsf{id}} = \mathsf{id}||\ldots$ *that has* id *as prefix. Note that its caveats are exactly* $\vec{m}$*. The challenger initializes* $\mathcal{T}$ *to contain the empty string,* λ*, as root.*

Issue Queries: *On the* i*th query, the adversary chooses an identifier* $\mathsf{id}^{(i)}$*. The challenger creates a macaroon* $(\mu_0^{(i)}, \mathsf{ek}_0^{(i)}) \leftarrow \mathsf{Issue}(k, \mathsf{id}^{(i)})$*. It adds* $(\mu_0^{(i)}, \mathsf{ek}_0^{(i)})$ *as a child of the root under an edge labeled with* $\mathsf{id}^{(i)}$*.*

Extend Queries: *On the* i*th query, the adversary sends an identifier* $\mathsf{id}^{(i)}$*, a tuple of caveats* $\vec{m}^{(i)}$*, and a message* $m^{(i)} \in \{0,1\}^*$ *to the challenger. If* $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ *is a valid path in* $\mathcal{T}$ *leading to a node* (μ, ek)*, then the challenger computes* $(\mu^{(i)}, \mathsf{ek}^{(i)}) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m^{(i)})$*, and adds* $(\mu^{(i)}, \mathsf{ek}^{(i)})$ *to* $\mathcal{T}$ *as a child of* (μ, ek) *under an edge labeled with* $m^{(i)}$*.*

Macaroon Queries: *On the* i*th query, the adversary sends an identifier* $\mathsf{id}^{(i)}$ *and a tuple of caveats* $\vec{m}^{(i)}$ *to the challenger. If* $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ *is a valid path in* $\mathcal{T}$ *leading to a node* (μ, ek)*, then the challenger returns* (μ, ek) *to the adversary.*

Winning Condition: *The adversary* $\mathcal{A}$ *outputs a macaroon* $\mu^* := (\widetilde{\mathsf{id}}^*||\vec{m}^*||\sigma^*)$ *with* $\widetilde{\mathsf{id}}^* := \mathsf{id}^*||\ldots$*. He wins if* $\mathsf{Verify}(k, \mu^*) = 1$ *and* μ^* *is not an extension of a macaroon given to* $\mathcal{A}$ *in a macaroon query. This can be easily checked by checking if* $(\mathsf{id}^*, \vec{m}^*)$ *leads down a path that includes a macaroon that was given to* $\mathcal{A}$ *in a macaroon query, or equivalently, if the caveat list of any of the macaroons given to* $\mathcal{A}$ *in a macaroon query with identifier* id^**, is a prefix of* μ^**'s caveat list.*

3.1.1 Construction 1

Below we show how to construct a macaroon scheme from a variable-input PRF. This construction is precisely the MAC-based construction described in [BPUE+14], where the key identifier and the caveats are MAC'ed in a cascade fashion, i.e. the root key k is used to create a signature, σ_0, of the key identifier id; σ_0 is then used as a key to create a signature, σ_1, of the first caveat m_1, and so on. Here, we formalize the properties required from the "MAC" function, and show that using a variable-input PRF suffices. Intuitively, the pseudorandomness of the output guarantees that it is computationally indistinguishable from a valid and fresh key, and the unpredictability of the output ensures unforgeability.

Let $F : \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$ be a variable-input PRF. We define:

- $\mathsf{KeyGen}(1^\kappa)$: Output $k \leftarrow \{0,1\}^n$.
- $(\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(k, \mathsf{id})$: Output $\mu := (\mathsf{id} \mid \ \mid \sigma)$ and $\mathsf{ek} := \sigma$, where $\sigma := F(k, \mathsf{id})$.
- $(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$: Parse $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$. Output $\mu' := (\mathsf{id} \mid m_1, \ldots, m_n, m \mid \sigma')$ and $\mathsf{ek} := \sigma'$, where $\sigma' := F(\sigma, m)$.
- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: Output $\mu' := \mu$ (i.e. don't do anything)
- $b := \mathsf{Verify}(k, \mu)$: Parse $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$. Compute $\sigma_0 := F(k, \mathsf{id})$. For $i = 1, \ldots, n$, compute $\sigma_i := F(k, \sigma_i)$. Output 1 if $\sigma_n = \sigma$, and 0 otherwise.

Theorem 3.4. . *Let $F : \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$ be a variable-input PRF. Then the above construction is a secure macaroon scheme.*

To prove Theorem 3.4, we must assume that if an adversary will ask a macaroon queries $\mathsf{id}, \vec{m}_1$ and $\mathsf{id}, \vec{m}_2$ such that $\vec{m}_1$ is a prefix of $\vec{m}_2$, then it will ask the query $\mathsf{id}, \vec{m}_1$ *before* the query $\mathsf{id}, \vec{m}_2$. Even though making this assumption weakens the theorem statement, it might not be an unreasonable assumption to make in the setting of macaroons. This is because the macaroon μ_2 corresponding to $\mathsf{id}, \vec{m}_2$ is an extension of the macaroon μ_1 corresponding to $\mathsf{id}, \vec{m}_1$, and therefore μ_1 must be computed before μ_2. In assuming that the adversary learns μ_1 before μ_2, we are assuming *static* (vs. adaptive) corruptions of the parties, or in other words, we assume that if a party is corrupted at any point in time then it was corrupted from the beginning.

3.1.2 Construction 2

We now show a new construction of a privately-verifiable macaroon scheme. To issue a macaroon, one simply computes a public-key signature of the key identifier. Extending a macaroon requires sampling a new signature key pair, and signing the previous signature, together with the new verification key, and the new caveat. However, notice that this simple construction is insecure, as it allows an adversary to obtain a valid macaroon and remove its caveats, by simply removing the corresponding signatures. To safe-guard against this, we encrypt the intermediate signatures using a public-key encryption scheme. The macaroon can then be verified using the original verification key (corresponding to the secret key used to sign the key identifier), and the decryption key.

Let $\mathcal{S} = (\mathsf{KeyGen}_\mathcal{S}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme, and let $\mathcal{E} = (\mathsf{KeyGen}_\mathcal{E}, \mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure public-key encryption encryption scheme. We define:

- $\mathsf{KeyGen}(1^\kappa)$: Output $k := (\mathsf{vk}_\mathcal{S}, \mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{S}, \mathsf{sk}_\mathcal{E})$, where

$$(\mathsf{vk}_\mathcal{S}, \mathsf{sk}_\mathcal{S}) \leftarrow \mathsf{KeyGen}_\mathcal{S}(1^\kappa) \quad , \quad (\mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{E}) \leftarrow \mathsf{KeyGen}_\mathcal{E}(1^\kappa)$$

- $(\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(k, \mathsf{id})$: Parse $k := (\mathsf{vk}_\mathcal{S}, \mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{S}, \mathsf{sk}_\mathcal{E})$. Output $\mu = (\widetilde{\mathsf{id}} \mid \ \mid \sigma)$, where

$$\widetilde{\mathsf{id}} := \mathsf{id} || \mathsf{pk}_\mathcal{E} \quad , \quad \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_\mathcal{S}, \widetilde{\mathsf{id}}) \quad , \quad \mathsf{ek} := \lambda,$$

  and λ is the empty string.

- $(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$: Parse $\mu = (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$ and parse $\widetilde{\mathsf{id}} = \mathsf{id}||\mathsf{pk}_{\mathcal{E}}||\ldots$ Output $\mu' := (\widetilde{\mathsf{id}}' \mid m_1, \ldots, m_n, m \mid \sigma')$, where:

  $$(\mathsf{vk}', \mathsf{sk}') \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa) \quad , \quad c \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, \sigma) \quad , \quad \widetilde{\mathsf{id}}' := \widetilde{\mathsf{id}}||(\mathsf{vk}', c) \quad , \quad \sigma' \leftarrow \mathsf{Sign}(\mathsf{sk}', \mathsf{vk}'||c||m) \quad , \quad \mathsf{ek} := \lambda$$

  and λ is the empty string.
- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: Output $\mu' := \mu$ (i.e. don't do anything)
- $b := \mathsf{Verify}(k, \mu)$: Parse $\mu = (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$ and parse $\widetilde{\mathsf{id}} = \mathsf{id}||\mathsf{pk}_{\mathcal{E}}||(\mathsf{vk}_1, c_1)||\ldots||(\mathsf{vk}_n, c_n)$. For $i = 0, \ldots, n-1$, compute $\sigma_i := \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, c_{i+1})$, and let $\sigma_n := \sigma$. Check that $\mathsf{Verify}(\mathsf{vk}_{\mathcal{S}}, \mathsf{id}||\mathsf{pk}_{\mathcal{E}}, \sigma_0) = 1$ and $\mathsf{Verify}(\mathsf{vk}_i, \mathsf{vk}_i||c_i||m_i, \sigma_i) = 1$ for $i = 1, \ldots, n$.

Theorem 3.5. *Let $\mathcal{S} = (\mathsf{KeyGen}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Verify}_{\mathcal{S}})$ be an EU-CMA-secure signature scheme, and let $\mathcal{E} = (\mathsf{KeyGen}_{\mathcal{E}}, \mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure public-key encryption scheme. Then the above construction is a secure macaroon scheme.*

Trade-Offs. Though Construction 2 is considerably less efficient than Construction 1 (as it uses public-key primitives), it has one advantage. Notice that in Construction 1, the process of verifying a macaroon reveals all intermediate signatures, which double as extension keys in that construction. As observed in [BPUE+14], this can be a problem when used within macaroon-tree protocols, as is our goal (see Section 5). Construction 2 does not use extension keys, and therefore does not suffer from this weakness.

3.2 Publicly-Verifiable Macaroon Schemes

Similarly to privately-verifiable macaroons schemes, we define publicly-verifiable macaroon schemes. The only difference is that a macaroon is issued with a secret root key sk and verified with a corresponding public verification key vk.

Definition 3.6. *A* public-key macaroon scheme *is a tuple of algorithms $\mathcal{M} = (\mathsf{Setup}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Verify})$ with the following syntax:*

- $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$: *Given a security parameter, outputs a verification key vk and a secret key sk.*
- $(\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(\mathsf{sk}, \mathsf{id})$: *Given a secret key sk and an identifier id, outputs a macaroon $\mu = (\mathsf{id} \mid \mid \sigma)$ with no caveats, and an extension key ek.*
- $(\mu', \mathsf{ek}) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$: *Given an extension key ek, a macaroon $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$ and a message $m \in \{0,1\}^*$, outputs a new macaroon $\mu' := (\mathsf{id}' \mid m_1, \ldots, m_n, m \mid \sigma')$ with key identifier id' such that id is a prefix of id', and whose caveat list is the same as that of μ, with m appended at the end. In particular, it should be easily determined if a macaroon μ' is an extension of a macaroon μ.*
- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: *Given an extension key ek and a macaroon $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$, outputs a new macaroon $\mu' = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma')$ with the same key identifier and the same caveat list as μ.*
- $b := \mathsf{Verify}(\mathsf{vk}, \mu)$: *Given a verification key vk and a macaroon μ, outputs a bit b.*

We require that these algorithms satisfy correctness: for all $\mathsf{id} \in \{0,1\}^*$, for all $n \geq 1$ and all $m_1, \ldots, m_n \in \{0,1\}^*$, if $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$, $(\mu_0, \mathsf{ek}_0) \leftarrow \mathsf{Issue}(\mathsf{sk}, \mathsf{id})$, $(\mu_i, \mathsf{ek}_i) \leftarrow \mathsf{Extend}(\mathsf{ek}_{i-1}, \mu_{i-1}, m_i)$ for $i \in [n]$, $\mu_{\mathsf{final}} \leftarrow \mathsf{Finalize}(\mathsf{ek}_n, \mu_n)$, then $\mathsf{Verify}(\mathsf{vk}, \mu_{\mathsf{final}}) = 1$ with probability 1.

The definition of security of a public-key macaroon scheme is the analogous to that of a symmetric-key macaroon scheme. The only differences are that the adversary is also given the verification key vk, macaroons are issued with the secret key sk, and the adversary wins if μ^* verifies under the verification key vk.

3.2.1 Construction

Below we show how to construct a publicly-verifiable macaroon scheme from a public-key signature scheme. This construction is a formalization of the certificate-based construction described in [BPUE+14]. Intuitively, in issuing and extending macaroons, one constructs two "strands" of signatures, denoted here with

ν and τ. The first is a certificate chain that authenticates the second, and the second is used to sign the key identifier and each of the caveats.

Note that the $\mathsf{Finalize}$ algorithm is crucial in this construction, as it ensures knowledge of the last signing key in the chain. This ensures that caveats cannot be removed from a legally-obtained macaroon, except by the principal who created the macaroon in the first place.

Let $\mathcal{S} = (\mathsf{KeyGen}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Verify}_{\mathcal{S}})$ be a signature scheme. We define:

- $\mathsf{KeyGen}(1^\kappa)$: Output $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa)$.
- $(\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(\mathsf{sk}, \mathsf{id})$: Output $\mu := (\widetilde{\mathsf{id}} \mid \ \mid \sigma)$ and $\mathsf{ek} = \mathsf{sk}'$, where:

  $(\mathsf{vk}', \mathsf{sk}') \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa) \quad , \quad \nu \leftarrow \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}, \mathsf{vk}') \quad , \quad \tau \leftarrow \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}', \mathsf{id}) \quad , \quad \sigma := (\nu, \tau) \quad , \quad \widetilde{\mathsf{id}} := \mathsf{id}||\mathsf{vk}'$

- $(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$: Parse $\mu := (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$ and $\widetilde{\mathsf{id}} = \mathsf{id}||\ldots$. Output $\perp$ if $m = \mathsf{id}$. Otherwise, output $\mu' := (\widetilde{\mathsf{id}}' \mid m_1, \ldots, m_n, m \mid \sigma')$ and $\mathsf{ek}' = \mathsf{sk}'$, where:

  $(\mathsf{vk}', \mathsf{sk}') \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa) \quad , \quad \nu \leftarrow \mathsf{Sign}_{\mathcal{S}}(\mathsf{ek}, \mathsf{vk}') \quad , \quad \tau \leftarrow \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}', m) \quad , \quad \sigma' := \sigma||(\nu, \tau) \quad , \quad \widetilde{\mathsf{id}}' := \widetilde{\mathsf{id}}||\mathsf{vk}'$

- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: Parse $\mu := (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$. Output $\mu' := (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n, m \mid \sigma')$, where $\sigma' := \sigma||\sigma_{\mathsf{final}}$ and $\sigma_{\mathsf{final}} \leftarrow \mathsf{Sign}_{\mathcal{S}}(\mathsf{ek}, \mathsf{id})$.
- $b := \mathsf{Verify}(\mathsf{vk}, \mu)$: Parse $\mu := (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$, parse $\widetilde{\mathsf{id}} = \mathsf{id}||\mathsf{vk}_1||\ldots||\mathsf{vk}_{n+1}$, and parse $\sigma = (\nu_0, \tau_0)||\ldots||(\nu_n, \tau_n)||\sigma_{\mathsf{final}}$. For simplicity, let $\mathsf{vk}_0 := \mathsf{vk}$ and $m_0 := \mathsf{id}$. For $i = 0, \ldots, n$, check that $\mathsf{Verify}_{\mathcal{S}}(\mathsf{vk}_i, \mathsf{vk}_{i+1}, \nu_i) = 1$ and $\mathsf{Verify}_{\mathcal{S}}(\mathsf{vk}_{i+1}, m_i, \tau_i) = 1$. Finally, check that $\mathsf{Verify}_{\mathcal{S}}(\mathsf{vk}_{n+1}, \mathsf{id}, \sigma_{\mathsf{final}}) = 1$.

