AnoNotify: A Private Notification Service

Ania Piotrowska[2], Jamie Hayes[2], Nethanel Gelernter[3],
George Danezis[2], and Amir Herzberg[1]

[1] Bar Ilan University, IL
[2] University College London, UK
[3] College of Management, Academic Studies, IL

Abstract. AnoNotify is a service for private, timely and low-cost online notifications. We present the design and security arguments behind AnoNotify, as well as an evaluation of its cost. AnoNotify is based on mix-networks, Bloom filters and shards. We present a security definition and security proofs for AnoNotify. We then discuss a number of applications, including notifications for incoming messages in anonymous communications, updates to private cached web and Domain Name Service (DNS) queries and finally, a private presence mechanism.

1 Introduction

A number of on-line applications require timely notifications about remote events: for example common mail delivery protocols notify users when new mail has arrived so that it can be retrieved. Other examples include updates of presence associated with social networking or instant messaging applications, and even notifications related to updates of cached records such as DNS or cached web records.

Traditionally, notification services provide no privacy *vis-à-vis* the notification service itself, that can observe both the content and the routing of notifications from the emitter of the event to the receiver. However, privacy preserving alternatives, such as anonymous communication systems [7], or private presence systems [5], rely on private notifications: an adversary should not be able to observe what events a user subscribes to.

AnoNotify provides such a private notification service, based on cryptographic constructions and the use of anonymous communication channels. In brief, publishers and subscribers of events share cryptographic keys, that allow them to share ever changing and unpredictable identifiers for events of interest. Subscribers can poll for events of interest privately using an anonymity system. To scale, subscribers only retrieve small parts of the event database, which we refer to as *shards*; we use Bloom filters to compress the size of each shard. AnoNotify provides privacy properties, as defined using a cryptographic and differentially private metric.

This paper is organized as follows: Section 3 presents the design of AnoNotify. In Section 4 we describe the dynamics of AnoNotify. Section 5 defines private

notification systems, and games used to show security. Section 6 contains the key security theorems. Section 7 discusses the costs of AnoNotify and compares it to PIR / DP5 [5]. Section 8 measures costs in the implementation of AnoNotify. Finally, in Section 9, we discuss applications in detail.

2 Related Work

Bloom Filters. The AnoNotify design is based on Bloom filters [3], used for representing a set membership of elements with some tunable *false positive* probability. The *false positive* error means that even if the element was not added to the filter, the check procedure may indicate its presence. A Bloom filter is defined as a bit array and constructed using several hash functions, which map an inserted element to the filter bits. To test if a particular element was added to the set, one should compute values of the hash functions for this element and check if all corresponding bits in the filter are set. A positive answer of the check procedure may yield a *false positive* error. Extensions of Bloom filters support additional functionalities, like deletion [27,15,4] or representing multisets [9]. In [2] authors present metrics as K-anonymity and $\gamma-$deniability to measure the privacy and utility of Bloom filters but the resulting privacy properties are weak. RAPPOR [14] allows the private collection of crowd sourced statistics as randomized responses in Bloom filters, while guaranteeing ϵ-differential privacy. RAPPOR uses input perturbation locally on the client side, however extracting results requires sophisticated statistical techniques.

Privacy in Remote Storage. Private information retrieval (PIR) allows a client to retrieve privately a single record from a remote public database. The naive solution retrieves all records from the database, but PIR protocols are more efficient in terms of bandwidth [8,19,10].

Social applications have required private presence notifications. Traditional implementations of presence give a central server the social graph of users. Protocols like Apres [22] and DP5 [5] offer privacy-preserving notification services. Apres splits the time into epochs and hides the correlation between the connectivity of the clients in every two epochs. DP5 offers stronger privacy guarantees, by using PIR to hide all information about the social graph.

Anonymity. The most widely deployed anonymity system is Tor [12]. In Tor, communications are routed through a network of relays using onion routing, which hides the senders location and ensures unlinkability between the user and the visited website. Although Tor is popular it is vulnerable to traffic analysis attacks, and for stronger anonymity properties mix networks have to be used [7] at the expense of latency. Receiver anonymity systems, such as nymservers [24], may also be used to route notifications to users. Pynchon Gate [28] proposes a pseudonymous message retrieval system based on a distributed PIR scheme. It allows the pseudonym holders to receive messages, while ensuring unlinkability among messages and their recipients with forward secrecy.

