A Contractual Anonymity System

Edward J. Schwartz David Brumley Jonathan M. McCune
Carnegie Mellon University
{edmcman,dbrumley,jonmccune}@cmu.edu

Abstract

We propose, develop, and implement techniques for achieving contractual anonymity. *In contractual anonymity, a user and service provider enter into an anonymity contract. The user is guaranteed anonymity and message unlinkability from the contractual anonymity system unless she breaks the contract. The service provider is guaranteed that it can identify users who break the contract. The significant advantages of our system are that 1) the service provider is not able to take any action toward a particular user (such as revealing her identity or blacklisting her future authentications) unless she violates her contract, 2) our system can enforce a variety of policies, and 3) our system is efficient.*

1 Introduction

Internet services such as chat rooms for victims of violence, abuse information and support message boards, and whistle-blowing services are more compelling to users if they provide *anonymity*. Despite users' desire for anonymity, service providers must grapple with the need to identify misbehaving users in order to protect their service. For example, a provider may need to identify and stop undesired behavior such as using the anonymity service to launch denial of service attacks or threaten other users.

Thus, an anonymity service must strike a balance between accountability and anonymity. Users of such services want as much anonymity as possible, and ideally, should not have to trust the service provider. The service provider, however, must retain some ability to identify misbehaving users to protect the value of the service.

Previous anonymity protocols [10, 31, 32] resolved this tension decidedly in favor of the service provider by allowing the service provider to *subjectively judge* whether a user misbehaved. Subjective judging does not provide adequate anonymity guarantees to the user in many scenarios because the provider can arbitrarily decide to blacklist (deny future use of the service to) the user for any reason and thus can discriminate, e.g., treat each user differently based on her past actions.

In this paper, we propose *contractual anonymity*, which offers a wider range of options for resolving the accountability vs. anonymity tension. In contractual anonymity, the user and service provider (SP) enter into a binding anonymity contract. The user is guaranteed to remain anonymous and not blacklisted as long as she follows the contract policy, while the service provider is guaranteed to be able to identify (and blacklist, if desired) users that break their contract.

In a contractual anonymity scheme, the contract policy can be an arbitrary boolean function $f : \{msg_1, \ldots, msg_n\} \rightarrow \{\text{ALLOWED}, \text{VIOLATION}\}$. If the function returns VIOLATION on a message (or messages), the message is deemed malicious and the user is de-anonymized. Example policy scenarios include:

Matching-based A policy may state that any message matching a pre-defined pattern is considered malicious. Such policies could de-anonymize messages that match an intrusion detection rule or malware signature, disallow messages containing a predefined set of profane words, and so on.

Consensus-based A policy may require that a threshold of users sign a petition to de-anonymize a user. In particular, if n unique users anonymously flag a message, it is considered malicious and the sender is de-anonymized. Note that in our protocol the message will not cause de-anonymization if $n - 1$ users sign the message, or if the same user signs a message n times.

Subjective-based A policy may state that any message selected by a special privileged entity is considered malicious. For example, any user that sends a message which is later designated (e.g., signed)

Published in the Proceedings of the 2010 Network and Distributed System Security Syposium

by an appropriate law-enforcement agency's key would be considered malicious. These policies can also enable subjective judging (if it is desired) in a manner similar to related systems [10, 31, 32], e.g., by allowing a SP to de-anonymize messages of its choice. However, in contractual anonymity, the user must explicitly agree to a contract that allows subjective judging.

Previous Systems Some of these policies can be enforced using previous systems. For instance, consensus-based policies can conceivably be enforced using threshold cryptography [17], and subjective-based policies have been considered by past works [10, 31, 32]. Subjective-based systems can enforce a variety of policy functions. However, in such a system, the SP can decide to change the policy function at any time; the user is not guaranteed access to the service if she behaves. In contractual anonymity, the user is guaranteed anonymity and access to the service if she does not break the contract, and neither the user nor SP can change the contract without the approval of the other party.

Previous subjective judging protocols also require all messages to be rate-limited [10, 31, 32]. These restrictions prevent the protocols from being used in a variety of settings, including those where users do not trust the service provider to make fair subjective judgements, and for services that send messages at a high rate.

A Protocol for Contractual Anonymity We develop and implement a contractual anonymity protocol called RECAP. We show through our implementation of RECAP that it is feasible to build a secure contractual anonymity implementation with a small trusted computing base while simultaneously achieving better performance than prior approaches (Section 7).

In RECAP, each user is given an anonymous credential that allows the user to send messages anonymously. RECAP implements anonymous credentials using group signatures (Section 2.1). At a high level, a group signature scheme allows any member of the group to sign on behalf of the group. Individual signatures from unrevoked members (i.e., users who have not broken the contract) are indistinguishable from any other unrevoked member's signatures. Group signatures allow the SP to efficiently authenticate messages without needing to know each sender's identity (and still reject messages from revoked users).

However, group signature schemes are not sufficient to achieve contractual anonymity. Group signatures require a group manager who is capable of de-anonymizing users at will. To achieve contractual anonymity, this entity must be constrained to only de-anonymize users that violate their contract. We address this in RECAP by leveraging trusted computing [30] to implement a *verifiable* third party, called the accountability server (AS), that acts as group manager and knows the mapping between users' real identities and anonymous credentials. Specifically, the AS is a software module that will only reveal a user's real identity if the SP provides message(s) that prove the user has violated the contract.

Note that the AS is *not* arbitrarily trusted by either the user or SP. We construct the AS with a small trusted computing base (TCB) that does not include the operating system or BIOS, and allow the user and SP to verify the exact code the AS runs (Section 6). We term this *verifiable trust* since all parties can verify that the trusted party is running correctly and in the pre-agreed manner.

Contributions We introduce the concept of contractual anonymity, in which users are guaranteed anonymity as long as they do not violate the policy of their pre-negotiated contract with the SP. The SP is guaranteed that it can learn a user's real identity and identify that user's past and future messages if the user breaches the contract. We design the RECAP protocol, which is the first protocol that provides contractual anonymity. We also implement RECAP with a very small trusted computing base. Through our implementation, we show that RECAP is more efficient and offers a wider spectrum of solutions to the accountability vs. anonymity tension than competing approaches [9, 10, 31, 32].

Organization The remainder of the paper is organized as follows. In Section 2, we present relevant background on group signatures and trusted computing. We discuss how our system operates at a high level in Section 3, and then in more detail in Section 4. We describe several advantages of and extensions to our system in Section 5. Our implementation and evaluation results are described in Sections 6 and 7, respectively. The discussion is in Section 8. We explore related work in Section 9. Finally, we conclude in Section 10.

2 Primitives

2.1 Anonymity and Group Signatures

RECAP uses group signatures [2, 5–8, 13–15] to implement anonymous credentials. In a group signature scheme, each group member has a unique private signing key that allows them to sign messages on behalf of the entire group. There is a single group public key which can be used to verify any member's signature. Group signature schemes provide anonymity among members of the group, since a verifier cannot distinguish which group member signed a particular message. The group manager is provided with a special trapdoor that can undo the signature anonymity. In RECAP, the AS, acting as a *verifiable* third party, acts as the group manager.

A group signature scheme suitable for RECAP must support verifier-local revocation [8]. Verifier-local revocation allows the signature verifier to determine if a message was signed by a revoked user without communicating with the group manager. The group manager can revoke a user by publicly disclosing a special token which verifiers add to their local blacklist. In RECAP, verifier-local revocation allows the SP to efficiently detect and disregard messages from blacklisted users.

Such a scheme has four procedures: GS_KEYGEN, GS_SIGN, GS_VERIFY, and GS_OPEN. We describe these algorithms below at a high level. We refer the reader to previous work [8] that provides the the full specification including security proofs.

GS_KEYGEN(n) The GS_KEYGEN algorithm takes in the number of group members n. The algorithm outputs a group public key K_{GPK}, the group manager secret key K^{-1}_{GMSK}, an n-element vector $K^{-1}_{GSK}[1 \ldots n]$ of user secret keys, and an n-element vector of user revocation tokens $RT[1 \ldots n]$.

GS_SIGN$(K_{GPK}, K^{-1}_{GSK}[i], M)$ GS_SIGN takes a message $M \in \{0,1\}^*$, group member i's private key $K^{-1}_{GSK}[i]$, and the group public key K_{GPK}, and returns a group signature σ.

GS_VERIFY(K_{GPK}, M, σ, BL) GS_VERIFY takes as input the group public key K_{GPK}, a message M, an alleged signature σ, and a blacklist BL that consists of zero or more revocation tokens, and returns one of $\{$VALID, INVALID$\}$. An output of INVALID means that either the signature σ is invalid, or that the signer is on the blacklist BL. In the latter case, the signer's identity is also returned.

