Contractual Anonymity
An INI Master's Thesis

Edward Schwartz

May 11, 2009

Contents

Acknowledgements

I would like to thank several people for their time, suggestions, and influence. Creating this thesis would not have been possible without them.

First, I would like to thank my thesis committee members, Drs. Adrian Perrig and Lujo Bauer. A short year ago, I was learning much of the knowledge and skills from these two professors that I used while creating my thesis. It seems very fitting to me that they are now also on my thesis committee. Thank you both for agreeing to be on my committee, and for your lasting influence in my academic career.

My sincere thanks also go to Drs. David Brumley and Jon McCune, who worked with me on a day to day basis to create this thesis. Their guidance and direction was instrumental in all parts of the thesis, from brainstorming the initial idea to helping polish the final written work.

I look forward to continue working with Drs. Brumley and McCune as well as Drs. Perrig and Bauer when I begin working on my PhD in the fall of 2009. Judging from how much they have impacted my life in the short time that I have known them, I am sure that I will owe many gratitudes to them in the future.

List of Figures

List of Tables

Abstract

Anonymity benefits a variety of applications, such as websites for whistle-blowers who want to report abuses without fear of reprisal, online support groups for sensitive issues such as victims of violence, and network privacy services like Tor. However, there is a tension between the need for anonymity for well-behaved users and the need to identify users who misbehave, e.g., by using the service to send spam or distribute malware. Unfortunately, in most current anonymity schemes, this balance is tipped in favor of the service providers. In particular, in most current schemes the service provider can take action such as de-anonymizing or blacklisting a user for any reason and at any time.

We propose and develop techniques for achieving *contractual anonymity.* In contractual anonymity, a user and service provider enter into an anonymity contract. The user is guaranteed anonymity and unlinkability from the contractual anonymity system unless they break the contract (though information that is external to the system, like network addresses, can leak identifying or linking information). Service providers are guaranteed that they can identify users who break the contract. Both parties are bound to the contract, and neither can change the contract without the other's permission. In particular and unlike other schemes, the service provider is bound to the agreed-upon definition of malicious behavior as stated in the contract. A 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.

We demonstrate our techniques for contractual anonymity by building a prototype system. In our system, anonymity of individual users is provided by group signatures. Users are guaranteed that their anonymity will remain intact (assuming they abide by the contract) via TPM (Trusted Platform Module) attestations to the correct operation of the identity-maintaining system components. Our approach does not require unjustified trust in third-party servers, and is significantly more efficient than previous related approaches.

Chapter 1

Introduction

There are a variety of reasons why users may wish to use a network service anonymously, and in turn why service providers may then wish to offer anonymous access to their service. Anonymous services enable users to discuss sensitive personal issues via a message board or chat room, such as victims of violence, cancer or AIDS patients, and child or spousal abuse information and support groups. Anonymous services also allow people to report abuses by governments and companies (i.e., whistle-blowing) without fear of retaliation. In some cases, a user may not have an a priori motivation, but may simply wish to retain some degree of personal privacy on the Internet as a matter of preference.

A service provider (SP), however, must be able to identify misbehaving users in order to protect their service. For example, a service may want to identify users that use chat rooms to threaten others, use anonymous networks for denial of service attacks, or send spam to message boards. The ability to identify misbehaving users is important even in otherwise anonymous services, since this may be necessary to keep the service functioning. There are a number of existing *anonymous authentication* schemes that are designed to allow users to authenticate anonymously and also allow the SP to disallow access to misbehaving users.

Unfortunately, there are currently no anonymous authentication schemes that simultaneously guarantee that: 1) the identity of legitimate users will remain anonymous and indistinguishable as long as they abide by a *pre-agreed* terms of service contract, 2) network providers can identify users who misbehave by violating the same contract, and 3) the contract is immutable and binding once all parties agree. In particular, existing schemes for achieving anonymous authentication either rely on a trusted third party [5, 10, 11], or *are* mutable and allow SPs to blacklist a user at any time and for subjective reasons [8, 24, 25]. An immutable contract is necessary for a user to be assured that the SP will not change the system in a way that affects that user's anonymity without her permission.

We propose the notion of *contractual anonymity*. In contractual anonymity, a user and a SP agree to a contract such that the user's anonymity is guaranteed as long as they do not violate the contract, and the SP is guaranteed that they can identify the user upon producing proof that the user violated the contract. However, neither party can change the anonymity contract at some later date without the other's consensus. Service providers and users must negotiate a new contract if either

side wishes to change the terms. Users who abide by the contract are guaranteed a SP learns nothing about their identity from CAP when they authenticate[1], and that their authentications are indistinguishable from other users' that have not broken their contract.

More specifically, we propose the following properties for a contractual anonymity scheme:

Anonymity The user can authenticate anonymously, i.e., users have an anonymous identity. No one can relate a user's real identity to her anonymous identity if she has not broken her contract.

Contract-based The user and SP enter in a contract, and both parties are bound by the contract. A contract specifies unambiguously the agreed-upon terms of service. In contract-based anonymous authentication, the rules for revealing a user's real identity are specified by the contract. Neither the user nor the SP can modify the contract; they must explicitly agree to a new contract if they wish to change the terms.

Unlinkability Separate authentications of a single anonymous identity that has not violated the contract cannot be correlated. For example, it should not be possible for the SP to attribute a group of authentications (e.g., logins or message signatures) that do not violate the contract to the same anonymous identity. Furthermore, a SP should be unable to provide special treatment to a user based on her past behavior, assuming that she has not broken her contract.

Revocability The SP is able to obtain a user's identity if the SP has proof that the user broke her contract. The SP can then take appropriate action, e.g., blacklisting the user.

Efficiency The protocol should be as efficienct as possible.

Our Approach In this thesis, we present the Contractual Anonymity Protocol (CAP), a protocol for achieving contractual anonymity with all of the above properties. At a high level, CAP enables a user and a SP to agree on an anonymity contract where the terms of the contract can be any boolean function. For example, the contract may stipulate that the user should not send packets that match known attack patterns. Once the contract is agreed upon, the user receives an anonymous identity.

[1] In a contractual anonymity system, every unlinkable message that is sent to the service provider must be separately authenticated, and thus such a message is called an authentication. Alternatively, an authentication can be thought of as an authenticated message.

An anonymous identity (e.g., set of credentials) is not linkable to the real user. The service provider is given proof that they can recover the user's real identity from the anonymous identity if the contract is broken.

In CAP, we base anonymous credentials on a cryptographic primitive called group signatures. Group signatures give CAP users cryptographically secure anonymity and unlinkability, and guarantee to the SP the ability to detect authentications from blacklisted users. We discuss how we use group signatures in detail in Section 2.3. At a high level, a group signature scheme allows any member of the group (a user in our scheme) to sign on behalf of the group. Individual members' signatures are indistinguishable from any other members' signatures.

However, group signature schemes are not sufficient to offer all of the properties required for a contractual anonymity system. We address this in CAP by leveraging trusted computing to construct a verifiable third party, called the *accountability server* (AS), that manages anonymous identities. The AS is a software module implementing an algorithm that will only reveal a user's real identity if the SP provides message(s) that prove the user has violated the contract.

The particular server that an AS runs on is *not* arbitrarily trusted by either the user or SP. Instead, remote parties trust that the CAP software is implemented correctly (e.g., that user identities will only be revealed when given proof of misbehavior), and the AS proves that it is running *that* software. This is accomplished by leveraging recent advances in trusted computing, including *remote attestation* and *dynamic root of trust*. We discuss these primitives in detail in Chapter 2.4. In particular, the AS proves via attestation that it is running known software in an isolated execution environment with an extremely small trusted computing base (TCB) (e.g., leveraging dynamic root of trust to eliminate all but a few hundred lines of additional trusted code beyond the logic of the AS itself), and that they are interacting directly with the AS's TCB. The TCB code can be verified by anyone (e.g., external security experts) to show that it returns a user's real identity if and only if a contract is broken. Thus, the SP and user in our protocol receive proof that the AS will enforce exactly the desired contract.