Theorem 3.7. *Let $\mathcal{S} = (\mathsf{KeyGen}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Verify}_{\mathcal{S}})$ be an EU-CMA-secure signature scheme. Then the above construction is a secure macaroon scheme.*

4 Discharge Protocols

Having defined and constructed macaroon schemes, we now turn to studying the "discharge" of so-called "third-party" caveats. Third-party caveats require a third-party to assert that a given condition holds true. As an example (previously used in [BPUE+14]), a third-party caveat can require proof of membership in a group G at service A. Before presenting a macaroon to a target service for verification, a holder of a macaroon must obtain "discharges" from the appropriate third-parties, asserting that the conditions specified in the third-party caveats holds true. These discharges are also macaroons.

In what follows, we formally define *discharge protocols*, specifying the syntax of third-party caveats and their corresponding discharges. After an initial discharge setup that creates a discharging key sk_D and an initial issuing setup that creates an issuing secret key sk_I, tokens can be issued using the issuing key and discharged using the discharging key. Every invocation to the token-issuing algorithm outputs two tokens - a state token α_S and a discharge token α_D. The discharge token α_D is used to obtain a discharge macaroon and the state token α_S is used to verify the validity of the discharge obtained.

Looking ahead, a third-party caveat will include both the state token α_S, as well as the discharge token α_D. The discharge token α_D can be used by the holder of the macaroon to obtain the discharge from the discharging third-party who holds the discharging key sk_D, and the state token α_S can be used by the target service to verify the validity of the discharge presented. See Section 5 for more details.

Definition 4.1. *A* macaroon discharge protocol *is a tuple of algorithms $\mathcal{D} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Discharge}, \mathsf{Verify})$ with the following syntax:*

- $(\mathsf{params}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{SetupDischarge}(1^\kappa)$: *Given a security parameter κ, outputs public parameters* params *and a discharging secret key sk_D. All other algorithms take* params *as argument, even if not explicitly written.*
- $\mathsf{sk}_\mathsf{T} \leftarrow \mathsf{SetupToken}(1^\kappa)$: *Given a security parameter κ, outputs a token-issuing secret key sk_T.*

- $(\alpha_{\mathsf{S}}, \alpha_{\mathsf{D}}) \leftarrow \mathsf{Token}(\mathsf{sk}_{\mathsf{T}})$*: Given the token-issuing secret key* sk_{T}*, outputs a state token* α_{S} *and a discharge token* α_{D}*.*
- $\mu \leftarrow \mathsf{Discharge}(\mathsf{sk}_{\mathsf{D}}, \alpha_{\mathsf{D}})$*: Given a discharge token* α_{D} *and the discharging secret key* sk_{D}*, outputs a macaroon* μ*.*
- $b := \mathsf{Verify}(\mathsf{sk}_{\mathsf{T}}, \alpha_{\mathsf{S}}, \mu)$*: Given a macaroon* μ*, a state token* α_{S}*, and the token-issuing secret key* sk_{T}*, outputs a bit* b*.*

We require that these algorithms satisfy correctness*: if* $(\mathsf{params}, \mathsf{sk}_{\mathsf{T}}, \mathsf{sk}_{\mathsf{D}}) \leftarrow \mathsf{Setup}(1^{\kappa})$*,* $(\alpha_{\mathsf{S}}, \alpha_{\mathsf{D}}) \leftarrow \mathsf{Token}(\mathsf{sk}_{\mathsf{T}})$*,* $\mu \leftarrow \mathsf{Discharge}(\mathsf{sk}_{\mathsf{D}}, \alpha_{\mathsf{D}})$*, then* $\mathsf{Verify}(\mathsf{sk}_{\mathsf{T}}, \alpha_{\mathsf{S}}, \mu) = 1$ *with probability 1.*

Note that in the use-case of discharge protocols, a principal will present a macaroon and the corresponding discharges to a target service. The target service will then use the state tokens that are part of the third-party caveats in the macaroon and use these to verify the discharges that are presented along with the macaroon. Therefore, for the security of a discharge protocol, we wish to ensure that without knowing the issuing or discharge keys, sk_{I} and sk_{D}, a principal cannot forge a state token α^*_{S} and a discharge macaroon μ^* such that μ^* is a valid discharge macaroon for α^*_{S}, unless they were legally obtained.

To ensure this, in our definition we allow the adversary to obtain tokens and discharges via *token queries* and *discharge queries*, and only consider a forgery $(\alpha^*_{\mathsf{S}}, \mu^*)$ to be valid if μ^* is a valid discharge macaroon for α^*_{S}, and α^*_{S} and μ^* were not *both* legally obtained from token and discharge queries. That is, we allow α^*_{S} to be the output of a token query $(\alpha^*_{\mathsf{S}}, \alpha^*_{\mathsf{D}})$, as long as μ^* was not the output of a discharge query using the corresponding token α^*_{D}. Thus, security is broken if a principal is able to obtain a valid discharge for a third-party caveat that it did not create (i.e. for a state token that it did not issue), without the help of the third-party who owns the discharging key sk_{D}. Furthermore, we consider security broken if a principal is able to forge a state token $\mathsf{sk}^*_{\mathsf{S}}$.

Definition 4.2. *We say that a macaroon discharge protocol* $\mathcal{D} = (\mathsf{Setup}, \mathsf{Issue}, \mathsf{Discharge}, \mathsf{Verify})$ *is* secure *if for all PPT adversaries* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins in the following game is negligible in* κ*.*

Setup: *The challenger chooses* $(\mathsf{params}, \mathsf{sk}_{\mathsf{D}}) \leftarrow \mathsf{SetupDischarge}(1^{\kappa})$ *and* $\mathsf{sk}_{\mathsf{T}} \leftarrow \mathsf{SetupToken}(1^{\kappa})$ *and gives* params *to* $\mathcal{A}$*. The challenger keeps track of a set of tokens* $\mathcal{Q}_{\mathsf{T}}$ *and a set of discharges* $\mathcal{Q}_{\mathsf{D}}$*, both of which start out empty.*

Token Queries: *The adversary* $\mathcal{A}$ *asks the challenger to issue a token. On the* i*th query, the challenger runs* $(\alpha^{(i)}_{\mathsf{S}}, \alpha^{(i)}_{\mathsf{D}}) \leftarrow \mathsf{Token}(\mathsf{sk}_{\mathsf{T}})$ *and returns* $(\alpha^{(i)}_{\mathsf{S}}, \alpha^{(i)}_{\mathsf{D}})$ *to* $\mathcal{A}$*. It also adds* $(\alpha^{(i)}_{\mathsf{S}}, \alpha^{(i)}_{\mathsf{D}})$ *to* $\mathcal{Q}_{\mathsf{T}}$*. Issuing queries and discharge queries can be interleaved.*

Discharge Queries: *On the* i*th query, the adversary* $\mathcal{A}$ *sends a token* $\alpha^{(i)*}_{\mathsf{D}}$ *to the challenger, who runs* $\mu^{(i)} \leftarrow \mathsf{Discharge}(\mathsf{sk}_{\mathsf{D}}, \alpha^{(i)*}_{\mathsf{D}})$ *and returns* $\mu^{(i)}$ *to* $\mathcal{A}$*. The challenger also adds* $(\alpha^{(i)*}_{\mathsf{D}}, \mu^{(i)})$ *to* $\mathcal{Q}_{\mathsf{D}}$*. Issuing queries and discharge queries can be interleaved.*

Winning condition: *The adversary* wins *if it produces a token* α^*_{S} *and a macaroon* μ^* *such that* μ^* *is not an extension of any macaroon* μ *such that* $(\alpha^*_{\mathsf{D}}, \mu) \in \mathcal{Q}_{\mathsf{D}}$*, where* α^*_{D} *is a corresponding discharge token to* α^*_{S} *in* $\mathcal{Q}_{\mathsf{T}}$*, and* $\mathsf{Verify}(\mathsf{sk}_{\mathsf{I}}, \alpha^*_{\mathsf{S}}, \mu^*) = 1$*. In other words,* $\mathcal{A}$ *wins if it outputs a valid macaroon that is not an extension of a macaroon that was the output of a discharge query with token* α^*_{D} *corresponding to* α^*_{S}*.*

4.1 Construction 1

We first present the encryption-based construction that was described in [BPUE+14]. The idea is for the state and discharge tokens to encode a key k, which will be the root key for the discharge macaroon. The state token is a symmetric-key authenticated encryption of k, whereas the discharge token encrypts k using the third-party's public key. Given a discharge token, the third-party obtains k by decrypting the token, and issues a discharge macaroon using k as root key. A verifying party first checks the integrity of a state token, decrypts it to obtain the root key k, and uses this key to verify the validity of the discharge macaroon.

Let $\mathcal{E}_1 := (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ be a CPA-secure symmetric-key encryption scheme, let $\mathcal{E}_2 := (\mathsf{KeyGen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ be a CCA2-secure public-key encryption scheme, let $\mathcal{T} := (\mathsf{KeyGen}_{\mathcal{T}}, \mathsf{Mac}, \mathsf{Verify}_{\mathcal{T}})$ be a message authentication code, and let $\mathcal{M} = (\mathsf{KeyGen}_{\mathcal{M}}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Verify}_{\mathcal{M}})$ be a macaroon scheme. We define:

- $\mathsf{SetupDischarge}(1^\kappa)$: Output $\mathsf{params} := \mathsf{pk}_2$, $\mathsf{sk}_\mathsf{D} := \mathsf{sk}_2$, where $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{KeyGen}_2(1^\kappa)$.
- $\mathsf{SetupToken}(1^\kappa)$: Output $\mathsf{sk}_\mathsf{T} := (\mathsf{sk}_1, k)$, where

$$\mathsf{sk}_1 \leftarrow \mathsf{KeyGen}_1(1^\kappa) \quad , \quad k \leftarrow \mathsf{KeyGen}_\mathcal{T}(1^\kappa)$$

- $\mathsf{Token}(\mathsf{sk}_\mathsf{T})$: Parse $\mathsf{params} = \mathsf{pk}_2$, $\mathsf{sk}_\mathsf{T} = (\mathsf{sk}_1, k)$ and output $\alpha_\mathsf{S} := (c_1, \sigma)$ and $\alpha_\mathsf{D} := c_2$, where

$$r \leftarrow \mathsf{KeyGen}_\mathcal{M}(1^\kappa) \quad , \quad c_1 \leftarrow \mathsf{Enc}_1(\mathsf{sk}_1, r) \quad , \quad c_2 \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, r) \quad , \quad \sigma \leftarrow \mathsf{Mac}(k, c_1)$$

- $\mathsf{Discharge}(\mathsf{sk}_\mathsf{D}, \alpha_\mathsf{D})$: Parse $\mathsf{sk}_\mathsf{D} = \mathsf{sk}_2, \alpha_\mathsf{D} = c_2$ and output $\beta = (c_2, t)$ where

$$r \leftarrow \mathsf{Dec}_2(\mathsf{sk}_2, c_2) \quad , \quad \mu \leftarrow \mathsf{Issue}(r, c_2)$$

- $\mathsf{Verify}(\mathsf{sk}_\mathsf{T}, \alpha_\mathsf{S}, \mu)$: Parse $\alpha_\mathsf{D} = (c_1, \sigma)$ and $\mathsf{sk}_\mathsf{T} = (\mathsf{sk}_1, k)$. Check that $\mathsf{Verify}_\mathcal{T}(k, c_1, \sigma) = 1$. Let $r = \mathsf{Dec}_1(\mathsf{sk}_1, c_1)$ and check that $\mathsf{Verify}_\mathcal{M}(r, \mu) = 1$. Output 0 if either of the checks fail; otherwise output 1.

Theorem 4.3. *Let $\mathcal{E}_1 := (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ be a CPA-secure symmetric-key encryption scheme, let $\mathcal{E}_2 := (\mathsf{KeyGen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ be a CCA2-secure public-key encryption scheme, let $\mathcal{T} := (\mathsf{KeyGen}_\mathcal{T}, \mathsf{Mac}, \mathsf{Verify}_\mathcal{T})$ be a message authentication code, and let $\mathcal{M} = (\mathsf{KeyGen}_\mathcal{M}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Verify}_\mathcal{M})$ be a macaroon scheme. Then the above construction is a secure macaroon discharge protocol.*

NEED FOR CCA2-SECURE PKE. We highlight the fact that the proof of Theorem 4.3 would not go through if $\mathcal{E}_2$ is not CCA2-secure but only CPA (or even CCA1) secure. The reason for this is that we need the encryption to be non-malleable. If an adversary were able to tamper with a ciphertext c_2 and obtain a ciphertext c_2' that encrypts a related plaintext, then he could submit c_2' as a discharge query and obtain a macaroon under a related key. If the macaroon is implemented using a signature or MAC (as proposed in Section 3), then the adversary obtains a signature or tag under a related key. In short, using an encryption scheme that permits malleability could allow the adversary to mount a *related-key attack* against the underlying signature or MAC scheme of the macaroon scheme. See, e.g [PSW12], for related-key attacks on HMAC.

USING SYMMETRIC-KEY ENCRYPTION FOR $\mathcal{E}_2$. It is possible, and possibly preferable for efficiency concerns, to have $\mathcal{E}_2$ be a symmetric-key encryption scheme instead of a public-key one. In this case, it is possible to achieve the required CCA2-security by encrypting the plaintext using a CPA-secure symmetric-key encryption scheme and concatenating a MAC tag of the ciphertext to the ciphertext. The new shared key includes both the encryption key and the MAC key.

4.2 Construction 2

We now present a new construction of a discharge protocol based on MACs and a publicly-verifiable macaroon scheme. In this construction, tokens are issued by first sampling a nonce r. The state token is a MAC of r, and the discharge token in simply r. The third-party "authenticates" itself by presenting a valid macaroon with r as key identifier.

Let $\mathcal{T} := (\mathsf{KeyGen}_\mathcal{T}, \mathsf{Mac}, \mathsf{Verify}_\mathcal{T})$ be a message authentication code, and let $\mathcal{M} = (\mathsf{KeyGen}_\mathcal{M}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Verify}_\mathcal{M})$ be a publicly-verifiable macaroon scheme. We define:

- $\mathsf{SetupDischarge}(1^\kappa)$: Output $\mathsf{params} := \mathsf{vk}$ and $\mathsf{sk}_\mathsf{D} := \mathsf{sk}$, where $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}_\mathcal{M}(1^\kappa)$.
- $\mathsf{SetupToken}(1^\kappa)$: Output $\mathsf{sk}_\mathsf{T} := k \leftarrow \mathsf{KeyGen}_\mathcal{T}(1^\kappa)$.
- $\mathsf{Token}(\mathsf{sk}_\mathsf{T})$: Output $\alpha_\mathsf{S} := (r, \sigma)$ and $\alpha_\mathsf{D} := r$, where

$$r \leftarrow \{0,1\}^\kappa \quad , \quad \sigma \leftarrow \mathsf{Mac}(\mathsf{sk}_\mathsf{T}, r)$$

- $\mathsf{Discharge}(\mathsf{sk}_\mathsf{D}, \alpha_\mathsf{D})$: Output $\mu \leftarrow \mathsf{Issue}(\mathsf{sk}_\mathsf{D}, r)$.