GS_OPEN$(K^{-1}_{GMSK}, M, \sigma)$ GS_OPEN takes as input the group manager secret key K^{-1}_{GMSK}, a message M and a corresponding signature σ. If (M, σ) is a valid message-signature pair, GS_OPEN outputs a revocation token $RT[s]$ for the signer s. The group manager can distribute revocation tokens to a verifier, allowing them to detect messages signed by s using the GS_VERIFY algorithm.

The properties of modern group signature schemes are often based on a framework introduced by Bellare et al [5]. The group signature scheme we use in our implementation, the Boneh-Shacham group signature scheme [8], bases its formal definitions in this framework. A group signature scheme suitable for RECAP must have the following properties (as described in the original Boneh-Shacham work [8]):

Correctness For any K_{GPK}, $K^{-1}_{GSK}[1 \ldots n]$, and $RT[1 \ldots n]$ returned by the GS_KEYGEN algorithm, any signature produced by the GS_SIGN algorithm must return VALID when verified using the GS_VERIFY algorithm, unless the user has been revoked. Specifically, $\forall i \in \{1 \ldots n\}$, $\text{GS\_VERIFY}(K_{GPK}, M, \text{GS\_SIGN}(K_{GPK}, K^{-1}_{GSK}[i], M), BL) = \text{VALID} \Leftrightarrow RT[i] \notin BL$.

Traceability Traceability is defined in terms of a game that takes place between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The traceability property holds if no adversary $\mathcal{A}$ can win the traceability game with more than negligible probability. In the traceability game, $\mathcal{A}$ wins if it can forge a signature that cannot be traced to any user in a coalition of users that $\mathcal{A}$ controls. The traceability game consists of three stages:

Setup $\mathcal{C}$ runs the GS_KEYGEN algorithm, and provides K_{GPK} and $RT[1 \ldots n]$ to $\mathcal{A}$. U, the set of users in $\mathcal{A}$'s coalition, is initially set to $\emptyset$. S, the set of message-signature tuples $\mathcal{A}$ obtained from oracles, is also set to $\emptyset$.

Queries $\mathcal{A}$ is allowed to query the GS_SIGN and GS_CORRUPT oracles. The GS_SIGN oracle takes as input a message M, a user $i \in \{1 \ldots n\}$, and outputs $\sigma \leftarrow \text{GS\_SIGN}(K_{GPK}, K^{-1}_{GSK}[i], M)$. $\mathcal{C}$ sets $S \leftarrow S \cup \{(M, \sigma)\}$ for each message-signature pair returned by the GS_SIGN oracle. The GS_CORRUPT oracle allows $\mathcal{A}$ to corrupt a user into joining the coalition. GS_CORRUPT takes a user $i \in \{1 \ldots n\}$ as input, and outputs $K^{-1}_{GSK}[i]$, the user's private key. The challenger sets $U \leftarrow U \cup \{i\}$ for each user i corrupted by the GS_CORRUPT oracle.

Response $\mathcal{A}$ outputs a message M', a set of revocation tokens that forms a blacklist BL', and a signature σ'.

An adversary $\mathcal{A}$ wins the game if all of the following conditions hold:

- GS_VERIFY$(K_{GPK}, M', \sigma', BL') =$ VALID
- GS_OPEN$(K^{-1}_{GMSK}, M', \sigma') = i \notin U$
- $(M', \sigma') \notin S$

Selfless-anonymity Selfless-anonymity, like traceability, is defined in terms of a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. In the selfless-anonymity game, $\mathcal{A}$ tries to determine which of two keys was used to generate a signature σ. The selfless-anonymity property holds if no adversary $\mathcal{A}$ can win the selfless-anonymity game with more than negligible advantage over random guessing. The game has five stages:

Setup $\mathcal{C}$ runs the GS_KEYGEN algorithm and obtains K_{GPK}, $K^{-1}_{GSK}[1 \ldots n]$, $RT[1 \ldots n]$. $\mathcal{C}$ gives K_{GPK} to $\mathcal{A}$. $\mathcal{C}$ sets U, the set of users that $\mathcal{A}$ has compromised or revoked, to $\emptyset$.

Queries The adversary can query the GS_SIGN, GS_CORRUPT and GS_REVOKE oracles. $\mathcal{C}$ runs the GS_SIGN oracle by computing $\sigma \leftarrow$ GS_SIGN$(K_{GPK}, K^{-1}_{GSK}[i], M)$ where user i and message M are inputs, and returns σ to $\mathcal{A}$. $\mathcal{A}$ can obtain the private key of user $i \in \{1 \ldots n\}$ using the GS_CORRUPT oracle. $\mathcal{C}$ runs this oracle by returning $K^{-1}_{GSK}[i]$ and setting $U \leftarrow U \cup \{i\}$. The GS_REVOKE oracle allows the adversary to obtain the revocation token for user $i \in \{1 \ldots n\}$. $\mathcal{C}$ simulates this oracle by returning $RT[i]$ and setting $U \leftarrow U \cup \{i\}$.

Challenge $\mathcal{A}$ chooses a message M and user indices i_0 and i_1 where $i_0 \notin U$ and $i_1 \notin U$. $\mathcal{C}$ chooses a random bit $b \overset{R}{\leftarrow} \{0, 1\}$ and returns $\sigma' \leftarrow$ GS_SIGN$(K_{GPK}, K^{-1}_{GSK}[i_b], M)$ to $\mathcal{A}$.

Restricted Queries $\mathcal{A}$ is allowed to make queries as in the **Queries** stage. However, in this stage the GS_CORRUPT and GS_REVOKE oracles cannot be queried for users i_0 and i_1.

Output $\mathcal{A}$ outputs a bit b'. If $b = b'$, $\mathcal{A}$ wins.

The Boneh-Shacham [8] scheme is an efficient group signature scheme that provides these properties. Specifically, signing a message takes about[1] eight modular exponentiations and two computations of a bilinear map. Verification with an empty blacklist (BL) requires approximately six modular exponentiations and three computations of a bilinear map. There are two ways of adding local revocation to the verification algorithm. The first, which provides the above properties of correctness, traceability, and selfless-anonymity, can be achieved using a $O(|BL|)$ algorithm, which performs two additional bilinear map computations for each entry in the verifier's blacklist.

However, the second type of revocation can be done in $O(1)$ time by using a precomputed revocation table at the expense of allowing a small number of messages to be linked. In the $O(|BL|)$ scheme, each signature contains a random identifier r that ranges over a large group. The identifier r is used when performing revocation checks. The $O(1)$ scheme constrains this identifier to range from $1 \ldots k$, which allows the revocation table to be computed in advance. The downside is that a verifier can determine that two signatures with the same value of r that are signed by the same user were in fact signed by the same user – this is called *partial unlinkability*, Unfortunately, there is no formal definition of partial unlinkability [8]. Informally, partial unlinkability ensures that a verifier can only link (e.g., determine that $signer(m_1) = signer(m_2)$) one out of every k signatures signed by the same user for the same site (e.g., a SP in our scheme), where k is a security parameter. For instance, if $k = 100$, 1% of the signatures are linkable. We believe that in many cases the benefits of constant time revocations outweigh the downside of having a small number of linkable messages, and so we consider the $O(1)$ scheme in our implementation. We discuss the repercussions of this further in Section 8.4.

2.2 Trusted Computing and Contract Enforcement

RECAP uses a *verifiable* third party (the AS) to securely bind user identities to contract policies and convince remote parties that it has done so. These properties can be achieved using platform security technologies built on the Trusted Platform Module (TPM) which is available in many recent commodity platforms [1, 20, 30]. Alternatively, secure coprocessors like the IBM 4758 [27] provide similar properties and stronger resistance to physical attacks, but are more expensive and not as readily available (the pros and cons of each are discussed further in Section 8.2). Specifically, RECAP will work on any trustworthy computing mechanism that provides the following properties:

Isolation Isolation allows execution of software com-

[1] This assumes that computing a group isomorphism takes roughly the same amount of time as computing a modular exponentiation.

ponents to take place in an isolated, verifiable environment such that any OS, Virtual Machine Monitor (VMM) or BIOS code that is running cannot affect or observe the execution in the isolated environment. Software running in isolation can have a small, self-contained Trusted Computing Base (TCB) that does not include the OS, BIOS, or device firmware. This is useful because the TCB of software running on commodity operating systems is generally very large, usually including the operating system, BIOS, etc.

Sealed Storage Sensitive data is protected using sealed storage, whereby data can be encrypted such that subsequent decryption is only possible if the platform is executing specific software. For instance, this can be used with isolation to ensure that sensitive information can only be decrypted by a specific software component running in isolation.

Attestation One system can prove to another that it has loaded certain code for execution within an isolated environment using attestation. An attestation demonstrates to a remote verifier that the attesting platform instantiated an execution environment with a particular code module, along with its input and output values. We denote the process of creating an attestation of the currently running code module with input i as $\textsc{Gen\_Attest}(i)$. A third party can compute the value an attestation should have for a *code module* running with input i as $\textsc{Exp\_Attest}(\textit{code module, i})$. Only an attestation from a platform executing the *code module* on input i should be equal to the verifier's, i.e., $\textsc{Gen\_Attest}(i)$ = $\textsc{Exp\_Attest}(\textit{code module, i})$.

