Anon-Pass: Practical Anonymous Subscriptions

Michael Z. Lee*, Alan M. Dunn*, Jonathan Katz†, Brent Waters*, Emmett Witchel*
*The University of Texas at Austin, †University of Maryland
{mzlee, adunn, bwaters, witchel}@cs.utexas.edu, jkatz@cs.umd.edu

Abstract

We present Anon-Pass, a protocol and system for anonymous subscription services that allow users to anonymously authenticate while preventing mass sharing of credentials. Service providers cannot correlate users' actions, yet service providers are guaranteed that each account is in use at most once at a given time.

A central tension in anonymous subscription services is balancing a service provider's computational resource use with users' desire for flexible access. Anon-Pass focuses on practical anonymity, for example, in multi-media services, making all accesses to different items (e.g. articles, songs) appear to be from different users, but not decorrelating access to different parts of the same item. This level of practical anonymity allows Anon-Pass to provide users with flexible service at low cost to the provider. We measure the performance of a prototype and use it in several services including a music streaming service and an unlimited-use subway pass.

I. Introduction

Electronic subscriptions are widespread and quickly becoming the dominant mode of access for services like streaming music and video, news, and academic articles. While electronic subscriptions are convenient for users, they also reveal a lot of information, ranging from users' personal preferences to geographic movements. Many users want electronic services, but also want privacy. Simply "anonymizing" data does not always protect users' privacy: indeed, multiple "anonymized" datasets released for research purposes (e.g., the AOL search dataset [8] and the Netflix Prize dataset [13]) have been partially deanonymized through use of correlation or understanding of the semantics of the released data.

It is difficult to create systems that protect user privacy and simultaneously control admission (keeping out users who have not paid). Foregoing one of these two goals makes achieving the other considerably easier. If we require a user to simply login to an account (foregoing anonymity), a service can enforce that no user is simultaneously logged in twice. On the other hand, a subscription system could use a single shared identity for every user, preventing user identification (though traffic-anonymization via a system like Tor [6] may also be required to mask network-level identifiers). However, any subscribing user could share the secret with non-subscribers.

Ideally, we would have an *anonymous subscription system* that protects the interests of both the service and the users. At a high level, such a service needs two operations: **registration** and **login**. Registration allows a user to sign up for the service, at which point they may need to provide details that are identifying (e.g. credit card number, public key).

Login allows a registered user to access protected resources via their subscription. Informally, an anonymous subscription service ensures that a user's logins are not linked to the information they provided at registration, and login sessions are not linkable with each other.

While cryptographic protocols for providing anonymous credentials services already exist [9], there are problems with putting these protocols into practice. Many of these protocols are designed for thousands of concurrent users, but Netflix streamed one billion hours of content in July 2012 and has millions of subscribers[1]. At this scale, some proposed cryptographic operations require too much compute. Existing work does not focus on realistic evaluation scenarios, making it difficult to understand what performance effects would arise in a deployed system.

In this article, we describe Anon-Pass, our contribution toward making anonymous subscription services practical. Anon-Pass implements a new protocol for anonymous subscription services that achieves significant improvements in efficiency over prior protocols. Our insight is that not all accesses to protected resources need to be unlinkable. For example, a user probably does not care whether an adversary is able to determine that she watched both the third and fourth minutes of a particular movie as much as whether she watched both of two separate movies. This insight appears in Anon-Pass as a protocol for an *anonymous subscription with conditional linkage*, wherein some accesses to protected resources are linked to reduce computational cost when privacy is less important. We have formalized the notion of an anonymous subscription with conditional linkage and proven that the protocol used in Anon-Pass obeys this security definition under certain environmental and cryptographic hardness assumptions.

We first describe the formalization of an anonymous subscription with conditional linkage and discuss our design which integrates the cryptographic protocol into real systems. We then discuss both the cryptographic and system implementation. The code for Anon-Pass is freely available at https://github.com/ut-osa/anon-pass to encourage future work in this area.

