2013 IEEE Symposium on Security and Privacy

Anon-Pass: Practical Anonymous Subscriptions

Michael Z. Lee, Alan M. Dunn, Brent Waters, Emmett Witchel
The University of Texas at Austin
{mzlee, adunn, waters, witchel}@cs.utexas.edu

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

Abstract

We present the design, security proof, and implementation of an anonymous subscription service. Users register for the service by providing some form of identity, which might or might not be linked to a real-world identity such as a credit card, a web login, or a public key. A user logs on to the system by presenting a credential derived from information received at registration. Each credential allows only a single login in any authentication window, or *epoch*. Logins are anonymous in the sense that the service cannot distinguish which user is logging in any better than random guessing. This implies unlinkability of a user across different logins.

We find that a central tension in an anonymous subscription service is the service provider's desire for a long epoch (to reduce server-side computation) versus users' desire for a short epoch (so they can repeatedly "re-anonymize" their sessions). We balance this tension by having short epochs, but adding an efficient operation for clients who do not need unlinkability to cheaply re-authenticate themselves for the next time period.

We measure performance of a research prototype of our protocol that allows an independent service to offer anonymous access to existing services. We implement a music service, an Android-based subway-pass application, and a web proxy, and show that adding anonymity adds minimal client latency and only requires 33 KB of server memory per active user.

I. Introduction

Today, widespread electronic-subscription services are used to manage access to streaming music and video, journalistic and academic articles, Internet hotspots, and public transportation. In such systems there is a fundamental tension between enforcing admission control and providing a user with anonymity and privacy. Both of these goals are important. Admission control can ensure that a service provider receives adequate compensation and the system remains economically viable. On the other hand, if a user's behavior in a subscription service is tracked, it creates a hoard of private information ranging from the user's personal tastes to geographic movements, depending on the service.

Foregoing one of these two goals makes achieving the other considerably easier. If we require a user to simply login to an account, we can make sure that no user is simultaneously logged in twice. On the other hand, if a subscription system requires no logins then anyone can access it anonymously, perhaps with the assistance of auxiliary tools such as a traffic-anonymization system like Tor [14]. However, achieving both admission control and anonymity together is difficult.

Ideally, we want an *anonymous subscription system* that protects the interests of both the service and the users. This problem was considered previously in the work of Damgård, Dupont, and Pedersen [12], who showed what they called an uncloneable identification scheme. At a high level, in their system there is a registration phase in which a client chooses a secret and the server "blindly" signs it using a two-party protocol. During time period (or *epoch*) t, a client can then login to the server using her acquired signature. The login protocol is such that the server cannot distinguish which user logged in (from all the registered users) nor link a user's login to any past logins. However, if a client attempts to login twice with the same credentials during the same epoch, the client will be detected and denied access. While the protocols of Damgård et al. [12] were cryptographically heavy, Camenisch et al. [4] gave asymptotic improvements resulted in a more practical scheme. Neither protocol, however, was implemented.

Our aim is to design and implement an anonymous subscription system which is practical and deployable for existing subscription services. We start by looking at the construction of Camenisch et al. [4], which is in turn based on ideas from e-cash [5]. In their system, the registration protocol involves the server issuing the client a blind signature on a pseudorandom function (PRF) key d. To login during epoch t, the server and client run a two-party protocol in which the server learns $y = Y_d(t)$ (where Y represents the PRF). In addition, the client proves to the server (in zero knowledge) that $y = Y_d(t)$ for some key d on which the client has a valid signature. If this proof succeeds, the server checks a table it maintains for the current time period. If y is not already in the table, it is simply added and the login proceeds. However, if it already exists in the table, then its presence is evidence that a login has already occurred during that epoch for the same (unknown) registered user and the login attempt is rejected.

Even though the system of Camenisch et al. is significantly more efficient than that of Damgård et al. (Camenisch et al. [4] show an order-of-magnitude reduction in the number of modular exponentiations), it is not clear that their improvements make the scheme practical. The computational cost of a cryptographic login can still be a limiting factor in system scalability, since it can limit the number of users that a service can handle for a fixed set of computational resources, or impact the battery life of a client on a mobile device. Indeed, even for

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.29

our scheme (which is more efficient than prior schemes), we find that a login requires approximately 8 ms of computation per core on a quad-core Intel 2.66 GHz Core 2 CPU (cf. Table II in §VI-A). This machine can service at most 496 logins per second.

If a login is too costly for the service, then the service must either buy more servers or increase the length of an epoch to reduce the number of logins per fixed time period. Increasing the length of an epoch negatively impacts usability, because the length of an epoch is approximately how long a user will have to wait if she wishes to unlink herself from past activity. Consider a video streaming service where the time epoch is 15 seconds. If a client wishes to load a new video and dissociate herself from past videos watched, waiting up to 15 seconds will not be too noticeable relative to other delays. However, a time epoch (and hence delay) of over a minute is likely to be unacceptable to the user.

To put the epoch length and the maximum number of logins per second in perspective, consider that users of the Netflix streaming service watched 1 billion hours of content in July 2012 [19]. With an epoch length of 1 minute, which is still rather high, this leads to 60 billion authentications per month, or 22,815 per second assuming that their distribution is uniform over time.

We believe the central tension in an anonymous subscription service is the service provider's desire for a long time epoch (to improve efficiency) versus the user's desire for a short epoch (to improve anonymity). Yet while users might occasionally want a short time epoch so they can quickly "re-anonymize" (e.g., when browsing through a collection of short videos), in the typical case such re-anonymization may not be necessary (e.g., if a user is watching a 90-minute movie straight through). Our central insight is to balance the tension by providing short epochs, giving users the ability to re-anonymize if they so choose, while also providing an efficient method for clients who do *not* need unlinkability to cheaply re-authenticate themselves for the next epoch.

A. Our Contributions

We introduce a new primitive that we call an *anonymous subscription scheme with conditional linkage*. Such a scheme has registration and login operations as described above. In addition, it offers a *re-up* operation that allows a client who is logged in at (current) epoch t to authenticate itself (more cheaply) for time period $t + 1$ with the tradeoff that the server is able to link these sessions. In practice, we find that allowing such an operation has a significant performance benefit because re-up in our scheme is over eight times faster than login.

Anon-Pass is designed for anonymous access to modern web services like audio streaming, video streaming, and reading articles. These services contain a large number of subscribers, only a small portion of which are active at any particular time. Users sign up for these services for a set amount of time, but during that time they can expect to use the service freely. The service provider cannot blacklist or deny service to an individual user. Anon-Pass is designed so a given service provider can provide anonymous access (perhaps as part of a premium package), or a partner organization could sell anonymous access to a range of subcontracted services.

We provide a formal definition of an anonymous subscription scheme with conditional linkage along with a cryptographic construction. We also provide a design and implementation for Anon-Pass, a system that implements our scheme. We demonstrate and evaluate Anon-Pass for scenarios including a streaming music service, an anonymous unlimited-use public transit pass, and a third-party authentication proxy. We now briefly overview these contributions.

At an intuitive level we desire our anonymous subscription system to have the following properties:

- **Correctness.** An honest service provider will accept any well-formed login request from a client that is not logged in, and any well-formed re-up request from a client that is currently logged in.
- **Unforgeability.** An honest service provider will only accept login or re-up requests that are derived from secrets of registered clients.
- **Sharing resistance (admission control).** In a given epoch, an honest service provider will allow at maximum one client to receive service per registered client secret.

We refer to unforgeability and sharing resistance as *soundness*.

- **Pseudonymity.** Any service provider will not be able to identify the client that originated a particular request. By identify, we mean associate a request with the information that the client submitted at registration.
- **Unlinkability.** The service provider cannot correlate a user's sessions (each session being a login and associated re-ups) any better than guessing.

We refer to pseudonymity and unlinkability as *anonymity*. In Section II we formalize the notion of an anonymous subscription scheme with conditional linkage, and provide formal security definitions for soundness and anonymity.

There are two main limitations to the anonymity guarantees provided by our system. First, the exact probability with which the server can "break anonymity" depends on various aspects of the system outside our model. As an extreme case, for example, if the service has only one registered user, then the service provider knows who is logging in with perfect accuracy. As a less obvious example: if all users are logged in (and remain logged in), and one user logs out and then another login occurs in the next time epoch, this new login must belong to the user who logged out. Second, there might be other ways – external to our system – in which a user's anonymity can be violated, e.g., by using network-traffic analysis or by correlating a user's observable behavior across sessions. Our anonymous subscription service is only intended to not "make the problem worse" by giving the server additional means to discern user identities. We note that our system could be coupled with other techniques (e.g., a network-anonymity service like Tor [14], or a private information retrieval scheme [11]) to anonymize other aspects of the user's interaction with the server.

Section III contains a description of our cryptographic construction. We then present the design (§ IV) and implementation (§ V) of our system. The implementation (§ V) discusses our usage scenarios: a streaming music service, an unlimited-use public-transit pass, and a third-party authentication proxy. Our evaluation (§ VI) shows that a single modern CPU can support almost 500 logins per second, and 4,000 re-ups per second. We demonstrate the practicality of our system by showing that the performance overheads for our macrobenchmarks are reasonable, e.g., 33 KB in extra memory resources per user and only a 11.8% increase in CPU utilization on the application server while serving 12,000 clients. Finally, we demonstrate the importance of re-ups for the music streaming service: having re-ups available can decrease average CPU utilization from 77.9% to 16.7% for the same number of user requests.

We review other related work in Section VII.

II. Anonymous Subscriptions with Conditional Linkage

In this section, we formally define the pieces of our scheme and the security properties that it provides. Note that the re-up operation in our system is referred to in our formal constructions as "linking."