- $\mathsf{Verify}(\mathsf{sk}_\mathsf{T}, \alpha_\mathsf{S}, \mu)$: Parse $\mathsf{sk}_\mathsf{T} = (\mathsf{vk}, k)$ and $\alpha_\mathsf{S} = (r, \sigma)$. Check that $\mathsf{Verify}_\mathcal{T}(k, r, \sigma) = 1$, μ's key identifier is r, and $\mathsf{Verify}_\mathcal{M}(\mathsf{vk}, \mu) = 1$. Output 0 if either of the checks fail; otherwise output 1.

Theorem 4.4. *Let $\mathcal{S} := (\mathsf{KeyGen}_\mathcal{S}, \mathsf{Sign}, \mathsf{Verify}_\mathcal{S})$ be an EU-CMA-secure signature scheme, let $\mathcal{T} := (\mathsf{KeyGen}_\mathcal{T}, \mathsf{Mac}, \mathsf{Verify}_\mathcal{T})$ be a message authentication code, and let $\mathcal{M} = (\mathsf{KeyGen}_\mathcal{M}, \mathsf{Issue}, \mathsf{Extend}, \mathsf{Verify}_\mathcal{M})$ be a macaroon scheme. Then the above construction is a secure macaroon discharge protocol.*

5 Macaroon-Tree Protocols

In this section, we show how to combine macaroon schemes and discharge protocols to obtain *macaroon-tree protocols*, which have the full functionality of macaroons described in [BPUE+14]. In particular, macaroons can be issued, extended with a first-party caveat, extended with a third-party caveat, discharged, finalized, and verified. The security definition of macaroon-tree protocols also follows directly from combining the security definitions of macaroon schemes and discharge protocols.

Definition 5.1. *A* macaroon-tree protocol *is a tuple of algorithms $\mathcal{M} = (\mathsf{Setup}, \mathsf{Issue}, \mathsf{Extend1stParty}, \mathsf{Extend3rdParty}, \mathsf{Verify})$ with the following syntax:*

- $(\mathsf{params}, \mathsf{sk}_\mathsf{I}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{Setup}(1^\kappa)$: *Given a security parameter κ, outputs public parameters* params*, an issuing secret key sk_I and a discharge secret key sk_D. All other algorithms take* params *as argument, even if not explicitly written.*
- $(k, \mu, \mathsf{ek}) \leftarrow \mathsf{Issue}(\mathsf{sk}_\mathsf{I})$: *Given an issuing secret key sk_I, outputs a key k and a macaroon μ.*
- $(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend1stParty}(\mathsf{ek}, \mu, m)$: *Given an extension key* ek*, a macaroon $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$ and a message $m \in \{0,1\}^*$, outputs a new macaroon $\mu' := (\mathsf{id}' \mid m_1, \ldots, m_n, m \mid \sigma')$ with key identifier id' such that* id *is a prefix of id', and whose caveat list is the same as that of μ, with m appended at the end. In particular, it should be easily determined if a macaroon μ' is an extension of a macaroon μ.*
- $(\mu', \mathsf{ek}', \alpha_\mathsf{D}) \leftarrow \mathsf{Extend3rdParty}(\mathsf{ek}, \mu)$: *Given an extension key* ek*, a macaroon $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$ and a message $m \in \{0,1\}^*$, outputs a new macaroon $\mu' := (\mathsf{id}' \mid m_1, \ldots, m_n, m \mid \sigma')$ with key identifier id' such that* id *is a prefix of id', and whose caveat list is the same as that of μ, with a new message m appended at the end. In particular, it should be easily determined if a macaroon μ' is an extension of a macaroon μ. This algorithm also outputs a discharge token α_D.*
- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: *Given an extension key* ek *and a macaroon $\mu = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma)$, outputs a new macaroon $\mu' = (\mathsf{id} \mid m_1, \ldots, m_n \mid \sigma')$ with the same key identifier and the same caveat list as μ.*
- $\mu \leftarrow \mathsf{Discharge}(\mathsf{sk}_\mathsf{D}, \alpha_\mathsf{D})$*: Given a discharge token α_D, outputs a macaroon μ. The key identifier of μ must be α_D.*
- $b := \mathsf{Verify}(\mathsf{sk}_\mathsf{I}, \mu, M = \{\mu_j\})$: *Given an issuing secret key sk_I, a macaroon μ, and a set of discharges $M = \{\mu_j\}$, outputs a bit b.*

We require that these algorithms satisfy correctness. To state the correctness requirement, we first define the algorithm $(\mu', \mathsf{ek}', \alpha) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m)$ that first checks if $m = \lambda$, where λ is the empty string. If it is, it outputs $\mathsf{Extend3rdParty}(\mathsf{ek}, \mu)$, otherwise it outputs $\mathsf{Extend1stParty}(\mathsf{ek}, \mu, m)$ and $\alpha = \lambda$. The correctness requirement is then stated as follows: if $(\mathsf{params}, \mathsf{sk}_\mathsf{I}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{Setup}(1^\kappa), (k, \mu_0, \mathsf{ek}_0) \leftarrow \mathsf{Issue}(\mathsf{sk}_\mathsf{I})$, $(\mu_i, \mathsf{ek}_i, \alpha_i) \leftarrow \mathsf{Extend}(\mathsf{ek}_{i-1}, \mu_{i-1}, m_i)$ for $i \in [n]$, $\mu_\mathsf{final} \leftarrow \mathsf{Finalize}(\mathsf{ek}_n, \mu_n)$, $\mu_j \leftarrow \mathsf{Discharge}(\mathsf{sk}_\mathsf{D}, \alpha_j)$ for every j such that $\mu_j = \lambda$ and $M = \{\mu_j\}$, then $\mathsf{Verify}(\mathsf{sk}_\mathsf{I}, \mu_\mathsf{final}, M) = 1$ with probability 1.

Definition 5.2. *We say that a macaroon-tree protocol is* secure *if for all PPT adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins in the following game is negligible in κ.*

Setup: *The challenger chooses $(\mathsf{params}, \mathsf{sk}_\mathsf{I}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{Setup}(1^\kappa)$ and gives* params *to $\mathcal{A}$. The challenger keeps a tree of macaroons $\mathcal{T}$, which has macaroon/extension key pairs as nodes and ids/caveats as edges. As a general rule, a macaroon in $\mathcal{T}$ whose path from the root is $(\mathsf{id}, \vec{m})$ has the form $(\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma)$ for some signature σ and key identifier $\widetilde{\mathsf{id}} = \mathsf{id}||\ldots$ that has* id *as prefix. Note that its caveats are exactly $\vec{m}$. The challenger initializes $\mathcal{T}$ to contain the empty string, λ, as root. The challenger also keeps track of a set of discharges $\mathcal{Q}_\mathsf{D}$, which starts out empty.*

Issue Queries: *On the ith query, the challenger creates a macaroon* $(\mu_0^{(i)}, \mathsf{ek}_0^{(i)}) \leftarrow \mathsf{Issue}(\mathsf{sk}_\mathsf{I})$. *It adds* $(\mu_0^{(i)}, \mathsf{ek}_0^{(i)})$ *as a child of the root under an edge labeled with* $\mu_0^{(i)}$*'s key identifier. All types of queries can be interleaved.*

Extend Queries: *On the ith query, the adversary sends an identifier* $\mathsf{id}^{(i)}$*, a tuple of caveats* $\vec{m}^{(i)}$*, and a message* $m^{(i)} \in \{0,1\}^* \cup \lambda$ *to the challenger, where* λ *is the empty string. If* $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ *is a valid path in* $\mathcal{T}$ *leading to a node* (μ, ek)*, then the challenger computes* $(\mu^{(i)}, \mathsf{ek}^{(i)}, \alpha_\mathsf{D}^{(i)}) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m^{(i)})$*, and adds* $(\mu^{(i)}, \mathsf{ek}^{(i)}, \alpha_\mathsf{D}^{(i)})$ *to* $\mathcal{T}$ *as a child of* (μ, ek) *under an edge labeled with* $m^{(i)}$*. Here,* $\mathsf{Extend}(\mathsf{ek}, \mu, m)$ *is defined as follows: If* $m = \lambda$*, it outputs* $\mathsf{Extend3rdParty}(\mathsf{ek}, \mu)$*, otherwise it outputs* $\mathsf{Extend1stParty}(\mathsf{ek}, \mu, m)$ *and* $\alpha = \lambda$*. All types of queries can be interleaved.*

Discharge Queries: *On the ith query, the adversary* $\mathcal{A}$ *sends a token* $\alpha_\mathsf{D}^{(i)*}$ *to the challenger, who runs* $\mu^{(i)} \leftarrow \mathsf{Discharge}(\mathsf{sk}_\mathsf{D}, \alpha_\mathsf{D}^{(i)*})$ *and returns* $\mu^{(i)}$ *to* $\mathcal{A}$*. The challenger also adds* $(\alpha_\mathsf{D}^{(i)*}, \mu^{(i)})$ *to* $\mathcal{Q}_\mathsf{D}$*. All types of queries can be interleaved.*

Macaroon Queries: *On the ith query, the adversary sends an identifier* $\mathsf{id}^{(i)}$ *and a tuple of caveats* $\vec{m}^{(i)}$ *to the challenger. If* $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ *is a valid path in* $\mathcal{T}$ *leading to a node* (μ, ek)*, then the challenger returns* (μ, ek) *to the adversary. All types of queries can be interleaved.*

Winning condition: *The adversary* $\mathcal{A}$ *outputs a macaroon* μ^* *and a set of discharges* $M^* = \{\mu_j^*\}$*. He wins if* $\mathsf{Verify}(\mathsf{sk}_\mathsf{I}, \mu^*, M^*) = 1$ *and either (1).* μ^* *is not an extension of a macaroon given to* $\mathcal{A}$ *in a macaroon query, or (2).* μ^* *is an extension of a macaroon* μ *given to* $\mathcal{A}$ *in a macaroon query, and there exists a macaroon* $m_j^* \in M^*$ *corresponding to a third-party caveat* $3||c_j||\alpha_\mathsf{S}^{(j)}||\alpha_\mathsf{D}^{(j)}$ *in* μ *such that* μ_j^* *is not an extension of any macaroon* μ' *such that* $(\alpha_\mathsf{D}^{(j)}, \mu') \in \mathcal{Q}_\mathsf{D}$*. In other words,* $\mathcal{A}$ *wins if it outputs a valid macaroon that is not an extension of a macaroon revealed to* $\mathcal{A}$ *in a macaroon query, or if* μ^* *is an extension of a macaroon* μ *revealed to* $\mathcal{A}$ *in a macaroon query and there is a discharge macaroon in* M^* *that corresponds to a third-party caveat in* μ *and that is not an extension of a macaroon that was the output of a discharge query for that caveat.*

5.1 Construction

We describe a construction of a macaroon-tree protocol that "glues" together a macaroon scheme and a discharge protocol using symmetric-key and public-key encryption, and a message authentication code. The symmetric-key encryption and the MAC are used to create the key identifier of a macaroon; in other words, the key identifier of a macaroon is a symmetric-key encryption of the root key k, together with a tag of this ciphertext. Finally, the public-key encryption is used to encrypt the token issuing key of the discharge protocol under the verifying party's public key. We note that this encryption can be removed in certain cases and will discuss this optimization later in this section.

Let $\mathcal{E}_1 = (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ be a CPA-secure symmetric-key encryption scheme, let $\mathcal{E}_2 = (\mathsf{KeyGen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ be a CPA-secure public-key encryption scheme, let $\mathcal{T} = (\mathsf{KeyGen}_\mathcal{T}, \mathsf{Mac}_\mathcal{T}, \mathsf{Verify}_\mathcal{T})$ be a message authentication code, let $\mathcal{M} = (\mathsf{KeyGen}_\mathcal{M}, \mathsf{Issue}_\mathcal{M}, \mathsf{Extend}_\mathcal{M}, \mathsf{Finalize}_\mathcal{M}, \mathsf{Verify}_\mathcal{M})$ be a macaroon scheme, let $\mathcal{D} = (\mathsf{SetupDischarge}_\mathcal{D}, \mathsf{SetupToken}_\mathcal{D}, \mathsf{Token}_\mathcal{D}, \mathsf{Discharge}_\mathcal{D}, \mathsf{Verify}_\mathcal{D})$ be a macaroon discharge protocol. We define:

- $\mathsf{Setup}(1^\kappa)$: Output $(\mathsf{params}, \mathsf{sk}_\mathsf{I}, \mathsf{sk}_\mathsf{D})$, where:

$$(\mathsf{params}_\mathsf{D}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{SetupDischarge}_\mathcal{D}(1^\kappa) \quad , \quad \mathsf{sk}_1 \leftarrow \mathsf{KeyGen}_1(1^\kappa) \quad , \quad (\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{KeyGen}_2(1^\kappa)$$

$$k_\mathcal{T} \leftarrow \mathsf{KeyGen}(1^\kappa) \quad , \quad \mathsf{sk}_\mathsf{I} := (\mathsf{sk}_1, \mathsf{sk}_2, k_\mathcal{T}) \quad , \quad \mathsf{params} := (\mathsf{params}_\mathsf{D}, \mathsf{pk}_2)$$

- $\mathsf{Issue}(\mathsf{sk}_\mathsf{I})$: Parse $\mathsf{sk}_\mathsf{I} = (\mathsf{sk}_1, \mathsf{sk}_2)$. Output (k, μ, ek), where:

$$k \leftarrow \mathsf{KeyGen}_\mathcal{M}(1^\kappa) \quad , \quad c \leftarrow \mathsf{Enc}(\mathsf{sk}_1, k) \quad , \quad \sigma \leftarrow \mathsf{Mac}(k_\mathcal{T}, c) \quad , \quad \mathsf{id} := c||\sigma \quad , \quad (\mu, \mathsf{ek}) \leftarrow \mathsf{Issue}_\mathcal{M}(k, \mathsf{id})$$

- $\mathsf{Extend1stParty}(\mathsf{ek}, \mu, m)$: Output $(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend}_\mathcal{M}(\mathsf{ek}, \mu, 1||m)$.