Previous Approaches There have been several previous approaches to anonymous authentication. However, none have satisfied the requirements of contractual anonymity. At a high level, these approaches differ from CAP by not binding anonymity to a pre-negotiated contract. For example, many existing anonymity systems require a trusted third party (TTP) that is capable of deanonymizing or linking users [6,10,11]. These systems differ from ours in that our AS is constrained to enforce only the terms

of the contract; previous work allowed the TTP either to link authentications through blacklisting at any time or deanonymize users for any reason.

Recently, a number of anonymous authentication and blacklisting schemes have been proposed that do not require a TTP [8, 24, 25]. However, these systems provide no way to constrain the conditions for a SP to link a user through blacklisting, i.e., they allow the SP to subjectively judge malicious behavior. We can also allow for subjective judging. However, contractual anonymity is also capable of providing a stronger sense of security for the user, because the user and SP can agree to the conditions for breaking unlinkability guarantees before the user commits to using the service at all.

Finally, CAP is much more efficient than previous competing approaches. Based on our measurements (Chapter 4.2), a single machine can perform about 10^6 CAP authentications per day. A user of PEREA [8, 24, 25], the most closely related work, can only carry out only about $2 * 10^4$ anonymous authentications per day (about 2% of the number we can carry out)[2]. These limitations motivate the creation of a new protocol that addresses these challenges.

Caveat: CAP, like previous anonymity schemes, provides a building block for users to anonymously communicate with a SP. It does not automatically take care of anonymizing the complete protocol stack; see Chapter 5.1 for details.

Caveat: CAP provides strong anonymity guarantees at the authentication level. Although contractual anonymity provides stronger guarantees against SP misbehavior than previous work, it does not make any guarantees about the behavior of SP backend software. For instance, while a SP cannot blacklist a user for posting a message allowed by the contract policy to an anonymous message board, the SP can still remove that posting for subjective reasons.

Contributions

Contractual Anonymity We introduce the concept of contractual anonymity. In a contractual anonymity system, users are guaranteed anonymity as long as they do not violate the policies of their predetermined contract with the SP. The SP is guaranteed that it can learn a user's real identity (and thus take appropriate action such as blacklisting) if the user breaches the contract.

Contractual Anonymity Protocol We design the Contractual Anonymity Protocol (CAP), which is the first protocol that provides contractual anonymity.

[2]For PEREA parameters K=30, and T = one hour.

Efficient Implementation We implement a prototype of CAP and show that our system is more efficient than previous competing approaches [7, 8, 24, 25]. Our system is designed to be used with many anonymous applications, including ones with moderate authentication rates (around 10^6 unlinkable messages per day), and does not have significant rate-limiting or scalability problems.

Verifiable Escrow Unlike systems that simply assume a trusted third party (TTP), our system runs the AS software in an isolated and verifiable execution environment. This enables users and SPs to decide whether they want to trust the AS based on the software it is running.

Chapter 2

Design Overview

2.1 CAP Overview

At a high level, the Contractual Anonymity Protocol is between three parties:

The Service Provider (SP) The SP wants to provide an anonymous service, such as an anonymous message board, chat room, or network. For now, we assume the SP defines the contract policy, since this situation naturally maps to current networks where the SP decides the terms of service. (Alternatives for contract negotiation are discussed in Chapter 5.3.) In CAP, the SP is guaranteed the ability to obtain the identity of a user that has broken an agreed-upon contract.

The User (U) Users wish to partake in the anonymous service offered by the SP, e.g., to be able to post anonymously to a message board. When users agree to the contract policy, they are given an anonymous identity. The identity serves as credentials which allow them to anonymously use the service. Users are guaranteed to remain anonymous as long as they do not break the contract. In our system, a user's real identity is the unique certified, public endorsement key associated with their computer's trusted platform module (TPM). A user's real identity is revealed to the SP if she breaks her contract.

The Accountability Server (AS) In CAP, the AS serves as a verifiable third party for enforcing contracts. Specifically, the AS 1) issues anonymous credentials to the user as described above, and 2) reveals the user's real identity if and only if given proof that the user violated her contract. We allow the AS to be a distinct entity so that a SP can operate its own AS, or a single AS can provide anonymous identity services for many SPs. A mapping from each user's anonymous credentials to her real identity is stored on the AS, so that the AS can determine a malicious user's real identity after being given evidence of misbehavior. Further, users and SPs can remotely verify that the software running on the AS will only reveal this mapping if given proof that the user has broken her contract.

An overview of CAP is shown in Figure 1. There are three stages in the protocol: registration, anonymous communication, and contract breach. These phases are discussed in much greater detail in Chapter 3.

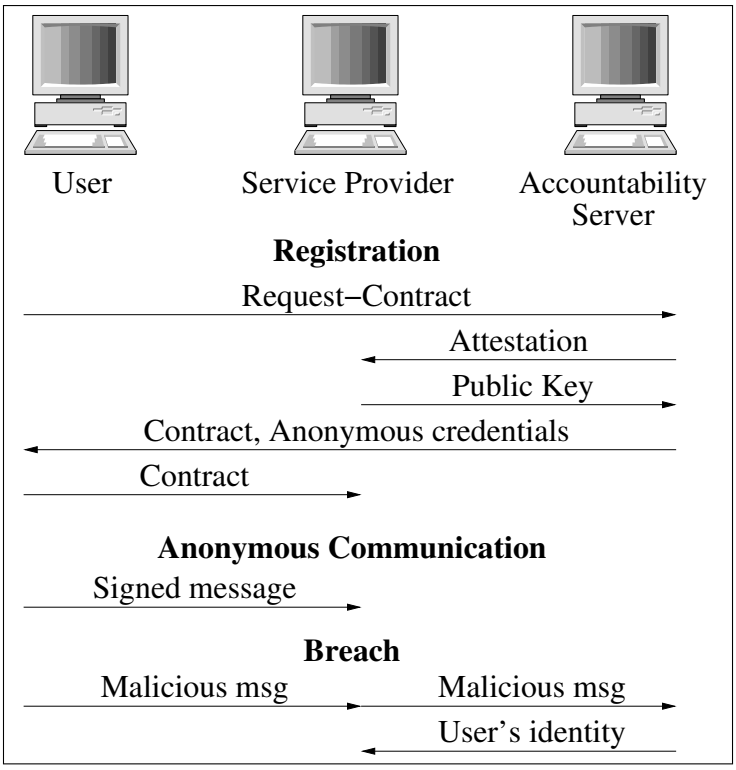

Figure 1: The three stages of CAP: registration, anonymous communication, and breach.

Setup Before the protocol starts, all the participants must engage in some form of setup. The user, AS, and SP must all generate public/private keypairs that can be used for digital signatures and asymmetric encryption. The SP must also obtain a certificate that binds its identifying name to its public key. The AS must run the group signature key generation algorithm for each group that is needed.

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 adhere to. For example, the contract policy may specify that users should not send known attack messages. We discuss policies further in Section 2.2.

In the default CAP implementation, the user receives a contract policy proposed by the SP. If she agrees to the policy, then she requests a contract with that policy from the AS. (Alternatives to this arrangement are discussed in Section 5.3.) The AS returns an anonymous identity and a contract for the user to be used with the SP. The contract is a statement that the AS has bound the user's real identity, anonymous identity, and the contract policy. It provides assurance to the SP that the AS knows the true identity of the user with the anonymous credentials, and will reveal the identity if given proof that the user has broken the policy in her contract (we often refer to this as simply breaking or breaching the contract). As mentioned previously, we use group signatures to implement anonymous identities. We describe group signatures and how CAP uses them below in Section 2.3.