A. Syntax

We first define the syntax of an **anonymous subscription scheme with conditional linkage**. Such a scheme consists of the following algorithms:

- The **setup algorithm** Setup is run by the authorization server S to initialize the system. It takes as input the security parameter 1^n and outputs a service public key spk along with an associated service secret key ssk, and the initial server local state σ.
- Client **registration** is done using two algorithms $\mathsf{Reg}_C, \mathsf{Reg}_S$ run by a client and server, respectively. The client takes as input the service public key, and the server takes as input the service secret key. Reg_C outputs a client secret key sk or an error symbol $\perp$.
- The **login protocol** is defined via two algorithms $\mathsf{Login}_C, \mathsf{Login}_S$ run by a client and server, respectively. The client takes as input a secret key sk, the service public key spk, and the current epoch t; the server takes as input the service secret key ssk, local state σ, a counter cur, and the current epoch t. Login_S outputs updated values σ', cur'.
- The **link protocol** provides an alternative way for a client who is logged in during epoch t to re-authenticate for epoch $t+1$. This protocol is defined by a pair of algorithms $\mathsf{Re\text{-}Up}_C, \mathsf{Re\text{-}Up}_S$ run by the client and server, respectively. The client takes as input a secret key sk, the service public key spk, and the current epoch t; the server takes as input the service secret key ssk, local state σ, a counter next, and the current epoch t. $\mathsf{Re\text{-}Up}_S$ outputs updated values σ', next'.

  The registration, login, and link protocols may fail if the client behaves incorrectly. For these protocols, the server outputs an additional bit which is 1 if and only if the protocol runs to completion, in which case we say the protocol **succeeds**. We say the protocol **fails** otherwise.
- The **end-of-epoch** algorithm $\mathsf{EndEpoch}$ provides a way for the server to end the current epoch, refresh its state, and begin the next epoch. This algorithm takes as input the current epoch t, local state σ, and counters $\mathsf{cur}, \mathsf{next}$; it outputs updated values $\sigma', \mathsf{cur}', \mathsf{next}'$.

Intended usage and correctness. System initialization begins by having the server run $\mathsf{Setup}(1^n)$ to generate spk, ssk and initial server state σ. The server also sets $\mathsf{cur} = \mathsf{next} = t = 0$.

Following setup, clients can register at any time; *client i* refers to the ith client who registers, and we denote the secret key of that client by sk_i. Independent of client registrations (which do not affect the server's state and may be performed at any time), there is some sequence of executions of the login, link, and end-of-epoch algorithms. In our formal model (unlike the implementation), we assume none of these are executed concurrently, and so there is a well-defined ordering among those events. We denote the period of time between two executions of $\mathsf{EndEpoch}$ (or between Setup and the first execution of $\mathsf{EndEpoch}$) as an **epoch**. We write Login_i (resp., $\mathsf{Re\text{-}Up}_i$) to denote an execution of Login (resp., $\mathsf{Re\text{-}Up}$) between the ith client and the server, with both parties using their prescribed inputs.

At some instant in an epoch, we (recursively) define that *client i is logged in* if either (1) Login_i was previously run during that epoch, or (2) at some point in the previous epoch, client i was logged in and $\mathsf{Re\text{-}Up}_i$ was run. At some instant during an epoch, *client i is linked* if at some previous point during that epoch client i was logged in and $\mathsf{Re\text{-}Up}_i$ was run.

Correctness requires that when honest clients interact with a server then, except with negligible probability, cur is always equal to the number of clients who are logged in, and next is always equal to the number of clients who are linked.

B. Security

We define two notions of security: one ensuring that malicious clients cannot generate more active logins than the number of times they have registered ("soundness"), and the other ("anonymity") guaranteeing anonymity and unlinkability for clients who authenticate using the Login protocol. (On the other hand, clients who re-authenticate using the $\mathsf{Re\text{-}Up}$ protocol will be linked to their session in the previous epoch.)

1) Soundness: A scheme is **sound** if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ *succeeds* in the following experiment is negligible:

1) $\mathsf{Setup}(1^n)$ is run to generate keys spk, ssk, and an initial state σ. Adversary $\mathcal{A}$ is given spk, and the experiment sets $\mathsf{cur} = \mathsf{next} = t = \mathsf{users} = 0$.
2) $\mathcal{A}$ may then do any of the following, where the server uses its prescribed inputs (based on its current state):

- $\mathcal{A}$ can interact with an oracle for Reg_S. (This represents a registration by a client whom $\mathcal{A}$ controls.) $\mathcal{A}$ need not run the registration protocol honestly. Following each such interaction, users is incremented.
- $\mathcal{A}$ can request that an honest client (one not controlled by $\mathcal{A}$) register. On the ith such request, the registration protocol is run honestly (using the prescribed inputs) and the resulting client key is denoted by sk_i. $\mathcal{A}$ cannot observe[1] the interaction between this client and the server, and sk_i is *not* given to $\mathcal{A}$.
- $\mathcal{A}$ can interact with an oracle for Login_S (resp., $\mathsf{Re\text{-}Up}_S$). This represents a login (resp., link) request by a client controlled by $\mathcal{A}$.
- $\mathcal{A}$ can request that client i Login (resp., Re-Up). In response, the login (resp., link) protocol is run honestly using sk_i (and the rest of the prescribed inputs). $\mathcal{A}$ cannot observe[2] this interaction.
- $\mathcal{A}$ can request to end the current epoch, in response to which t is incremented and $(\sigma', \mathsf{cur}', \mathsf{next}') \leftarrow \mathsf{EndEpoch}(\sigma, \mathsf{cur}, \mathsf{next})$ is executed.

In the above, we allow $\mathcal{A}$ only *sequential* access to its oracles.

3) $\mathcal{A}$ *succeeds* if at any point cur is greater than users plus the number of honest clients who are logged in.

2) Anonymity: A scheme is **anonymous** if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ *succeeds* in the following experiment is negligibly close to $1/2$:

1) A random bit c is chosen, and we set $t = 0$.
2) $\mathcal{A}$ outputs a service public key spk.
3) $\mathcal{A}$ runs two sequential interactions with $\mathsf{Reg}_C(spk)$. If either of these results in output $\perp$, then $c' = 0$ is output and the experiment ends. Otherwise, these interactions result in two secret keys sk_0, sk_1.
4) $\mathcal{A}$ then runs in three phases. In the first phase, $\mathcal{A}$ may do any of the following:
   - Increment the epoch number t.
   - Query oracle $\mathsf{Login}(\cdot)$. On input a bit b, this begins executing the client login protocol Login_C using inputs sk_b, spk, and the current epoch number t.
   - Query oracle $\mathsf{Re\text{-}Up}(\cdot)$. On input a bit b, if client b is not logged in, $\mathsf{Re\text{-}Up}(b)$ does nothing. Otherwise, it begins executing the client link protocol $\mathsf{Re\text{-}Up}_C$ using inputs sk_b, spk, and the current epoch number t.
5) When the second phase begins, both clients must not be logged in. Then $\mathcal{A}$ may:
   - Increment the epoch number t.
   - Query oracle $\mathsf{ChallengeLogin}(\cdot)$. $\mathsf{ChallengeLogin}(b)$ responds as $\mathsf{Login}(b \oplus c)$ does.
   - Query oracle $\mathsf{ChallengeRe\text{-}Up}(\cdot)$. $\mathsf{ChallengeRe\text{-}Up}(b)$ responds as $\mathsf{Re\text{-}Up}(b \oplus c)$ does.

   The second phase ends once an epoch begins in which neither client is logged in.
6) In the third phase, $\mathcal{A}$ interacts as in the first phase.
7) $\mathcal{A}$ outputs a bit c', and *succeeds* if $c' = c$.

In all the above, $\mathcal{A}$ is again given only sequential access to its oracles.

[1]We assume registration is done over a private, authenticated channel.

[2]We assume logins/links are done over a private channel. Note that there is no client authentication when setting up this channel (since the client wishes to remain anonymous); thus, we assume $\mathcal{A}$ only passively eavesdrops but does not actively interfere.

III. Construction

In this section we provide a construction for a secure anonymous subscription scheme with conditional linkage. Our construction uses a number of primitives – bilinear groups, zero-knowledge proofs of knowledge, and a particular pseudorandom function family – and cryptographic assumptions from prior work. We provide relevant background in Appendix A.

Similar to [4], our construction works by associating a unique token, $Y_d(t)$, with each client secret, d, in each epoch, t. Registration works by allowing a client to obliviously obtain a signature on a secret. To log in, a client sends a token and proves in zero-knowledge that (1) it knows a server signature on a secret, and (2) this secret corresponds to the token that was sent. The tokens are used to determine admission to the service; the server accepts a token only if it has not been presented before in that epoch. Intuitively, soundness follows from the difficulty of generating signatures; anonymity follows from pseudorandomness of the tokens. (Formal proofs of security can be found in the full version of this paper [18].)

On a technical level, we use the Dodis-Yampolskiy PRF [15] and an adapted version of one of the signature schemes proposed by Camenisch and Lysyanskaya [7] (CL signatures). These building blocks are themselves efficient, and also enable efficient zero-knowledge proofs as needed for our construction.

As noted above, a client can authenticate during epoch t by sending the token $Y_d(t)$ and proving in zero-knowledge that the token is "correct." If a client is already logged in during epoch $t-1$, however, an alternative way of authenticating is to send $Y_d(t)$ and prove that this token is "linked" to the token $Y_d(t-1)$ (which was already proven correct). This can be done much more efficiently, with the tradeoff that the two user sessions are now explicitly linked to each other. In an epoch where the client is not logged in, it can perform a fresh Login to "re-anonymize" itself.

A. Notation

Throughout, $G = \langle g \rangle$ is a bilinear group of prime order q, with target group G_T. $e(\cdot, \cdot)$ denotes the bilinear map, and we let $g_T = e(g, g)$. We denote by $a \leftarrow S$ the selection of an element a uniformly at random from the set S.