[1]http://usatoday30.usatoday.com/tech/news/story/2012-07-03/netflix-online-video/56009322/1

II. Anonymous Subscriptions with Conditional Linkage

How do we achieve both anonymity and admission control? Ideally, we want each new client operation to appear to be from a new user, unrelated to previous users; however, when two operations are authorized with the same credential at the same time, then it must be clear they are from the same credential. Note that in our text "user" will denote the person and "client" the program (and sometimes the machine) performing actions. To keep credentials verifiable but also make them changeable, we divide time into equal length *epochs*, agreed on by both clients and servers. Clients use the current epoch number as input to a pseudorandom function (PRF) which allows them to change the login credential for each epoch. Changing login credentials allows each client to appear to be from a completely new user for the service in every new epoch, but each user can only create one unique login credential per epoch, preventing multiple simultaneous logins with the same credentials. A login credential only provides access for the duration of an epoch.

There is a tension between a service provider's desire for a long time epoch (to reduce server load) versus users' desire for a short epoch (to improve anonymity). The service needs to perform cryptographic checks during login, making login a computationally expensive operation. Consequently, the service provider wants to maximize epoch length. However, a user is only unlinked from previous activity once an epoch boundary has passed, and hence prefers a shorter epoch length. For example, when listening to a streaming music service, a user probably does not want to wait five minutes (or even one minute) for the next track to play.

We believe that users want unlinkability across accesses to distinct pieces of content, e.g., a movie, song, or news article. Therefore, we provide short epochs, giving users the ability to re-anonymize quickly if they so choose, while also providing an efficient method for users who do *not* need unlinkability to cheaply re-authenticate themselves for the next epoch. Users who are still watching the same movie or listening to the same song in a new epoch do not need unlinkability. Users want to decorrelate their watching "The Godfather" from listening to "Teenage Dream"; they do not need to decorrelate watching the first minute of "The Godfather" from watching the second.

In the rest of this section, we briefly define the cryptographic syntax and discuss the intuition behind our security guarantees.

A. Syntax and intended usage

An **anonymous subscription scheme with conditional linkage** consists of two algorithms (Setup and $\mathsf{EndEpoch}$) and three protocols (Reg, Login, and $\mathsf{Re\text{-}Up}$). Setup and $\mathsf{EndEpoch}$ are used for bookkeeping of internal service state and allow us to clearly define the operations for a service provider when used in our proofs. The three protocols comprise the primary functionality of the scheme: adding new users to the system (Reg) and authenticating clients to the system (Login and $\mathsf{Re\text{-}Up}$). $\mathsf{Re\text{-}Up}$ is not like normal authentication because it requires that the client is already logged into the system. The protocol implements the conditional linkage aspect of our scheme by extending the current user session into the next epoch.

System initialization begins by having the server run $\mathsf{Setup}(n)$ (where n is the security parameter) to generate the service public and secret keys (spk, ssk). Following setup, clients can register new users at any time; we denote the secret key of *client* i as sk_i. Independent of user 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, re-up, and end-of-epoch algorithms. 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.

B. Security

We define two notions of security: one ensuring that malicious users cannot generate more active logins than the number of times they have registered ("soundness"), and the other guaranteeing unlinkability for clients who authenticate using the Login protocol ("anonymity"). (On the other hand, clients who re-authenticate using the $\mathsf{Re\text{-}Up}$ protocol will be linked to their actions 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. $\mathcal{A}$ takes the role of a malicious user or users and tries to authenticate without using a valid credential it knows. $\mathcal{A}$ can perform a number of actions against the service: registering polynomially-many (malicious) users, controlling honest clients to login and re-up, and globally incrementing the current epoch. $\mathcal{A}$ succeeds if at any point in time, the number of logged in clients is greater than the number of $\mathcal{A}$'s users plus the number of honest clients currently 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 $1/2$. $\mathcal{A}$ takes the role of a malicious service trying to link users' access patterns. During setup, a random bit c is chosen, $\mathcal{A}$ sets the service public key, and two clients, U_0 and U_1 are registered. The game then proceeds in three phases.

Phase one: $\mathcal{A}$ may increment the epoch, query oracle $\mathsf{Login}(b)$ which performs a login for U_b, and query oracle $\mathsf{Re\text{-}Up}(b)$ which performs a re-up for U_b. In essence, $\mathcal{A}$ has full knowledge of the access pattern for U_0 and U_1. If both U_0 and U_1 are not currently logged in, then $\mathcal{A}$ is also allowed to proceed to phase two.