Unique Identifiers Each user has a real identity that is revealed if she violates the contract. For concreteness, we assume RECAP uses the unique, unspoofable identifier found as part of various trusted computing platforms, called the endorsement certificate. We also refer to the endorsement certificate as the trusted computing identifier. Section 5 discusses the importance of practical unique identifiers in the context of Sybil attacks.

Since the user and AS both have different roles in RECAP, it may be desirable to use different trusted computing implementations for each party. For instance, since the AS stores sensitive information, it may be worthwhile to use a secure coprocessor that is designed to withstand physical attacks for the AS (discussed more in Section 8.2), but use the more readily available TPM-based platform for the user.

3 Design Overview

3.1 Contractual Anonymity Requirements

A contractual anonymity protocol should have the following properties:

Unlinkability We consider a user u to be unlinkable in an unlinkability set $\mathcal{S}_{\mathcal{L}}$ if, given any two messages m_1 and m_2 such that $signer(m_1) = signer(m_2) = u$, an adversary can determine that $signer(m_1) \in \mathcal{S}_{\mathcal{L}}$ and $signer(m_2) \in \mathcal{S}_{\mathcal{L}}$, but the probability that the adversary can determine $signer(m_1) = signer(m_2)$ is $\leq \epsilon$, a security parameter in our system. $\epsilon = 0.01$ means that 1% of messages from the same user can be linked. A user is unlinkable unless she breaks her contract.

Note that unlinkability implies the weaker notion of anonymity, e.g., that an adversary cannot learn the identity of the user that sent a message.

Contract-based The user and SP enter in a contract, and both parties are bound by the contract. A contract unambiguously specifies the agreed-upon terms of service with a policy function $f : \{msg_1, \ldots, msg_n\} \rightarrow \{\textsc{Allowed}, \textsc{Violation}\}$. If the user signs some messages m such that $f(m)$ returns $\textsc{Violation}$, these signed messages can prove misbehavior to the AS, who will de-anonymize the user that signed m. Once a user is de-anonymized, unlinkability no longer holds for that user. Neither the user nor the SP can modify an accepted contract; they must explicitly agree to a new contract if they wish to change the policy function f.

Revocability The SP is able to obtain a user's identity if the SP has proof that the user broke her contract. Specifically, the proof is a set of messages m signed by the user such that $f(m)$ returns $\textsc{Violation}$, where f is the user's contract policy function. The SP can then take appropriate action, e.g., blacklist the user.

Efficiency The protocol should be as efficient as possible. This includes, but is not limited to, scaling well with respect to the number of blacklisted users and not requiring stringent rate limiting.

At a high level, previous approaches fail to meet these requirements since they do not bind anonymity to a pre-negotiated contract. For example, many existing anonymity systems require an unverifiable trusted third party (TTP) that is capable of de-anonymizing users at will [8, 12, 13]. More recent systems [10, 32] allow for subjective judgement and anonymous blacklisting, which allow a SP to blacklist a user for any reason and at any time without completely de-anonymizing her. In these systems, the user is never guaranteed unlinkability, even if she follows the SP's posted policies.

3.2 RECAP Overview

We propose RECAP, a protocol for achieving the desired contractual anonymity properties. RECAP involves three parties.

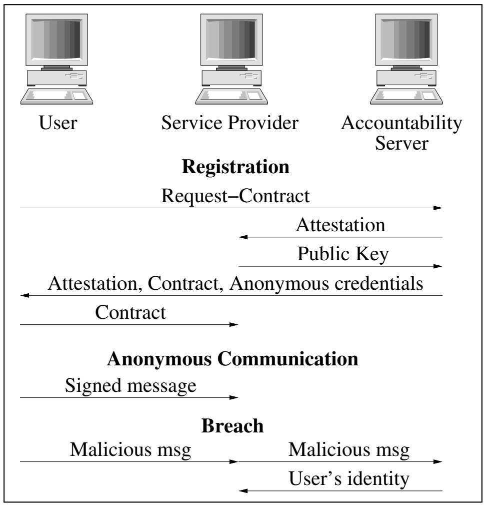


Figure 1. The three stages of RECAP are registration, anonymous communication, and breach.

User The user wants to access the service that is provided by the service provider. However, she may not trust the SP or the other users, and thus wants to be anonymous and unlinkable when using the service.

Service Provider The service provider (SP) wants to provide anonymous and unlinkable access to its service. However, the SP also wants the ability to blacklist users who threaten the utility of the service.

Accountability Server The accountability server (AS) is a *verifiable* third party that manages users' anonymous credentials and de-anonymizes users that violate their contracts. Specifically, the user and SP can verify that the AS de-anonymizes users if and only if they break their contract.

An overview of the different stages and participants of RECAP is shown in Figure 1. We provide an overview of the stages below, and then discuss them in greater detail in Section 4.

3.2.1 Protocol Stages

Setup During setup, the parties must generate two types of keys. The user, AS, and SP must generate public/private keypairs that can be used for digital signatures and asymmetric encryption (e.g., RSA keypairs). The user only uses these asymmetric keys when registering with the AS. The AS generates group signature keys for each group required. The user and AS generate and seal their keypairs (and related sensitive information) so that only the trusted RECAP code can decrypt it. The SP obtains a certificate that binds its identifying name to its public key (like a SSL certificate).

Registration Phase In the initial registration phase a SP and user agree on a specific contract policy. The contract policy stipulates the rules that users are expected to follow. We discuss policies further in Sections 3.3 and 8.3.

In RECAP, the user receives a contract policy proposed by the SP. If she agrees to the policy, then she requests a contract containing that policy from the AS. The AS returns an anonymous credential and a contract for the user that can be used with the SP.

The contract can be thought of as proof that the AS has bound together the user's real identity, anonymous credential, and the contract policy. It provides assurance to the SP that the AS knows the true identity of the user assigned the anonymous credential, and will reveal that identity if given proof that the user has broken the policy in her contract.

At the completion of the registration phase, the SP and user have a contract that guarantees the user's real identity will only be revealed by the AS if the SP can submit a set of messages signed by the user that violate the contract.

Anonymous Communication Phase In the anonymous communication phase the user uses her anonymous credential to interact with the SP. In particular, the user communicates with the service by digitally signing a message with her group private key. The SP then verifies that the message was created by a user with a valid contract by verifying the signed message with the group public key specified in the contract.

Since each message must prove it came from an authorized user, we also refer to communicating messages to the SP as performing an *authentication*. An anonymous communication operation is analogous to the authentication operation of an anonymous authentication protocol [31, 32].

Contract Breach A breach of contract happens when the user sends message(s) prohibited by the contract policy to the SP. The SP can identify which user violated the contract by presenting the prohibited message(s) to the AS. Upon confirming that the message(s) violate the agreed-upon contract, the AS reveals the user's real identity and group signature revocation token to the SP. This allows the SP to identify any subsequent and prior communication using the anonymous credential.

At the end of the breach phase, the SP has the capability to identify the user who breached her contract, and thus can take appropriate action. To be concrete, we assume the SP will blacklist the user. The blacklist (BL) is a list of users who have violated the contract and are no longer allowed to use the service. The SP can blacklist the user by adding the user's anonymous credential to her group signature blacklist. To prevent the user from obtaining a new anonymous credential, we allow the SP to control the AS's blacklist, which is a list of real identities that are not allowed to obtain new anonymous credentials. RECAP can easily be extended to support other actions as well, such as anonymous blacklisting, in which SPs are given the ability to blacklist users without needing to know their real identities.

3.3 Contract Policies

An anonymity contract is a binding agreement that states that a user's real identity may be exposed if she violates the contract terms. We call those terms the contract policy.

A contract policy is a boolean predicate $f : \{msg_1, msg_2, \ldots, msg_n\} \rightarrow \{\textsc{Allowed}, \textsc{Violation}\}$. The status VIOLATION indicates that the messages violate the contract terms, and thus the user is in breach of contract. ALLOWED indicates that the messages do not violate the policy. We do not make any attempt to model a "morality function"; such policies are outside the scope of our system. However, there are many classes of policies that do fit our model, several of which were described in Section 1. We discuss policies further in Section 8.3.

4 Architecture

4.1 Establishing a Secure Channel

Why is a New Protocol Needed? Many parts of RECAP rely on the ability to create a secure channel between the protocol participants. A RECAP secure channel must be able to provide 1) confidentiality and integrity of any messages sent inside the channel, and 2) assurance that the remote party's private key is sealed so that only the trusted RECAP software can access it.