We denote an interactive protocol executed by two probabilistic algorithms A (with private input a) and B (with private input b) by

$$(x, y) \leftarrow \langle A(a), B(b) \rangle,$$

where x (resp., y) denotes the local output of A (resp., B).

We denote zero-knowledge proofs of knowledge where a prover convinces a verifier of knowledge of values $(a_1, \ldots, a_n)$ that satisfy a predicate P by

$$\text{PoK}\{(a_1, \ldots, a_n) \mid P(a_1, \ldots, a_n)\}.$$

This notation is taken from Camenisch and Stadler [8] (modified to use PoK instead of PK).

B. Main construction

We assume zero-knowledge proofs of knowledge as building blocks, and describe them in a separate section (§III-D). The zero-knowledge proofs of knowledge presented there are non-interactive proofs that are secure in the random oracle model. In our security proofs, we assume the use of interactive versions of these protocols that do not rely on the random oracle model and can in turn be made zero knowledge using standard execution of protocols, this implies sequential execution of the zero-knowledge proofs. When the interactive zero-knowledge proofs of knowledge are instantiated using the Fiat-Shamir heuristic in the random oracle model, and protocols may be executed concurrently, our proof breaks down for technical reasons but we nevertheless view our proof as heuristic evidence for the security of our implementation.

Setup: $(spk, ssk, \sigma) \leftarrow \mathsf{Setup}$

The server chooses $x, y, z \leftarrow \mathbb{Z}_q$ and sets $X = g^x$, $Y = g^y$, and $Z = g^z$. The service public key is $spk = (q, G, G_T, g, X, Y, Z)$, and the service secret key is $ssk = (x, y, z)$. The server state σ will be a pair of sets. They are both initialized to be empty, i.e., $\sigma = (\{\}, \{\})$. We refer to the first component as $\sigma.cur$ and the second as $\sigma.next$. Throughout, $\mathsf{cur} = |\sigma.cur|$ and $\mathsf{next} = |\sigma.next|$.

Registration: $(\phi, sk) \leftarrow \langle \mathsf{Reg}_S(ssk), \mathsf{Reg}_C(spk) \rangle$

1) The client chooses $d, r \leftarrow \mathbb{Z}_q$. It constructs $M = g^d Z^r$ and sends this to the server.
2) The client acts as prover and the server as verifier in the zero-knowledge proof of knowledge
$$\text{PoK}\{(d, r) \mid M = g^d Z^r\}.$$
If the proof fails, registration fails.
3) The server generates $a \leftarrow \mathbb{Z}_q^*$ and sets $A = g^a$. Then it forms signature $s = (A, B = A^y, Z_B = Z^{ay} (= B^z), C = A^x M^{axy})$ and returns it to the client.
4) The client verifies that it has received a legitimate signature by checking
$$A \neq 1, \quad e(g, B) = e(Y, A), \quad e(g, Z_B) = e(Z, B),$$
$$e(g, C) = e(X, A) e(X, B)^d e(X, Z_B)^r.$$
Otherwise, Reg_C outputs $\perp$.
5) The client sets $sk = (s, d, r)$.

Login: $((\sigma', \mathsf{cur}'), \phi) \leftarrow$
$$\langle \mathsf{Login}_S(ssk, \sigma, \mathsf{cur}, t), \mathsf{Login}_C(sk, spk, t) \rangle$$

1) The client uses its secret key $(s = (A, B, Z_B, C), d, r)$ to create a blinded signature. The client chooses $r_1, r_2 \leftarrow \mathbb{Z}_q^*$ and creates blinded signature $\tilde{s} = (\tilde{A}, \tilde{B}, \tilde{Z}_B, \tilde{C})$, where $\tilde{A} = A^{r_1}$, $\tilde{B} = B^{r_1}$, $\tilde{Z}_B = Z_B^{r_1}$, and $\tilde{C} = C^{r_1 r_2}$.
2) The client creates login token $Y_d(t) = {g_T}^{1/(d+t)}$.
3) The client submits $\tilde{s}, Y_d(t)$ to the server.
4) If $Y_d(t) \in \sigma.cur$, login fails.
5) Otherwise, the server verifies that
$$\tilde{A} \neq 1, \quad e(g, \tilde{B}) = e(Y, \tilde{A}), \quad \text{and} \quad e(g, \tilde{Z}_B) = e(Z, \tilde{B}).$$
If not, login fails.
6) The client and server each compute
$$v = e(g, \hat{C})$$
$$v_x = e(X, \tilde{A})$$
$$v_{xy} = e(X, \tilde{B})$$
$${v'}_{xy} = e(X, \tilde{Z}_B)$$
7) The client acts as prover and the server as verifier in the zero-knowledge proof of knowledge
$$\text{PoK}\{(d, r, r') \mid v^{r'} = v_x v_{xy}^d {v'}_{xy}^r \wedge Y_d(t) = {g_T}^{1/(d+t)}\}.$$
(The client uses $r' = \frac{1}{r_2}$.) If the proof fails, login fails.
8) The server sets $\sigma' = (\sigma.cur \cup \{Y_d(t)\}, \sigma.next)$.

Link: $((\sigma', \mathsf{next}'), \phi) \leftarrow$
$$\langle \mathsf{Re\text{-}Up}_S(ssk, \sigma, \mathsf{next}, t), \mathsf{Re\text{-}Up}_C(sk, spk, t) \rangle$$

1) The client with $sk = (s, d, r)$ submits $Y_d(t) = g_T^{1/(d+t)}, Y_d(t+1) = g_T^{1/(d+(t+1))}$ to the server.
2) The server checks that $Y_d(t) \in \sigma.cur$ and $Y_d(t+1) \notin \sigma.next$. If not, linking fails.
3) The client acts as prover and the server as verifier in the zero-knowledge proof of knowledge
$$\text{PoK}\{d \mid Y_d(t) = g_T^{1/(d+t)} \wedge Y_d(t+1) = g_T^{1/(d+(t+1))}\}.$$
If the proof fails, linking fails.
4) The server adds $Y_d(t+1)$ to $\sigma.next$.

End epoch: $(\sigma', \mathsf{cur}', \mathsf{next}') \leftarrow \mathsf{EndEpoch}(\sigma, \mathsf{cur}, \mathsf{next})$
$\sigma' = (\sigma.next, \{\})$.

Proofs of the following can be found in the full version of this paper [18].

Theorem (soundness): If the LRSW assumption holds in G, the construction above is sound.

Theorem (anonymity): If the DDHI assumption holds in G, the construction above is anonymous.

C. Efficiency improvements

Our protocol incorporates several efficiency improvements over the base primitives that it uses:

Improved CL signatures: The base CL signature incorporates a fifth element: $A^z \equiv Z_A$ in our notation, where A is part of a client sk, and $z \in ssk$. Z_A, and a blinded version

$\tilde{Z}_A$ that the client would need to send in proof of knowledge of a signature, can be eliminated by restructuring checks of signature validity. Instead of checking

$$e(\tilde{A}, Z) = e(g, \tilde{Z}_A), \quad e(\tilde{Z}_A, Y) = e(g, \tilde{Z}_B),$$

to prove that $\tilde{Z}_A$ and $\tilde{Z}_B$ are formed correctly, we eliminate $\tilde{Z}_A$ and the former check, and for the latter, check

$$e(\tilde{B}, Z) = e(g, \tilde{Z}_B)$$

to prove that $\tilde{Z}_B$ is formed correctly. Removing this element eliminates two pairing operations – the check that this element is properly formed – from server verification of logins. Pairing operations dominate the computational cost of login, so this change is significant. A login operation on the server consists of 8 pairings and 6 exponentiations in G_T. We measure that a pairing operation takes an average of 1950 μs, while a G_T exponentiation takes 232 μs. (See §VI for a full description of the settings used to acquire the timing of pairing and exponentiation operations.) Thus, we expect this change improves efficiency of login on the server by a factor of 1.2.

Simultaneous login and linking: Some of our applications (§IV-E) involve linking for the next epoch immediately upon logging in for the current epoch. We modify the protocol to improve efficiency in this case. We are able to eliminate the repeated computation of exponentiated $Y(t)$ values that occur for separate links: A login using $Y(t)$ and sequence of links $Y(t)$ to $Y(t+1)$, $Y(t+1)$ to $Y(t+2)$, ..., $Y(t+(n-1))$ to $Y(t+n)$ with separate login and link operations would duplicate exponentiations of $Y(t+1), \ldots, Y(t+(n-1))$. By eliminating these repeated exponentiations, the time for two link operations is reduced from 2566 μs to 1392 μs on the client and from 1412 μs to 921 μs on the server. This is an improvement of $1.8\times$ and $1.5\times$, respectively. However, the overall time is still dominated by the cost of login.

D. Zero-knowledge proofs of knowledge

We present non-interactive zero knowledge proofs of knowledge that are secure in the random oracle model; these are the protocols as implemented in Anon-Pass.

Registration PoK: $\text{PoK}\{(d, r) \mid M = g^d Z^r\}$

Prover:

1) Choose $r_d, r_r \leftarrow \mathbb{Z}_q$, calculate $R = g^{r_d} Z^{r_r}$.
2) Set $c = H(g, Z, M, R)$.
3) Send $(R, a_d = cd + r_d, a_r = cr + r_r)$ to the verifier.

Verifier:

1) Calculate $c = H(g, Z, M, R)$.
2) Check that $M^c R = g^{a_d} Z^{a_r}$.