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 evidence (i.e., a set of messages) that the user violated the contract.

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

Authenticated messages are both anonymous and unlinkable. For this reason, we also refer to communicating messages to the SP as performing a separate *authentication*. An anonymous communication operation is analogous to the authentication operation of an anonymous authentication protocol.

Breach of Contract Phase A breach of contract happens when the user sends messages prohibited by the contract policy to the SP. The SP can identify which user violated the contract by presenting the prohibited messages to the AS. Upon confirming that the messages violate the agreed-upon contract, the AS reveals the user's identity to the SP, and also allows the SP to identify any subsequent and prior communication using the revealed anonymous credentials.

At the end of the breach phase, the SP has the capability to identify the user that breached their 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. However, CAP can certainly be extended to support other actions, such as anonymous blacklisting, in which SPs are given the ability to blacklist users without needing to know their identities.

2.2 Contract Policies

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

A contract policy is a Boolean predicate $f : \{msg_1, msg_2, \ldots, msg_n\} \rightarrow \{\text{BREACH}, \text{OK}\}$. The status BREACH indicates that the messages violate the contract terms, and thus the user is in breach of contract. OK indicates that the messages do not violate the policy.

Recall that CAP allows for messages by the same user to be unlinkable. Note that f does not reveal which user created the messages, but only whether the included messages are in violation of the policy. This allows the SP to run f without the aide of the AS. However, the SP may have difficulty enforcing policies that require analyzing multiple messages from the same user, because the SP cannot determine if the messages were created by the same user or not. Instead, if the SP suspects that the messages violate the policy and were created by the same user, it must send them to the AS which can determine if they were created by the same user using the AS's group manager abilities. As an example, CAP has difficulty enforcing threshold policies, which are described in Section 5.4.

One advantage of contractual anonymity is that contract policies are immutable once both parties have agreed to the contract. In contrast, other anonymity schemes such as PEREA [25] allow the SP to decide at any time that a user is misbehaving and blacklist them.

Filter-Based Policies One type of policy may be to disallow messages that are known to be malicious. For example, the SP may have an intrusion detection system that implements a set of rules for determining when messages are malicious. Note that such a policy should have one-sided error, e.g., never mistake a safe message for a malicious one. Previous work has shown how to automatically generate rules that have one-sided error, e.g., researchers have shown [9, 12] how to automatically generate rules that will only match exploits for a known vulnerability, and never match a safe input.

Digital Signature Policies Another type of policy could be to allow privileged users to map an anonymous message to a real user. For example, an anonymous message board intended to provide anonymity to victims of cancer could be misused to post unrelated content anonymously (e.g., posting terrorist threats). Such a service may wish to allow law enforcement the ability to relate particular posts to individuals. The SP for the service can achieve this by specifying in the contract the ability for a pre-arranged key belonging to law enforcement to retrieve the mapping between anonymous ID and user ID. Note that the users of the service would have had to explicitly agree to such a policy as a term for using the service. We can use digital signature policies to implement subjective judging [24] by specifying that a pre-arranged key belonging to the SP can be used to deanonymize a user.

2.3 Anonymity and Group Signatures

CAP uses group signatures [2, 5, 6] to implement anonymous identities. We describe group signature schemes and show how we use them to implement anonymous identities.

Group Signatures Group signature schemes provide anonymity among members of the group. Each group member has a private signing key which allows them to sign messages on behalf of the group. We require the group signature scheme to have a designated group manager [2, 5, 6]. Group signature schemes provide anonymity because given a signed message, it is computationally infeasible for anyone other than the group manager to determine which group member signed the message.

A group signature scheme suitable for CAP has four procedures: *GS_KeyGen*, *GS_Sign*, *GS_Verify*, and *GS_Open*.

GS_KeyGen(n, k) The *GS_KeyGen* algorithm takes in a security parameter k and the number of group members n. The algorithm outputs a group public key K_{GPK}, the group private key K^{-1}_{GMSK}, and a list of n group member private

keys $K_{GSK}^{-1}[i]$. In a regular group signature setting (but not in CAP), the group member private keys and group public key are distributed to group members. The group manager retains the group private key K_{GMSK}^{-1}.

GS_Sign$(K_{GPK}, K_{GSK}^{-1}[i], M)$ *GS_Sign* signs a message M with the group member i's private key $K_{GSK}^{-1}[i]$.

GS_Verify$(K_{GPK}, M_{K_{GSK}^{-1}[i]}, BL)$ *GS_Verify* ensures that the given signed message $M_{K_{GSK}^{-1}[i]}$ has a valid signature for message M. We use a variant of group signatures that allow for verifier-local revocation [6]. In a verifier-local revocation scheme, *GS_Verify* also checks if the user who signed $M_{K_{GSK}^{-1}[i]}$ is on the list of blacklisted users BL. Note that the verify algorithm cannot add members to the BL without possessing a special revocation token that must be released by the group manager, and cannot distinguish between signatures made by members not on BL.

GS_Open$(K_{GMSK}^{-1}, M_{K_{GSK}^{-1}[i]})$ *GS_Open* determines which group member signed message M and outputs a revocation token for that user. The revocation token can be distributed to group members and added to their BL.

Group signature schemes, and thus authentications in CAP have the following security properties:

Correctness Signatures produced by a group member using the *GS_Sign* algorithm are accepted as valid signatures by the *GS_Verify* algorithm, as long as the group member is not on the blacklist.

Unforgeability It is computationally infeasible for an adversary who is not a group member to produce a signature that is accepted by the *GS_Verify* algorithm.

Anonymity It is also computationally infeasible to determine which member of a group created a particular signature without the use of K_{GMSK}^{-1}.

Traceability By using the *GS_Open* algorithm, the group manager can always identify at least one member of a coalition of one or more dishonest members that collude to produce a signature.

Unlinkability It is computationally infeasible to determine if two messages were signed by the same group member without possessing K_{GMSK}^{-1} or the signer's revocation token.

Exculpability Group member i cannot create a signature $M_{K_{GSK}^{-1}[i]}$ such that *GS_Open* $(M_{K_{GSK}^{-1}[i]}) = j$ if $i \neq j$.

Readers interested in the security arguments are invited to consult the appropriate references [5, 6].

Anonymous Identities We use group signatures to create anonymous identities. Each user is a member of a group for her particular SP and contract policy. The AS is the group manager. Our use of group signatures achieves the desired properties (from Chapter 1) of anonymity, unlinkability, revocability, and efficiency.

The CAP protocol provides anonymity because each message endorsed by a user is a group signature, and signatures among (unrevoked) group members are indistinguishable. The endorsement signatures also provide unlinkability between messages, even if those messages are signed by the same user. Users' keys can be revoked via the *GS_Open* algorithm. Finally, group signatures are efficient. For example, we use a group signature scheme [6] that requires about 8 exponentiations (and an additional 2 bilinear map computations) to sign a message. Verification takes 6 exponentiations. Verification with revocation with perfect unlinkability can be performed in $O(|BL|)$ time, while revocation which slightly decreases unlinkability (see Section 5.5) can be done in $O(1)$ time.