Phase two: $\mathcal{A}$ can perform the same operations, but query $\mathsf{ChallengeLogin}(b)$ and $\mathsf{ChallengeRe\text{-}Up}(b)$ instead. $\mathsf{ChallengeLogin}(b)$ is the same as $\mathsf{Login}(b \oplus c)$ and $\mathsf{ChallengeRe\text{-}Up}(b)$ is the same as $\mathsf{Re\text{-}Up}(b \oplus c)$. The second phase ends when neither client is logged at the beginning of an epoch.

Phase three: $\mathcal{A}$ interacts with the same power as in phase one.

At any point, $\mathcal{A}$ may output a bit c'. $\mathcal{A}$ succeeds if $c = c'$.

III. Design

This section describes the design and implementation 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 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 access 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 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 or BitCoins.

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.

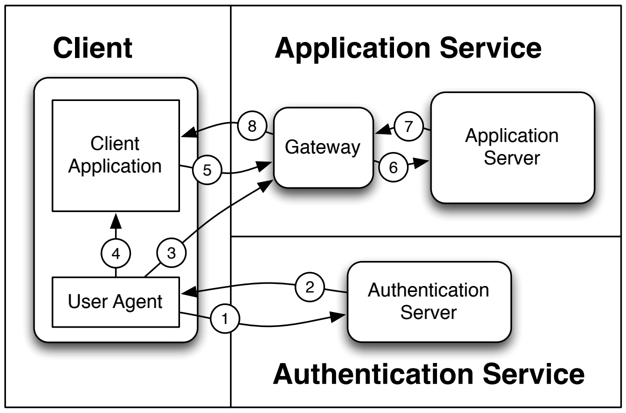

Fig. 1. The communication between the authentication server, resource gateway, and user agent with respect to the client and the service. ① Communication is initiated by the user agent and the authentication server verifies the credentials. ② The authentication server verifies the credentials and returns 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.

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[2] 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[3] 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. 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 user secret, communicating with the authentication server, and

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

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

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 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 user's 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 certificate that includes 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 session's initial login request.

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 clients 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 is the current epoch and PRF values of all logged-in users (e.g., by using a distributed hash table). Only storing information about currently authenticated users relieves a service provider from having to store all spent tokens, which requires unbounded storage (as is the case for some prior work, see §VII).

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 re-authenticate (login or re-up).

IV. Construction

In this section, we provide an overview of a cryptographic construction for a secure anonymous subscription scheme with conditional linkage (see [11] for details) that allows us to formalize and prove properties of Anon-Pass. Our construction uses a number of primitives – bilinear groups, zero-knowledge proofs of knowledge, a particular pseudorandom function family, and cryptographic assumptions from prior work. In our formal model (unlike the implementation), we assume protocols are not executed concurrently, and so there is a well-defined ordering among those events.

Similar to [1], our construction works by associating a unique token $Y_d(t)$, with each client secret d, in each epoch t. To register, a client obtains a blind signature from the service on a secret of its choosing. To log in, a client sends a token and proves in zero-knowledge that (1) it knows a service's signature on a secret, and (2) this secret was used to compute to the token that was sent. These tokens are used to determine admission to the service; the service accepts a token only if it has not been presented before in that epoch. Intuitively, soundness follows from the difficulty of generating signatures and anonymity follows from pseudorandomness of the tokens. (Further details and formal proofs of security can be found in the full version of this paper [11].)

On a technical level, we use the Dodis-Yampolskiy PRF [7] and an adapted version of one of the signature schemes proposed by Camenisch and Lysyanskaya [2] (CL signatures). These building blocks are themselves efficient, and also enable efficient zero-knowledge proofs as needed for our construction.

A client can authenticate during epoch t by sending the token $Y_d(t)$ and proving in zero-knowledge that the token is "correct." However, if a client is already logged in during epoch $t-1$, it can authenticate by sending $Y_d(t)$ and proving that $Y_d(t-1)$ uses the same client secret. 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.