Login PoK: $\text{PoK}\{(d, r, r') \mid v^{r'} = v_x v_{xy}^d {v'}_{xy}^r \wedge Y(t) = {g_T}^{1/(d+t)}\}$

We rewrite this as

$$\text{PoK}\{(d, r, r') \mid v^{r'} = v_x v_{xy}^d {v'}_{xy}^r \wedge Y(t)^d = g_T Y(t)^{-t}\}$$

Prover:

1) Choose $r_d, r_r, r_{r'} \leftarrow \mathbb{Z}_q$, and then compute $R_1 = v^{r_{r'}} v_{xy}^{r_d} {v'}_{xy}^{r_r}$ and $R_2 = Y(t)^{r_d}$.
2) Set $c = H(v, v_x, v_{xy}, {v'}_{xy}, R_1, g_T, Y(t), R_2)$.
3) Send $(R_1, a_{r'} = cr' + r_{r'}, a_d = -cd + r_d, a_r = -cr + r_r, R_2)$ to the verifier.

Verifier:

1) Calculate $c = H(v, v_x, v_{xy}, {v'}_{xy}, R_1, g_T, Y(t), R_2)$.
2) Check whether $v_x^c R_1 = v^{a_{r'}} {v_{xy}}^{a_d} {v'}_{xy}^{a_r}$ and $(g_T Y(t)^{-t})^{-c} R_2 = Y(t)^{a_d}$.

Link PoK: $\text{PoK}\{d \mid Y(t) = g_T^{1/(d+t)} \wedge Y(t+1) = g_T^{1/(d+(t+1))}\}$

We rewrite this as

$$\text{PoK}\{d \mid Y(t)^d = g_T Y(t)^{-t} \wedge Y(t+1)^d = g_T Y(t+1)^{-(t+1)}\}$$

Prover:

1) Choose $r \leftarrow \mathbb{Z}_q$, set $R_t = Y(t)^r$ and $R_{t+1} = Y(t+1)^r$.
2) Set $c = H(g_T, Y(t), Y(t+1), R_t, R_{t+1})$.
3) Send $(a = cd + r)$ to the verifier.

Verifier:

1) Calculate $c = H(g_T, Y(t), Y(t+1), R_t, R_{t+1})$.
2) Check whether $(g_T Y(t)^{-t})^c R_t = Y(t)^a$ and $(g_T Y(t+1)^{-(t+1)})^c R_{t+1} = Y(t+1)^a$.

IV. Design

This section describes the design of the Anon-Pass system. The system is intended to instantiate our protocol in a way that is practical for deployment. We present a conceptual framework for the system in which the various functionalities of the system are separated.

There are three major pieces of Anon-Pass functionality: client authentication management, server authentication management, and service provider admission control. In our design, we call the these pieces the **client user agent**, the **authentication server**, and the **resource gateway**. The client user agent and the authentication server correspond to the client and server in the cryptographic protocol. The resource gateway enforces access to the underlying service, denying service to users who are not properly authenticated. A **session** in Anon-Pass is a sequence of epochs beginning when a user logs in and ending when the user stops re-upping.

Figure 1 shows the major components of the Anon-Pass system. We depict the most distributed setting, where each of the three functions is implemented separately from existing services, though a deployment might merge functionality. For example, the resource gateway might be folded into an already existing component for session management.

Our system supports internal and external authentication servers. An internal authentication server corresponds to a service provider offering anonymous access themselves, e.g., the New York Times website might offer anonymous access at a premium. An external authentication server corresponds

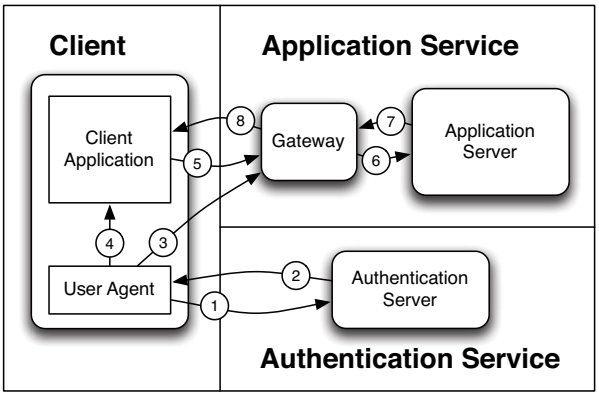

Fig. 1. The communication between the authentication server, resource gateway, and user agent with respect to the client and the service. ① Communication is initiated by the user agent and the authentication server verifies the credentials. ② The authentication server verifies the credentials and returns a sign-in token to the user agent. ③ The user agent communicates this sign-in token to the gateway and, afterward, ④ passes this information to the client application for use. ⑤ The client application includes the token as a cookie along with its normal request. ⑥ The gateway checks that the sign-in token has not already been used in the current epoch and then proxies the connection to the application server. ⑦ The application server returns the requested content and ⑧ the gateway verifies that the connection is still valid before returning the response to the client.

to an entity providing anonymous access to already existing web services. For example, a commercial anonymous web proxy (like proxify.com or zend2.com) might offer anonymous services.

Our system implements registration, though it is not depicted in the figure. We do not discuss the payment portion of the registration protocol. Anonymous payment is a separate and orthogonal problem. Possible solutions include paying in some form of e-cash [10] or BitCoins [25].

A service might allow multiple re-ups within a single epoch. If a user application knows it will not need to disassociate its current actions from prior actions for a while, it could batch several epochs worth of re-up operations. The server would have to allow such batching, but might put the requests in a queue to remain responsive to requests for the next epoch.

We want to allow services to use our authentication scheme without much modification, so we provide a simple interface: authorized clients during a time period are allowed to contact the service and are cut off as soon as the session is no longer valid. Services might have to accommodate Anon-Pass's access control limitations. For example, a streaming media service might want to limit how much data can be buffered within a given epoch. The service provider loses the ability to enforce any access control for buffered data.

A. Timing

Anon-Pass requires some time synchronization between clients and servers because both client and server must agree on epoch boundaries, and Anon-Pass supports short epochs. To support a 15 second epoch, clients and servers should be synchronized within about a second. The network time protocol (NTP) is sufficient, available and scalable for this task. The `pool.ntp.org` organization[3] runs a pool of NTP servers that keep the clocks of 5–15 million machines on the Internet synchronized to within about 100 ms.

The server response to a login request includes a timestamp. Clients verify that they agree with the server on the current epoch. Client anonymity could be violated[4] if the epoch number ever decreases, so clients must track the latest timestamp from every server they use and refuse to authenticate to a server that returns a timestamp that is earlier than a prior timestamp from that server. This ensures that regardless of any time difference between server and client, anonymity is preserved.

Clients who will re-up choose a random time during the epoch to send the re-up request in order to prevent repetitive behavior that becomes identifying. However, clients avoid re-upping at the end of the epoch to avoid service interruption (e.g., in our prototype, clients re-up in the first $\frac{4}{5}$ of the epoch). Randomizing the re-up request time also has the benefit of spreading the computational load of re-ups on the server across the entire epoch.

B. Client user agent

The client user agent is responsible for establishing the client secret, communicating with the authentication server, and maintaining a session for the client. Separating it from the client application achieves two goals: it minimizes the amount of code that needs to be trusted by the user to handle her secrets and and it lowers the amount of modification necessary to support new client applications.

Once the user agent establishes a connection with the authentication server, it runs our login protocol, and the user agent receives a (standard, public-key) signature on the PRF value and the current epoch. The user agent sends this certificate to the resource gateway as proof that it is authenticated for the current epoch. The resource gateway uses the signature to determine token validity. The user agent cannot use this certificate in a later epoch.

When the user agent and authentication server run our re-up protocol, the user agent receives a signature that includes both the current epoch and the next epoch, as well as the two corresponding PRF values. These additional values allow the resource gateway to link the re-up operations back to the original request.

The user agent handles almost all of the protocol state, but the original client application still needs to identify itself as authenticated. Thus, the user agent transforms the signed certificate from a login, into a per-session user credential (e.g., a cookie for HTTP-based services). The only operation most client applications need to support is the ability to send this credential along with its request. The client application does not need to make any changes as the user agent re-ups; the

[3] http://www.ntp.org/ntpfaq/NTP-s-algo.htm

[4] Anonymity would not necessarily be completely broken, but the server could link the current session of a client with a prior one.

user agent's actions ensure that the same session credential remains valid for the session's duration.

C. Authentication Server

The authentication server is separated from the service to provide greater flexibility for service providers. The server's primary task is to run the authentication protocols and ensure that users are not authenticating more than once per epoch. Since the protocol's cryptographic operations use a lot of computational resources, Anon-Pass was designed so that an authentication service provider can distribute the work among multiple machines. The only information that needs to be shared between processes are the PRF values and the epoch of currently authenticated users (e.g., by using a distributed hash table (DHT)). Only storing information about currently authenticated users relieves a service provider from having to store all spent tokens, which requires unbounded storage.

D. Resource Gateway

The resource gateway is designed to perform a lightweight access check before sending data back to a client. Only if a client is authenticated for an epoch can it receive data during that epoch. Therefore the epoch length (which is determined by the service provider) bounds how much data can go to a client before the client must reauthenticate (login or re-up).

When the authentication server is external to the service, the authentication server never talks directly to the resource gateway. User misbehavior (i.e., a double authentication attempt) will not cause the user to be immediately disconnected. The authentication server will refuse any re-up request from a misbehaving client, disconnecting them at the start of the next epoch.

A resource gateway is composed of two logical parts – one handles the user agent updates, the other part handles the client request. In a large distributed system, a service provider might split these into different parts to place access control on the outer perimeters and the user agent update handler off the critical processing and request path.

E. Multi-epoch login

As we discuss in Section III-C, we can combine a login with multiple re-up operations. Allowing re-up with login provides the benefits of long epochs that start on demand and can provide request rate limiting by preventing reauthentication for at least a known period of time. It also reduces the total computation done by the server.

Multi-epoch logins allows Anon-Pass to be used for unlinkable resource reservation of digital and even physical goods, for example, to reserve computer access at an Internet cafe. Users can reserve a resource for a variable number of epochs, without needing to periodically extend access.