2.4 Trusted Computing and Contract Enforcement

CAP meets several requirements for a contractual anonymity scheme by leveraging trusted computing, specifically the contract-based and efficiency properties. The contract-based property is achieved by using a verifiable third party that can securely bind user identities to contract policies and convince remote parties that it will do so. As a side-effect of using trusted computing, we avoid the need for Zero-Knowledge Proofs in CAP, since an AS can prove that it will not misuse sensitive information. This makes our system efficient and scalable. These properties can be achieved using technologies that are available on recent commodity platforms [1, 17, 23]. Specifically, trusted computing is used (1) to execute sensitive code in an isolated, verifiable environment on the AS's and user's platforms; (2) to cryptographically *seal* data so that it is only available to specific code that executes within the isolated environment; (3) to *attest* to a remote party what code executed within the isolated environment, and its inputs and outputs; and (4) to provide unique identifiers for the AS and users.

Isolated and Verifiable Execution The AS is a third-party server that users must trust to protect their anonymity, and SPs simultaneously trust to reveal misbehaving

users' identities. CAP builds trust in the AS by executing its critical components (i.e., those which handle users' true identities) in an isolated, verifiable environment. The isolation is achieved using the *dynamic root of trust* primitive [16] available on commodity systems today [1, 17]. This primitive reinitializes the platform into a known trusted state and records a measurement (cryptographic hash) of the code that will be executed in the isolated environment. Once the execution has been recorded, *sealed storage* and *remote attestation* become possible.

Protecting Sensitive Data Sensitive data is protected using TPM-based *sealed storage*, whereby data can by encrypted such that subsequent decryption is only possible if the platform is executing specific software. Thus, AS code can seal the mapping between users' true identities and their anonymous credentials such that only that same AS code running in the isolated execution environment will be able to unseal (decrypt) it.

Proving the AS Behaves Properly to a Remote Party One system can prove to another that it has loaded certain code for execution within an isolated environment using TPM-based *attestation*. An attestation demonstrates to a remote verifier that the platform in which a particular TPM resides has instantiated an isolated execution environment with a particular code module. In CAP, this is the AS's identity-mapping module.

A remote verifier can make a trust decision regarding the operation of the attesting AS based on its knowledge of the attested software. We expect that a known-good implementation of CAP will be released. Rather than releasing a formal proof that the implementation is correct, we expect that this implementation will be endorsed by security experts to have the desired properties: that a user's identity will be released if and only if there is evidence that the user broke her contract. This enables users to achieve a similar level of control over their identity as in a TTP-less scheme, because they can control what code has access to their identity. SPs gain a similar level of control over the code that can issue anonymous credentials allowing access to their service. The verification process takes place during the establishment of the secure channel in CAP (see Section 3.2.1).

We denote the process of creating an attestation of the currently running code module with input i as *Gen_Attest*(i). A third party can compute the value an attestation should have for *code module* running with input i as *Ver_Attest*(*code module, i*). A remote party can verify an attestation by confirming that the attestation's value is correct, and then verifying that there is valid signature on the attestation from a

key whose private component is known only to a TPM.

Unique Identities For a contractual anonymity scheme to be deployed, users must have a unique identity that they cannot readily change. If users could change their identity, they could register the new identities at will to bypass blacklisting. CAP uses the Endorsement Credential [23] that is included in Trusted Platform Modules (TPMs) as the globally unique identity for the users. It is this identity that the AS keeps in escrow. If a user violates their contract, this identity is released to the SP, which can add it to its blacklist to prevent the user from obtaining new anonymous credentials. Alternatively, users' identities can be readily tied to some other unique identifier (e.g., driver's license or social security number) if desired. We discuss this further in Section 3.3.

Chapter 3

Architecture

We now describe CAP in detail. We describe the setup and roles of the user, service provider, and accountability server, and also layout the protocols that they use.

3.1 Setup

Before the user, service provider, and accountability server begin participating in CAP, they must complete the necessary setup steps. The user and the AS must a generate public/private keypair that is valid for digital signatures and asymmetric encryption. The private components of these keys must kept secret to the trusted CAP code, however, so they must be generated and sealed using the TPM when the trusted code is running in isolation. The user and AS do not need certificates other than their TPM endorsement key certificates; they are able to convince others of their public keys using attestation.

The SP must also generate a public/private keypair, but does not do so inside the trusted execution environment. Since the SP lacks any TPM support, it must also obtain a certificate that binds its name to a public key (e.g., a commodity SSL certificate).

The AS is also responsible for generating group signature keys for each contract policy and SP combination in use. As with the private components of the signature/encryption keypairs, the group manager secret key and group private keys must be generated and sealed in the isolated environment. Although there are a number of ways to arrange for this to occur, CAP's standard behavior is to generate a new group when a user is requesting a contract for a previously unseen (SP, contract policy) tuple.

3.2 Operation of the System

We now describe how a secure channel is established between parties, and then describe the phases of CAP. Figure 1 gives a depiction of the different phases (registration, anonymous communication, and breach).

3.2.1 Establishing a Secure Channel

Many parts of CAP rely on the ability to create a secure channel between the protocol participants. Although the general problem of creating a secure channel between participants possessing authenticated public keys is well understood, CAP must create

a secure channel between actors whose corresponding private keys are sealed (using the TPM) so that only the CAP software can access them, which requires a slightly different protocol. Specifically, a CAP 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 CAP software can access it. These combined properties 1) allow messages to be delivered confidentially to the trusted CAP software without being read or modified (even though they must pass through untrusted components such as the operating system), and 2) allow a remote party to certfify that messages received from the channel could only have been generated by the trusted CAP software. This is needed when communicating with the AS, because the user and SP's trust in the AS is based on the trusted CAP software.

Before the secure channel is established, participants with a TPM (e.g., the user and AS) must possess an endorsement key certificate issued by a trusted manufacturer that indicates that the TPM is created by that manufacturer. The user and SP must have a list of known trusted code signatures for the AS, and the AS must have a list of known trusted code signatures for the user. All the trusted code signatures should be written in such a way that the trusted private key they utilize is sealed so that only the trusted code can access it. All participants must have a public/private keypair that can be used for asymmetric encryption and digital signatures. After the secure channel is established, any participant communicating with a TPM-equipped participant will believe that the TPM participant's private key is sealed so that only the trusted code can access it, because the AS has proved it is running code that is trusted to do this. Because of this, the remote party can infer that any data encrypted to the corresponding public key will be known only to the trusted code, and any message signed with the private key could only have been created by the trusted code.

We now explain the secure channel establishment in the registration protocol between the user and AS (Figure 2, Lines 4–17). A secure channel is established in the breach protocol as well, but the process is very similar (Figure 4, Lines 1–11).

On Lines 4–5, the user generates an untrusted nonce[1] and sends the nonce and its public key to the AS. The AS generates its own nonce and creates an attestation to prove that is running the CAP software in response to the user's request (Lines 6–7).

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 execu-

[1]The nonces in CAP are known by the untrusted software. This is in contrast to the random numbers, which are known only by the trusted part of the code.

tion 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 the AS keeps security-sensitive data like K_{AS}^{-1} sealed so that only CAP 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 TPM Endorsement Key Certificate (Line 8). At this point, the user verifies that the attestation is correct; if it is not, she aborts the protocol (Lines 9–10). 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 11). In response, the AS sends $R_U + 1$ and its own random number (Line 12). When the user receives its incremented random number, it believes that K_{AS} is bound to the trusted CAP code, and is willing to send its Endorsement Key (EK) Certificate encrypted under that key as part of an attestation, because the CAP code will only disclose the user's EK if the user breaks her contract. The user generates an attestation which provides similar properties to the AS, and sends it with its EK certificate, and the AS's incremented random number (Lines 13–14). Upon receiving the incremented random number, the AS verifies the user's attestation (Lines 15–16), and both parties then switch to more efficient symmetric cryptography (Line 17). This can be done with standard techniques [20].

3.2.2 Protocol Phases

We now present CAP in detail. The protocol is split into three parts: the registration, anonymous communication, and breach phases. The registration phase is required before a user can start sending messages to the SP. The anonymous communication phase serves to mark messages as coming 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 CAP begins with the user connecting to the SP (Line 1 in Figure 2). The user does 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 (Lines 2–3). 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.