- $\mathsf{Extend3rdParty}(\mathsf{ek}, \mu)$: Output $(\mu', \mathsf{ek}', \alpha_{\mathsf{D}})$, where:

$$\mathsf{sk}_{\mathsf{T}} \leftarrow \mathsf{SetupToken}_{\mathsf{D}}(1^{\kappa}) \quad , \quad c \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, \mathsf{sk}_{\mathsf{T}}) \quad , \quad (\alpha_{\mathsf{S}}, \alpha_{\mathsf{D}}) \leftarrow \mathsf{Token}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}})$$

$$(\mu', \mathsf{ek}') \leftarrow \mathsf{Extend}_{\mathcal{M}}(\mathsf{ek}, \mu, 3||c||\alpha_{\mathsf{S}}||\alpha_{\mathsf{D}})$$

- $\mu' \leftarrow \mathsf{Finalize}(\mathsf{ek}, \mu)$: Output $\mathsf{Finalize}_{\mathcal{M}}(\mathsf{ek}, \mu)$.
- $\mathsf{Discharge}(\mathsf{sk}_{\mathsf{D}}, \alpha_{\mathsf{D}})$: Output $\mu \leftarrow \mathsf{Discharge}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{D}}, \alpha_{\mathsf{D}})$.
- $b := \mathsf{Verify}\,(\mathsf{sk}_{\mathsf{I}}, \mu, M = \{\mu_j\})$: Parse $\mathsf{sk}_{\mathsf{I}} = (\mathsf{sk}_1, \mathsf{sk}_2)$ and $\mu := (\widetilde{\mathsf{id}} \mid m_1, \ldots, m_n \mid \sigma)$ and $\widetilde{\mathsf{id}} = c||\sigma|| \ldots$. Check that $\mathsf{Verify}(k_{\mathcal{T}}, c, \sigma) = 1$ and if so, compute $k := \mathsf{Dec}(\mathsf{sk}_1, \mathsf{id})$ and verify that $\mathsf{Verify}_{\mathcal{M}}(k, \mu) = 1$. For every third-party caveat m_i in μ (ie. every caveat prefixed with 3), parse $m_i := 3||c_i||\alpha_{\mathsf{S}}^{(i)}||\alpha_{\mathsf{D}}^{(i)}$ and compute $\mathsf{sk}_{\mathsf{T}}^{(i)} := \mathsf{Dec}(\mathsf{sk}_2, c_i)$. Finally, verify that there exists $\mu_j \in M$ with key identifier $\alpha_{\mathsf{D}}^{(i)}$, and verify that $\mathsf{Verify}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}}^{(i)}, \alpha_{\mathsf{S}}^{(i)}, \mu_j) = 1$.

Theorem 5.3. *Let $\mathcal{E}_1 = (\mathsf{KeyGen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ be a CPA-secure symmetric-key encryption scheme, let $\mathcal{E}_2 = (\mathsf{KeyGen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ be a CPA-secure public-key encryption scheme, let $\mathcal{M} = (\mathsf{KeyGen}_{\mathcal{M}}, \mathsf{Issue}_{\mathcal{M}}, \mathsf{Extend}_{\mathcal{M}}, \mathsf{Verify}_{\mathcal{M}})$ be a macaroon scheme, let $\mathcal{D} = (\mathsf{SetupDischarge}_{\mathcal{D}}, \mathsf{SetupToken}_{\mathcal{D}}, \mathsf{Token}_{\mathcal{D}}, \mathsf{Discharge}_{\mathcal{D}}, \mathsf{Verify}_{\mathcal{D}})$ be a macaroon discharge protocol. Then the above construction is a secure macaroon-tree protocol.*

Optimization. If the macaroon scheme has *pseudorandom* extension keys, as is the case in the first macaroon-scheme construction (see Section 3.1.1), then in the algorithm $\mathsf{Extend3rdParty}$, instead of sampling and encrypting sk_{T}, it possible to use a (strong) computational extractor or a KDF to expand the extension key into randomness to run $\mathsf{SetupToken}_{\mathcal{D}}$, as well as a new pseudorandom extension key ek' to use when running $\mathsf{Extend}_{\mathcal{M}}$. This allows Verify to recover sk_{T} and ek' when verifying the validity of a macaroon, and does not require the expensive encryption step.

Acknowledgements

We are very grateful to Úlfar Erlingsson for valuable discussions about the definitions and constructions. We also thank Tyler Close, Edward Knapp, and Yevgeniy Dodis for helpful discussions and feedback.

References

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the 37th Symposium on Foundations of Computer Science, IEEE*, pages 514–523. IEEE, 1996.

[BPUE+14] Arnar Birgisson, Joe Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[PSW12] Thomas Peyrin, Yu Sasaki, and Lei Wang. Generic related-key attacks for hmac. In *ASIACRYPT*, volume 7658, pages 580–597. Springer, 2012.

A Proof of Theorem 3.4: Security of SK Macaroons, Construction 1

Proof. We use a series-of-games argument to show the security of the public-key macaroon scheme construction. In all games, the setup phase is the same: the challenger runs $k \leftarrow \mathsf{KeyGen}(1^{\kappa})$.

We let p_i be the probability that $\mathcal{A}$ wins in Game i. For simplicity, we do not write extension keys since these are always the same as the corresponding signature.

Game 0: Let Game 0 be the original security game from Definition 3.3. In this game, we have:

Issue Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}$ to the challenger who adds $\mu_0^{(i)}$ to $\mathcal{T}$, where:

$$\sigma^{(i)} \leftarrow F(k, \mathsf{id}^{(i)}) \quad , \quad \mu_0^{(i)} = (\mathsf{id}^{(i)} \mid \quad \mid \sigma^{(i)})$$

Extend Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}, \vec{m}^{(i)}, m^{(i)}$ to the challenger, who checks that $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a macaroon $\mu := (\mathsf{id}^{(i)} \mid \vec{m}^{(i)} \mid \sigma)$. If it is, it adds $\mu^{(i)}$ to $\mathcal{T}$ as a child of μ with edge $m^{(i)}$, where:

$$\sigma^{(i)} \leftarrow F(\sigma, m^{(i)}) \quad , \quad \mu^{(i)} := (\mathsf{id}^{(i)} \mid \vec{m}^{(i)}, m^{(i)} \mid \sigma^{(i)})$$

Macaroon Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}, \vec{m}^{(i)}$ to the challenger, who checks that $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a macaroon $\mu := (\mathsf{id}^{(i)} \mid \vec{m}^{(i)} \mid \sigma)$. If it is, it sends μ to the adversary.

Winning Condition: The adversary $\mathcal{A}$ outputs a macaroon $\mu^* = (\mathsf{id}^* \mid m_1^*, \ldots, m_n^* \mid \sigma^*)$. He wins if $\sigma^* = F^*(k, \mathsf{id}^*, m_1^*, \ldots, m_n^*)$ and μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query. Such a μ^* we call a valid macaroon forgery.

Game 1: In Game 1, we change how the challenger answers macaroon queries. At the beginning of the game, the challenger chooses a random function f. It receives issue and extend queries from $\mathcal{A}$ and builds $\mathcal{T}$ but does not compute the signatures immediately. When it receives a macaroon query $\mathsf{id}, \vec{m}$, it checks if there exists a revealed macaroon $\mu' \in \mathcal{T}$ whose path from the root is $\mathsf{id}, \vec{m}'$ and $\vec{m}'$ is a prefix of $\vec{m}$. Or in other words, it checks if any of the ancestors of the macaroon corresponding to $\mathsf{id}, \vec{m}$ in $\mathcal{T}$ have been revealed to $\mathcal{A}$ in a macaroon query. If no ancestor has been revealed, then the challenger computes $\sigma = f(\mathsf{id}, \vec{m})$. If an ancestor $\mu := (\mathsf{id} \mid \vec{m}' \mid \sigma')$ has already been revealed, then the challenger computes the new signature as an extension of σ', that is, it outputs $\mu := (\mathsf{id}, \vec{m}, \sigma)$ where $\sigma = F^*(\sigma', m_{i+1}, \ldots, m_n)$ and $\vec{m} := (m_1, \ldots, m_n)$ and $\vec{m}' := (m_1, \ldots, m_i)$. The adversary wins if $\sigma^* = f(\mathsf{id}^*, \vec{m}^*)$. Note that f can be simulated efficiently by independently sampling a uniformly random value in $\{0,1\}^n$ every time the function is evaluated on a new point.

We claim that $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the pseudorandomness of F^* (see Theorem 2.3). The reduction behaves the same as the challenger in Game 1, except that when it receives a macaroon query and no ancestor has been revealed, it queries its oracle at point $(\mathsf{id}, \vec{m})$ to obtain σ. In the case that the oracle function is F^*, the reduction perfectly simulates Game 0, and if the oracle function is random, then the reduction perfectly simulates Game 1. Therefore, $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the pseudorandomness of F^*. Notice that this is only true since we assume that an adversary will never ask to reveal a macaroon that is an ancestor of a macaroon that has already been revealed. This guarantees that the reduction's query set is prefix-free.

$\mathcal{A}$ **Can't Win:** We argue that $p_1 = \mathsf{negl}(\kappa)$. Since f is a truly random function, then $f(\mathsf{id}^*, \vec{m}^*)$ is uniformly random in $\{0,1\}^n$, and therefore $p_1 = 2^{-n} = \mathsf{negl}(\kappa)$.

We conclude that $p_0 \leq |p_0 - p_1| + p_1 = \mathsf{negl}(\kappa)$.

□

B Proof of Theorem 3.5: Security of SK Macaroons, Construction 2

Proof. We use a series-of-games argument to show the security of the public-key macaroon scheme construction. In all games, the setup phase is the same: the challenger runs $k \leftarrow \mathsf{KeyGen}(1^\kappa)$ and obtains $k := (\mathsf{vk}_\mathcal{S}, \mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{S}, \mathsf{sk}_\mathcal{E})$ where $(\mathsf{vk}_\mathcal{S}, \mathsf{sk}_\mathcal{S}) \leftarrow \mathsf{KeyGen}_\mathcal{S}(1^\kappa)$ and $(\mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{E}) \leftarrow \mathsf{KeyGen}_\mathcal{E}(1^\kappa)$.

We let p_i be the probability that $\mathcal{A}$ wins in Game i. For simplicity, we do not write extension keys since these are always λ, the empty string.

Game 0: Let Game 0 be the original security game from Definition 3.3. In this game, we have:

Issue Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}$ to the challenger who adds $\mu_0^{(i)}$ to $\mathcal{T}$, where:

$$\widetilde{\mathsf{id}}^{(i)} := \mathsf{id}^{(i)} || \mathsf{pk}_\mathcal{E} \quad , \quad \sigma^{(i)} \leftarrow \mathsf{Sign}(\mathsf{sk}_\mathcal{S}, \widetilde{\mathsf{id}}^{(i)}) \quad , \quad \mu_0^{(i)} = (\widetilde{\mathsf{id}}^{(i)} \mid \quad \mid \sigma^{(i)})$$

Extend Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}, \vec{m}^{(i)}, m^{(i)}$ to the challenger, who checks that $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m}^{(i)} \mid \sigma)$. If it is, it adds $\mu^{(i)}$ to $\mathcal{T}$ as a child of μ with edge $m^{(i)}$, where:

$$(\mathsf{vk}^{(i)}, \mathsf{sk}^{(i)}) \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa) \quad , \quad c^{(i)} := \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, \sigma) \quad , \quad \widetilde{\mathsf{id}}^{(i)} := \widetilde{\mathsf{id}} || (\mathsf{vk}^{(i)}, c^{(i)})$$

$$\sigma^{(i)} \leftarrow \mathsf{Sign}(\mathsf{sk}^{(i)}, \mathsf{vk}^{(i)} || c^{(i)} || m^{(i)}) \quad , \quad \mu^{(i)} := (\widetilde{\mathsf{id}}^{(i)} \mid \vec{m}^{(i)}, m^{(i)} \mid \sigma^{(i)})$$

Macaroon Queries: On the ith query, $\mathcal{A}$ gives $\mathsf{id}^{(i)}, \vec{m}^{(i)}$ to the challenger, who checks that $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m}^{(i)} \mid \sigma)$. If it is, it sends μ to the adversary.

Winning Condition: The adversary $\mathcal{A}$ outputs a macaroon $\mu^* = (\widetilde{\mathsf{id}}^* \mid m_1^*, \ldots, m_n^* \mid \sigma^*)$. He wins if $\mathsf{Verify}(k, \mu^*) = 1$ and μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query. Such a μ^* we call a valid macaroon forgery. In more detail, it must be the case that, $\widetilde{\mathsf{id}}^* = \mathsf{id}^* || \mathsf{pk}_{\mathcal{E}} || (\mathsf{vk}_1^*, c_1^*) || \ldots || (\mathsf{vk}_n^*, c_n^*)$ and $\mathsf{Verify}(\mathsf{vk}_{\mathcal{S}}, \mathsf{id} || \mathsf{pk}_{\mathcal{E}}, \sigma_0) = 1$ and $\mathsf{Verify}(\mathsf{vk}_i^*, \mathsf{vk}_i^* || c_i^* || m_i^*, \sigma_i^*) = 1$ for $i = 1, \ldots, n$, where: $\sigma_i^* := \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, c_{i+1}^*)$ for $i = 0, \ldots, n-1$ and $\sigma_n^* := \sigma^*$.

Game 1: In Game 1, we change the winning condition so that $\mathcal{A}$ wins only if $\mathsf{id}^* = \mathsf{id}^{(i)}$ for some $\mathsf{id}^{(i)}$ given by $\mathcal{A}$ in an issuing query. This guarantees that μ^* is in $\mathcal{T}$ or is an extension of some macaroon in $\mathcal{T}$.

We argue that $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the security of the signature scheme $\mathcal{S}$ under keys $(\mathsf{vk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}})$. Let ε be the probability that $\mathcal{A}$ outputs a valid macaroon forgery and $\mathsf{id}^* \neq \mathsf{id}^{(i)}$ for all $\mathsf{id}^{(i)}$ given by $\mathcal{A}$ in an issuing query. Then $|p_0 - p_1| = \varepsilon$ since the games are equivalent otherwise. We argue that $\varepsilon = \mathsf{negl}(\kappa)$ by the security of $\mathcal{S}$. The reduction receives the verification key from the signature challenger and uses it as $\mathsf{vk}_{\mathcal{S}}$. It computes $(\mathsf{pk}_{\mathcal{E}}, \mathsf{sk}_{\mathcal{E}}) \leftarrow \mathsf{KeyGen}_{\mathcal{E}}(1^\kappa)$ on its own. When $\mathcal{A}$ asks for an issuing query $\mathsf{id}^{(i)}$, the reduction answers it by querying the signature oracle on $\mathsf{id}^{(i)} || \mathsf{pk}_{\mathcal{E}}$. The reduction answers extend and macaroon queries by computing everything as specified. When $\mathcal{A}$ outputs a forgery $\mu^* := (\widetilde{\mathsf{id}}^* \mid m_1^*, \ldots, m_n^* \mid \sigma^*)$, it parses $\widetilde{\mathsf{id}}^* = \mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}^* || (\mathsf{vk}_1^*, c_1^*) || \ldots$, decrypts $\sigma := \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, c_1^*)$ and submits σ as a signature forgery for message $\mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}$. We argue that the probability that the reduction outputs a valid forgery is at least ε. The reduction's forgery is valid only if $\mathcal{A}$ outputs a macaroon μ^* such that $\sigma := \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, c_1^*)$ is a valid signature for $\mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}$, and $\mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}$ was not queried to the signing oracle. But note that μ^* is valid only if σ is a valid signature for $\mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}$ since this is part of the macaroon's verification. Furthermore, if $\mathsf{id}^* \neq \mathsf{id}^{(i)}$ for all issuing queries, then $\mathsf{id}^* || \mathsf{pk}_{\mathcal{E}}$ is never queried to the signing oracle. Therefore, by the security of $\mathcal{S}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$.