Consider a subway system that supports month-long subscription passes. Because the transit authority does not want riders to all enter with one pass, it limits the access that each pass can grant. The New York City MTA lists 18 minutes as the lockout period[5] between uses of an unlimited ride card. Anon-Pass can provide anonymous authentication for a transit subscription pass. A long epoch helps limit how often a user can access; however, this is not enough to prevent two users from sharing a single credential wherein the first user uses the credential immediately before an epoch change, and a second uses the same credential immediately afterward. If a service requires multiple authenticated tokens for the same credential upon access, then this form of sharing is prevented.

Operation	Baseline	Pairing Preprocessing	Signature Precompute
Client Login	19.9	15.8	13.5
Server Login	16.0	7.9	7.9

TABLE I
PAIRING AND SIGNATURE PREPROCESSING DIFFERENCES. ALL TIMES ARE IN MILLISECONDS.

V. IMPLEMENTATION

We implement the cryptographic protocol in a library, `libanonpass`, using the Pairing Based Cryptographic Library [21], PolarSSL[6] for clients, and OpenSSL[7] for the server. Both the client and server operations are encapsulated in this library's 1,434 lines of code.[8] The library includes a number of management functions for initializing and clearing data structures, and protocol functions for creating and verifying requests for registration, login, and re-up.

We arrange terms to minimize exponentiations and we reuse partial computations in the login and re-up zero-knowledge proofs to make them more efficient. We further improve performance by implementing a multi-epoch authorizing zero-knowledge proof for verifying re-up tokens during a login. In addition, we use two different forms of preprocessing: preprocessing the pairing operation and precomputing a known portion of the client login message. Table I shows the improvements of these optimizations.

To show the flexibility of our protocol, we implement a number of usage scenarios including a streaming music service, an anonymous unlimited-use public transit pass, and a third party authentication proxy. These applications are all large enough to highlight implementation issues specific to each context.

The authentication server is implemented as a 926 line module for the lightweight HTTP server Nginx[9]. Nginx uses a process pool rather than a thread pool for handling concurrency and therefore minimizes synchronization. The only shared state for Anon-Pass is a hash table of currently active login tokens. In our prototype, we dedicate a server to maintaining this hash table whose contents are get and set using RPCs. The performance of this hash table server is not a

[5]http://www.mta.info/metrocard/compare.htm
[6]https://polarssl.org/
[7]http://www.openssl.org/
[8]Counted by SLOCCount. http://www.dwheeler.com/sloccount/
[9]http://nginx.org/

bottleneck for any of our workloads. In a deployment situation, the hash table could be a distributed hash table (DHT) [28], [22] run by the service provider. DHTs are a common part of the software infrastructure in data centers.[10]

The resource gateway is implemented as a 443 line Nginx module. The module performs all of the operations needed by the application service. It can checks and updates session information for clients, and terminates connections when an unauthenticated request is made or a response is returned. Each of these operations is designed to be as simple as possible and could be merged with a frontend server or load balancer.

The basic client user agent consists of two pieces. There are 789 lines of code that handle the client connection and interface with `libanonpass`, and there are 357 lines of code that handle configuration parsing and the client state machine. The protocol messages are sent by using cookies to simplify server-side parsing and minimize client application modifications.

In the rest of this section, we will talk about the structure of each of these applications and the specific implementation changes needed for each.

A. Streaming Music Service

We implement a streaming music service over HTTPS by exposing media from web accessible URIs. The service directly implements our anonymous credential scheme and allows a user to choose the granularity of an anonymous session as either a full playlist or as an individual song. We modify 54 lines of VLC[11] to communicate with our user-agent and pass our session token as an additional cookie.

Our music service allows users to download songs, but we rate-limit playback. Rate limiting reduces network bandwidth usage, which allows our service to support more clients with jitter-free service. Rate limiting also reduces the amount of data a client can buffer during an epoch. If a client loses its anonymous service in the next epoch, it will only have a small amount of buffered data. The music service has no ability to enforce access control for that buffered data.

B. Public Transit Pass

We implement a public transit pass as an Android application. Currently, public transit providers who issue month long or week long "unlimited" access passes limit user access to prevent cheating. Without safeguards, a user could give her pass to all of her friends to ride for free. Anonymous subscriptions are able to provide these safeguards without revealing user's identity (so users' movements cannot be tracked).

As an example, the average daily ridership of BART in the San Francisco Bay Area for the months of August through October, 2012 is 401,323 people on weekdays [30]. While we do not have data on traffic peaks, the total load is easily handled by Anon-Pass. A single three-epoch login verification takes approximately 8.4 ms on our system which is very close to the base cost of login verification of 7.9 ms. One modern CPU core on a server can run the 400,000 verifications in just a little under an hour. These operations are trivially parallelizable across multiple cores and machines.

We use the Java Native Interface (JNI) to call Anon-Pass from an Android application. The Android application has a simple interface with a single button to generate a login and two re-up additional PRFs. It then displays this data as a quick response (QR) code for a physical scanner to read. If a transit provider chooses a 6 minute epoch length, then this would create a 12 to 18 minute window in which a login attempt from the same phone would fail.

Other anonymous subscription systems such as Unlinkable Serial Transactions [29] or anonymous blacklisting systems such as Nymble [17] or BLAC [31] require network connectivity at the time when a client uses an authentication token. When using a blacklisting system, a user wants to proactively fetch the blacklist to ensure that she is not on the list prior to contacting a server, otherwise she could be deanonymized. The size of a blacklist can grow quickly; for example, BLAC adds 0.27KB of overhead per blacklist entry. When using a UST-like system, the user must receive the next token when a prior token is used up (but not before). Anon-Pass is ideal for subway systems where network phone coverage is spotty at best, since it only needs to communicate in one direction at the subway entry gate.

C. Access Proxy

We implement a server to allow users to proxy access to websites. In addition, the server could authenticate for users with legitimate accounts provided by the service to access news sites and other content. All traffic and accesses appear to originate from the same entity and it is up to the proxy service to multiplex the user credentials. Users' anonymity leverages both the wide variety of accessible services as well as the number of proxy users.

To approximate an access proxy, we sign up for accounts at a number of news websites. The service consists for two parts: the scraper logs into the news sites using valid credentials and caches the results for later use, and the proxy injects cached cookies into authenticated user requests. A legitimate service running this type of proxy would likely need to work with the news sites to better control creation of user accounts and history. Ideally, the proxy service would provide ephemeral user accounts for a client session; however, current systems do not allow us to easily accomplish this task.

As an approximation of the necessary steps, we use a cache of cookies, but allow more than one client to share a real user account. The proxy cycles through its list of cookies for a given news site rather than generating ephemeral accounts or registering for new legitimate accounts.

VI. Evaluation

We evaluate Anon-Pass through a series of micro-benchmarks and several larger systems. The authentication and

[10]For example, the Cassandra key-value store, which can easily be used as a DHT, is used by a wide variety of commercial data centers. See http://www.datastax.com/cassandrausers.

[11]http://www.videolan.org/vlc/index.html

application servers run on two Dell Optiplex 780s, each of which has a quad-core 2.66 GHz Intel Core 2 CPU, 8 GB of RAM, and uses Ubuntu Linux 12.04. The hash server runs on a Dell 755 with an older generation quad-core 2.66 GHz Intel Core 2 CPU, only 4 GB of RAM, and also uses Ubuntu Linux 12.04. The elliptic pairing group is a Type A (in the naming conventions of the PBC library) symmetric pairing group with a 160-bit group order and 512-bit base field, and the ECDSA signature uses a 160-bit key.

A. Comparison to Prior Work

We compare the computational complexity of our scheme to prior work by counting the computationally expensive operations (i.e., group exponentiations and pairings) in each. We only examine server-side computation, as this is the limiting factor in the scalability of the system. The main competing schemes are those from Camenisch et al. [4] and an adaptation of a scheme from Brickell and Li [3].

Camenisch et al. [4] mention two alternatives for their construction, using either an RSA-based signature scheme [6] or the CL signature scheme we use. We found the description of their RSA-based instantiation insufficient to produce an implementation, but note that the performance of this variant will be hurt by the need to use large moduli to prevent known factorization attacks. The second variant, using the same CL signatures we use, is not described fully in their paper.

Brickell and Li [3] propose a scheme for direct anonymous attestation (DAA) with controlled linkability which could be adapted to give an anonymous subscription scheme with conditional linkage. The correspondence between their scheme and ours is that the basename which controls linkability in their scheme corresponds to the epoch number in our scheme. Using different basenames per epoch ensures that the (B, K) components of their signature are equal for the same client secret if submitted in the same epoch and unlinkable otherwise. Re-up for client secret f can be performed between signatures (B_1, K_1) and (B_2, K_2) in their scheme via the proof of knowledge

$$\text{PoK}\{f \mid B_1^f = K_1 \wedge B_2^f = K_2\},$$

though we stress that they do not consider this idea in their work. Their scheme requires the use of asymmetric pairings. With the PBC library, we measured that asymmetric pairings lead to prohibitively slow login operations. (See Table III.) Additionally, many of the curve families supported in PBC, including all those with asymmetric pairings, have high embedding degree. While high embedding degree leads to lower field sizes for G_1 and G_2 for a given level of security, it complicates multiplication in G_T, which can lead to a slower re-up operation.

In comparing the above to our scheme we use notation from Brickell and Li [3] to describe operation counts: Each count is written as a sum of individual operation types. A term nG^x indicates n multiexponentiations in group G with x bases. P indicates a pairing, and Pp indicates a preprocessable pairing, that is, one with one argument fixed after server-side setup is complete. Results are given in Table II.