To obtain a contract, the client connects to the AS and begins the U-AS protocol (Line 4). As was described in Section 3.2.1, the user and AS establish a secure channel (Lines 4–17). Once the channel is established, the client sends 1) the contract policy that the SP requires, and 2) the address of the SP (Line 18). The AS maintains a list of users that have been blacklisted by the SP, and aborts if one of those users is attempting to register (Line 19).

At this point in the protocol, the AS connects directly to the SP and executes the key binding protocol (Line 20). This allows the SP to ensure that the AS is running the CAP software, and to verify that K_{AS}^{-1} is bound to that software. This convinces the SP that a user's identity will be revealed if she breaks her contract. The AS also learns the SP's public key during this protocol. This protocol is expressed in detail in Figure 3, and is actually a subset of the messages needed to establish a secure channel (Section 3.2.1).

After the key binding, the AS will proceed to create a contract. The contract consists of the contract policy the user agrees to, the public key of the group signature group that the user is part of, and the SP's public key. The AS sends the contract and a group private key to the user (Figure 2, Line 21). Finally, the user sends the contract to the SP, and she is ready to start endorsing messages (Line 22). The SP ensures that the contract is signed by a key that it knows about from the AS-SP key binding protocol (Figure 3).

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 22). When the SP receives a signed message, it ensures that it has received a valid contract that included that group public key. The SP also verifies that the message has a valid signature by executing the group signature verification operation (Line 23).

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 (Line 14). The AS obtains the group private key[2] that violated the contract, by using the *GS_Open* operation (Line 15).

[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.

It then reveals 1) the user's group signature revocation token, and 2) the user's TPM Endorsement Key Certificate 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 her messages will no longer be accepted.

3.3 Features

Contract + Unlinkability = Anti-discrimination CAP prevents a SP and its AS from discriminating against anonymous users based on their past messages. Previous systems with TTP's have not appropriately limited the power of the TTP to revoke [6, 10, 11], thus the TTP could potentially revoke well-behaved users. For example, someone could compromise the TTP and revoke users, or bribe the TTP itself to misbehave. Thus, such systems can discriminate.

Previous TTP-free systems allowed subjective judging [24, 25], 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 for any reason whatsoever. For example, a user could post a message the SP simply decides it does not like, and the SP would be free to block all future authentication. Thus, the SP could discriminate against a user without knowing their real identity in such systems.

In CAP, anyone can verify that the AS will only reveal a user's identity if their contract is violated. Further, the AS seals the contract, which results in an encrypted blob that can only be decrypted when the trusted CAP code is running in a verifiable execution environment. Thus, even if the untrusted part of the CAP software, the operating system, or the BIOS is compromised, the AS cannot reveal a behaving user's real identity. A SP cannot discriminate against users not in breach of contract because multiple authentications are unlinkable; there is no way to link multiple anonymous authentications of a single user.

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. CAP 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 their TPM's Endorsement Key (EK) Certificate C_{TPM-U} with the AS and receive an anonymous credential $K^{-1}_{GSK}[i]$. During contract breach, the AS is provided with a sequence of signed messages that violate the contract. A blacklist will

contain both $K_{GSK}^{-1}[i]$ (so that subsequent messages from the user can be identified) and C_{TPM-U} (so that the blacklisted user cannot ask another AS to create a new credential).

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

Solving the Sybil Attack In the Sybil attack [15] a user can subvert security by forging new identities. In our system, users cannot create new identities themselves without breaking the security of group signatures. Thus, in our setting a Sybil attack corresponds to a user successfully receiving a new identity from an AS, since the newly forged identity could bypass the blacklist.

To the best of our knowledge, no other implemented anonymous authentication system has solved this problem in a practical manner. For instance, in PEREA [25], a suggested method is for the user to register with the SP by presenting her driver's license in person[3]. However, we argue that this is impractical for typical network services.

Our architecture solves the Sybil problem by binding the anonymous identity to the the unique endorsement key found in each user's TPM as the user's identity. A user cannot practically obtain a new endorsement key without replacing the TPM (since the TPM is a physical device and there is no programmatic way to replace it). Thus, while an attacker can bypass blacklists by purchasing new equipment, she cannot do so without incurring relative expense.

[3]This still preserves anonymity because the registration is not linkable to future authentications.

	U-SP registration protocol
1. U $\rightarrow$ SP:	$\{message, contract = \perp\}$
2. SP:	if $contract = \perp$, execute Line 3, else Line 23
3. SP $\rightarrow$ U:	$\{Get\text{-}Contract, AS, CP\}_{K_{SP}^{-1}}$
	U-AS registration protocol
4. U:	$N_U \xleftarrow{R} \{0,1\}^\alpha, R_U \xleftarrow{R} \{0,1\}^\alpha$
5. U $\rightarrow$ AS:	$\{K_U, N_U\}$
6. AS:	$N_{AS} \xleftarrow{R} \{0,1\}^\alpha, R_{AS} \xleftarrow{R} \{0,1\}^\alpha$
7. AS:	$a \leftarrow Gen\_Attest(K_U \vert N_U \vert K_{AS} \vert N_{AS})$
8. AS $\rightarrow$ U:	$\{K_{AS}, N_{AS}, a, C_{TPM-AS}\}$
9. U:	$a' \leftarrow Ver\_Attest($*Trusted CAP Code*, $K_U \vert N_U \vert K_{AS} \vert N_{AS})$
10. U:	abort if $a \neq a'$
11. U $\rightarrow$ AS:	$\{\{R_U\}_{K_U^{-1}}\}_{K_{AS}}$
12. AS $\rightarrow$ U:	$\{\{R_U + 1, R_{AS}\}_{K_{AS}^{-1}}\}_{K_U}$
13. U:	$a \leftarrow Gen\_Attest(K_U \vert N_U \vert K_{AS} \vert N_{AS})$
14. U $\rightarrow$ AS:	$\{\{R_{AS} + 1, a, C_{TPM-U}\}_{K_U^{-1}}\}_{K_{AS}}$
15. AS:	$a' \leftarrow Ver\_Attest($*Trusted CAP Code*, $K_U \vert N_U \vert K_{AS} \vert N_{AS})$
16. AS:	abort if $a \neq a'$
17.	Setup symmetric encryption and MAC
18. U $\rightarrow$ AS:	$\{Addr_{SP}, \{Get\text{-}Contract, AS, CP\}_{K_{SP}^{-1}}\}$
19. AS:	abort if C_{TPM-U} on SP's blacklist
20. AS:	executes key binding protocol
21. AS $\rightarrow$ U:	$\{\{CP, K_{GPK},\ K_{SP}\}_{K_{AS}^{-1}}, K_{GSK}^{-1}[i]\}$
	U-SP anonymous communication protocol
22. U $\rightarrow$ SP:	$\{\{message\}_{K_{GSK}^{-1}[i]}, contract = \{CP, K_{GPK}, K_{SP}\}_{K_{AS}^{-1}}\}$
23. SP:	if $GS\_Verify(K_{GPK}, \{message\}_{K_{GSK}^{-1}[i]}, BL) = 1$, accept message, else abort.

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 17 are implicitly encrypted and MACed using symmetric cryptography.