3 The Design of AnoNotify

AnoNotify protects the relationship privacy of publishers that send notifications, and subscribers that request notifications, from a hostile network and infrastructure. The system operates in sequential epochs denoted by t, for time. Our techniques are, in this respect, inspired by Ben Laurie's Apres [22], and the subsequent DP5 [5].

Straw-man Design. We first present a straw-man design, loosely inspired by DP5, and argue informally for its security but also its inefficiency. A single server acts as the infrastructure. Publishers and subscribers of notifications privately agree on a secret random identifier for a specific notification event. When a publisher wishes to send a notification, she simply sends the pre-arranged random identifier to the server which stores it forever. Meanwhile, subscribers of notifications access the single server, and periodically download the full database of stored notification identifiers, looking for identifiers they recognize as events.

This naïve design is secure: since subscribers of notification always download the full database, an adversary at the server could not distinguish which notification they seek. However, performance is very poor: the database grows forever, and downloading the full database is very expensive. One option is to use PIR, for more efficient private download. However, DP5 [5] illustrates that PIR also has scalability limitation.

3.1 Goals and Setting

AnoNotify is a service connecting *notification publishers* with specific *notification subscribers*. By convention we call a notification source Alice (as the traditional publisher in cryptography) and a notification subscriber Bob.

The AnoNotify protocols make use of a number of infrastructure *servers* managing the routing of one or more *shards* each. We denote those shards as s_i and their total number as S. Publishers and subscribers connect through this infrastructure, as illustrated in Figure 1. AnoNotify aims to scale well: the capacity of the system should increase as we add more servers to the infrastructure. The goal is to keep the relationship between publishers and subscribers of notifications private from the servers.

3.2 The AnoNotify System Design

The (basic) AnoNotify system. A number of servers collaborate to provide the private notification system by each managing one or more shards of notifications. A publisher Alice who wishes to send a notification to a subscriber Bob, simply provides Bob with a secret channel key (ck) (which could also be the result of a Diffie-Hellman [11] key exchange). Upon sending a notification to Bob, Alice derives an epoch specific identifier for the event as $ID^t = PRF_{ck}(t)$.

The publisher Alice then selects a shard s_i, using $i \leftarrow ID^t \bmod S$, and sends ID^t *anonymously* to the server managing this shard. This spreads different

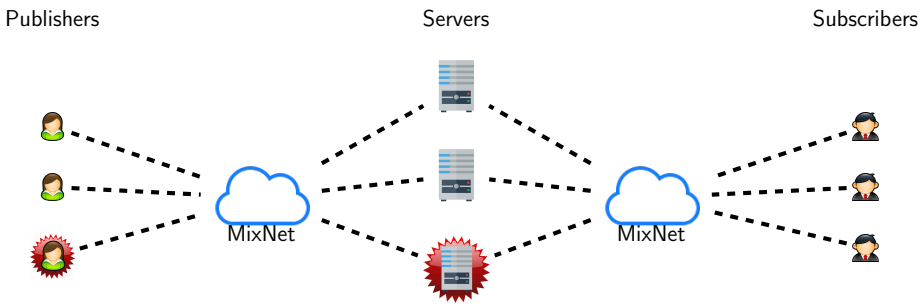


Fig. 1. The AnoNotify architecture

identifiers for notifications across servers. Upon receiving ID^t a server adds it to a Bloom filter for the shard s_i at epoch t, of bit size l, which includes all received notification identifiers for a particular epoch t. The server makes this available to download immediately or at the start of the next epoch.

At the beginning of epoch $t+1$ Bob reconstructs ID^t for the notifications he wishes to check for the previous period by computing $ID^t = PRF_{ck}(t)$, and computes the shard i in which ID^t might be stored. Bob then *anonymously* downloads the filter for shard s_i and epoch t and checks whether ID^t is in the filter or not. This may yield a false positive match, misleading Bob into thinking that the notification was present when it was not.