Although confidentiality and integrity of messages inside the channel can be achieved using standard techniques (i.e., SSL/TLS), RECAP has several requirements which motivate a new secure channel protocol:

- The user's endorsement certificate is used as the user's unique identifier and as a required component for verifying user attestations. Thus, the secure channel protocol must not require the user to reveal her endorsement certificate, which is confidential, until it has been established that only the trusted RECAP code can access it.
- The untrusted part of the RECAP software (e.g., the part not running in trusted computing-enabled isolation) must demonstrate that it runs the trusted RECAP software in response to the secure channel establishment. In addition, the trusted RECAP software must prove it has access to its trusted long term RSA key. Thus, two sets of challenges are needed.
- Any agent running the secure RECAP software in an isolated, verifiable environment is considered to be trusted. Because of this, it is not only important to establish confidentiality and integrity among the possessors of the keys used to setup the channel, but also establish that these keys are *only accessible by the trusted* RECAP *software.*

RECAP achieves these requirements using the protocol described below. RECAP's secure channel protocol is similar to existing secure channel protocols, modulo the changes needed to add the above features.

Protocol Details Lines 3–16 of Figure 2 show how we establish a secure channel in the registration protocol between the user and AS. A secure channel is established

	U-SP registration protocol	
1. U → SP:	$\{message, \sigma = \bot, contract = \bot\}$	
2. SP → U:	$\{\textit{Get-Contract}, AS, SP, Addr_{SP}, CP\}_{K_{SP}^{-1}}$	
	U-AS registration protocol	
3. U:	$N_U \overset{R}{\leftarrow} \{0,1\}^\alpha, R_U \overset{R}{\leftarrow} \{0,1\}^\alpha$	
4. U → AS:	$\{K_U, N_U\}$	
5. AS:	$N_{AS} \overset{R}{\leftarrow} \{0,1\}^\alpha, R_{AS} \overset{R}{\leftarrow} \{0,1\}^\alpha$	
6. AS:	$a \leftarrow \textsc{Gen\_Attest}(K_U \mid N_U \mid K_{AS} \mid N_{AS})$	
7. U ← AS:	$\{K_{AS}, N_{AS}, a, C_{Endorse-AS}\}$	
8. U:	$a' \leftarrow \textsc{Exp\_Attest}(\textit{Trusted}\ \text{RECAP}\ \textit{Code}, K_U \mid N_U \mid K_{AS} \mid N_{AS})$	
9. U:	abort if $a \neq a'$	User knows AS running trusted code.
10. U → AS:	$\{\{R_U\}_{K_U^{-1}}\}_{K_{AS}}$	
11. U ← AS:	$\{\{R_U + 1, R_{AS}\}_{K_{AS}^{-1}}\}_{K_U}$	User knows K_{AS} bound to trusted AS code.
12. U:	$a \leftarrow \textsc{Gen\_Attest}(K_U \mid N_U \mid K_{AS} \mid N_{AS})$	
13. U → AS:	$\{\{R_{AS} + 1, a, C_{Endorse-U}\}_{K_U^{-1}}\}_{K_{AS}}$	
14. AS:	$a' \leftarrow \textsc{Exp\_Attest}(\textit{Trusted}\ \text{RECAP}\ \textit{Code}, K_U \mid N_U \mid K_{AS} \mid N_{AS})$	
15. AS:	abort if $a \neq a'$	AS knows user running trusted user code and K_U bound to trusted user software.
16.	Setup symmetric encryption and MAC	Secure channel established.
17. U → AS:	$\{\textit{Get-Contract}, AS, SP, Addr_{SP}, CP\}_{K_{SP}^{-1}}$	
18. AS:	abort if $C_{Endorse-U}$ on SP's blacklist	
19. AS:	execute key binding protocol	
20. U ← AS:	$\{\{CP, K_{GPK}, K_{SP}\}_{K_{AS}^{-1}}, K_{GSK}^{-1}[i]\}$	
	U-SP anonymous communication protocol	
21. U → SP:	$\{message, \sigma = \text{GS\_}\textsc{Sign}(K_{GPK}, K_{GSK}^{-1}[i], message), contract = \{CP, K_{GPK}, K_{SP}\}_{K_{AS}^{-1}}\}$	
22. SP:	if $\text{GS\_}\textsc{Verify}(K_{GPK}, message, \sigma, BL) = \textsc{Valid}$, accept message.	

Figure 2. The registration and anonymous communication protocols. The user obtains a contract using the registration protocol. The anonymous communication protocol is then used to send anonymous messages to the service provider (SP). All messages after Line 16 are implicitly encrypted and MACed using symmetric cryptography.

in the breach protocol as well, but the process is very similar. On Lines 3–4 of Figure 2, the user generates a nonce N_U and sends the nonce and its public key K_U to the AS. The AS generates its own nonce N_{AS} and an attestation to prove that is running the RECAP software in response to the user's request (Lines 5–6).

A verified attestation proves several important facts to the verifier (in this case, the user). First, by including N_U and N_{AS} in the attestation, the AS proves that it is responding to the user's request, which ensures freshness, i.e., the isolated execution environment ran in response to the user's request. Second, the AS proves that messages encrypted to K_{AS} or digitally signed by K_{AS}^{-1} can only be read or created by the AS. This is because a correct AS keeps security-sensitive data like K_{AS}^{-1} sealed so that only the trusted RECAP software can access it. The user can verify this, because they can verify the exact code the AS is running. Last, the AS also conveys that it has received the user's key, K_U.

After creating the attestation, the AS sends the attestation, nonce, its public key and its trusted computing device endorsement certificate (Line 7). The endorsement certificate is issued by a trusted computing device's manufacturer and usually indicates that the private component of a keypair is known only to that device. This lets a verifier confirm that an attestation came from a legitimate trusted computing device. The user's endorsement certificate also doubles as her unique identity, which is discussed further in Section 5. At this point, the user verifies that the attestation is correct; if it is not, she aborts the protocol (Lines 8–9). Otherwise, she encrypts and signs her random number, and expects the AS to increment it in response to prove that it can decrypt and sign using K_{AS}^{-1} (Line 10). It is worth noting that the random number is generated and handled in plaintext form only by the trusted RECAP code. In contrast, the nonce used earlier was not a secret. In response, the AS sends $R_U + 1$ and its own random number (Line 11). When the user receives its incremented random number, it believes that K_{AS} is bound to the trusted RECAP code, and is willing to send its endorsement certificate encrypted under that key as part of an attestation, because the RECAP code will only disclose the user's endorsement certificate (which is also her unique identity) if she breaks her contract. The user generates an attestation which provides similar properties to the AS, and sends it with its own endorsement certificate, and the AS's incremented random number (Lines 12–13). Upon receiving the incremented random number, the AS verifies the user's attestation (Lines 14–15), and both parties then switch to more efficient symmetric cryptography (Line 16). This can be done with standard techniques [24].

4.2 Protocol Phases

The RECAP protocol is split into three phases: the registration, anonymous communication, and breach phases. The registration phase is required before a user interacts with the SP. The anonymous communication phase serves to mark messages as originating from a user that has a valid contract. The breach phase takes place when the SP wants to know who created messages that are in violation of the contract.

Registration Phase RECAP begins with the user connecting to the SP (Line 1 in Figure 2). The user will not have a contract since it is her first time connecting, and indicates this in her initial message. The SP replies with a message indicating that a contract is required to use the service (Line 2). Specifically, the user must obtain a contract from the SP-specified AS and the contract must have the SP-specified contract policy (CP), which is the policy that the user must agree to. If the user agrees to abide by the CP, she connects to the AS and begins to create a contract. Otherwise, she aborts. We allow the SP to choose the policy, since that maps most closely to existing services. However, an alternate version of RECAP might be more flexible, i.e., allow a user to choose one of several policies, negotiate policy terms, etc.

To obtain a contract, the client connects to the AS and begins the U-AS protocol (Line 3). As was described in Section 4.1, the user and AS establish a secure channel (Lines 3–16). Once the channel is established, the client sends the contract policy that the SP requires (Line 17). The AS maintains a list of users that have been blacklisted by the SP, and aborts if one of those users is attempting to re-register (Line 18).

	AS-SP key binding protocol
1. AS:	$N_{AS} \overset{R}{\leftarrow} \{0,1\}^{\alpha}$
2. AS → SP:	$\{K_{AS}, N_{AS}\}$
3. SP:	$N_{SP} \overset{R}{\leftarrow} \{0,1\}^{\alpha}$
4. AS ← SP:	$\{\{N_{SP}, N_{AS}\}_{K_{SP}^{-1}}, K_{SP}, C_{SP}\}$
5. AS:	$a \leftarrow \textsc{Gen\_Attest}(K_{SP}\|N_{SP}\|K_{AS}\|N_{AS})$
6. AS → SP:	$\{a, C_{Endorse-AS}\}$
7. SP:	$a' \leftarrow \textsc{Exp\_Attest}($*Trusted* RECAP *Code*, $K_{SP}\|N_{SP}\|K_{AS}\|N_{AS})$
8. SP:	abort if $a \neq a'$

Figure 3. The key binding protocol.