Game 2 : Let q_{I} be an upper bound on the number of issuing queries that $\mathcal{A}$ makes, and let q_{E} be an upper bound on the number of extension queries that it makes. In Game 2, the challenger picks a uniformly random index $t \in \{1, \ldots, q_{\mathsf{I}} + q_{\mathsf{E}}\}$. This number identifies a unique macaroon $\mu_{\mathsf{ch}} := (\widetilde{\mathsf{id}}_{\mathsf{ch}} \mid \vec{m}_{\mathsf{ch}} \mid \sigma_{\mathsf{ch}}) \in \mathcal{T}$: if $t \leq q_{\mathsf{I}}$, then t corresponds to the node created in the tth issue query. If $t > q_{\mathsf{I}}$, then t corresponds to the node created in the $(t - q_{\mathsf{I}})$th extension query. We change the winning condition so that $\mathcal{A}$ only wins if $\widetilde{\mathsf{id}}_{\mathsf{ch}} = \mathsf{id}^* || \ldots$ and μ_{ch} is the last macaroon in $\mathcal{T}$ whose caveat list, $\vec{m}_{\mathsf{ch}}$, is a prefix of $\vec{m}^*$. In other words, $\mathcal{A}$ only wins if $\vec{m}_{\mathsf{ch}}$ is a prefix of $\vec{m}^*$ and there does not exist another macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma) \in \mathcal{T}$ such that $\widetilde{\mathsf{id}} = \mathsf{id}^* || \ldots$ and $\vec{m}$ is a longer prefix of $\vec{m}^*$.

We argue that $p_2 = p_1/(q_{\mathsf{E}} + q_{\mathsf{I}})$. If $\mathcal{A}$ outputs a valid forgery $\mu^* = (\mathsf{id}^* || \ldots \mid \vec{m}^* \mid \sigma^*)$, then there is a unique macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma) \in \mathcal{T}$ such that $\widetilde{\mathsf{id}} = \mathsf{id}^* || \ldots$ and $\vec{m}$ is the longest prefix of $\vec{m}^*$ in any macaroon in $\mathcal{T}$. Since t was sampled uniformly at random, the probability that $\mu_{\mathsf{ch}} = \mu$ is $1/(q_{\mathsf{E}} + q_{\mathsf{I}})$.

Game $3.i$ **for** $i \in 1, \ldots, q_{\mathsf{E}}$**:** Let q_{E} and μ_{ch} be as in Game 2. In Game $3.i$ we change how the challenger answers the ith extension query that extends μ_{ch}. Since q_{E} is an upper bound on the number of extension queries, in particular it is an upper bound on the number of extensions of μ_{ch}. On the ith extension to μ_{ch}, instead of encrypting σ_{ch}, the challenger encrypts 0.

Extend Queries: Upon receiving $\widetilde{\mathsf{id}}_{\mathsf{ch}}, \vec{m}_{\mathsf{ch}}, m^{(i)}$, the challenger adds $\mu^{(i)}$ to $\mathcal{T}$ as a child of μ_{ch} with edge $m^{(i)}$, where:

$$(\mathsf{vk}^{(i)}, \mathsf{sk}^{(i)}) \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^\kappa) \quad , \quad c^{(i)} \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, 0) \quad , \quad \widetilde{\mathsf{id}}^{(i)} := \widetilde{\mathsf{id}}_{\mathsf{ch}} || (\mathsf{vk}^{(i)}, c^{(i)})$$

$$\sigma^{(i)} \leftarrow \mathsf{Sign}(\mathsf{sk}^{(i)}, \mathsf{vk}^{(i)}||c^{(i)}||m^{(i)}) \quad , \quad \mu^{(i)} := (\widetilde{\mathsf{id}}^{(i)} \mid \vec{m}_{\mathsf{ch}}, m^{(i)} \mid \sigma^{(i)})$$

For simplicity let Game 3.0 be Game 2. We argue that $\left|p_{3.(i-1)} - p_{3.i}\right| = \mathsf{negl}(\kappa)$ for all $i \in 1, \ldots, q_{\mathsf{E}}$ by the CPA-security of $\mathcal{E}$. The reduction receives pk from the CPA-challenger and uses it as $\mathsf{pk}_{\mathcal{E}}$. It samples $(\mathsf{vk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}) \leftarrow \mathsf{KeyGen}_{\mathcal{S}}(1^{\kappa})$ on its own. It answers issue queries using $\mathsf{sk}_{\mathcal{S}}$ to sign. It answers the jth extension query as follows: if $j < i$, it lets $c^{(j)} \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, \sigma)$. For the ith query, it sends $m_0 = \sigma, m_1 = 0$ to the CPA challenger, and receives a challenge ciphertext $c^* \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, m_b)$. It lets $c^{(i)} = c^*$. Finally, if $j > i$, it lets $c^{(j)} \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}, 0)$. The challenger answers macaroon queries as specified. If $b = 0$, the reduction correctly simultes Game $3.(i-1)$, whereas if $b = 1$, the reduction correctly simulates Game $3.i$. Therefore, by the CPA-security of $\mathcal{E}$, we must have $\left|p_{3.(i-1)} - p_{3.i}\right| = \mathsf{negl}(\kappa)$.

$\mathcal{A}$ **Can't Win:** We argue that $p_3 = \mathsf{negl}(\kappa)$ by the security of signature scheme $\mathcal{S}$. Let $\widetilde{\mathsf{id}}^* = \mathsf{id}^*||\mathsf{pk}_{\mathcal{E}}||(\mathsf{vk}_1, c_1)||\ldots||(\mathsf{vk}_n, c_n)$, and let j be such that $\mathsf{id}_{\mathsf{ch}} = \mathsf{id}^*||\mathsf{pk}_{\mathcal{E}}||(\mathsf{vk}_1, c_1)||\ldots||(\mathsf{vk}_j, c_j)$, or let $j = 0$ if $\mathsf{id}_{\mathsf{ch}} = \mathsf{id}^*||\mathsf{pk}_{\mathcal{E}}$ (in this case, define $\mathsf{vk}_0 = \mathsf{vk}_{\mathcal{S}}$). We know such a j exists by the definition of μ_{ch}. In the reduction, the challenger will receive vk from the signing challenger, and use $\mathsf{vk}_j = \mathsf{vk}$. It computes $(\mathsf{pk}_{\mathcal{E}}, \mathsf{sk}_{\mathcal{E}}) \leftarrow \mathsf{KeyGen}_{\mathcal{E}}(1^{\kappa})$ on its own. When $\mathcal{A}$ asks for an issuing query or extension query that does not involve signing with sk_j (the corresponding secret key to vk_j), the challenger answers them accordingly. If a query involves signing a message with sk_j, then the challenger uses the signing oracle to compute the signature. However, the challenger never computes σ_{ch}. This means that the reduction never queries the signing oracle on $\mathsf{id}||\mathsf{pk}_{\mathcal{E}}$ if $j = 0$ or on $\mathsf{vk}_j||c_j||m_j$ if $j > 0$. This is possible since all extensions of μ_{ch} encrypt 0 instead of σ_{ch}, and if the adversary wins the game, then σ_{ch} is never revealed in a macaroon query. When $\mathcal{A}$ outputs its forgery, the reduction computes $\sigma_j = \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, c_{j+1})$, and outputs σ_j as its forgery. If $\mathcal{A}$ wins, then σ_j must be a valid signature under vk_j for $\mathsf{id}||\mathsf{pk}_{\mathcal{E}}$ if $j = 0$ or for $\mathsf{vk}_j||c_j||m_j$ if $j > 0$ since this check is part of the verification algorithm. Since the signing oracle was never queried by the reduction on that message, σ_j is also a valid signature forgery. Therefore, by the security of $\mathcal{S}$, we must have $p_3 = \mathsf{negl}(\kappa)$.

We wish to prove that $p_0 = \mathsf{negl}(\kappa)$. We know that $|p_0 - p_1| = \mathsf{negl}(\kappa)$, $p_1 = (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot p_2$, $|p_2 - p_3| = \mathsf{negl}(\kappa)$, and $p_3 = \mathsf{negl}(\kappa)$. Putting it all together, we have:

$$|p_0 - p_1| + (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot |p_2 - p_3| \geq (p_0 - p_1) + (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot (p_2 - p_3) = p_0 - (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot p_3$$

Since q_{I} and q_{E} are number of queries made by $\mathcal{A}$ and $\mathcal{A}$ runs in polynomial time, we must have that $(q_{\mathsf{I}} + q_{\mathsf{E}}) = \mathsf{poly}(\kappa)$. We can therefore conclude that

$$p_0 \leq |p_0 - p_1| + (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot |p_2 - p_3| + (q_{\mathsf{I}} + q_{\mathsf{E}}) \cdot p_5 = \mathsf{negl}(\kappa),$$

as required.

□

C Proof of Theorem 3.7: Security of PK Macaroons

Proof. We use a series-of-games argument to show the security of the macaroon discharge protocol construction. In all games, the setup phase is the same: the challenger runs $(\mathsf{params}, \mathsf{sk}_I, \mathsf{sk}_D) \leftarrow \mathsf{Setup}(1^{\kappa})$ and gives params to $\mathcal{A}$. We let p_i be the probability that $\mathcal{A}$ wins in Game i.

Game 0: Let Game 0 be the original game from the definition of security of public-key macaroons (analogous to Definition 4.2).

Game 1 : Let q_{I} be an upper bound on the number of issuing queries that $\mathcal{A}$ makes, and let q_{E} be an upper bound on the number of extension queries that it makes. In Game 1, the challenger picks a uniformly random index $t \in \{1, \ldots, q_{\mathsf{I}} + q_{\mathsf{E}}\}$. This number identifies a unique macaroon $\mu_{\mathsf{ch}} := (\widetilde{\mathsf{id}}_{\mathsf{ch}} \mid \vec{m}_{\mathsf{ch}} \mid \sigma_{\mathsf{ch}}) \in \mathcal{T}$: if $t \leq q_{\mathsf{I}}$, then t corresponds to the node created in the tth issue query. If $t > q_{\mathsf{I}}$, then t corresponds to the node created in the $(t - q_{\mathsf{I}})$th extension query. We change the winning condition so that in addition, $\mathcal{A}$ only wins if $\widetilde{\mathsf{id}}_{\mathsf{ch}} = \mathsf{id}||\mathsf{vk}_1, \ldots, \mathsf{vk}_i$ is the longest prefix in $\mathcal{T}$ of $\widetilde{\mathsf{id}}^*$.

This means $\widetilde{\mathsf{id}}^* = \mathsf{id}||\mathsf{vk}_1||\ldots,\mathsf{vk}_i,\mathsf{vk}_{i+1}^*,\ldots\mathsf{vk}_n^*,\mathsf{vk}_{n+1}^*$, and there does not exist another macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma) \in \mathcal{T}$ such that $\widetilde{\mathsf{id}}$ is a longer prefix of $\widetilde{\mathsf{id}}^*$.

We argue that $p_1 = p_0/(q_\mathsf{E} + q_\mathsf{I})$. If $\mathcal{A}$ outputs a valid forgery $\mu^* = (\mathsf{id}^*||\ldots \mid \vec{m}^* \mid \sigma^*)$, then there is a unique macaroon $\mu := (\widetilde{\mathsf{id}} \mid \vec{m} \mid \sigma) \in \mathcal{T}$ such that $\widetilde{\mathsf{id}}$ the longest prefix of $\widetilde{\mathsf{id}}^*$ in any macaroon in $\mathcal{T}$. Since t was sampled uniformly at random, the probability that $\mu_\mathsf{ch} = \mu$ is $1/(q_\mathsf{E} + q_\mathsf{I})$.

Game 2: We change the winning condition so that $\mathcal{A}$ only wins if $\widetilde{\mathsf{id}}^* = \widetilde{\mathsf{id}}_\mathsf{ch}$. Let ε be the probability that $i < n$. Then $|p_1 - p_2| = \varepsilon$ since the games are equivalent otherwise. We argue that $\varepsilon = \mathsf{negl}(\kappa)$ by the security of $\mathcal{S}$ under verification key vk_i. The reduction receives the verification key from the signature challenger and uses it as vk_i. The reduction answers issue, extension, and macaroon queries as specified, except when extending μ_ch. In this case, it samples a key pair $\mathsf{vk}^{(j)}, \mathsf{sk}^{(j)}$ as usual, and uses the signing oracle to compute the signature $\nu^{(j)}$ on $\mathsf{vk}^{(j)}$. When the adversary outputs a macaroon forgery $\mu^* = (\widetilde{\mathsf{id}}^* \mid \vec{m}^* \mid \sigma^*)$, it parses $\widetilde{\mathsf{id}}^* = \mathsf{id}||\mathsf{vk}_1||\ldots,\mathsf{vk}_i,\mathsf{vk}_{i+1}^*,\ldots\mathsf{vk}_n^*$ and $\sigma = (\nu_0^*,\tau_0^*)||\ldots(\nu_i^*,\tau_i^*)||\ldots$ and outputs ν_i^* as a signature forgery of vk_{i+1}^*. If μ^* is a valid macaroon forgery, then ν_i^* is a valid signature since the macaroon verification checks this. Moreover, since μ_ch is the macaroon with the largest prefix, then $\mathsf{vk}_{i+1}^* \neq \mathsf{vk}^{(j)}$ for all j and therefore vk_{i+1}^* was never queried to the signature oracle. We thus conclude that $\varepsilon = \mathsf{negl}(\kappa)$.

Game 3: Let $\mu_\mathsf{ch} = (\mathsf{id}||\ldots \mid \vec{m} \mid \sigma)$. In Game 3, we change the winning condition so that when $\mathcal{A}$ submits a forgery $\mu^* := (\widetilde{\mathsf{id}}^* \mid \vec{m}^* \mid \sigma^*)$, it wins only if $\widetilde{\mathsf{id}}^* = \mathsf{id}^*||\ldots$ and $(\mathsf{id}^*,\vec{m}^*) = (\mathsf{id},\vec{m})$. In other words, the key identifier and the caveats of the adversary's forgery must match those of μ_ch.

Let i be the last index at which $(\mathsf{id}^*,\vec{m}^*)$ and $(\mathsf{id},\vec{m})$ are the same (if $\mathsf{id}^* \neq \mathsf{id}$ then $i = 0$). Let ε be the probability that $i < n$. Then $|p_1 - p_2| = \varepsilon$ since the games are equivalent otherwise. We argue that $\varepsilon = \mathsf{negl}(\kappa)$ by the security of the signature scheme $\mathcal{S}$ under key vk_{i+1}. The reduction answers issue, extend, and macaroon queries as usual, except when extending μ_ch. In this case, instead of sampling a new key pair, it queries the signature oracle. When $\mathcal{A}$ outputs a forgery $\mu^* := (\widetilde{\mathsf{id}}^* \mid m_1^*,\ldots,m_n^* \mid \sigma^*)$, the reduction parses $\sigma^* = (\nu_0^*,\tau_0^*)||\ldots(\nu_{i+1}^*,\tau_{i+1}^*)||\ldots$, and outputs τ_{i+1}^* as a signature forgery for message m_{i+1}^*.