V. Implementation

We implement the cryptographic protocol in a library, `libanonpass`, using the Pairing Based Cryptographic Library [12], PolarSSL[4] for clients, and OpenSSL[5] for the server. To show the flexibility of our protocol, we implement a number of usage scenarios including a streaming music service and an anonymous unlimited-use public transit pass. These applications are all large enough to highlight implementation issues specific to each context.

The authentication server and resource gateway are implemented as two separate Nginx[6] modules. The design does not need to share much state: the only state Anon-Pass needs to track is the current set of active login tokens. Both the authentication server and the resource gateway need to track this information; however, this can be consolidated to a single distributed hash table if both are run by the same service. The authentication server performs the cryptographic operations to try to keep expensive computations out of users' critical data path. Instead, the resource gateway only needs to verify a standard ECDSA signature and verify and update the table of active sessions.

The basic client is a wrapper around `libanonpass` and PolarSSL which provides an encrypted connection. The protocol messages are sent by using cookies to simplify server-side parsing and minimize client application modifications.

[4]https://polarssl.org/
[5]http://www.openssl.org/
[6]http://www.nginx.org

We briefly discuss two applications: a music streaming service and an unlimited use, public transit pass.

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 VLC,[7] a popular media player, to communicate with our user-agent and pass our session token as a cookie to the resource gateway.

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 weeklong "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).

We use the Java Native Interface (JNI) to call into `libanonpass` 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 period in which a login attempt from the same phone would fail. Though this guarantee is not precisely the same guarantee an unlimited ride transit pass currently provides, it does present an alternative that allows riders anonymity which still enforcing the lockout period.

Other anonymous subscription systems such as Unlinkable Serial Transactions [14] or anonymous blacklisting systems such as Nymble [10] or BLAC [15] 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.

There is a caveat however, because though Anon-Pass may be able to simulate the lockout period a transit provider needs to implement an unlimited use pass, it does not map directly to the model enforced today. This is because it, by definition, Anon-Pass cannot prevent time-sharing over a longer period of time. Once the 12-18 minute period is up, another user could use a duplicated pass, but the original rider would not have necessarily exited the system. The requirement of one physical device allowing access to one person is broken because the credential could be copied in an untraceable manner.

VI. Evaluation

We evaluate Anon-Pass through a series of micro-benchmarks and through its use in several applications. Further results, including a theoretical cost comparison to to prior work and an additional example service, can be found the full version of this paper [11].

A. *Measured Operation Costs*

There are overheads when integrating the protocols into a full system. Figure 2 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. Re-up is 7.7× faster than login.

B. *Public Transit Pass*

To evaluate the public transit pass scenario, we use the Android application and compute the time it takes to generate the login QR code on a commodity phone. Recall, the login QR code consists of a normal client login and three re-up tokens. The time to generate a login QR code on an HTC Evo 3D is 222 ± 24 ms. Power usage on this platform is minimal because the application does not need access to any radios on the phone.

On our server, the login and token verification costs 8.4 ms of CPU time, most of which is the cost of verifying the login portion. Putting this into perspective, in 2012 the Bay Area Rapid Transit had an average of 401,323 riders on weekdays for the months of August through October[8]. While we do not have data on traffic peaks, the total load is easily handled by Anon-Pass. One modern CPU core on our server can perform the approximately 400,000 verifications in just a little under an hour. These operations are trivially parallelizable across multiple cores and machines.