At this point in the protocol (Line 19), the AS connects directly to the SP and executes the key binding protocol shown in Figure 3. The key binding protocol allows the SP to ensure that the AS is running the RECAP software, and to verify that K_{AS}^{-1} is bound to that software. This verification is proof to the SP that a user's identity will be revealed if that user breaks her contract. This protocol only needs to be executed once per (AS,SP) pair, since the result can be cached. Note that the key-binding protocol is similar to establishing a secure channel as described in Section 4.1. The central difference is there is no need to switch to symmetric cryptography since no messages are transmitted after the AS's key is shown to be bound to the trusted AS software.

After the key binding, the AS proceeds to create a contract. The contract consists of the contract policy the user agrees to, the public key of the user's group signature group, and the SP's public key. The AS sends the contract and a group private key to the user (Figure 2, Line 20). Finally, the user sends the contract to the SP, and she is ready to start endorsing messages (Line 21).

Anonymous Communication Phase To endorse a message, the user simply signs the message m using her group private key $K_{GSK}^{-1}[i]$, and sends the signed message to the SP (Line 21). When the SP receives a signed message, it ensures that it has received a valid contract with the corresponding group public key. The SP also verifies that the message has a valid signature by executing the group signature verification operation (Line 22).

Breach Phase When a user generates message(s) that violate the SP's policies, the SP delivers the offending message(s) to the AS. This protocol is shown in Figure 4. After establishing a secure channel (Lines 1–11), the AS verifies that the received messages are signed by a group that the AS manages (Lines 12–13). Then, the AS verifies that the messages violate the contract

	Breach protocol
1. AS:	$N_{AS} \overset{R}{\leftarrow} \{0,1\}^{\alpha}, R_{AS} \overset{R}{\leftarrow} \{0,1\}^{\alpha}$
2. AS → SP:	$\{K_{AS}, N_{AS}\}$
3. SP:	$N_{SP} \overset{R}{\leftarrow} \{0,1\}^{\alpha}, R_{SP} \overset{R}{\leftarrow} \{0,1\}^{\alpha}$
4. AS ← SP:	$\{K_{SP}, N_{SP}\}$
5. AS:	$a \leftarrow \textsc{Gen\_Attest}(K_{SP} \| N_{SP} \| K_{AS} \| N_{AS})$
6. AS → SP:	$\{a, C_{Endorse-AS}, \{\{R_{AS}\}_{K_{AS}^{-1}}\}_{K_{SP}}\}$
7. SP:	$a' \leftarrow \textsc{Exp\_Attest}(\textit{Trusted}\ \text{RECAP}\ \textit{Code},$ $K_{SP} \| N_{SP} \| K_{AS} \| N_{AS})$
8. SP:	abort if $a \neq a'$
9. AS ← SP:	$\{\{R_{AS}+1, R_{SP}\}_{K_{SP}^{-1}}\}_{K_{AS}}$
10. AS → SP:	$\{\{R_{SP}+1\}_{K_{AS}^{-1}}\}_{K_{SP}}$
11.	Setup symmetric encryption and MAC
12. AS ← SP:	$\{m = \{message_1, \sigma_1, \ldots, message_n, \sigma_n\}\}$
13. AS:	$\forall m_i \in m$, abort if $\text{GS\_}\textsc{Verify}(K_{GPK}, m_i, \sigma_i, BL) = \textsc{Invalid}$
14. AS:	abort if $CP(m) \neq \textsc{Violation}$
15. AS:	$gid \leftarrow \text{GS\_}\textsc{Open}(K_{GMSK}^{-1}, msg_1, \sigma_1)$
16. AS → SP:	$\{gid, RT[gid], GidToEKcert[gid]\}$

Figure 4. The breach protocol. The service provider (SP) submits any messages suspected to be in violation of the contract to the accountability server (AS). The AS verifies the messages, and returns the identity of the users that violated their contracts, if any. All messages after Line 11 are implicitly encrypted and MACed using symmetric cryptography.

(Line 14). The AS obtains the group private key[2] that violated the contract, by using the GS_OPEN operation (Line 15). It then reveals 1) the user's group signature revocation token, and 2) the user's real identity to the SP (Line 16). With that information, the SP can add the user's current group key to the group signature revocation list so that messages signed with her anonymous credential will no longer be accepted. The SP can also add the user's real identity to a blacklist on the AS that prevents the user from obtaining a new contract and anonymous credential.

4.3 Security Overview

4.3.1 Trusted Computing

RECAP builds some of the properties required for a contractual anonymity system from the properties of trusted computing:

Contract-based and Revocability These properties rely on the trusted computing properties of attestation, isolation, and sealed storage. The contract policies can be fairly enforced by running the trusted RECAP software in isolation and keeping sensitive information in sealed storage, and then proving this using attestation. This allows the user and SP to ensure that the AS is running a known-good implementation in hardware-assisted isolation, e.g., that the AS behaves as described in this paper and is not a malicious or incorrect implementation.

4.3.2 Group Signatures

RECAP also inherits properties from group signatures:

Unlinkability Unlinkability comes directly from group signature properties. For instance, if the $O(1)$ revocation scheme is being used, unlinkability with $\epsilon = \frac{1}{k}$ comes from partial unlinkability. If the $O(|BL|)$ revocation scheme is used, unlinkability with negligible ϵ comes from selfless-anonymity. We discuss this choice further in Section 8.4.

Contract-based Users must obtain anonymous credentials from the AS, who can hold each user accountable to the contract policy. This follows from traceability, which ensures a user cannot forge a signature without having anonymous credentials.

Revocability Again from traceability, any signature produced can be traced back to the key used to produce it. Thus, any message submitted by a user that violates her contract can be traced back to that user. That user can then be revoked from the service.

Protocol Correctness Although a formal security proof of the RECAP protocol is outside the scope of the paper, we note the protocol is similar to well-known secure protocols. The high-level semantics of the protocol (e.g., actions that take place after the secure channel is established) are straight-forward and can be manually verified. The setup of the secure channel can be more easily understood if one considers the basic steps of each party. Specifically, each party performs the following steps:

1. Attests to the code she is running in an isolated environment (Figure 2, Lines 6, 8, 12, and 14)

2. Sends her public key and certificate (Lines 4, 7, and 13)

[2] We assume for simplicity here that all messages in violation of the contract policy are signed with the same private key, i.e., that there is only a single malicious user.

3. Issues and responds to a challenge (Lines 10, 11, and 13)
4. Sets up symmetric cryptography (Line 16).

In other words, our secure channel protocol is similar to standard secure channel protocols (such as SSL 3.0), but does not negotiate which ciphers are used[3], and includes attestations. For instance, a model checking approach to verifying SSL has shown that the high level semantics of SSL can be reduced to similar simple steps [25]. We leave the augmentation of existing similar security proofs with our additional attestation and TPM steps as future work.

5 Features

Anti-discrimination RECAP prevents a SP and its AS from discriminating against anonymous users based on their past messages. Previous systems with TTPs have not appropriately limited the power of the TTP to blacklist [8, 12, 13], and so the TTP could potentially blacklist well-behaved users. For example, someone could compromise the TTP and blacklist users, or bribe the TTP itself to misbehave. Thus, such systems can discriminate.

Previous TTP-free systems allowed subjective judging [31,32], i.e., users can be blacklisted for any reason. The ability to subjectively judge means that a SP can block all future authentications from a user based on her past actions. For example, a user could post a message the SP dislikes, and the SP would be free to block all future authentication. Thus, the SP could discriminate against a user without knowing her real identity in such systems.

In RECAP, anyone can verify that the AS will only de-anonymize a user if her contract is violated. Further, the AS seals each user's real identity, which results in an encrypted blob that can only be decrypted when the trusted RECAP code is running in a verifiable execution environment. Thus, even if the untrusted part of the RECAP software, the operating system, or the BIOS is compromised, a collusion between the AS and SP cannot reveal a behaving user's real identity. RECAP also provides unlinkability of multiple authentications. Thus, a SP cannot discriminate against users who have not broken their contract. The SP would have to deny service to all behaving users in order to deny service to one.

Verifiable Blacklists Blacklists are commonly used in network services to block known malicious identities. Current blacklists, however, typically do not provide much information as to why a particular identity is on the list. RECAP can easily be extended to implement *verifiable blacklists*. We say a blacklist is verifiable if each identity on the blacklist is accompanied by a proof of the malicious activity that led to its being blacklisted.

During registration, a user and SP agree to the contract policy. The user will register her trusted computing device endorsement certificate $C_{Endorse-U}$ with the AS and receive her anonymous credential $K_{GSK}^{-1}[i]$. During contract breach, the AS is provided with a set of signed messages that violate the contract. When provided with evidence of a breach, the AS responds with both $K_{GSK}^{-1}[i]$ (so that subsequent messages from the user can be identified) and $C_{Endorse-U}$ (so that the blacklisted user cannot obtain a new anonymous credential).