Scheme	Login	Re-up
Anon-Pass	$G_T^3 + G_T^4 + 8Pp$	$2G_T^2$
Pairing-based DAA ([3])	$G_1^2 + G_2^2 + G_T^4 + P$	$2G_T^{2\,\dagger}$
Clone Wars ([4])	Comparison impossible, see text	

TABLE II
SERVER-SIDE OPERATION COUNTS FOR THE DIFFERENT CRYPTOGRAPHIC SCHEMES. † INDICATES ADDITIONS THAT WE PROPOSE TO EXISTING SCHEMES.

Group Type	G_1 Exp	G_2 Exp	G_T Exp	Pairing (Preprocessed)
A512 ECC	2.4	2.4	0.2	1.8 (0.8)
D159 ECC	0.8	6.5	1.5	5.1 (3.9)
F160 ECC	0.8	1.5	6.0	27.7 (27.7)

TABLE III
OPERATION COSTS FOR DIFFERENT GROUPS PROVIDING 80-BIT SECURITY. ALL TIMES ARE IN MILLISECONDS.

We give measured operation costs in terms of CPU utilization in Table III for different families of curves where all components of the system are at least an 80-bit security level. This implies EC group orders of at least 160 bits and finite field sizes of at least 1024 bits [26]. From these raw operation times, we estimate CPU utilization for operations in Table IV. We estimate multiexponentiation times based on ratios of the number of multiplications required when we use a simultaneous k-ary exponentiation for the optimal k value. For 2, 3, and 4 bases with 1024-bit exponents, the costs are 1.1, 1.3, and 1.4 times that of an optimal k single base exponentiation respectively. See [23] for information on basic exponentiation and multiexponentiation techniques.

B. Measured Operation Costs

Table V presents the base time for the protocol operations. Here the operations are run in isolation. We did not use multiexponentiation in our prototype because the PBC library does not implement the operations; however, this could be added to further reduce the cost. In addition, there are overheads when integrating the protocols into a full system. Figure 2

Scheme	Group	Login	Re-up
Anon-Pass	A512	6.9	0.44
Pairing-Based DAA [3]	D159	15.2 (2.2×)	3.3 (7.5×)

TABLE IV
SERVER-SIDE OPERATION COSTS FOR DIFFERENT SCHEMES USING AN OPTIMAL GROUP FOR RE-UP. ALL TIMES ARE IN MILLISECONDS.

Protocol	Client		Server	
Registration	Create message	10.4	Verify message	7.3
	Verify signature	13.0	Sign message	12.5
Login	Create message	13.5	Verify message	7.9
Re-up	Create message	1.3	Verify message	0.7

TABLE V
RAW PROTOCOL OPERATION TIME IN MILLISECONDS.

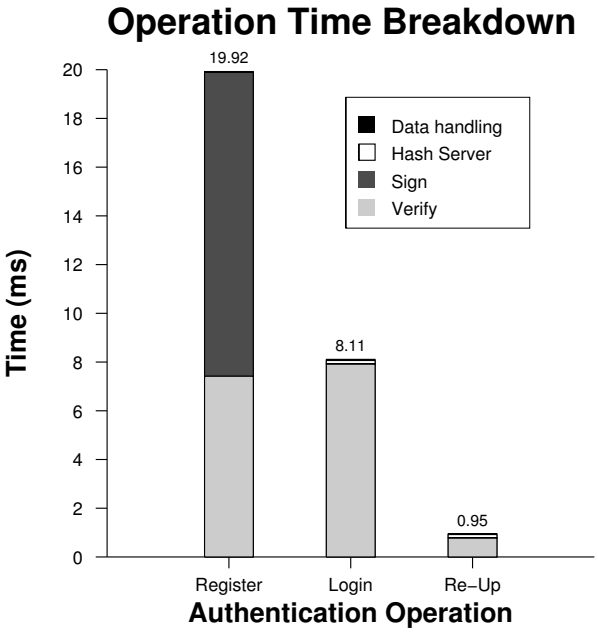

Fig. 2. The average cost of different requests on an unsaturated server. The bulk of the time is spent in signature verification.

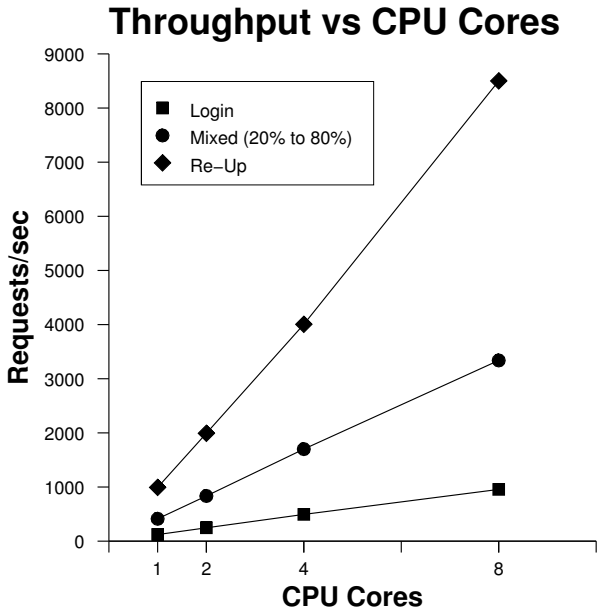

Fig. 3. The maximum throughput for a different mix of operations. There is a $3.5\times$ difference between just login and the 20% to 80% mix and a $8.9\times$ different between login and re-up.

shows a break down of each authentication operation and how time is spent on the server. For registration, the signature operation is our modified CL signature on the blinded client secret, whereas the signature for login and re-up are standard ECDSA signatures. The majority of the work for the ECDSA signature can be precomputed, and hence takes almost no time to compute. There is also a small amount of time spent contacting the hash server during login and re-up that does not happen for registration. Re-up is $8.5\times$ faster than login.

C. Authentication Server Scaling

We run the authentication service in different configurations to see how well the system scales with the addition of more cores or more machines in the system. For these experiments, we artificially restrict computation to a subset of the possible cores from one to eight cores. We precompute a number of valid login and re-up tokens and measure the maximum capacity of the servers.

Figure 3 shows the throughput scaling across two four-core machines. Note that for an average mix of 20% logins to 80% re-up operations, the servers can handle over $3\times$ as many requests per second as 100% logins. The re-up line shows the upper bound on the number of operations our servers can handle per second, approximately 8,500 requests/sec for the two servers. If we consider an epoch length of 15 seconds, this implies a re-up capacity of over 120,000 concurrent user sessions for the two quad-core CPUs.

D. Gateway Cost

Figure 4 shows the relative latency overhead of a request compared with simply downloading a number of different sized files. The experiment is run on a local area network to isolate the computational overhead; however, the authentication server, gateway, and hash server are all hosted on separate machines to better simulate a real deployment. Overhead for reasonably sized files is low. And, although the worst case of accessing a 1 byte file suffers a $1.30\times$ overhead, the time difference is only 0.15 milliseconds. In comparison, receiving 16 MB of data takes on average 194ms, dwarfing the additional cost of the hash server query.

E. Streaming music service

We build an example streaming music service. We lack datacenter-level resources, and so must adapt the benchmark to run on our local cluster of machines: network bandwidth is limited to 1 Gbps; we run the authentication server and application server on the Dell Optiplex 780s; and the hash server on the Dell 755. Clients run on 10 other machines. Each client randomly chooses a song and fetches it using pyCurl rather than a more memory-intensive media player like VLC. Avoiding VLC allows us to scale to a greater number of clients for our testbed.

We serve a media library consisting of 406 MP3 files, whose length is drawn from the most popular 500 songs on the Grooveshark music service, eliminating duplicates and songs that are over 11 minutes long. The average length of a song is $4{:}05 \pm 64.38$ s. We represent the music files using white noise

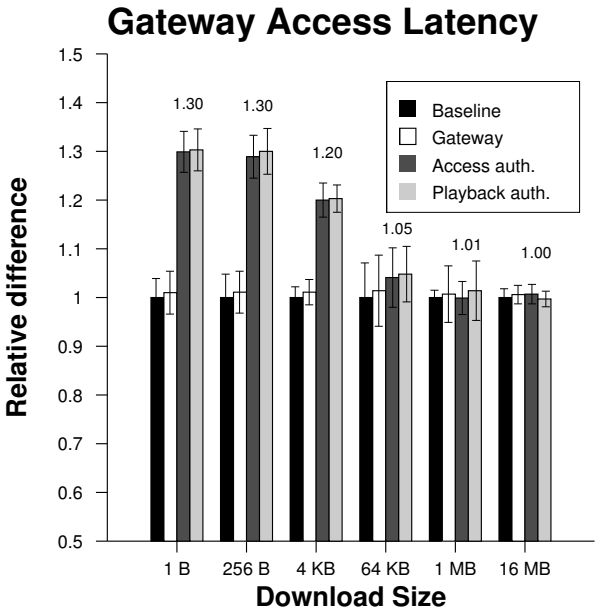

Fig. 4. Average latency overhead versus different sized client requests. Access authentication only verifies client requests while playback authentication also verifies returned data.