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

[8]http://www.bart.gov/about/reports/ridership.aspx

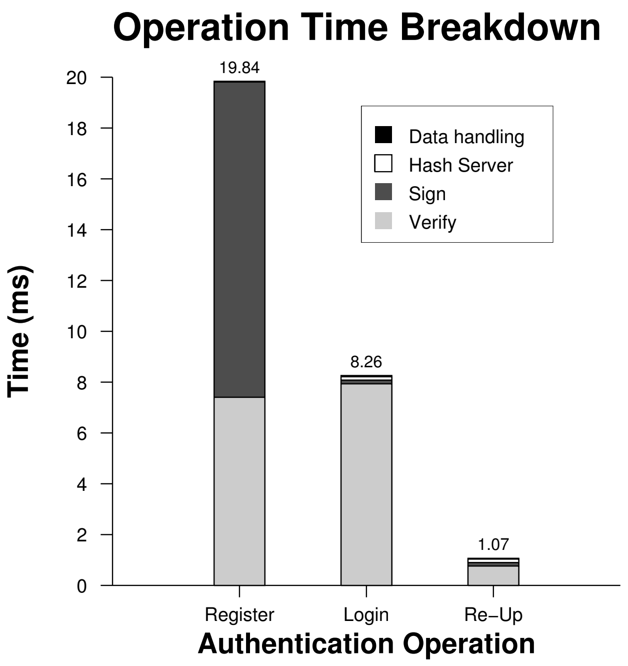

Fig. 2. The average cost of different requests on an unsaturated server. The bulk of the time is spent in signature verification.

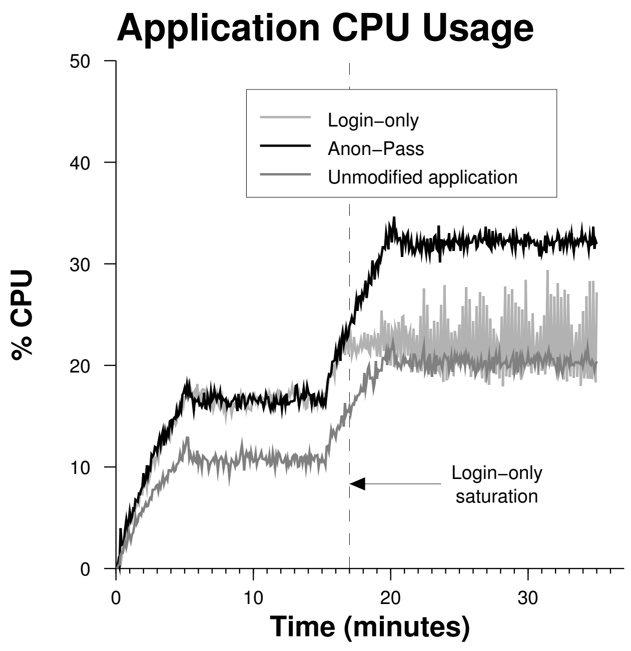

Fig. 3. The CPU usage on the application server measured every 5 seconds. The CPU usage with login-only follows the Anon-Pass behavior until the authentication server reaches saturation. Clients timeout and the application server has an overall drop in CPU utilization due to the lower number of clients successfully completing requests.

C. Streaming music service

We build an example streaming music service; however, we lack datacenter-level resources, and so must adapt the benchmark to run on our local cluster of machines. Our cluster's main constraints are the limited network bandwidth (1 Gbps) and memory available to run clients. 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 test bed.

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 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.

We simulate three different scenarios: a baseline system without any authentication, a login-only system in which users are not able to cheaply re-authenticate, and the full Anon-Pass system including re-up. When authentication is involved, we use an epoch length of 15 seconds for a better user experience. The scenario begins with 6,000 clients gradually logging over a period of five minutes. After a song is finished, the client unlinks itself and chooses a new song to stream. After 10 more minutes, we have an additional 6,000 clients login also over a five-minute period. We were not able to scale the experiment further because we exhausted the resources that could be devoted to additional clients.

Figure 3 shows the CPU utilization on the application server sampled once every five seconds. Anon-Pass uses more CPU resources than the baseline application because the service must perform an additional ECDSA verification once per epoch. Up until approximately 17 minutes into the experiment, a login-only service shows a very similar CPU utilization graph. However, at 17 minutes, the utilization graph drops lower and is much more jagged.

To see why this happens, we look at a graph of the CPU utilization on the authentication server (Figure 4). This 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 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. When a user is not able to re-authenticate, the music playback is cut off and the client is forced to retry. With 12,000 clients attempting to concurrently stream music, the login-only configuration has a client failure rate of 34% as compared to only 0.02% when also using re-up.

VII. Related Work

Our work continues research into *anonymous credentials* [3], which allow admission control while maintaining anonymity. We describe several themes of research in anonymous credential schemes.