AS-SP key binding protocol	
1. AS:	$N_{AS} \xleftarrow{R} \{0,1\}^{\alpha}$
2. AS $\rightarrow$ SP:	$\{K_{AS}, N_{AS}\}$
3. SP:	$N_{SP} \xleftarrow{R} \{0,1\}^{\alpha}$
4. SP $\rightarrow$ AS:	$\{K_{SP}, N_{SP}, C_{SP}\}$
5. AS:	$a \leftarrow$ *Gen_Attest*$(K_{SP}\|N_{SP}\|K_{AS}\|N_{AS})$
6. AS $\rightarrow$ SP:	$\{a, C_{TPM-AS}, \{\{R_{AS}\}_{K_{AS}^{-1}}\}_{K_{SP}}\}$
7. SP:	$a' \leftarrow$ *Ver_Attest*(*Trusted CAP Code*, $K_{SP}\|N_{SP}\|K_{AS}\|N_{AS}$)
8. SP:	abort if $a \neq a'$

Figure 3: The key binding protocol.

Breach protocol

1. AS:	$N_{AS} \xleftarrow{R} \{0,1\}^{\alpha}, R_{AS} \xleftarrow{R} \{0,1\}^{\alpha}$
2. AS $\rightarrow$ SP:	$\{K_{AS}, N_{AS}\}$
3. SP:	$N_{SP} \xleftarrow{R} \{0,1\}^{\alpha}, R_{SP} \xleftarrow{R} \{0,1\}^{\alpha}$
4. SP $\rightarrow$ AS:	$\{K_{SP}, N_{SP}\}$
5. AS:	$a \leftarrow \mathit{Gen\_Attest}(K_{SP}\|N_{SP}\|K_{AS}\|N_{AS})$
6. AS $\rightarrow$ SP:	$\{a, C_{TPM-AS}, \{\{R_{AS}\}_{K_{AS}^{-1}}\}_{K_{SP}}\}$
7. SP:	$a' \leftarrow \mathit{Ver\_Attest}(\mathit{Trusted\ CAP\ Code}, K_{SP}\|N_{SP}\|K_{AS}\|N_{AS})$
8. SP:	abort if $a \neq a'$
9. SP $\rightarrow$ AS:	$\{\{R_{AS}+1, R_{SP}\}_{K_{SP}^{-1}}\}_{K_{AS}}$
10. AS $\rightarrow$ SP:	$\{\{R_{SP}+1\}_{K_{AS}^{-1}}\}_{K_{SP}}$
11.	Setup symmetric encryption and MAC
12. SP $\rightarrow$ AS:	$\{m = \{\{message_1\}_{K_{GSK}^{-1}[x]}, \ldots, \{message_n\}_{K_{GSK}^{-1}[z]}\}\}$
13. AS:	$\forall msg_i \in m$, abort if $\mathit{GS\_Verify}(K_{GPK}, msg_i, BL) = 0$
14. AS:	abort if $CP(m) \neq 1$
15. AS:	$gid \leftarrow \mathit{GS\_Open}(message_1, K_{GMSK}^{-1})$
16. AS $\rightarrow$ SP:	$\{gid, \mathit{GidToEKcert}[gid]\}$

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.

Chapter 4

Implementation

We now describe our implementation of the CAP system, and note security-relevant implementation decisions.

We implemented CAP using two cryptographic libraries: the PBC_SIG group signature library [18] that implements the BBS [6] group signature scheme, and the XySSL library [27] 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. The BBS signature scheme is configured to use a "Type A" pairing that offers security similar to that of a 1024-bit RSA key [5]. We do not currently implement efficient verifier-local revocation for the group signature keys [6], because we are unaware of a publicly available implementation, although we intend to implement this ourselves in future work. We note that we plan on implementing a $O(1)$-time revocation scheme that adds a single table look-up per verification, which is unlikely to significantly change our performance measurements. Options for implementing such revocation are discussed in Section 5.5.

Portions of the code that execute on the user's and AS's platforms constitute the security-sensitive, trusted components of CAP. Our implementation uses the Flicker system [19] to provide verifiable, hardware-supported isolation of security-sensitive code from all other software and devices on a platform by using a TPM [23] and hardware-supported dynamic root of trust [16]. As a result, the Trusted Computing Base (TCB) for security-sensitive CAP code includes only the Flicker stub code, and excludes the legacy operating system, BIOS, and all DMA-capable devices. These trusted components will be the same across all uses of CAP, i.e., their code will be publicly known and evaluated to be "known-good."

The registration and breach phases of CAP (Figure 1) involve processing inside the Flicker isolation environment, because the protocol requires access to information that must be kept secret. 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 CAP 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 power-off attack described in Section 5.2, but the untrusted code cannot impersonate trusted code.

The trusted CAP 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 N_U, N_{AS}, 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: N_{AS}, N_{U_i}, R_{AS}, R_{U_i}, K_{AS}^{-1}, K_{U_i}, K_{AS-U_i}, and the registered users' endorsement key certificates (true identities) C_{TPM-U_i}. It further includes the entire set of private group signature keys $K_{GSK}^{-1}[i]$ (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}.

4.1 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 at a time.

4.2 Performance

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)[1]. Consequently, the common-case operation of CAP is efficient. On average, message endorsement takes 86 ms $\pm$ 0.4 ms on the user's platform, and message verification takes 87 ms $\pm$ 0.2 ms on the SP's platform. Signature generation requires calculations equivalent to 8 exponentiations and 2 bilinear map computations, and verification takes 6 exponentiations and 3 computations of the bilinear map [6]. Figures 5 and 6 show that the endorsement time of CAP scales well with the size of the blacklist for

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

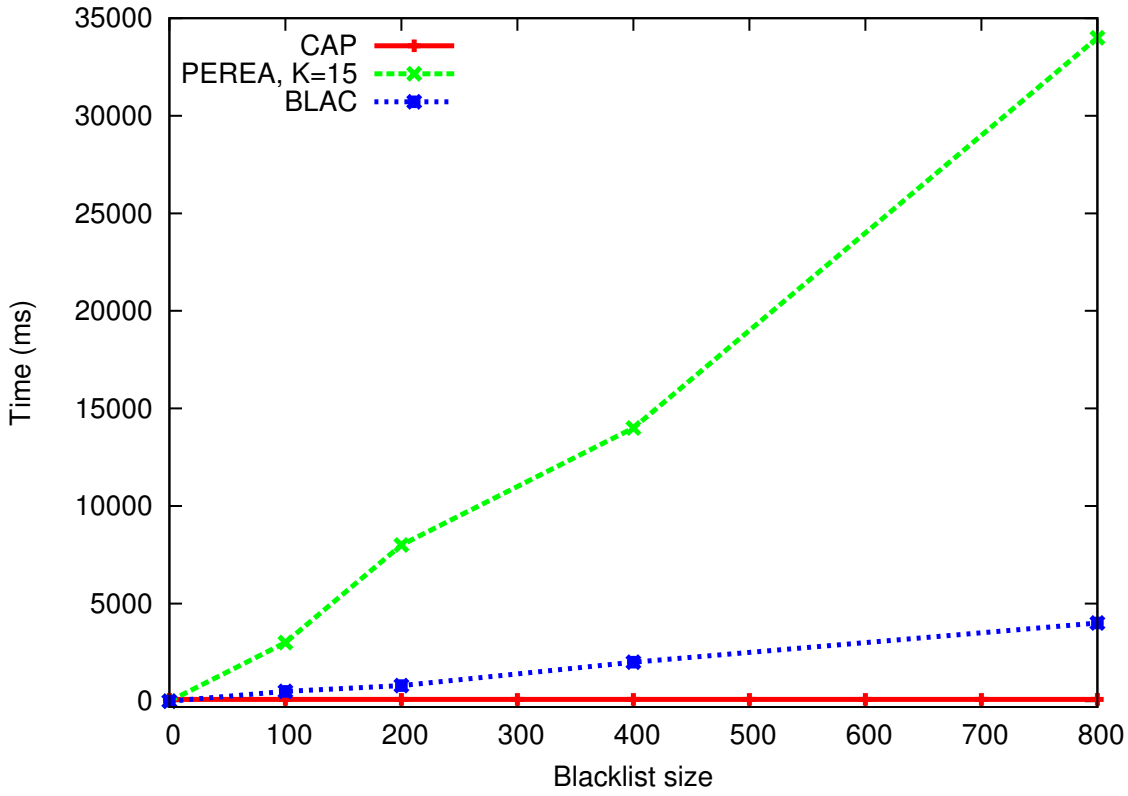

Figure 5: Comparison of anonymous communication time at the user for CAP, PEREA [25], and BLAC [24]. Note that data points for BLAC and PEREA were extrapolated from figures in the original publications.

both the user and SP. Table 1 compares the asymptotic and empirical performance measurements reported by prior works [24, 25].