The (continuous) AnoNotify system. The basic system results in a long waiting period of one epoch between when a notification is sent and when a notification is received, depending on the epoch length at the server. We present a variant of the AnoNotify (basic) system that does not require the server and publisher-subscriber epochs to be synchronized and allows the servers to serve notifications corresponding to an arbitrary moving time-window: over time each server constructs a Bloom filter that contains all notifications within a fixed past period. It does so incrementally, and may also wish to modify the Bloom filter parameters to ensure a fixed error rate – which adds security considerations.

Instead of having a server side fixed epoch each served by a fresh Bloom filter, notifications are added and deleted continuously as events are included and excluded from a time-window. The length of this time window is application dependent. Since standard Bloom filters cannot be efficiently updated, we use a counting Bloom filter: the server keeps a count of the number of items contributing to each bin of the Bloom filter [15,4]. Hence, adding an element to the Bloom filter involves increasing the appropriate bins by one, and removing an element decreasing them. The counting Bloom filter can then be quantized to take no more space, and is similarly susceptible to false positives.

4 Dynamics of AnoNotify

We assume that all servers globally agree on an acceptable range of false positive error rate for the Bloom filter $[f_{min}, f_{max}]$. Servers then compute the optimal parameters of the filter, given at any time the number of notifications to be included to maintain the error rate within acceptable range. However, all servers must agree on identical parameters for the filter, to ensure that its length does not leak the shard Alice requests. Thus, we propose a distributed protocol for servers to communicate with each other their predicted error rates, and reach a consensus on the parameters of the Bloom filter used by all.

A server, decides whether it needs to change the new Bloom filter parameters in the following manner:

- At time t_i with incoming notification rate v_i, the server has Bloom filter parameters (l_i - length of Bloom filter, k_i - number of hashes), that have a false positive rate of f_i which is within some system-wide agreed false positive range $f_{min} \leq f_i \leq f_{max}$.
- If at time t_{i+1} with rate v_{i+1}, $f_{i+1} \in [f_{min}, f_{max}]$ the original Bloom filter parameters are kept, since the error is within bounds.
- If however, at time t_{i+1} and rate v_{i+1}, the error rate given the old parameters (l_i, k_i) is not acceptable ($f_{i+1} \notin [f_{min}, f_{max}]$) then the server initiates a re-parameterization (l'_{i+1}, k'_{i+1}) of the Bloom filter to yield an error rate f'_{i+1} within bounds. Only upon correct completion of the consensus protocol can new filter parameters be used.

The aim of the above mechanism is to minimize communication costs (Bloom filter size), and thus the parameters of the filter will fluctuate upwards and downwards to keep the error rate within acceptable bounds and communication costs low.

If a servers notification rate fluctuates to the extent that the false positive rate falls outside of the system-wide error bounds, the server will notify all other servers in the system that they must change their Bloom filter parameters and all other servers must choose the largest parameter. Once all servers have agreed on the new Bloom filter length, this is posted to a public bulletin board and digitally signed by all servers. If a subscriber subsequently receives a Bloom filter of a different length to the one they expect, they mark that server as untrusted. The server can also send, along with the Bloom filter, the expected false positive rate to the subscriber. This allows the subscriber to be judicious about whether to perform a subsequent action based on whether the Bloom filter informs them of a notification.

5 Secure Notification System

5.1 Entities, Threats and Assumptions

In AnoNotify we assume, that the system might be exposed to both a passive adversary, corrupted servers that observe all shards, or malicious users. A passive

adversary observes part or the whole network and will try to learn relationships between publishers and subscribers. However, we also assume, that a large number of AnoNotify concurrent users (publishers and subscribers) are honest, and follow the protocol faithfully. In fact the security theorem we present relies on the fact that target users can hide amongst other honest users' notifications and queries. In AnoNotify the communication among the publishers and the infrastructure as well among the requesting subscribers and the servers should be done using an anonymity network. The anonymity system is immune to traffic analysis, namely, from the point of view of the adversary it provides a perfect secret permutation between its input and output messages. The messages sent among publishers, servers and subscribers are padded into equal length blocks to prevent privacy leakage through message size.