In RECAP, the AS can publish those messages as proof that a breach has occurred to enable verifiable blacklists. More specifically, the AS publishes the tuple $(\{CP, K_{GPK}, K_{SP}\}_{K_{AS}^{-1}}, M, \sigma,$ $K_{GSK}^{-1}[i], \textsc{Gen\_Attest}(K_{GSK}^{-1}[i] \;\rightarrow\; C_{Endorse-U}))$, such that $\{CP, K_{GPK}, K_{SP}\}_{K_{AS}^{-1}}$ is the user's contract, M and σ are the offending message(s) and signature(s), and the AS attests to the fact that the anonymous credential (group signature private key) was issued to the referenced real identity. No trusted maintainer is required because the blacklist entries contain proof that the contract was violated.

Practical Unique Identifiers In the Sybil attack [18] a user can subvert security by forging new identities. In our system, users cannot create new identities themselves without breaking the traceability property of group signatures. Thus, in our setting a Sybil attack corresponds to a user successfully obtaining access to a new real identity, since a new identity allows her to obtain a new contract even if her old identity is on the blacklist.

Our architecture mitigates the Sybil problem by leveraging the unique identifier found in each user's trusted computing device as the user's real identity. A user cannot practically obtain a new identifier for her computer without replacing the trusted computing device (since it is a physical device and there is no programmatic way to replace it).

We believe that our solution is more practical than solutions in other systems. For instance, in PEREA [32], a suggested method is for the user to register with the SP by presenting her driver's license in person[4]. However, we argue that this is impractical for most services.

[3] RECAP always uses the same ciphers for simplicity.

[4] This still preserves anonymity because the registration is not link-

We do note that our solution is not perfect, since one computer can be shared by multiple users, and one user can own multiple computers. In particular, well-funded attackers may be able to purchase such a large number of computers that blacklisting all of them would be difficult. Services concerned about this type of adversary can use a more precise identifier as a unique identity, such as a driver's license or passport.

6 Implementation

We implemented RECAP using two cryptographic libraries: the PBC_SIG group signature library [22] that is a framework for implementing pairing based group signature schemes, and the XySSL library [34] for implementations of RSA, AES, SHA-1, and HMAC. We use 256-bit AES keys, HMAC keys, nonces and random values. RSA keys are 1024-bit. We use the Boneh-Shacham group signature scheme [8] with a Type-A pairing. We do not currently implement any local revocation checking for the group signature scheme [8], although we intend to implement this in future work. Instead, we consider the effects of using the $O(1)$-time revocation scheme that adds a table-lookup per verification, which is unlikely to significantly change our performance measurements. We assume that this table-lookup will have a negligible effect on our measurements. Options for implementing such revocation are discussed in Section 8.4.

Portions of the code that execute on the user's and AS's platforms constitute the security-sensitive, trusted components of RECAP. Our implementation uses the Flicker system [23] to provide the attestation, sealed storage, and isolation properties specified in Section 2.2 by using a TPM [30] and hardware-supported dynamic root of trust [19]. Datta et al. have proven that dynamic root of trust systems like Flicker allow a verifier to make strong conclusions about the software state of an attesting platform. The Trusted Computing Base (TCB) for security-sensitive RECAP code includes only the Flicker stub code, and excludes the legacy operating system, BIOS, and all DMA-capable devices. We expect the trusted RECAP code to be the same across all uses of RECAP, i.e., the code will be publicly known and evaluated to be "known-good" by manual or formal security analysis. Our implementation bases each user's unique identity on the Endorsement Credential found in each TPM (which is discussed further in Section 5). One benefit of using the Flicker system is that RECAP can run on commodity systems that are widely available.

The registration and breach phases of RECAP involve processing inside the Flicker isolation environment, because the protocol requires access to information that must be kept secret using sealed storage. However, Flicker does not support direct access to a network stack. Therefore, software that directly interfaces with the network stack must run on the untrusted host operating system. The untrusted portion of RECAP is responsible for launching the Flicker sessions on the user's and AS's platforms. We note that the untrusted code could choose not to launch the Flicker session. This is equivalent to the availability attack described in Section 8.2, but more importantly, the untrusted code cannot impersonate trusted code.

The trusted RECAP components that run in the Flicker environment are responsible for protecting their state using TPM_SEAL and TPM_UNSEAL. Many protocol messages in the registration and breach phases are passed as input to the Flicker environment, along with the sealed copy of any sensitive data that may be required. The trusted code will then unseal the information it needs and create its reply message. It will then output the reply message to be sent over the network, and seal and output any updated sensitive state before returning to the host operating system.

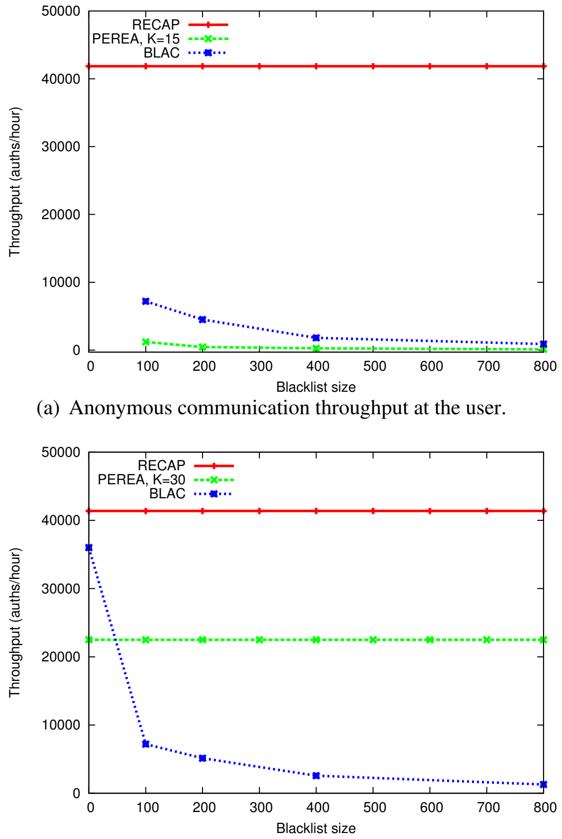
Sealed state on the user's platform includes R_U, R_{AS}, K_U^{-1}, K_{AS}, and K_{U-AS}. Sealed state on the AS's platform includes, for each registered user U_i: R_{AS}, R_{U_i}, K_{AS}^{-1}, K_{U_i}, K_{AS-U_i}, and the registered users' endorsement key certificates (real identities) $C_{Endorse-U_i}$. It further includes the entire set of private group signature keys $K_{GSK}^{-1}[1 \ldots n]$ (i.e., keys for each registered member, and unused keys that may be assigned to future members), and the group manager secret key K_{GMSK}^{-1}.

7 Evaluation

Our test machine is an off-the-shelf Lenovo Thinkpad T400 with a 2.53 GHz Intel Core 2 Duo processor and 2 GB of RAM. It runs Ubuntu 8.10 with Linux kernel 2.6.24. Our current implementation only utilizes one core, but a more sophisticated implementation could use multiple CPUs to improve performance. We perform all of our experiments on this one machine, i.e., we execute the SP, AS, and user code on the same machine. This configuration gives a conservative estimate of the protocol's end-to-end running time in a real system (excluding network latency), since only one Flicker session can be running in isolation at a time.

able to future authentications.

7.1 Performance



(a) Anonymous communication throughput at the user.

(b) Anonymous communication throughput at the service provider.

Figure 5. Comparison of anonymous communication throughput at the user (5(a)) and service provider (5(b)) for RECAP, PEREA [32], and BLAC [31]. Note that data points for BLAC were extrapolated from figures in the original publication. We do not consider the effects of rate limiting in PEREA and BLAC.

Anonymous Communication Once a contract is established, no Flicker sessions are needed to anonymously endorse messages by the user. We do not protect the user's private group signing key within Flicker because it is not required for the security of the system (although it *is* the user's responsibility to safeguard their keys)[5]. Consequently, the common-case operation of RECAP is efficient. On average, message endorsement takes 86 ms ± 0.4 ms on the user's platform, and message verification takes 87 ms ± 0.2 ms on the SP's platform. Note that our implementation is not actually checking for revoked users. Thus, these measurements are very close to what the $O(1)$ revocation scheme would yield. In this scheme, a small number of messages are linkable (see Section 8.4). Figures 5(a) and 5(b) show that the endorsement throughput of RECAP scales well with the size of the blacklist $|BL|$ for both the user and SP. Table 1 compares the asymptotic and empirical performance measurements. We use the numbers reported in prior works [31, 32].

Note that PEREA and BLAC both require additional rate limiting not shown in Figures 5(a) and 5(b), which limits the throughput between a particular user and SP. For instance, in PEREA, users must be rate-limited to k authentications per detection time. For $k = 30$ and a detection time of one hour, this yields only 30 authentications per hour. RECAP does not require rate limiting, and a particular user and SP can authenticate approximately 40,000 times per hour.