The reduction's forgery is valid only if $\mathcal{A}$ outputs a macaroon μ^* such that τ_{i+1}^* is a valid signature for m_{i+1}^*, and m_{i+1}^* was not queried to the signing oracle. Note that if μ^* is valid then the first condition is satisfied since this is part of the macaroon's verification. Furthermore, if $m_{i+1}^* \neq m_{i+1}$ then m_{i+1}^* is never queried to the signing oracle. Therefore, by the security of $\mathcal{S}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$.

$\mathcal{A}$ Can't Win: We argue that $\mathcal{A}$ can't win by the security of $\mathcal{S}$ under verification key vk_{n+1}. The reduction uses the verification key from the signature challenger as vk_{n+1}, and answers issue, extend, and macaroon queries as usual, using the signing oracle when needed. When $\mathcal{A}$ outputs a forgery $\mu^* := (\widetilde{\mathsf{id}}^* \mid m_1^*,\ldots,m_n^* \mid \sigma^*)$, the reduction parses $\widetilde{\mathsf{id}}^* = \mathsf{id}^*||\ldots$ and $\sigma^* = (\nu_0^*,\tau_0^*)||\ldots||\sigma_\mathsf{final}$, and outputs σ_final as a signature forgery for message id^*.

The reduction's forgery is valid only if $\mathcal{A}$ outputs a macaroon μ^* such that σ_final is a valid signature for id^*, and id^* was not queried to the signing oracle. Note that if μ^* is valid then the first condition is satisfied since this is part of the macaroon's verification. Furthermore, the challenger never extends a macaroon with a caveat equal to its key identifier, so the reduction never queried id^* to the signing oracle. Therefore, by the security of $\mathcal{S}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$.

□

D Proof of Theorem 4.3: Security of Discharge Protocol, Construction 1

Proof. We use a series-of-games argument to show the security of the macaroon discharge protocol construction. In all games, the setup phase is the same: the challenger runs $(\mathsf{params},\mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{SetupDischarge}(1^\kappa)$ and $\mathsf{sk}_\mathsf{T} \leftarrow \mathsf{SetupToken}(1^\kappa)$ and gives params to $\mathcal{A}$. We let p_i be the probability that $\mathcal{A}$ wins in Game i.

Game 0: Let Game 0 be the original game from Definition 4.2. In this game, we have:

Token Queries: On the ith query, $\mathcal{A}$ receives $\alpha_{\mathsf{S}}^{(i)} := (c_1^{(i)}, \sigma^{(i)})$, and $\alpha_{\mathsf{D}}^{(i)} := c_2^{(i)}$, where

$$r^{(i)} \leftarrow \mathsf{KeyGen}_{\mathcal{M}}(1^\kappa) \quad , \quad c_1^{(i)} \leftarrow \mathsf{Enc}_1(\mathsf{sk}_1, r^{(i)}) \quad , \quad \sigma^{(i)} \leftarrow \mathsf{Mac}(k, c_1^{(i)}), \quad , \quad c_2^{(i)} \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, r^{(i)})$$

Discharge Queries: On the ith query $\alpha_{\mathsf{D}}^{(i)*} = c_2^{(i)*}$, $\mathcal{A}$ receives $\mu^{(i)}$, where

$$r^{(i)} \leftarrow \mathsf{Dec}_2(\mathsf{sk}_2, c_2^{(i)*}) \quad , \quad \mu^{(i)} \leftarrow \mathsf{Issue}_{\mathcal{M}}(r^{(i)}, c_2^{(i)*})$$

Winning Condition: $\mathcal{A}$ *wins* if it produces a token $\alpha_{\mathsf{S}}^* = (c_1^*, \sigma^*)$ and a macaroon μ^* such that $\mathsf{Verify}(r^*, \mu^*) = 1$ where $r^* = \mathsf{Dec}_1(\mathsf{sk}_1, c_1^*)$, and μ^* is not an extension of any macaroon μ such that $(\alpha_{\mathsf{D}}^*, \mu) \in \mathcal{Q}_{\mathsf{D}}$, where α_{D}^* corresponds to α_{S}^* in $\mathcal{Q}_{\mathsf{T}}$. In other words, $\mathcal{A}$ wins if it outputs a valid macaroon that is not an extension of a macaroon that was the output of a discharge query with a discharge token corresponding to α_{S}^*.

Game 1: Let q_{T} be an upper bound on the number of issuing queries that $\mathcal{A}$ makes. In Game 1, the challenger remembers each $r^{(i)}$ that it samples in an issuing query, and we change the winning condition so that $\mathcal{A}$ only wins if μ^* is a valid macaroon under key $r^{(i)}$ for some $i \in \{1, \ldots, q_{\mathsf{T}}\}$. In other words, $\mathcal{A}$ only wins if it produces a valid macaroon under a key that the challenger sampled in a token query.

Winning Condition: $\mathcal{A}$ *wins* if it outputs $(\alpha_{\mathsf{S}}^*, \mu^*)$ such that $\mathsf{Verify}(r^{(i)}, \mu^*) = 1$ for some $i \in \{1, \ldots, q_{\mathsf{T}}\}$ and μ^* is not an extension of any macaroon μ such that $(\alpha_{\mathsf{D}}^*, \mu) \in \mathcal{Q}_{\mathsf{D}}$, where α_{D}^* corresponds to α_{S}^* in $\mathcal{Q}_{\mathsf{T}}$.

We argue that $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the security of the MAC $\mathcal{T}$. Consider the probability ε that $\mathcal{A}$ outputs a valid forgery that contains $c_1^* \neq c_1^{(i)}$ for all $i \in \{1, \ldots, q_{\mathsf{T}}\}$, or in other words, that c_1^* is a ciphertext that was not given to $\mathcal{A}$ as a response to an token query. It is easy to see that $|p_0 - p_1| = \varepsilon$ since by correctness of decryption of $\mathcal{E}_1$, the games are equivalent if $c_1^* = c_1^{(i)}$ for some $i \in \{1, \ldots, q_{\mathsf{T}}\}$. By the security of the MAC $\mathcal{T}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$. The reduction samples $\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{sk}_2$, answers token queries by computing $c_1^{(i)}$ and $c_2^{(i)}$ on its own and using the MAC tagging oracle to obtain $\sigma^{(i)}$. It answers discharge queries using sk_2, and given $\mathcal{A}$'s forgery (c_1^*, σ^*, μ^*), it outputs c_1^*, σ^*. Since this simulates $\mathcal{A}$'s view perfectly, the reduction produces a tag on a new message with probability ε.

Game 2: In Game 2, the challenger chooses an index $i_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{T}}\}$ uniformly at random. We change the winning condition so that $\mathcal{A}$ only wins if μ^* is a valid macaroon under key $r^{(i_{\mathsf{ch}})}$.

Winning Condition: The challenger picks $i_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{T}}\}$ uniformly at random. $\mathcal{A}$ *wins* if it outputs $(\alpha_{\mathsf{S}}^*, \mu^*)$ such that $\mathsf{Verify}(r^{(i_{\mathsf{ch}})}, \mu^*) = 1$ and μ^* is not an extension of any macaroon μ such that $(\alpha_{\mathsf{S}}^*, \mu) \in \mathcal{Q}_{\mathsf{D}}$. In other words, $\mathcal{A}$ wins if it outputs a valid macaroon under $r^{i_{\mathsf{ch}}}$ that is not an extension of a macaroon that was the output of a discharge query with token α_{S}^*.

Since i_{ch} is chosen uniformly at random, we have that $p_2 = p_1/q_{\mathsf{T}}$.

Game 3: In Game 3, we change how the i_{ch}th token query is answered by the challenger. In particular, $c_1^{(i_{\mathsf{ch}})}$ will now be an encryption of 0 instead of an encryption of $r^{(i_{\mathsf{ch}})}$.

i_{ch}th Token Query: On the i_{ch}th query, $\mathcal{A}$ receives $\alpha_{\mathsf{S}}^{(i_{\mathsf{ch}})} := (c_1^{(i_{\mathsf{ch}})}, \sigma^{(i_{\mathsf{ch}})})$, and $\alpha_{\mathsf{D}}^{(i_{\mathsf{ch}})} := c_2^{(i_{\mathsf{ch}})}$, where

$$r^{(i_{\mathsf{ch}})} \leftarrow \mathsf{KeyGen}_{\mathcal{M}}(1^\kappa) \quad , \quad c_1^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Enc}_1(\mathsf{sk}_1, 0) \quad , \quad \sigma \leftarrow \mathsf{Mac}(k, c_1^{(i_{\mathsf{ch}})}), \quad , \quad c_2^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, r^{(i_{\mathsf{ch}})})$$

We argue that $|p_2 - p_3| = \mathsf{negl}(\kappa)$ by the CPA-security of $\mathcal{E}_1$. The reduction samples $\mathsf{pk}_2, \mathsf{sk}_2, k$ and chooses a random $i_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{T}}\}$. It answers the ith token query as follows: if $i \neq i_{\mathsf{ch}}$, it generates $r^{(i)}$ and obtains $c_1^{(i)}$ from the encryption oracle. It computes $\sigma^{(i)}, c_2^{(i)}$ on its own. On the i_{ch}th token query, the reduction gives $m_0 = r^{i_{\mathsf{ch}}}$ and $m_1 = 0$ to the CPA-challenger as the challenge plaintexts. It obtains a challenge ciphertext $\mathsf{Enc}_1(\mathsf{sk}_1, m_b)$ in return, which it uses as $c_1^{(i_{\mathsf{ch}})}$. It then computes $\sigma^{(i_{\mathsf{ch}})}, c_2^{(i_{\mathsf{ch}})}$ on its own. The reduction answers discharge queries as usual, decrypting with sk_2.

If $b = 0$, then the reduction perfectly simulates Game 2 and if $b = 1$, the reduction perfectly simulates Game 1. Therefore, we conclude that $|p_2 - p_3| = \mathsf{negl}(\kappa)$ by the CPA-security of $\mathcal{E}_1$.

Game 4: In Game 4, we change how we answer discharge queries. More specifically, the challenger checks if $c_2^{(i)*} = c_2^{(i_{\mathsf{ch}})}$, or in other words, if the query ciphertext is the same as the c_2 ciphertext given as a response to the i_{ch}th token query. If $c_2^{(i)*} \neq c_2^{(i_{\mathsf{ch}})}$, then the discharge query is answered as usual, but if $c_2^{(i)*} = c_2^{(i_{\mathsf{ch}})}$, then the challenger issues a macaroon under key $r^{(i_{\mathsf{ch}})}$.

Games 3 and 4 are identically distributed by the correctness of decryption of $\mathcal{E}_2$. Thus, $p_3 = p_4$.

Game 5: In Game 5, we again change how the i_{ch}th token query is answered by the challenger. In particular, $c_2^{(i_{\mathsf{ch}})}$ will now be an encryption of 0 instead of an encryption of $r^{(i_{\mathsf{ch}})}$.

i_{ch}th Token Query: On the i_{ch}th query, $\mathcal{A}$ receives $\alpha_{\mathsf{S}}^{(i_{\mathsf{ch}})} := (c_1^{(i_{\mathsf{ch}})}, \sigma^{(i_{\mathsf{ch}})})$, and $\alpha_{\mathsf{D}}^{(i_{\mathsf{ch}})} := c_2^{(i_{\mathsf{ch}})}$, where

$$r^{(i_{\mathsf{ch}})} \leftarrow \mathsf{KeyGen}_{\mathcal{M}}(1^\kappa) \quad , \quad c_1^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Enc}_1(\mathsf{sk}_1, 0) \quad , \quad \sigma \leftarrow \mathsf{Mac}(k, c_1^{(i_{\mathsf{ch}})}), \quad , \quad c_2^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, 0)$$

We argue that $|p_4 - p_5| = \mathsf{negl}(\kappa)$ by the CCA2-security of $\mathcal{E}_2$. The reduction obtains pk_2 from the CCA2-challenger, and samples sk_1 and k. To answer the ith token query, it computes $c_1^{(i)}, \sigma$ on its own. If $i \neq i_{\mathsf{ch}}$, it also computes $c_2^{(i)}$. On the other hand, if $i = i_{\mathsf{ch}}$, then it sends $m_0 = r^{(i_{\mathsf{ch}})}, m_1 = 0$ to the CCA-challenger as the challenge plaintexts. It obtains a challenge ciphertext $\mathsf{Enc}(\mathsf{pk}_2, m_b)$, which it uses as $c_2^{(i)}$. It answers discharge queries as follows: If $c_2^{(i)*} \neq c_2^{(i_{\mathsf{ch}})}$, then it gives $c_2^{(i)*}$ to the decryption oracle and uses the plaintext it receives back as the key to issue the macaroon. If $c_2^{(i)*} = c_2^{(i_{\mathsf{ch}})}$, then the challenger simply issues the macaroon under key $r^{(i_{\mathsf{ch}})}$. Note that this guarantees that we never query the decryption oracle with the challenge ciphertext.

If $b = 0$, then the reduction perfectly simulates Game 4 and if $b = 1$, the reduction perfectly simulates Game 5. Therefore, we conclude that $|p_4 - p_5| = \mathsf{negl}(\kappa)$ by the CCA2-security of $\mathcal{E}_2$.

$\mathcal{A}$ Can't Win: We argue that $p_5 = \mathsf{negl}(\kappa)$ by the security of the macaroon scheme $\mathcal{M}$. The reduction samples $\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_2, k$. It answers token queries by computing $c_1^{(i)}, c_2^{(i)}, \sigma^{(i)}$ on its own. Note that this does not require encrypting $r^{(i_{\mathsf{ch}})}$. It answers discharge queries as follows: If $c_2^{(i)*} \neq c_2^{(i_{\mathsf{ch}})}$, it answers the query as usual: decrypts $c_2^{(i)*}$ and creates a macaroon using the plaintext. But if $c_2^{(i)*} = c_2^{(i_{\mathsf{ch}})}$, it calls the issuing oracle with $c_2^{(i_{\mathsf{ch}})}$ and the asks for it with a macaroon query. Finally, when $\mathcal{A}$ outputs $\alpha_{\mathsf{S}}^*, \mu^*$, it outputs μ^* as its forgery. If $\mathcal{A}$ wins, then μ^* is a valid macaroon under $r^{(i_{\mathsf{ch}})}$ and is not an extension of a macaroon μ such that $(\alpha_{\mathsf{S}}^*, \mu) \in \mathcal{Q}_{\mathsf{D}}$. In particular, this means that it is not an extension of a macaroon given to the reduction as a macaroon query. Therefore, the probability that the reduction gives a valid forgery is exactly p_5 since it simulated Game 5 perfectly for $\mathcal{A}$. By the security of the macaroon scheme $\mathcal{M}$, we conclude $p_5 = \mathsf{negl}(\kappa)$.