5.2 Notification System: Definition

A notification system $\mathcal{N}$ is a set of seven probabilistic algorithms:

- $\mathcal{N}$.GenSystem takes as input $n \in \mathbb{N}$ users, $S \in \mathbb{N}$ shards, and security parameters $\kappa \in 1^*,\ \epsilon > 0,\ \delta \geq 0$, and outputs server initial state σ, packet length l and public information $\pi \in \{0,1\}^*$.
- $\mathcal{N}$.GenChannel takes as input public information π and outputs channel key $ck \in \{0,1\}^*$.
- $\mathcal{N}$.Notify takes as input channel key ck, epoch $t \in \mathbb{N}$ and state σ, and outputs a notification μ and new state σ'.
- $\mathcal{N}$.ProcNotify takes as input notification μ, epoch t, state σ, and server key sk, and outputs response ρ and new state σ'.
- $\mathcal{N}$.Query takes as input channel key ck, epoch t and state σ, and outputs a query ϕ, and new state σ'.
- $\mathcal{N}$.ProcQuery takes as input query ϕ, epoch t, state σ, and server key sk, and outputs response ρ and new state σ'.
- $\mathcal{N}$.ProcRespose takes as input response ρ, channel key ck, epoch t and state σ, and outputs return code ψ and new state σ'.

For simplicity we assume that the length of all notifications, queries and responses, is always a fixed value l, generated as part of the system parameters (by the $\mathcal{N}$.GenSystem algorithm).

5.3 Security Definition

In this subsection we define an *Indisitinguishable-Notification Experiment* (IndNotExp), addressing the threats identified in 5.1. In this experiment multiple rounds of notifications and queries are executed, and the adversary has full control over which honest users notify and which honest users query for their respective notifications. Then, the adversary chooses an epoch, and two target honest users $n-1$ and n, of which the adversary knows the secret notification keys. One of them is included in the round at random. Then, given all information gained by observing all epochs queries and notifications, the adversary tries to guess which of the two users was present in the last round.

```
procedure IndNotExp(N, A, n, κ, Δ, b ∈ {0,1})
    (σ, l, π) ← N.GenSystem(n, κ, Δ).
    for i = 0, ..., n do
        ck_i ← N.GenChannel(π)
    end for
    Give ck_{n−1}, ck_n, n, κ, Δ, π to A
    for t = 0, ... do
        (notifications, queries)← (0, 0)
        for i = 0, ..., n do
            if A(i, 'notify') = 1 then                    ▷ Adv. chooses notifications.
                Increment C_n
                μ_i ← N.Notify(ck_i, t)
                σ ← N.ProcNotify(μ_i, t, σ)
            end if
        end for
        for i = 0, ..., n do
            if i = n − 1 ∧ A('done?') = 1 then
                φ_T ← N.Query(c_{n−b}, t, σ)
                Give φ_T to A
                ρ_T, σ ← N.ProcQuery(φ_T, t, σ, sk)
                return (A(φ_T, ρ_T), C_n, C_r)            ▷ Guess of the Adversary.
            end if
            if A(i, 'req') = 1 then
                if A(i, 'notify') ≠ 1 then
                    Increment C_r          ▷ Number of queries with unseen notifications.
                end if
                φ_i ← N.Query(ck_i, t, σ)                 ▷ A(φ_i): Adv. sees all queries.
                ρ_i, σ ← N.ProcQuery(φ_i, t, σ, sk)
                N.ProcRespose(ρ_i, ck_i, t, σ)
            end if
        end for
    end for
end procedure
```

We now define a $\Delta-$private notification system. Note that we restrict the adversary in the epoch it issues the challenge: in that round a minimum of $u < C_r$ queries from honest users must be issued, of which the corresponding notifications were not observed. This can be relaxed, but requires a more sophisticated security proof which we leave as future work.

Definition 1. *A notification system $\mathcal{N}$ is (u, n, Δ)-private if for any PPT adversary $\mathcal{A}$ holds:*

$$\Pr\left[\begin{array}{c}(g, C_n, C_r)) \leftarrow \textsc{IndNotExp}(\mathcal{N}, \mathcal{A}, n, \kappa, \Delta, b) : \\ g = b \wedge u < C_r\end{array}\right] \leq \frac{1}{2} + \Delta + \mathrm{negl}(\kappa)$$

The probability is taken over all coin tosses, including uniform choice of bit b, and where $\mathrm{negl}(\cdot)$ *is a negligible function; the inequality should hold for sufficiently large security parameter κ. For simplicity, we write Δ-private.*

The threat model captured by the *Indisitinguishable-Notification Experiment* is very generous to the adversary: the adversary has full visibility into the processing of all notifications and all query requests at all shards of the system for as many epochs as she wishes. Furthermore, the adversary is given the secrets associated with the notifications of the two potential target users. Ultimately, the adversary needs to decide which user took part in the protocol run with full knowledge of the secrets they share with senders of notifications.