Clearly, there is a performance-unlinkability tradeoff between our implementation of RECAP and existing systems such as PEREA and BLAC. RECAP scales extremely well, however it does so in part by utilizing a $O(1)$ revocation scheme which favors performance over perfect unlinkability. Thus, existing systems may be preferable when performance is not an issue, and when perfect unlinkability is required. RECAP is better suited for services with high rates, and in which a small number of linked messages does not harm the user. However, subjective-based systems like PEREA and BLAC suffer from several factors unrelated to performance (which have been already discussed in Section 5), including the lack of guarantees for well-behaved users and the difficulty of finding practical identifiers.

Registration We have measured the end-to-end time it takes for a user to negotiate a contract using the registration protocol. Although the contract negotiation protocol takes $O(|BL|)$ time between the AS and SP to determine if the user is on the blacklist, the total time is largely dominated by the time it takes to execute the TPM_SEAL and TPM_UNSEAL commands. The blacklist would have to be impractically large for the linear time component of the runtime to have any impact on the total runtime. In our implementation, contract negotiation takes 7.99 ± 0.04 s. Although this may seem like a long time, this protocol only executes when a user registers to use a new service, or the user and SP negotiate a new contract.

[5] It is straightforward to put the user's private group key inside Flicker at the cost of invoking a Flicker session for every group signature operation.

System	Auth. (U)	Auth. (SP)	Auth. (U)	Auth. (SP)	Parameters
RECAP	86 ms	87 ms	$O(1)$	$O(1)$	
PEREA [32]	5900 ms	160 ms	$O(k\|BL\|)\dagger$	$O(k)$	$k_{SP} = 30, k_U = 10$
BLAC [31]	1450 ms	870 ms	$O(\|BL\|)$	$O(\|BL\|)$	

Table 1. Comparison of authentication time between RECAP and other systems for reasonable parameter choices ($|BL| = 800$). Measurements for PEREA and BLAC are taken from the relevant works, as we were unable to obtain the source code for these schemes [31, 32]. †: The amount of computation needed for PEREA is $O(k\Delta_{|BL|})$, but the actual time required to authenticate is $O(k|BL|)$ because of the risk of timing attacks. k is a window parameter used only in PEREA.

The majority of the time spent during registration is spent executing the TPM_UNSEAL command. Thus, by batching multiple requests together in a single Flicker session, the cost of unsealing data can be amortized to achieve improved throughput. It may also be possible to replace the use of the TPM's (relatively slow) sealed storage with its (relatively fast) non-volatile RAM facilities [23], though our current implementation does not support TPM NVRAM. We leave this for future work.

Breach Last, we also examine the end-to-end time for a SP to determine the identity of a misbehaving user. Our implementation of the breach protocol takes 0.32 ± 0.09 s on average from the time the SP detects a malicious message to the time it receives the user's identity from the AS, excluding the time needed to establish the secure channel as described in Section 4.

7.2 Trusted Computing Base (TCB)

RECAP has a relatively small trusted computing base that needs to run in the Flicker isolated execution environment [23]. Table 2 shows the number of lines of code in the TCB for the user and the AS. The majority of the code is the PBC cryptographic libraries for implementing group signatures, which also depend on portions of the GNU Multiple Precision Arithmetic Library. RSA and the symmetric cryptographic functions, as well as the TPM driver and supporting code for TPM_SEAL and TPM_UNSEAL also make significant contributions to code size. The actual logic for RECAP comprises a relatively small overall portion of the TCB, suggesting that formal verification or manual audit are realistic options. We also note that we have made no effort to strip unused content from the cryptographic and mathematical libraries. Significant additional reductions in code size are readily attainable. Even so, our entire TCB measures in a few tens of thousands of lines. This is orders of magnitude less than the TCB for code running on top of a commodity operating system.

Component	**Lang.**	**SLoC**
Flicker: User	.c/.S	953
Flicker: User	.h	1590
Flicker: AS	.c/.S	1173
Flicker: AS	.h	1549
Flicker: Shared		
Crypto / TPM	.c	4134
Crypto / TPM	.h	202
Crypto	.c	2698
Crypto	.h	1791
PBC	.c/.S	11527
PBC	.h	1160
GMP	.c/.S	4859
GMP	.h	5802

Table 2. Lines of code in the trusted computing base (TCB) of our implementation as measured by `sloccount` [33]. PBC = pairing based cryptography library. GMP = GNU multiple precision arithmetic library.

8 Discussion

8.1 RECAP as a Primitive

RECAP provides a mechanism for users to anonymously use a service, and thus it is a component in a larger, overall protocol stack. For example, RECAP may be run on top of TCP/IP, and as part of a larger chat protocol.

We only make guarantees about the RECAP component. For example, a user who types in their personal information to a chat service could circumvent any security otherwise offered from RECAP. Similarly, the chat protocol could run RECAP on top of TCP/IP, which may allow chat servers to log IP addresses. Although RECAP does not solve the complete protocol stack problem, RECAP can be used at each layer of the stack. For example, Tor is a widely-used network-level service that is intended to help create network-level privacy for higher-level services by preventing a network server from learning the IP address of a network client. Tor could use RECAP to enforce policies regarding proper use. A chat application could run on top of Tor, and use RECAP to provide contractual anonymity for chat sessions.

8.2 Attacks

Availability There are several potential attacks against RECAP. The first potential attack is that the AS could be powered off or otherwise made unavailable. An AS that is unavailable cannot reveal the identity of users who misbehave. There are several possible ways to counter this problem. An SP can insist upon an AS that has been designed with high availability in mind, e.g., an AS with redundant network links, power, etc. Attacks on availability are present in most protocols, and can be also addressed by standard methods in fault-tolerant computing and cryptography.

Small Groups In RECAP, user authentications are unlinkable among all other registered users in the group, e.g., the unlinkability set $\mathcal{S}_\mathcal{L}$ (from Section 3.1) is the set of registered users with the same SP and policy. If the group is small, and the SP knows this, then the SP knows that any two requests are likely to be from the same user. This concern is not unique to RECAP; the problem of small group sizes also occurs in closely related works [9, 10, 31, 32]. Because RECAP relies on trusted computing, it can mitigate the weakness of unlinkability for small groups: the AS can reveal the number of active users upon request. Each user can then create her own threshold for $|\mathcal{S}_\mathcal{L}|$ (e.g., the user may wish to be unlinkable among at least 20 users). If the number of active users is below the user's threshold, then the service should not be used. To the best of our knowledge, similar systems [31, 32] do not consider this weakness in their design. Note that in RECAP, a behaving user's real identity (e.g., trusted computing identifier) is known only to the AS, and thus is secret regardless of the number of active users.

Physical Attacks on TPM TPMs provide strong guarantees about programmatic or software-based attacks. However, TPMs are not designed to withstand a continued physical attack from a determined adversary. If a TPM is physically compromised, its security properties are lost. Since RECAP relies on those properties *if* TPMs are used to provide the desired trusted computing properties, it is important to consider how to mitigate physical attacks. There are several options. The first option is to do nothing; users would have to trust that the AS they use will not be physically compromised. There may be a popular AS (or several) that is believed to not be physically compromised. Note that the only requirement for this AS is that it should not be physically compromised; the operating system, BIOS, human operator, etc. can all be compromised. Second, RECAP can be extended to use threshold cryptography [17] such that a coalition of ASes are needed to reveal a misbehaving user's identity. If a user only uses ASes that are controlled by distinct entities, it would be difficult for an attacker to physically compromise all of the ASes. Last, the AS could use a secure coprocessor (like the IBM 4758 cryptographic processor [27]) that is designed to withstand physical attacks, instead of the TPM.

8.3 Policies and Unlinkability

In our explanation of RECAP, we have abstracted away some of the details of policies. There are two practical issues that arise in practice. The first problem is the unlinkability problem, or how to implement policies that rely on linkability. The second problem is how message matching should be implemented.

Unlinkability Problem In our description of RECAP, we have not specified how the SP knows when a policy is violated. If the policy only deals with one message, then it is simple: the SP can run the policy function locally on each message. For instance, a matching policy that forbids the string "badword" can be implemented using a function $f(msg_1)$: if $\textsc{HasWord}(msg_1,$ "badword") then $\textsc{Violation}$ else $\textsc{Allowed}$. Since the policy only needs to match one message, the SP can simply run $\textsc{HasWord}$ itself on each message.

However, consider a policy that forbids a user from including "badword" in one message, and "terribleword" in another message; it is okay to send one message, but not both. Such a policy might look like: $g(msg_1, msg_2)$: if $\textsc{HasWord}(msg_1,$ "badword") and $\textsc{HasWord}(msg_2,$ "terribleword") then

Violation else Allowed. Consider what happens if the SP receives two messages: "...badword..." and "...terribleword...". The SP can easily verify that the two messages would violate the policy if sent by the same user, but cannot determine if they were sent by the same user because of the unlinkability property. We call this the unlinkability problem.