We wish to prove that $p_0 = \mathsf{negl}(\kappa)$. We know that $|p_0 - p_1| = \mathsf{negl}(\kappa)$, $p_1 = q_{\mathsf{T}} \cdot p_2$, $|p_2 - p_3| = \mathsf{negl}(\kappa)$, $p_3 = p_4$, $|p_4 - p_5| = \mathsf{negl}(\kappa)$, and $p_5 = \mathsf{negl}(\kappa)$. Putting it all together, we have:

$$|p_0 - p_1| + q_{\mathsf{T}} \cdot |p_2 - p_3| + q_{\mathsf{T}} \cdot |p_4 - p_5| \geq (p_0 - p_1) + q_{\mathsf{T}} \cdot (p_2 - p_3) + q_{\mathsf{T}} \cdot (p_4 - p_5) = p_0 - q_{\mathsf{T}} \cdot p_5$$

Since q_{T} is the number of queries that $\mathcal{A}$ makes and $\mathcal{A}$ runs in polynomial time, we must have that $q_{\mathsf{T}} = \mathsf{poly}(\kappa)$. We can therefore conclude that

$$p_0 \leq |p_0 - p_1| + q_{\mathsf{T}} \cdot |p_2 - p_3| + q_{\mathsf{T}} \cdot |p_4 - p_5| + q_{\mathsf{T}} \cdot p_5 = \mathsf{negl}(\kappa),$$

as required. □

E Proof of Theorem 4.4: Security of Discharge Protocol, Construction 2

Proof. We use a series-of-games argument to show the security of the macaroon discharge protocol construction. In all games, the setup phase is the same: the challenger runs $(\mathsf{params}, \mathsf{sk}_{\mathsf{D}}) \leftarrow \mathsf{SetupDischarge}(1^\kappa)$ and $\mathsf{sk}_{\mathsf{T}} \leftarrow \mathsf{SetupToken}(1^\kappa)$ and gives params to $\mathcal{A}$. We let p_i be the probability that $\mathcal{A}$ wins in Game i.

Game 0: Let Game 0 be the original game from Definition 4.2. In this game, we have:

Token Queries: On the ith query, $\mathcal{A}$ receives $\alpha_{\mathsf{S}}^{(i)} := (r^{(i)}, \sigma^{(i)})$, and $\alpha_{\mathsf{D}}^{(i)} := r^{(i)}$, where

$$r^{(i)} \leftarrow \{0,1\}^{\kappa} \quad , \quad \sigma^{(i)} \leftarrow \mathsf{Mac}(k, r^{(i)})$$

Discharge Queries: On the ith query $\alpha_{\mathsf{D}}^{(i)*} = r^{(i)*}$, $\mathcal{A}$ receives $\mu^{(i)}$, where

$$\mu^{(i)} \leftarrow \mathsf{Issue}_{\mathcal{M}}(\mathsf{sk}_{\mathsf{D}}, r^{(i)*})$$

Winning Condition: $\mathcal{A}$ *wins* if it produces a token $\alpha_{\mathsf{S}}^* = (r^*, \sigma^*)$ and a macaroon μ^* such that $\mathsf{Verify}(\mathsf{vk}, \mu^*) = 1$, μ^*'s key identifier is r^*, and μ^* is not an extension of any macaroon μ that was given to $\mathcal{A}$ as a discharge query with token r^*.

Game 1: Let q_{T} be an upper bound on the number of issuing queries that $\mathcal{A}$ makes. In Game 1, the challenger remembers each $r^{(i)}$ that it samples in a token query, and we change the winning condition so that $\mathcal{A}$ only wins if $r^* = r^{(i)}$ for some $i \in \{1, \ldots, q_{\mathsf{T}}\}$. In other words, $\mathcal{A}$ only wins if it produces a valid discharge macaroon for a token it received from a token query.

Winning Condition: $\mathcal{A}$ *wins* if it outputs $(\alpha_{\mathsf{S}}^*, \mu^*)$ such that $\mathsf{Verify}(\mathsf{vk}, \mu^*) = 1$, μ^* key identifier is r^*, $r^* = r^{(i)}$ for some $i \in \{1, \ldots, q_{\mathsf{T}}\}$, and μ^* is not an extension of any macaroon μ that was given to $\mathcal{A}$ as a discharge query with token r^*.

We argue that $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the security of the MAC $\mathcal{T}$. Consider the probability ε that $\mathcal{A}$ outputs a valid forgery that contains $r^* \neq r^{(i)}$ for all $i \in \{1, \ldots, q_{\mathsf{T}}\}$, or in other words, that r^* was not given to $\mathcal{A}$ as a response to an token query. It is easy to see that $|p_0 - p_1| = \varepsilon$ since the games are equivalent otherwise. By the security of the MAC $\mathcal{T}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$. The reduction samples $(\mathsf{vk}, \mathsf{sk})$ on its own, answers token queries by computing sampling random $r^{(i)}$ and using the MAC tagging oracle to compute $\sigma^{(i)}$. It answers discharge queries using sk. Given $\mathcal{A}$'s forgery (r^*, σ^*, μ^*), it outputs σ^* and its forgery on message r^*. Since this simulates $\mathcal{A}$'s view perfectly, the reduction produces a tag on a new message with probability ε.

Game 2: In Game 2, the challenger chooses an index $i_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{T}}\}$ uniformly at random. We change the winning condition so that $\mathcal{A}$ only wins if $r^* = r^{(i_{\mathsf{ch}})}$.

Winning Condition: $\mathcal{A}$ *wins* if it outputs $(\alpha_{\mathsf{S}}^*, \mu^*)$ such that $\mathsf{Verify}(\mathsf{vk}, \mu^*) = 1$, μ^* key identifier is r^*, $r^* = r^{(i_{\mathsf{ch}})}$, and μ^* is not an extension of any macaroon μ that was given to $\mathcal{A}$ as a discharge query with token r^*.

Since i_{ch} is chosen uniformly at random, we have that $p_2 = p_1/q_{\mathsf{T}}$.

$\mathcal{A}$ Can't Win: We argue that $p_2 = \mathsf{negl}(\kappa)$ by the security of the macaroon scheme $\mathcal{M}$. The reduction samples k and receives vk from the macaroon scheme challenger. It answers token queries by computing $r^{(i)}, \sigma^{(i)}$ on its own. It answers discharge queries by calling its issuing oracle with $r^{(i)*}$ and and immediately asking for the resulting macaroon with a macaroon query. Finally, when $\mathcal{A}$ outputs $\alpha_{\mathsf{S}}^*, \mu^*$, it outputs μ^* as its forgery. If $\mathcal{A}$ wins, then μ^* is a valid macaroon with $r^{(i_{\mathsf{ch}})}$ as its key identifier, and it is not an extension of a macaroon μ given to $\mathcal{A}$ as a discharge query for token $r^{(i_{\mathsf{ch}})}$. In particular, this means that it is not an extension of a macaroon given to the reduction as a macaroon query. Therefore, the probability that the reduction gives a valid forgery is exactly p_2 since it simulated Game 2 perfectly for $\mathcal{A}$. By the security of the macaroon scheme $\mathcal{M}$, we conclude $p_2 = \mathsf{negl}(\kappa)$.

We wish to prove that $p_0 = \mathsf{negl}(\kappa)$. We know that $|p_0 - p_1| = \mathsf{negl}(\kappa)$, $p_1 = q_{\mathsf{T}} \cdot p_2$, and $p_2 = \mathsf{negl}(\kappa)$. Putting it all together, we have:

$$|p_0 - p_1| + q_{\mathsf{T}} \cdot p_2 \geq (p_0 - p_1) + q_{\mathsf{T}} \cdot p_2 = p_0$$

Since q_{T} is the number of queries that $\mathcal{A}$ makes and $\mathcal{A}$ runs in polynomial time, we must have that $q_{\mathsf{T}} = \mathsf{poly}(\kappa)$. We can therefore conclude that

$$p_0 \leq |p_0 - p_1| + q_{\mathsf{T}} \cdot p_2 = \mathsf{negl}(\kappa),$$

as required. □

F Proof of Theorem 5.3: Security of our Macaroon-Tree Protocol

Proof. We use a series-of-games argument to show the security of the macaroon discharge protocol construction. In all games, the setup phase is the same: the challenger runs $(\mathsf{params}, \mathsf{sk}_\mathsf{I}, \mathsf{sk}_\mathsf{D}) \leftarrow \mathsf{Setup}(1^\kappa)$ and gives params to $\mathcal{A}$. We let p_i be the probability that $\mathcal{A}$ wins in Game i.

Game 0: Let Game 0 be the original game from Definition 5.2. In this game, we have:

Issue Queries: On the ith query, the challenger creates a macaroon $(\mu_0^{(i)}, \mathsf{ek}_0^{(i)})$ and adds it as a child of the root under an edge labeled with $c^{(i)}||\sigma^{(i)}$, where:

$$k^{(i)} \leftarrow \mathsf{KeyGen}_\mathcal{M}(1^\kappa) \quad , \quad c^{(i)} \leftarrow \mathsf{Enc}(\mathsf{sk}_1, k^{(i)}) \quad , \quad \sigma^{(i)} \leftarrow \mathsf{Mac}(k_\mathcal{T}, c^{(i)}) \quad , \quad \mathsf{id}^{(i)} := c^{(i)}||\sigma^{(i)}$$

$$(\mu_0^{(i)}, \mathsf{ek}_0^{(i)}) \leftarrow \mathsf{Issue}_\mathcal{M}(k^{(i)}, \mathsf{id}^{(i)})$$

Extend Queries: On the ith query, the adversary sends an identifier $\mathsf{id}^{(i)}$, a tuple of caveats $\vec{m}^{(i)}$, and a message $m^{(i)} \in \{0,1\}^* \cup \lambda$ to the challenger, where λ is the empty string. If $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a node (μ, ek), then the challenger computes $(\mu^{(i)}, \mathsf{ek}^{(i)}, \alpha_\mathsf{D}^{(i)}) \leftarrow \mathsf{Extend}(\mathsf{ek}, \mu, m^{(i)})$, and adds $(\mu^{(i)}, \mathsf{ek}^{(i)}, \alpha_\mathsf{D}^{(i)})$ to $\mathcal{T}$ as a child of (μ, ek) under an edge labeled with $m^{(i)}$. Here, $\mathsf{Extend}(\mathsf{ek}, \mu, m)$ is defined as follows: If $m = \lambda$, it outputs $\mathsf{Extend3rdParty}(\mathsf{ek}, \mu)$, otherwise it outputs $\mathsf{Extend}_\mathcal{M}(\mathsf{ek}, \mu, 3||c^{(i)}||\alpha_\mathsf{S}^{(i)}||\alpha_\mathsf{D}^{(i)})$ and $\alpha = \lambda$, where:

$$\mathsf{sk}_\mathsf{T}^{(i)} \leftarrow \mathsf{SetupToken}_\mathcal{D}(1^\kappa) \quad , \quad c^{(i)} \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, \mathsf{sk}_\mathsf{T}^{(i)}) \quad , \quad (\alpha_\mathsf{S}^{(i)}, \alpha_D^{(i)}) \leftarrow \mathsf{Token}_\mathcal{D}(\mathsf{sk}_\mathsf{T})$$

Discharge Queries: On the ith query, the adversary $\mathcal{A}$ sends a token $\alpha_\mathsf{D}^{(i)*}$ to the challenger, who runs $\mu^{(i)} \leftarrow \mathsf{Discharge}_\mathcal{D}(\mathsf{sk}_\mathsf{D}, \alpha_\mathsf{D}^{(i)*})$ and returns $\mu^{(i)}$ to $\mathcal{A}$. The challenger also adds $(\alpha_\mathsf{D}^{(i)*}, \mu^{(i)})$ to $\mathcal{Q}_\mathsf{D}$.

Macaroon Queries: On the ith query, the adversary sends an identifier $\mathsf{id}^{(i)}$ and a tuple of caveats $\vec{m}^{(i)}$ to the challenger. If $(\mathsf{id}^{(i)}, \vec{m}^{(i)})$ is a valid path in $\mathcal{T}$ leading to a node (μ, ek), then the challenger returns (μ, ek) to the adversary.

Winning condition: The adversary $\mathcal{A}$ outputs a macaroon μ^* and a set of discharges $M^* = \{\mu_j^*\}$. He wins if:

1. $\mathsf{Verify}(k_\mathcal{T}, c, \sigma) = 1$ and $\mathsf{Verify}_\mathcal{M}(k, \mu) = 1$ where $k := \mathsf{Dec}(\mathsf{sk}_1, \mathsf{id})$, and
2. For every third-party caveat $3||c_i||\alpha_\mathsf{S}^{(i)}||\alpha_\mathsf{D}^{(i)}$ in μ, there exists $\mu_j \in M$ with key identifier $\alpha_\mathsf{D}^{(i)}$ such that $\mathsf{Verify}_\mathcal{D}(\mathsf{sk}_\mathsf{T}^{(i)}, \alpha_\mathsf{S}^{(i)}, \mu_j) = 1$ where $\mathsf{sk}_\mathsf{T}^{(i)} := \mathsf{Dec}(\mathsf{sk}_2, c_i)$, and
3. Either
   (a) μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query, or
   (b) There exists a macaroon $m_j^* \in M^*$ corresponding to a third-party caveat $3||c_j||\alpha_\mathsf{S}^{(j)}||\alpha_\mathsf{D}^{(j)}$ in μ^* such that μ_j^* is not an extension of any macaroon μ such that $(\alpha_\mathsf{D}^{(j)}, \mu) \in \mathcal{Q}_\mathsf{D}$.

Game 1: Let q_I be an upper bound on the number of issuing queries that $\mathcal{A}$ makes. In Game 1, the challenger remembers each $k^{(i)}$ that it samples in an issuing query, and we change the winning condition so that $\mathcal{A}$ only wins if μ^* is a valid macaroon under key $k^{(i)}$ for some $i \in \{1, \ldots, q_\mathsf{I}\}$. In other words, $\mathcal{A}$ only wins if it produces a valid macaroon under a key that the challenger sampled in an issuing query.

Winning condition: The adversary $\mathcal{A}$ outputs a macaroon μ^* and a set of discharges $M^* = \{\mu_j^*\}$. He wins if:

1. $\mathsf{Verify}(k_\mathcal{T}, c, \sigma) = 1$ and $\mathsf{Verify}_\mathcal{M}(k^{(i)}, \mu) = 1$ for some $i \in \{1, \ldots, q_\mathsf{I}\}$, and
2. For every third-party caveat $3||c_i||\alpha_\mathsf{S}^{(i)}||\alpha_\mathsf{D}^{(i)}$ in μ, there exists $\mu_j \in M$ with key identifier $\alpha_\mathsf{D}^{(i)}$ such that $\mathsf{Verify}_\mathcal{D}(\mathsf{sk}_\mathsf{T}^{(i)}, \alpha_\mathsf{S}^{(i)}, \mu_j) = 1$ where $\mathsf{sk}_\mathsf{T}^{(i)} := \mathsf{Dec}(\mathsf{sk}_2, c_i)$, and
3. Either
   (a) μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query, or

(b) There exists a macaroon $m_j^* \in M^*$ corresponding to a third-party caveat $3||c_j||\alpha_{\mathsf{S}}^{(j)}||\alpha_{\mathsf{D}}^{(j)}$ in μ^* such that μ_j^* is not an extension of any macaroon μ such that $(\alpha_{\mathsf{D}}^{(j)}, \mu) \in \mathcal{Q}_{\mathsf{D}}$.