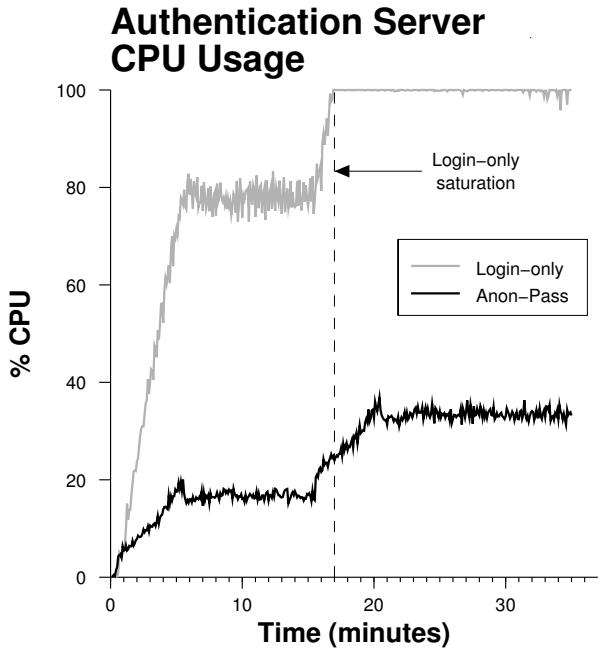

Fig. 5. The CPU usage on the authentication server measured every 5 seconds. The average CPU utilization for Login-only during the first stable segment (6,000 clients) is 77.9% (±2.42) and reaches saturation at about the 17 minute mark, or approximately 8,100 clients. The CPU utilization for Anon-Pass is 16.8% (±0.73) at 6,000 clients, and 33.4% (±0.96) at 12,000 clients (the second stable segment).

encoded at 32Kbps. The system dynamics are independent of the music content, and 32Kbps allows our server to saturate its CPU before saturating its outbound network bandwidth.

Media streaming servers want their clients to have enough data to buffer changing network conditions, but transferring too much data too quickly costs server resources with no end-user benefit. For our service, clients are allowed a burst of 32 KB at the beginning of each song request which helps to fill quickly the client song buffer. After this initial burst, the server aggressively throttles the download speed to only 5 KBps (or 40 Kbps) – enough to keep each stream playing, but not enough for a client to quickly download the entire library. We measure how much data each client downloaded as a function of time and infer the number of pauses for buffering that would have occurred during song playback.

We deploy a tightly integrated service running an authentication server, gateway, and data server with an epoch length of 15 seconds. We simulate two different scenarios: one case using only login and the other using both the login and re-up operations. We use an epoch length of 15 seconds as we believe this would be an acceptable delay for users to re-anonymize between songs.

Figures 5 and 6 show the performance of Anon-Pass and a modified login-only service that provides a linkable re-up service at the server CPU cost of a regular login. We modify the client programs to call the anonymous subscription service.

In both the login-only configuration and the Anon-Pass configuration, we ramp up the number of concurrent clients at a rate of 300 new clients every epoch (approximately 20 new clients a second) until we reach 6,000 total clients. After 10 minutes, we continue to increase the total number of clients until we reach 12,000 active clients. At 12,000 clients, the login-only configuration has a client failure rate of 34% due to CPU saturation. On the other hand, Anon-Pass only fails 0.02% of the requested songs.

Figure 5 shows the limited capacity of the login-only service. At 6,000 clients, the login-only service is able to keep up with authentication requests. However, the the steady-state average CPU utilization is already 77.9%. At the CPU saturation point, there are 8,100 clients attempting to connect to the service.

Figure 6 shows the CPU utilization on the application server and measures the impact of the gateway server. In addition to serving content, an authenticating application server must also receive client re-authentication updates and interact with the hash server. The intermittent client updates (once per epoch) each require an ECDSA signature verification which is relatively CPU intensive. Each update also requires at least one network round trip to the hash server (two in the case of re-up), and every active client connection also triggers a check (and hence network round trip) once every epoch. In combination, Anon-Pass adds an appreciable, but manageable amount of additional CPU utilization. On average, 6,000 clients adds 5.9 percentage points of CPU utilization and 12,000 clients

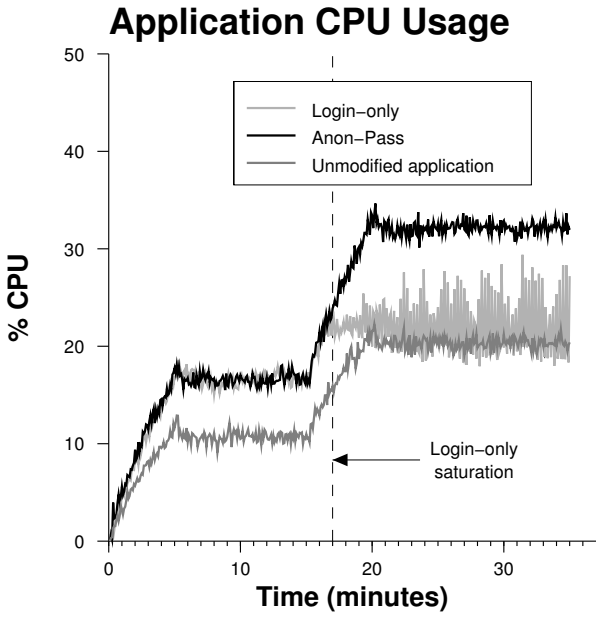

Fig. 6. The CPU usage on the application server measured every 5 seconds. The CPU usage with login-only follows the Anon-Pass behavior until the authentication server reaches saturation. Clients timeout and the application server has an overall drop in CPU utilization due to the lower number of clients successfully completing requests.

adds 11.8 percentage points. The login-only configuration adds approximately the same amount of overhead for as long as the authentication server can keep up, but as soon as clients begin to fail, the application server sees a decrease in overall CPU utilization due to the decrease in the number of successful clients. Clients request new songs causing a larger amount of variation in the application server CPU utilization.

Anon-Pass keeps its state in hash server memory, and does not require persistent storage. One average, the hash server memory utilization is only 2.1 KB per client. However, the authentication server requires an additional 23.8 KB of memory per client and the application server requires an additional 7.3 KB of memory per client. The unmodified server requires 52.1 KB of memory per client, so Anon-Pass has a memory overhead of $1.64\times$ per active client.

F. Public Transit Pass

We compute the amount of time it takes to generate a login QR code on an HTC Evo 3D. Recall, the login QR code consists a normal client login and three re-up tokens. The time to generate a login QR code is 222 ± 24 ms. Power usage is minimal because the the application does not need to communicate with any remote servers over the network.

G. Content proxy service

We set up a proxy to test how much latency our proposed proxy service adds to clients' requests. We host the proxy on an AWS micro instance[12] for ease of access and to better simulate a real deployment.

Website	Normal Access	Proxied Access
http://news.yahoo.com/	2.69 ($\pm$0.46)	2.89 ($\pm$0.42)
http://www.nytimes.com/	2.74 ($\pm$1.12)	3.20 ($\pm$0.76)
http://www.guardiannews.com/	3.03 ($\pm$0.27)	2.66 ($\pm$0.37)
http://abcnews.go.com/	2.35 ($\pm$0.55)	2.66 ($\pm$0.71)
http://espn.go.com/	1.67 ($\pm$0.14)	1.97 ($\pm$0.14)
http://www.npr.org/	1.24 ($\pm$0.19)	1.14 ($\pm$0.29)

TABLE VI
AVERAGE REQUEST LATENCY OVER 20 TRIALS IN SECONDS.

Table VI shows the average latency for accessing the sites using Firefox. The proxy generally increases page load latency by 7.4–18.0%. However, due to content variability, two of the sites load faster through the proxy.

In four of these cases, the proxy works without ever needing to send the authenticated session information back to the client. However, for npr.org and guardiannews.org, the proxy must return to the client some of the session cookies for the websites so the browser can indicate that the user is logged in. Giving session cookies to the client is unfortunate because depending on how a site formats its cookies, a user could potentially steal the cookie and attempt to change the login information related to the account. However, passing cookies is safe for these two sites because they require an additional reauthentication before account details may be modified.

VII. Related Work

Our work continues research into *anonymous credentials* [9], which allow access control while maintaining anonymity. We describe several themes of research in anonymous credential schemes and show the point that our system occupies in design space.

A. Flexible policy support

Handling credential abuse has been a central theme of much of the work on anonymous credentials. However, abuse of credentials takes on different meaning in many of the different systems. Early work (e.g. [13]) focused around *e-cash* [10], where credentials represented units of currency. Here the relevant policy is to prevent double spending of the same currency.

Recent work has focused on *anonymous blacklisting systems* (e.g. [31], [17]). In these systems, a service is capable of *blacklisting* a user, excluding her from future interactions with the service, based on her actions during a transaction. However, many anonymous blacklisting systems leave blacklisting decisions completely up to the service, as opposed to e-cash based systems, which only allow the service to enforce a specific policy. Some work [27] has been done to hold services to particular policies, though this work uses mechanisms beyond pure cryptography (trusted hardware).

[12]http://aws.amazon.com/ec2/instance-types/

Anon-Pass chooses to trade policy flexibility for performance. Nonetheless, we have shown (§V) that Anon-Pass is flexible enough to support a wide variety of applications.

B. Efficiency

Stubblebine, Syverson, and Goldschlag [29] propose unlinkable serial transactions to handle anonymous subscription. In their scheme, when users register they receive a blind signature. A user can use this blind signature to begin a transaction and receives a new signature upon transaction end. However, this means the system must store and be able to efficiently search through all used tokens while the system key material remains unchanged (likely the period of a subscription, which could be on order of months). Blanton [2] uses more advanced cryptographic techniques to support client secret expiration, but incurs the same space requirements. Anon-Pass requires only the ability to store tokens for a fixed number of epochs, which is storage proportional to the number of requests that can occur in a few minutes, rather than months.

While anonymous blacklisting techniques could possibly be used to provide anonymous subscription (i.e. by temporarily blacklisting logged-on clients), anonymous blacklisting schemes often suffer from poor scalability. For example, BLAC [31] requires time linear in the number of blacklisted users to check the blacklist. PEREA [32] reduces this to linear in the number of logged in users. Even the latest in this series of work BLACR [1] supports 26-38 authentications/minute on an 8-core machine with 5000 blacklisted users. Anon-Pass requires only hash table lookups to check for double usage, otherwise operations are constant in the number of registered and logged in users. With Anon-Pass on a 4-core machine, our micro-benchmark sustains almost 500 login operations a second, and scales up to 12,000 concurrent users in the music streaming benchmark.