6 The Security of AnoNotify

In this section we prove that AnoNotify is a secure (i.e., Δ-private) notification system, as in Def. 1 in the previous section.

Security Theorem 1 *The AnoNotify system is a Δ-private notification system, for $\Delta > 0$ satisfying the following inequality. For any $\epsilon > 0$,*

$$\Delta \leq \frac{1}{2}\tanh\left(\frac{\epsilon}{2}\right) + \exp\left(-\frac{(u-1)}{2S}\left(\frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)^2\right).$$

Proof. See Appendix A.

The proof depends on a key lemma, proving a differentially private [13] security bound for the final challenge round of IndNotExp. In the challenge round u honest users whose corresponding notifications were not observed send queries. These include either user n or user $n-1$ according to the challenge b. Since all other users query a previously unseen ID through an anonymous channel, they each download a random shard from S. Based on the volume of queries the adversary wants to distinguish which user, $n-1$ or n, was present. We can prove the following lemma for this simple setting:

Lemma 1. *Let X_0, X_1 be the query volumes observed by the adversary at shards s_0, s_1 and let $n-1, n$ define events when a particular challenge user participates in the round. An (ϵ, δ)-*differential privacy *bound by which:*

$$\Pr[X_0, X_1 | n-1] \leq e^{\epsilon} \Pr[X_0, X_1 | n]$$

is ensured for $\epsilon > 0$ with probability at least $1-\delta$:

$$\delta \leq \exp\left(-\frac{(u-1)\tau^2}{S}\right) + \exp\left(-\frac{(u-1)}{2S}\left(\frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)^2\right) \text{ for } \tau \in [0,1].$$

Proof. See Appendix B.

Interestingly, we also derive a generic result linking an (ϵ, δ)-differentially private mechanism to the advantage of an adversary in the context of trying to guess the outcome of a binary challenge.

Lemma 2. *Let $\mathcal{K}$ be an (ϵ, δ)-differentially private mechanism, on two private inputs E_1 and E_2 for which $\Pr[\mathcal{K}|E_1] \leq e^{\epsilon} \Pr[\mathcal{K}|E_2]$ with probability at least $1-\delta$. If the adversary is provided with $\mathcal{K}$ resulting from either E_1 or E_2, and tries to guess the input, she succeeds with probability:*

$$\Pr\left[\begin{array}{c} b \leftarrow \{0,1\}; \mathcal{K} \leftarrow \mathcal{K}(E_b); \\ \text{guess} \leftarrow \mathcal{A}(\mathcal{K}, E_1, E_2) : \\ b = \text{guess} \end{array}\right] \leq \frac{1}{2} + \frac{1}{2}\left(\frac{e^{\epsilon}-1}{e^{\epsilon}+1}\right) + \delta + \text{negl}(\kappa)$$

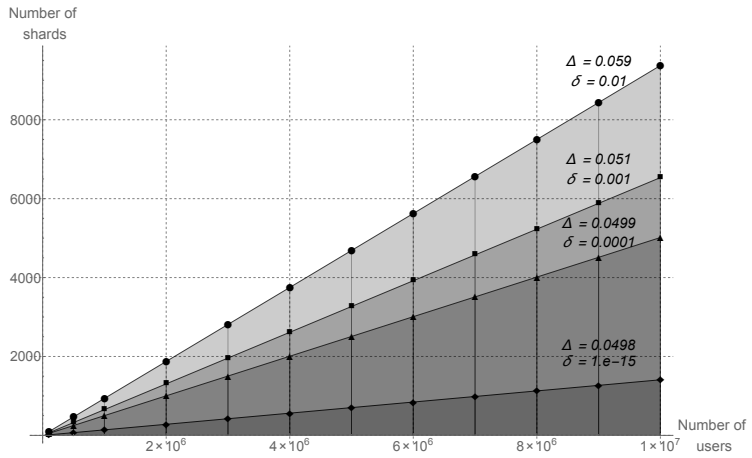


Fig. 2. The maximum number of shards for an increasing number u of requesting users, $\epsilon = 0.2$ and selected values of δ and Δ.