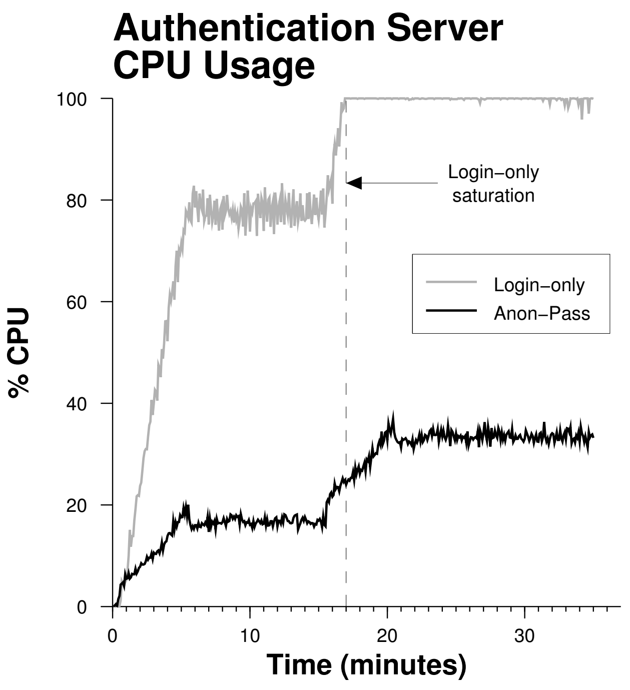

Fig. 4. The CPU usage on the authentication server measured every 5 seconds. The average CPU utilization for Login-only during the first stable segment (6,000 clients) is 77.9% (±2.42) and reaches saturation at about the 17 minute mark, or approximately 8,100 clients. The CPU utilization for Anon-Pass is 16.8% (±0.73) at 6,000 clients, and 33.4% (±0.96) at 12,000 clients (the second stable segment).

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. [5]) focused around *e-cash* [4], where credentials represented units of currency. The key task is to prevent double spending of the currency. However, currency-based systems are use-limited and do not translate to unlimited subscription services.

One of the earliest proposals for anonymous subscription services is "Unlinkable Serial Transactions" [14]. The system ensures a user can have only one valid credential at a time by recording every previously seen credential and issuing a new anonymous credential at the end of every transaction. One the other hand, Anon-Pass trades the potentially unbounded storage cost of UST with having users periodically contact the service.

More recent work has focused on *anonymous blacklisting systems* [9]. In these systems, a service is able to *blacklist* a user, excluding that user from accessing the service. Anonymous blacklisting systems usually allow a service to either reveal some form of linking information to prevent future access. Anon-Pass can only link a user when the user explicitly tells the service, which allows us to reduce the cryptographic cost.

One way to implement an anonymous subscription service is by blacklisting the user at login and removing the user at logout. However, a number of these schemes suffer from poor scalability. Indeed, BLAC [15] requires time linear in the number of blacklisted users to authenticate. A more recent system, BLACR, still measures scalability in terms of authentications per minute using 5,000 concurrent users. Contrastingly, Anon-Pass can sustain almost 500 login operations a second, and scales to 12,000 clients concurrently streaming music.

VIII. Conclusion

Anon-Pass is a building block for anonymous authentication and we have shown that it can be used in a range of applications. We have made our source code publicly available in the hopes of spurring further developments in this field. Anon-Pass demonstrates that it is possible to balance the tension between client flexibility and service load through the use of a lighter weight re-authentication operation for some usage models. However, there is still work to be done to convince services to provide unlinkability for their users.

IX. 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] 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.

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

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

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

[5] Ivan Damgård. 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.

[6] 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.

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

[8] Saul Hansell. Aol removes search data on vast group of web users, August 2006.

[9] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, 2011.

[10] Peter C. Johnson, Apu. Kapadia, Patrick P. Tsang, and Sean W. Smith. Nymble: Anonymous ip-address blocking. In *Privacy Enhancing Technologies*, pages 113–133. Springer, 2007.

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

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

[13] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 111–125, Washington, DC, USA, 2008. IEEE Computer Society.

[14] 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.

[15] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable Anonymous Credentials: Blocking Misbehaving Users Without TTPs. In *CCS*, 2007.