The obvious implementation of policies with the unlinkability problem is not always the most efficient. As one example, threshold policies are commonly used to prevent spamming, e.g., users should not send more than k messages per day. Unlinkability prevents the linking needed for the SP to easily count how many messages each user sent. The obvious RECAP implementation, in which the SP sends a large set of messages *suspected*[6] to be created by the same user to the AS, is clearly very inefficient. A more efficient implementation is to incorporate k-times anonymous authentication [28], a cryptographic protocol which provides efficient enforcement of threshold policies. This allows the SP to efficiently detect when a user exceeds her threshold without excessive communication with the AS. The RECAP anonymity guarantees still hold since k-TAA provides unlinkability for users who have not exceeded their thresholds (e.g., broken their contract).

Unfortunately, we are not aware of a more general solution to the unlinkability problem. For instance, one useful type of policy that we currently can not implement efficiently is finite state machine-based policies. We leave solutions for finite state machine-based policies and the more general unlinkability problem for future work. One potential research direction would be to allow the SP to run an efficient trusted computing supported hypervisor which could perform the necessary linking in a secure, isolated environment [21].

Policy Message Matching As has already been stated, a contract policy can be thought of as a function $f : \{msg_1, \ldots, msg_n\} \rightarrow \{\text{Allowed}, \text{Violation}\}$. However, if the SP submits 50 messages to the AS for a policy that takes only four messages, it is not clear how the correct messages are assigned to the inputs of the policy function. Even for this example, there are $\binom{n}{50} = 230300$ possible assignments of messages, assuming order does not matter; clearly, it is not practical to simply try all combinations.

Instead, when the AS receives signed messages, it should partition them into sets based on the user that signed them (using GS_Open). Then, the policy function is executed on each set of messages. This means that, when implemented, policy functions take a set of messages as input, and should not assume that only the correct messages are included. For instance, the matching policy described above could actually be implemented as in Figure 6.

```
$foundone \leftarrow$ false
$foundtwo \leftarrow$ false
for all messages $m$ do
    if HasWord($m$, "badword") then
        $foundone \leftarrow true$
    end if
    if HasWord($m$, "terribleword") then
        $foundtwo \leftarrow true$
    end if
end for
if $foundone \wedge foundtwo$ then
    return Violation
else
    return Allowed
end if
```

Figure 6. An Example Implementation of a Matching Policy

8.4 Verifier-local Revocation

In the group signature scheme we use there is a trade-off between unlinkability and the runtime of GS_Verify in the size of the blacklist [8]. Verifying that a message signer is not on the blacklist can be performed in $O(1)$ time by the SP if the scheme allows for a small proportion ϵ of messages signed by the same user to be linkable ($\epsilon \approx \frac{memory}{\#users}$), and in $O(|BL|)$ time for perfect unlinkability (ϵ is negligible).

In the $O(1)$ scheme, ϵ is controlled by the security parameter $k : \epsilon = \frac{1}{k}$. However, the SP must maintain a precomputed lookup table whose size is $O(k * |BL|)$, and therefore there is a linkability-memory tradeoff.

As an example, if $|BL| = 1024$, $k = 1024$, each table entry in a precomputed lookup table is about 128 bytes long, and the SP devotes 128MB to create a lookup table, then less than 0.1% of the messages sent by the same user can be linked by the SP. In RECAP, the SP, AS, and users will all know which scheme is used, and thus will know whether there is a chance messages will be linkable. Security-conscious users can always insist on using services that rely on the $O(|BL|)$ algorithm.

[6] For instance, all the messages sent during a high-traffic period.

9 Related work

Group signature schemes are often motivated by the need for anonymous authentication [2, 5–8, 13–15]. Group signature schemes typically assume a group manager. The group manager is trusted not to reveal the secret keys of group members. Our system provides a way of intelligently placing such trust. In RECAP, the AS becomes the group manager, but all parties can verify that the manager will act appropriately.

Several researchers have proposed schemes for anonymous authentication that do not involve a trusted third party (TTP). The most basic of these are e-cash schemes [3,4,11,26,28,29], and k-times anonymous authentication schemes [28]. Such schemes do not allow for richer contract policies, and are not appropriate for many types of Internet-based anonymous authentication.

Further, existing TTP-free schemes are less scalable than RECAP [9, 10, 31, 32]. Experimental results for BLAC [31] showed that the SP required 0.46 s of computation when the blacklist only contained 400 entries. Because an authentication must occur for each unlinkable message, these systems would not be practical for many applications. These schemes also do not achieve the property that anonymity is bound to a contract.

We use trusted computing so that the AS can be verified correct instead of simply trusted. In particular, we base our work on Flicker [23]. Datta et al. have proven that dynamic root of trust systems like Flicker allow verifiers to make strong conclusions about the software state on a machine performing an attestation [16]. Others have proposed using TPMs to help build anonymous authentication. For example, Direct Anonymous Attestation [9] can be used to anonymously attest to a software stack. However, these systems have slower performance, e.g., DAA and EPID require about 2 seconds of computation on the SP per authentication on a modern laptop [9, 10]. Further, these systems do not achieve all the contractual anonymity properties.

10 Conclusion

We introduced the notion of *contractual anonymity*, which provides strong guarantees for the user and service provider. Unlike other schemes, contractual anonymity requires a user and service provider to agree on a binding, immutable contract *before* the service is used. We designed the RECAP protocol to achieve the contractual anonymity properties, and implemented and evaluated RECAP to demonstrate that it is scalable and practical. Our end-to-end implementation of RECAP depends on a very small trusted computing base that excludes the operating system, BIOS, and DMA-capable devices, thereby enabling RECAP to use a *verifiable* third party to enforce contracts. Our experiments demonstrate that RECAP scales well, and is fully capable of supporting services with realistic message rates.

Acknowledgements

This research was supported by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by gifts from AMD and Intel. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, or the U.S. Government or any of its agencies.

We would like to thank our anonymous reviewers and our shepherd, Scott Coull, for their valuable feedback and suggestions. We also thank Bryan Parno and Thanassis Avgerinos for their comments and fruitful discussions.

References

[1] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[2] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO*, 2000.

[3] M. H. Au, S. S. M. Chow, and W. Susilo. Short e-cash. In *INDOCRYPT*, 2005.

[4] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In *Security and Cryptography for Networks*, 2006.

[5] M. Bellare, D. Micciancio, and B. Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT*, 2003.

[6] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, 2004.

[7] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *CRYPTO*, 2004.

[8] D. Boneh and H. Shacham. Group signatures with verifier-local revocation. In *CCS*, 2004.

[9] E. F. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS*, 2004.

[10] E. F. Brickell and J. Li. Enhanced privacy ID: a direct anonymous attestation scheme with enhanced revocation capabilities. In *Workshop on Privacy in the Electronic Society*, 2007.

[11] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In *Security and Cryptography for Networks*, 2006.

[12] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, 2002.

[13] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.

[14] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups (extended abstract). In *CRYPTO*, 1997.

[15] D. Chaum and E. van Heyst. Group signatures. In *EUROCRYPT*, 1991.

[16] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its applications to trusted computing. In *IEEE Symposium on Security and Privacy*, 2009.

[17] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO*, 1989.

[18] J. R. Douceur. The sybil attack. In *International Workshop on Peer-To-Peer Systems*, 2002.

[19] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2008.

[20] Intel Corporation. Trusted eXecution Technology – measured launched environment developer's guide. Document number 315168005, 2008.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of ACM SOSP*, 2009.

[22] B. Lynn, H. Shacham, and J. Cooley. PBC_sig group signature library. [Online]. Available: `http://crypto.stanford.edu/pbc/sig`. [Accessed: May 1, 2009].

[23] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. 1997.

[25] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *USENIX Security Symposium*, 1998.

[26] L. Nguyen and R. Safavi-Naini. Dynamic *k*-times anonymous authentication. In *Applied Cryptography and Network Security*, 2005.

[27] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. In *Computer Networks*, 1998.

[28] I. Teranishi, J. Furukawa, and K. Sako. *k*-times anonymous authentication (extended abstract). In *ASIACRYPT*, 2004.

[29] I. Teranishi and K. Sako. *k*-times anonymous authentication with a constant proving cost. In *Public Key Cryptography*, 2006.

[30] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. Version 1.2, Revision 103., 2007.

[31] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without TTPs. In *CCS*, 2007.

[32] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith. PEREA: towards practical TTP-free revocation in anonymous authentication. In *CCS*, 2008.

[33] D. A. Wheeler. Linux kernel 2.6: It's worth more! [Online]. Available: `http://www.dwheeler.com/essays/linux-kernel-cost.html`. [Accessed: May 1, 2009].

[34] XySSL Developers. XySSL cryptographic library. [Online]. Available: `http://polarssl.org`.