We argue that $|p_0 - p_1| = \mathsf{negl}(\kappa)$ by the security of the MAC $\mathcal{T}$. Consider the probability ε that $\mathcal{A}$ outputs a valid forgery μ^* with key identifier $\widetilde{\mathsf{id}}^* = c^*||\sigma^*||\ldots$ such that $c^* \neq c^{(i)}$ for all $i \in \{1, \ldots, q_{\mathsf{I}}\}$, or in other words, that c^* is a ciphertext that was not given to $\mathcal{A}$ as a response to an issue query. It is easy to see that $|p_0 - p_1| = \varepsilon$ since by correctness of decryption of $\mathcal{E}_1$, the games are equivalent if $c^* = c^{(i)}$ for some $i \in \{1, \ldots, q_{\mathsf{I}}\}$. By the security of the MAC $\mathcal{T}$, we must have $\varepsilon = \mathsf{negl}(\kappa)$. The reduction samples $\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{sk}_2$, answers token queries by sampling $k^{(i)}$ and computing $c^{(i)}$ on its own and using the MAC tagging oracle to obtain $\sigma^{(i)}$. It answers extend, macaroon, and discharge queries on its own. Given $\mathcal{A}$'s forgery μ^* with key identifier $\widetilde{\mathsf{id}}^* = c^*||\sigma^*||\ldots$, it outputs σ^* as a signature forgery on message c^*. Since this simulates $\mathcal{A}$'s view perfectly, the reduction produces a tag on a new message with probability ε.

Game 2: In Game 2, the challenger chooses an index $i_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{I}}\}$ uniformly at random. We change the winning condition so that $\mathcal{A}$ only wins if μ^* is a valid macaroon under key $r^{(i_{\mathsf{ch}})}$.

Winning condition: The adversary $\mathcal{A}$ outputs a macaroon μ^* and a set of discharges $M^* = \{\mu_j^*\}$. He wins if:

1. $\mathsf{Verify}(k_{\mathcal{T}}, c, \sigma) = 1$ and $\mathsf{Verify}_{\mathcal{M}}(k^{(i_{\mathsf{ch}})}, \mu) = 1$, and
2. For every third-party caveat $3||c_i||\alpha_{\mathsf{S}}^{(i)}||\alpha_{\mathsf{D}}^{(i)}$ in μ, there exists $\mu_j \in M$ with key identifier $\alpha_{\mathsf{D}}^{(i)}$ such that $\mathsf{Verify}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}}^{(i)}, \alpha_{\mathsf{S}}^{(i)}, \mu_j) = 1$ where $\mathsf{sk}_{\mathsf{T}}^{(i)} := \mathsf{Dec}(\mathsf{sk}_2, c_i)$, and
3. Either
   (a) μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query, or
   (b) There exists a macaroon $m_j^* \in M^*$ corresponding to a third-party caveat $3||c_j||\alpha_{\mathsf{S}}^{(j)}||\alpha_{\mathsf{D}}^{(j)}$ in μ^* such that μ_j^* is not an extension of any macaroon μ such that $(\alpha_{\mathsf{D}}^{(j)}, \mu) \in \mathcal{Q}_{\mathsf{D}}$.

Since i_{ch} is chosen uniformly at random, we have that $p_2 = p_1/q_{\mathsf{I}}$.

Game 3: In Game 3, we change how the challenger answers the i_{ch}th issuing query. Instead of encrypting $k^{(i_{\mathsf{ch}})}$, it encrypts 0.

Issue Queries: On the i_{ch}th query, the challenger creates a macaroon $(\mu_0^{(i_{\mathsf{ch}})}, \mathsf{ek}_0^{(i_{\mathsf{ch}})})$ and adds it as a child of the root under an edge labeled with $c^{(i_{\mathsf{ch}})}||\sigma^{(i_{\mathsf{ch}})}$, where:

$$k^{(i_{\mathsf{ch}})} \leftarrow \mathsf{KeyGen}_{\mathcal{M}}(1^\kappa) \quad , \quad c^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Enc}(\mathsf{sk}_1, 0) \quad , \quad \sigma^{(i_{\mathsf{ch}})} \leftarrow \mathsf{Mac}(k_{\mathcal{T}}, c^{(i_{\mathsf{ch}})}) \quad , \quad \mathsf{id}^{(i)} := c^{(i)}||\sigma^{(i)}$$

$$(\mu_0^{(i_{\mathsf{ch}})}, \mathsf{ek}_0^{(i_{\mathsf{ch}})}) \leftarrow \mathsf{Issue}_{\mathcal{M}}(k^{(i_{\mathsf{ch}})}, \mathsf{id}^{(i_{\mathsf{ch}})})$$

We argue that $|p_2 - p_3| = \mathsf{negl}(\kappa)$ by the CPA-security of $\mathcal{E}_1$. The reduction samples $\mathsf{pk}_2, \mathsf{sk}_2, k_{\mathcal{T}}$ on its own. For $i \neq i_{\mathsf{ch}}$, it answers the ith issuing query by sampling $k^{(i)}$ and asking its encryption oracle for $c^{(i)}$. It then computes $\sigma^{(i)}$ and $\mu_0^{(i)}, \mathsf{ek}_0^{(i)}$ on its own. On the i_{ch}th issuing query, it samples $k^{(i_{\mathsf{ch}})}$ and sends $m_0 := k^{(i_{\mathsf{ch}})}, m_1 := 0$ to the encryption challenger as the challenge plaintexs, and receives a challenge ciphertext $\mathsf{Enc}_1(\mathsf{sk}_1, m_b)$. It uses the challenge ciphertext as $c^{(i_{\mathsf{ch}})}$ and computes $\sigma^{(i)}$ on its own. The reduction answers extend, discharge, and macaroon queries on its own. Notice that if $b = 0$, the reduction perfectly simulates Game 2, whereas if $b = 1$, the reduction perfectly simulates Game 3. Therefore, by the CPA-security of $\mathcal{E}_1$ we must have $|p_2 - p_3| = \mathsf{negl}(\kappa)$.

Game 4: Let q_{E} be an upper on the number of extension queries made by $\mathcal{A}$ that add a third-party caveat. In Game 4, the challenger chooses an index $j_{\mathsf{ch}} \in \{1, \ldots, q_{\mathsf{E}}\}$ uniformly at random. We change the winning condition so that $\mathcal{A}$ only wins if either μ^* is not an extension of any macaroon μ revealed in a macaroon query, or it is such an extension, the j_{ch}th extension query added a third-party caveat $3||c_{j_{\mathsf{ch}}}||\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}||\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ caveat to μ, and there exists a macaroon $\mu^{(j)} \in M^*$ that is a valid discharge for this caveat and $\mu^{(j)}$ was not the output of a discharge query for token $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$.

Winning condition: The adversary $\mathcal{A}$ outputs a macaroon μ^* and a set of discharges $M^* = \{\mu_j^*\}$. He wins if:

1. $\mathsf{Verify}(k_{\mathcal{T}}, c, \sigma) = 1$ and $\mathsf{Verify}_{\mathcal{M}}(k^{(i_{\mathsf{ch}})}, \mu) = 1$, and
2. For the third-party caveat $3||c_{j_{\mathsf{ch}}}||\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}||\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ in μ, there exists $\mu_j \in M$ with key identifier $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ such that $\mathsf{Verify}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})}, \alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}, \mu_j) = 1$ where $\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})}$ is the key sampled by the challenger in the j_{ch}th extension query, and
3. Either
   (a) μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query, or
   (b) $\mu^{(j)}$ was not the output of a discharge query for token $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$.

Since j_{ch} is chosen uniformly at random, we have that $p_4 \geq p_3/q_{\mathsf{E}}$.

Game 5 : In Game 5, we change how the j_{ch}th extension query is answered. If is a query to add a third-party caveat, instead of encrypting $\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})}$, the reduction encrypts 0.

Extend Queries: On the j_{ch}th query that adds a third-party caveat, the adversary sends an identifier $\mathsf{id}^{(j_{\mathsf{ch}})}$, a tuple of caveats $\vec{m}^{(j_{\mathsf{ch}})}$ to the challenger. If $(\mathsf{id}^{(j_{\mathsf{ch}})}, \vec{m}^{(j_{\mathsf{ch}})})$ is a valid path in $\mathcal{T}$ leading to a node (μ, ek), then the challenger computes $(\mu', \mathsf{ek}')\mathsf{Extend}_{\mathcal{M}}(\mathsf{ek}, \mu, 3||c^{(j_{\mathsf{ch}})}||\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}||\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})})$ and adds (μ', ek') to $\mathcal{T}$ as a child of (μ, ek), where:

$$\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})} \leftarrow \mathsf{SetupToken}_{\mathcal{D}}(1^{\kappa}) \quad , \quad c^{(i)} \leftarrow \mathsf{Enc}_2(\mathsf{pk}_2, 0) \quad , \quad (\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}, \alpha_D^{(j_{\mathsf{ch}})}) \leftarrow \mathsf{Token}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}})$$

We argue that $|p_4 - p_5| = \mathsf{negl}(\kappa)$ by the CPA-security of $\mathcal{E}_2$.

Game 6 : In Game 6, we change the winning condition so that the adversary only wins if it outputs a macaroon μ^* that is valid under key $k^{(i_{\mathsf{ch}})}$, and is not an extension of a macaroon revealed to $\mathcal{A}$ in a macaroon query.

Winning condition: The adversary $\mathcal{A}$ outputs a macaroon μ^* and a set of discharges $M^* = \{\mu_j^*\}$. He wins if:

1. $\mathsf{Verify}(k_{\mathcal{T}}, c, \sigma) = 1$ and $\mathsf{Verify}_{\mathcal{M}}(k^{(i_{\mathsf{ch}})}, \mu) = 1$, and
2. μ^* is not an extension of a macaroon given to $\mathcal{A}$ in a macaroon query.

We argue that $|p_5 - p_6| = \mathsf{negl}(\kappa)$ by the security of the macaroon discharge protocol $\mathcal{D}$. Let E be the event that $\mathcal{A}$ outputs m^*, M^* such that for the third-party caveat $3||c_{j_{\mathsf{ch}}}||\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}||\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ in μ, there exists $\mu_j \in M$ with key identifier $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ such that $\mathsf{Verify}_{\mathcal{D}}(\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})}, \alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}, \mu_j) = 1$ where $\mathsf{sk}_{\mathsf{T}}^{(j_{\mathsf{ch}})}$ is the key sampled by the challenger in the j_{ch}th extension query, and $\mu^{(j)}$ was not the output of a discharge query for token $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$. Then $|p_5 - p_6| \leq \Pr[E]$, and we claim $\Pr[E] = \mathsf{negl}(\kappa)$ by the security of $\mathcal{D}$.
The reduction samples $\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_2, k_{\mathcal{T}}$ on its own and answers issuing and macaroon queries on its own. It also answers extend queries on its own if the query is adding a first-party caveat. It answers all extension queries on its own, except the j_{ch}th extension query that adds a third-party caveat. It answers the j_{ch}th extension query that adds a third-party caveat by calling the token oracle and receiving $\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}, \alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$. It answers all discharge queries by calling its discharge oracle. When $\mathcal{A}$ outputs μ^*, M^*, then the reduction looks for a discharge $\mu^{(j)} \in M^*$ with key identifier $\alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}$ and submits $\alpha_{\mathsf{S}}^{(j_{\mathsf{ch}})}, \alpha_{\mathsf{D}}^{(j_{\mathsf{ch}})}, \mu^{(j)}$ as its discharge forgery. If there is more than one, it picks one at random. If the event E occurs, then there is at least one valid forgery macaroon in M^* and the reduction picks it with probability $1/|M^*|$. Therefore, the probability that the reduction outputs a valid forgery is at least $\Pr[E]/|M^*|$. Since $|M^*| = \mathsf{poly}(\kappa)$, by the security of the discharge protocol $\mathcal{D}$, we must have $\Pr[E] = \mathsf{negl}(\kappa)$, as desired.

$\mathcal{A}$ Can't Win : We argue that $p_6 = \mathsf{negl}(\kappa)$ by the security of the macaroon scheme $\mathcal{M}$. The reduction samples $\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_2, k_{\mathcal{T}}, \mathsf{params}_{\mathsf{D}}, \mathsf{sk}_{\mathsf{D}}$ on its own. On the ith issue query, if $i \neq i_{\mathsf{ch}}$, the reduction answers it on its own as well. On the i_{ch}th query, the reduction computes $c_{i_{\mathsf{ch}}}$ and $\sigma_{i_{\mathsf{ch}}}$ on its own (note that this does not require it to know $k_{i_{\mathsf{ch}}}$), and calls its issuing oracle with $\mathsf{id}_{i_{\mathsf{ch}}} = c_{i_{\mathsf{ch}}}||\sigma_{i_{\mathsf{ch}}}$. The reduction answers extend queries by first checking if $\mathsf{id}^* = \mathsf{id}_{i_{\mathsf{ch}}}$. If it is, then it uses its extension oracle to complete the query, otherwise it computes everything on its own. It answers discharge queries on its own. When $\mathcal{A}$ outputs a macaroon forgery μ^*, then the reduction outputs μ^* as well. It is easy to see that if $\mathcal{A}$ wins, then the reduction outputs a valid forgery. Therefore, by the security of the macaroon scheme, we must have $p_6 = \mathsf{negl}(\kappa)$.

□

We wish to prove that $p_0 = \mathsf{negl}(\kappa)$. We know that $|p_0 - p_1| = \mathsf{negl}(\kappa)$, $p_1 = q_{\mathsf{I}} \cdot p_2$, $|p_2 - p_3| = \mathsf{negl}(\kappa)$, $p_3 \leq p_4 \cdot q_{\mathsf{E}}$, $|p_4 - p_5| = \mathsf{negl}(\kappa)$, $|p_5 - p_6| = \mathsf{negl}(\kappa)$, and $p_6 = \mathsf{negl}(\kappa)$. Putting it all together, we have:

$$|p_0 - p_1| + q_{\mathsf{T}} \cdot |p_2 - p_3| + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot |p_4 - p_5| + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot |p_5 - p_6| \geq$$

$$(p_0 - p_1) + q_{\mathsf{T}} \cdot (p_2 - p_3) + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot (p_4 - p_5) + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot (p_5 - p_6) = p_0 - q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot p_6$$

Since q_{T} and q_{E} are numbers of queries that $\mathcal{A}$ makes and $\mathcal{A}$ runs in polynomial time, we must have that $q_{\mathsf{T}} = \mathsf{poly}(\kappa)$ and $q_{\mathsf{E}} = \mathsf{poly}(\kappa)$. We can therefore conclude that

$$p_0 \leq |p_0 - p_1| + q_{\mathsf{T}} \cdot |p_2 - p_3| + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot |p_4 - p_5| + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot |p_5 - p_6| + q_{\mathsf{T}} \cdot q_{\mathsf{E}} \cdot p_6 = \mathsf{negl}(\kappa),$$

as required.