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 $O(|blacklist|)$ 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 enter and resume from the Flicker execution environment, including 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. In our implementation, contract negotiation averages 7.99 ± 0.04 s. Although this may seem like a long time, it is faster than the time it takes many users to enter their login information, and is needed infrequently (only when a user registers to use a new service, or the SP changes its policies). The majority of the time spent during registration is spent executing the *TPM_Unseal* command. Thus, by batching mul-

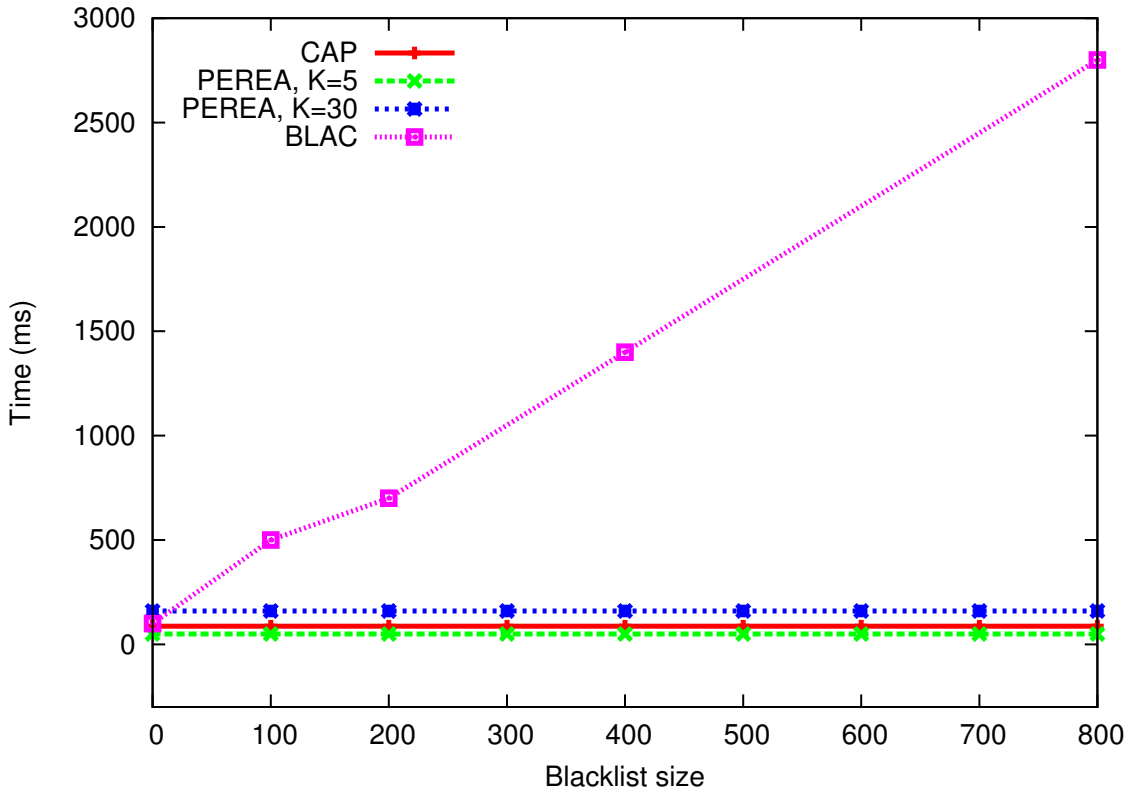

Figure 6: Comparison of anonymous communication time at the service provider (SP) for CAP, PEREA [25], and BLAC [24]. Note that data points for BLAC were extrapolated from figures in the original publication.

tiple requests together in a single Flicker session, the cost of unsealing data can be amortized to achieve reasonable 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 [19], though our current implementation does not support TPM NV RAM.

Breach Lastly, 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 $\pm$ 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 to establish the secure channel as described in Chapter 3 for the breach protocol.

4.3 Trusted Computing Base (TCB)

CAP has a relatively small trusted computing base that needs to run in the Flicker isolated execution environment [19]. Table 2 shows the lines of code in the TCB for

System	Auth. (U)	Auth. (SP)	Auth. (U)	Auth. (SP)	Parameters
CAP	86 ms	87 ms	$O(1)$	$O(1)$	
PEREA [25]	5900 ms	160 ms	$O(K\|BL\|)$†	$O(K)$	$K_{SP} = 30, K_U = 10$
BLAC [24]	1450 ms	870 ms	$O(\|BL\|)$	$O(\|BL\|)$	

Table 1: Comparison of authentication time between CAP and other systems for reasonable parameter choices ($|BL| = 800$). We do not include string comparisons in the asymptotic analysis. Measurements for PEREA and BLAC are taken from the relevant works, as we were unable to obtain the source code for these schemes [24,25]. †: 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.

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 CAP 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	**Language**	**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` [26].

Chapter 5

Discussion

5.1 CAP as a Primitive

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

We do not make anonymity guarantees about the complete protocol stack: only the CAP component. For example, a user who types in their personal information to a chat service could circumvent any security otherwise offered from CAP. Similarly, the chat protocol may run CAP on top of TCP/IP, which may allow chat servers to log IP addresses. Although CAP does not solve the complete protocol stack problem, CAP 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 CAP to enforce policies regarding proper use. A chat application could run on top of Tor, and use CAP for anonymous communication to provide contractual anonymity for chat sessions.

5.2 Security

There are several potential attacks against CAP. 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. First, an SP could insist upon an AS that has been designed with high availability in mind, e.g., by having a network of ASs that could each reveal a users identity. Attacks on availability are present in most protocols, and can be addressed by standard methods in fault-tolerant computing and cryptography.

Another attack is that a malicious SP could require each user to obtain a unique contract policy such that every user would be a member of a different group signature group. Although such a user would remain anonymous, all of their requests would be linkable. All anonymity systems offer weak unlinkability guarantees if some party can determine that the pool of active users is small. Because CAP relies on trusted computing, it can solve this problem in a unique way: the AS can reveal the number of active registered users upon request. Each user can then create her own threshold for the number of active 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, TTP-less systems [7,8,24,25] are also vulnerable to this type of attack, but have no defense against it since there is not a neutral third party that can attest to the number of active users.

We assume that the TPM on the AS is tamper-proof. However, real systems may only be able to provide tamper-resistant TPM's that are potentially vulnerable to physical attacks. CAP can easily be extended to use standard threshold cryptography techniques [14] such that a coalition of AS's are needed to reveal a misbehaving users identity. Threshold cryptography would distribute the secret AS key for unlocking a user's real identity across t servers where some number n of them need to be compromised to successfully carry out this attack.

5.3 Contract Negotiation

We assume that the user can retrieve the contract policy directly from the SP, and that the user will not be able to contribute to the policies. We make this assumption because there is a bootstrapping problem involved in having full negotiation of the contracts in which both parties can contribute, specifically: how can a user and SP negotiate a contract without the SP learning anything about the user during the negotiation? We leave negotiable contracts as future work.