Nymble [17] improves performance of blacklisting systems by adding a trusted third party that can revoke anonymity as needed. Follow-on projects try to divide trust among multiple parties [16] or reduce involvement of trusted third parties [20]. Anon-Pass maintains efficiency without needing any trusted third party.

Most papers from the cryptographic literature do not include implementations and benchmarks. More applied papers still do not include system use in actual scenarios. In this paper we describe how anonymous subscription primitives affect system performance in more realistic scenarios.

VIII. Acknowledgments

We thank Sangman Kim and Lara Schmidt for their kind help. We also thank our shepherd, Paul Syverson, and the useful feedback from the anonymous reviewers. This research was supported by funding from NSF grants IIS-#0964541, CNS-#0905602, CNS-#1223623, and CNS-#1228843 as well as NIH grant LM011028-01.

References

[1] Man Ho Au, Apu Kapadia, and Willy Susilo. BLACR: TTP-free blacklistable anonymous credentials with reputation. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, February 2012.

[2] M. Blanton. Online subscriptions with anonymous access. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 217–227. ACM, 2008.

[3] Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In *Conference on Trust and Trustworthy Computing*, 2010.

[4] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clone wars: Efficient periodic n-times anonymous authentication. In *ACM Conference on Computer and Communications Security*, pages 201–210, 2006.

[5] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *EUROCRYPT*, pages 302–321, 2005.

[6] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In *International Conference on Security in Communication Networks*, 2002.

[7] Jan Camenisch and Anna Lysyanskaya. Signature Schemes and Anonymous Credentials from Bilinear Maps. *CRYPTO*, 2004.

[8] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burt Kaliski, editor, *Advances in Cryptology - CRYPTO 97*, volume 1296 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[9] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[10] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of CRYPTO '82*, pages 199–203. Plenum, 1982.

[11] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 41–50. IEEE, 1995.

[12] Ivan Damgård, Kasper Dupont, and Michael Østergaard Pedersen. Unclonable group identification. In *EUROCRYPT*, pages 555–572, 2006.

[13] Ivan Damgrd. Payment systems and credential mechanisms with provable security against abuse by individuals. In *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1988, Proceedings*, volume 403 of *Lecture Notes in Computer Science*, pages 328–335. Springer, 1988.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

[15] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography*, pages 416–431, 2005.

[16] R. Henry, K. Henry, and I. Goldberg. Making a nymbler nymble using verbs. In *Privacy Enhancing Technologies*, pages 111–129. Springer, 2010.

[17] P. Johnson, A. Kapadia, P. Tsang, and S. Smith. Nymble: Anonymous ip-address blocking. In *Privacy Enhancing Technologies*, pages 113–133. Springer, 2007.

[18] Michael Z. Lee, Alan M. Dunn, Jonathan Katz, Brent Waters, and Emmett Witchel. AnonPass: Usable anonymous subscriptions - Full Version. http://z.cs.utexas.edu/users/osa/anon-pass/.

[19] Michael Liedtke. Netflix users watched a billion hours last month. http://usatoday30.usatoday.com/tech/news/story/2012-07-03/netflix-online-video/56009322/1.

[20] Z. Lin and N. Hopper. Jack: Scalable accumulator-based nymble system. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 53–62. ACM, 2010.

[21] B. Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.

[22] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[23] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. http://cacr.uwaterloo.ca/hac/.

[24] Shigeo Mitsunari, Ryuichi Sakai, and Masao Kasahara. A new traitor tracing. *IEICE Transactions on Fundamentals*, 2002.
[25] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.
[26] National Institute of Standards and Technology. Recommendation for Key Management - Part 1: General (Revision 3). http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf.
[27] Edward J. Schwartz, David Brumley, and Jonathan M. Mccune. A contractual anonymity system. In *NDSS*, 2010.
[28] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.
[29] Stuart G Stubblebine, Paul F Syverson, and David M Goldschlag. Unlinkable serial transactions: protocols and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2(4):354–389, 1999.
[30] Bay Area Rapid Transit. Monthly ridership reports. http://www.bart.gov/about/reports/ridership.aspx.
[31] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean 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 *Proceedings of the 15th ACM conference on Computer and communications security*, pages 333–344. ACM, 2008.

APPENDIX A
BACKGROUND

A. Bilinear Groups

Let G, G_T be two cyclic groups of the same prime order q, and let g be a generator of G. We say G is *bilinear* if there is an efficiently computable map $e(\cdot,\cdot) : G \times G \to G_T$ satisfying

1) *Bilinearity.* $e(g^a, g^b) = e(g,g)^{ab}$.
2) *Non-degeneracy.* $e(g,g) \neq 1$.

This map is also called a *pairing*. Note $g_T \equiv e(g,g)$ is then a generator of G_T.

B. Complexity Assumptions

We describe the LRSW and DDHI assumptions in a group G. Note that both assumptions imply that computing discrete logarithms in G is hard.

LRSW assumption [7]. Let G be a group of prime order q, with generator g. The *LRSW assumption* is that any efficient algorithm $\mathcal{A}$ succeeds in the following experiment with negligible probability:

1) Choose $x \leftarrow \mathbb{Z}_q$ and $y \leftarrow \mathbb{Z}_q$, and give g, $X = g^x$, and $Y = g^y$ to $\mathcal{A}$.
2) $\mathcal{A}$ can query an oracle that, on input $m \in \mathbb{Z}_q$, chooses $A \leftarrow G \setminus \{1\}$ and returns (A, A^y, A^{x+mxy}). We denote by $\mathcal{M}$ the set of inputs on which $\mathcal{A}$ queries its oracle.
3) $\mathcal{A}$ *succeeds* if it outputs (m, A, B, C) with $m \notin \mathcal{M}$ and such that $A \neq 1$, $B = A^y$, and $C = A^{x+mxy}$.

Decisional Diffie-Hellman inversion (DDHI) assumption [24]. Let G be a group of prime order q, with generator g. The *DDHI assumption* is that for any efficient algorithm $\mathcal{A}$ and any polynomial t the following is negligible:

$$\Pr[\mathcal{A}(g, g^x, \ldots, g^{x^t}, g^{1/x}) = 1] - \Pr[\mathcal{A}(g, g^x, \ldots, g^{x^t}, g^y) = 1],$$

where $x, y \leftarrow \mathbb{Z}_q^*$.

C. Zero-Knowledge Proofs, Proofs of Knowledge

Consider an interactive protocol between a *prover* $\mathcal{P}$ and *verifier* $\mathcal{V}$, where the output is one bit from the verifier. We let $\langle \mathcal{P}(x), \mathcal{V}(y) \rangle = 1$ (resp. $= 0$) denote the event that $\mathcal{V}$ outputs 1 (resp. 0) in the interaction, which we refer to as "accepting" (resp., "rejecting"). This forms an *interactive proof system* for a language L if $\mathcal{V}$ runs in probabilistic polynomial time and the following properties are satisfied:

- **Completeness.** If $x \in L$, then $\Pr[\langle \mathcal{P}(x), \mathcal{V}(x) \rangle = 1]$ is negligibly close to 1.
- **Soundness.** If $x \notin L$, then $\Pr[\langle \mathcal{P}^*(x), \mathcal{V}(x) \rangle = 1]$ is negligible for arbitrary $\mathcal{P}^*$.

A *distribution ensemble* $\{X(a)\}_{a \in S}$ is a function from $S \subset \{0,1\}^*$ to probability distributions. Two distribution ensembles $X = \{X(a)\}_{a \in S}, Y = \{Y(a)\}_{a \in S}$ are *computationally indistinguishable* if for all probabilistic polynomial-time algorithms D and all $a \in S$

$$\Big| \Pr[D(X(a), a) = 1] - \Pr[D(Y(a), a) = 1] \Big| < \mu(|a|),$$

for some negligible function μ.

An interactive proof system for a language L is *(computationally) zero knowledge*, if for every probabilistic polynomial time interactive algorithm $\mathcal{V}^*$ there exists a probabilistic polynomial-time algorithm Sim (a *simulator*) such that the following two distribution ensembles are computationally indistinguishable:

- $\{\text{view}^{\mathcal{P}}_{\mathcal{V}^*}(x)\}_{x \in L}$
- $\{\mathsf{Sim}(x)\}_{x \in L}$

where $\text{view}^{\mathcal{P}}_{\mathcal{V}^*}(x)$ is a random variable describing the content of the random tape of $\mathcal{V}^*$ and the messages $\mathcal{V}^*$ receives during interaction with $\mathcal{P}$ on common input x.

Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. Define $L_R = \Big\{ x : \exists w \mid (x,w) \in R \Big\}$. We say that R is an $\mathcal{NP}$*-relation* if

- There exists a polynomial p such that for all $(x,w) \in R$, $|w| \leq p(|x|)$.
- There exists a polynomial-time algorithm for deciding membership in R.

If $(x,w) \in R$, we refer to w as a *witness* for x.

A interactive proof system for a language L is a *proof of knowledge* if the following conditions hold:

- **Non-triviality.** There is an interactive algorithm $\mathcal{P}$ such that for every $(x,w) \in R$, $P(\langle \mathcal{P}(x,w), \mathcal{V}(x) \rangle = 1) = 1$.
- **Validity.** There exists a probabilistic interactive algorithm $\mathcal{K}$ such that for every interactive algorithm $\mathcal{P}^*$, every $(x,y) \in R$, if $p(x,y,r)$ is the probability that $\mathcal{V}$ accepts in $\langle P^*(x,y), \mathcal{V}(x) \rangle$ when $\mathcal{P}^*$ has random tape r, then $\mathcal{K}$ outputs y' such that $(x,y') \in R$ in expected time $q(|x|)/p(x,y,r)$ for polynomial q.