5.4 Threshold Policies

Some policies in our system may be inefficient to implement. For example, threshold policies are commonly used to prevent spamming, e.g., users should not send more than k messages per day. Our architecture provides message unlinkability, which by its very nature prevents the linking needed for the SP to check threshold policies. (One could implement a threshold policy by having the SP send a large set of messages to the AS, but doing so is clearly very inefficient.) We leave contractual anonymity that allows for threshold policies as an open problem.

5.5 Verifier-local Revocation

In the group signature scheme we use there is a tradeoff between unlinkability and the runtime of *GS_Verify* in the size of the blacklist [6]. 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 probability that messages can be linked, and in $O(|BL|)$ time for perfect unlinkability. For example, if $|BL| = 1024$, each table entry in a precomputed lookup table is about 128 bytes long, and the SP devotes 128Mb to create a lookup table, then there is about a 1/1024 chance that two messages sent by the same user

can be linked. In CAP, 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. Paranoid users can always insist on using services that rely on the $O(|BL|)$ algorithm.

Chapter 6

Related Work

CAP is similar to existing anonymous authentication systems. However, it should be noted that these existing systems, unlike CAP, do not provide all of the properties required to achieve contractual anonymity.

The existing anonymous authentication systems can be divided into several categories. Members of the first class of anonymous authentication systems utilize a trusted third party (TTP) server [10, 11]. In these schemes, users escrow their identities on a TTP, which is able to reveal the identity of the user to the service provider (SP) if there is evidence of misbehavior. However, a user has no assurance that the TTP will keep her identity secret if she behaves. Although TTPs can use rich policies for defining malicious behavior, they lack any mechanism for proving that a user's identity is bound to those policies. While CAP's accountability server (AS) is a third party server, the physical server and its operator *do not* have to be trusted. Instead, the user and service provider can remotely verify the code that the AS is running using trusted computing and choose to trust it based on their knowledge of the code's semantics.

Because of the risk of storing identities on a TTP that cannot be verified, researchers have created other anonymous authentication systems that eliminate the need for TTPs. One class of these TTP-less systems are those based on e-cash [3, 4, 21, 22]. In these schemes, users remain anonymous unless they authenticate too many times. In e-cash systems, spending currency is considered authentication, and so de-anonymization is important to thwart the double spending of e-cash. These systems have been generalized into anonymous credential systems that provide anonymity and unlinkability unless a user authenticates k or more times. Unfortunately, these systems can only define policies in terms of thresholds, and so many types of misbehavior can not be expressed. CAP is not restricted to enforcing any one type of policy.

The last class of systems are those that do not rely on a TTP but also allow for rich policies to be enforced. The systems in this class allow for *subjective judging*, in which the SP can choose to link a user's authentications for any reason by adding her to a blacklist. Unfortunately, there is no pre-negotiated anonymity policy, and so the user has no guarantees about when her access might be revoked. Systems in this category also have scalability issues [7, 8, 24] or require considerable rate-limiting to function [25]. Because an authentication must occur for each unlinkable message,

these systems are not practical for applications that send messages at moderate rates (10^6 unlinkable messages a day). We show that CAP is efficient and scales well (see Section 4.1) in comparison to PEREA [25] and BLAC [24], the most efficient of these schemes. Unlike PEREA and BLAC, CAP allows users and service providers to negotiate the terms for the user's anonymity such that a user can not be blacklisted unless she breaks her contract.

We use trusted computing so that the AS can be verified to be running a known implementation of the anonymity software. In particular, we base our work on Flicker [19]. Datta et al. [13] 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.

Chapter 7

Conclusion

We introduce the notion of *contractual anonymity*, a type of anonymous authentication that provides strong guarantees for the user and service provider, and present CAP, a protocol that achieves the contractual anonymity properties. We utilized trusted computing to overcome the scalability limitations of cryptography in other anonymous systems. CAP scales well with respect to the size of the blacklist, supports services with moderate message rates, and does not blindly rely on a trusted third party that can deanonymize well-behaved users. Finally, we implemented the end-to-end CAP system, and show through our experiments that CAP is scalable and practical.

Bibliography

[1] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual: Volume 2: System Programming*, Dec. 2005, [Online]. Available: AMD Publication no. 24594 rev. 3.11., http://www.amd.com. [Accessed: May 1, 2009].

[2] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik, "A practical and provably secure coalition-resistant group signature scheme," in *Advances in Cryptology – CRYPTO*, ser. Lecture Notes in Computer Science, vol. 1880. Springer, 2000, pp. 255–270.

[3] M. H. Au, S. S. M. Chow, and W. Susilo, "Short e-cash," in *Progress in Cryptology - INDOCRYPT*, ser. Lecture Notes in Computer Science, vol. 3797. Springer, 2005, pp. 332–346.

[4] M. H. Au, W. Susilo, and Y. Mu, "Constant-size dynamic k-TAA," in *Security and Cryptography for Networks*, ser. Lecture Notes in Computer Science, vol. 4116. Springer, 2006, pp. 111–125.

[5] D. Boneh, X. Boyen, and H. Shacham, "Short group signatures," in *Advances in Cryptology – CRYPTO*, ser. Lecture Notes in Computer Science, vol. 3152. Springer, 2004, pp. 41–55.

[6] D. Boneh and H. Shacham, "Group signatures with verifier-local revocation," in *ACM Conference on Computer and Communications Security*. ACM, 2004, pp. 168–177.

[7] E. F. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *ACM Conference on Computer and Communications Security*. ACM, 2004, pp. 132–145.

[8] 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*. ACM, 2007, pp. 21–30.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 2–16.

[10] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Advances in Cryptology –*

CRYPTO, ser. Lecture Notes in Computer Science, vol. 2442. Springer, 2002, pp. 61–76.

[11] ——, "Signature schemes and anonymous credentials from bilinear maps," in *Advances in Cryptology – CRYPTO*, ser. Lecture Notes in Computer Science, vol. 3152. Springer, 2004, pp. 56–72.

[12] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *ACM Symposium on Operating Systems Principles*. ACM, 2007, pp. 117–130.

[13] 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*. IEEE Computer Society, 2009.

[14] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *Advances in Cryptology – CRYPTO*, ser. Lecture Notes in Computer Science, vol. 435. Springer, 1989, pp. 307–315.

[15] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-To-Peer Systems*, ser. Lecture Notes in Computer Science, vol. 2429. Springer, 2002, pp. 251–260.

[16] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[17] Intel Corporation, "Trusted eXecution Technology – measured launched environment developer's guide," [Online]. Available: Document number 315168005, http://www.intel.com. [Accessed: May 1, 2009], Jun. 2008.

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

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *ACM European Conference in Computer Systems*. ACM, Apr. 2008, pp. 315–328.

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

[21] L. Nguyen and R. Safavi-Naini, "Dynamic k-times anonymous authentication," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, vol. 3531. Springer, 2005, pp. 318–333.

[22] I. Teranishi and K. Sako, "k-times anonymous authentication with a constant proving cost," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, vol. 3958. Springer, 2006, pp. 525–542.

[23] Trusted Computing Group, "Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands," [Online] Available: Version 1.2, Revision 103. http://www.trustedcomputinggroup.org. [Accessed: May 1, 2009], Jul. 2007.

[24] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith, "Blacklistable anonymous credentials: blocking misbehaving users without ttps," in *ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 72–81.

[25] ——, "PEREA: towards practical TTP-free revocation in anonymous authentication," in *ACM Conference on Computer and Communications security*. ACM, 2008, pp. 333–344.

[26] 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].

[27] XySSL Developers, "XySSL cryptographic library," [Online]. Available: http://polarssl.org. [Accessed: May 1, 2009].