Proof. See Appendix C.

In Figure 2, we illustrate the maximum number of shards allowed to ensure that AnoNotify guarantees (ϵ, δ)-*differential privacy* for specific values of ϵ, δ, with increasing number of querying users. We also estimate the concrete value of the adversary advantage Δ based on Theorem 1. Note that the upper bound on δ in Lemma 1 is constant as long as the ratio $\frac{u-1}{S}$ is constant, which justifies a linear plot.

7 Analytical Performance Evaluation

Bandwidth. We evaluate the bandwidth cost of multi-shard AnoNotify against the naïve design using a multi-server IT-PIR [8] scheme inspired by DP5 [5]. Let the number of servers in AnoNotify be S, and the number of servers in the PIR scheme be S'. Since in AnoNotify all Bloom filters are of equal size, say l, the number of bits transferred is $nl \cdot m_x$ where n is the number of subscribers that downloaded the Bloom filter and m_x is the cost of using a mixnet to transport data (to be fair we will assume $m_x = S'$). For the IT-PIR scheme the cost is $nS'\sqrt{v}$, where v is the number of bits in the server's database.

Additionally, since AnoNotify may yield false positives, we must consider the bandwidth cost of a subsequent action of a subscriber given that they received a notification, which we denote as a. We intentionally do not specify what this action is, as AnoNotify could be used in a variety of applications. Let $k \leq n$ be the number of subscribers who received a notification. Let f be the error rate of Bloom filter, then $h = nf$ subscribers will incorrectly think they have received

Table 1. The cost of action a for particular parameters of the system.

Servers PIR	Shards AnoNotify	Bloom filter size [bits]	Mixing Cost	**Notifications Number**	Error Rate	**a** [bits]	ϵ	δ	Δ
10	5000	10^4	10	10^7	0.091	$2.9 \cdot 10^5$	0.2	0.0001	0.049
10	6500	10^4	10	10^7	0.044	$6.1 \cdot 10^5$	0.2	0.001	0.051
10	9000	10^4	10	10^7	0.013	$2.0 \cdot 10^6$	0.2	0.01	0.057
10	16000	10^4	10	10^7	0.0005	$5.8 \cdot 10^7$	0.2	0.1	0.135
10	10^4	10^4	10	10^7	0.008	$3.2 \cdot 10^6$	0.2	0.014	0.063
10	10^7	10^4	10	10^{10}	0.008	$5.7 \cdot 10^8$	0.2	0.014	0.063

a notification. Hence the cost of performing actions in AnoNotify is $a(k+h)$, whereas in the PIR scheme the cost is ak since no false positives will occur.

The total cost of AnoNotify is $nl \cdot m_x + a(k+h) = nl \cdot m_x + a(k+nf)$. The total cost of the PIR scheme is $nS'\sqrt{v} + ak$. We want to estimate the cutoff cost a for AnoNotify to be less expensive than a PIR scheme, hence we require $nl \cdot m_x + a(k+nf) < nS'\sqrt{v} + ak$. This gives $a < \frac{S'\sqrt{v} - (l \cdot m_x)}{f}$.

We note that the false positive rate f and the size of the Bloom filter l are related by $f \approx (1/2)^{l \log 2/m}$, where m is the number of messages in the filter, that we assume are approximately N/S where N is the total number of notifications. Similarly, the database in an IT-PIR system would need at least $v = N \log N$ bits to store a list of up to N distinct notifications. Thus, it is preferable to use the AnoNotify system over IT-PIR when the cost of an action a is lower than the following threshold: $a < (S'\sqrt{N \log N} - (l \cdot m_x))2^{\frac{lS}{N} \log 2}$.

In Table 7, we present the threshold cost of action a in AnoNotify under different system and security parameters and estimate the values ϵ, δ, Δ using Theorem 1. AnoNotify can support thousands or even millions of users at a lower cost than PIR for actions costing kilobites to megabytes respectively. Furthermore, the security parameters are mostly affected by the ratio of shards to queries which can be kept constant to achieve satisfactory privacy.

Latency. In the basic AnoNotify, a notification sent by a publisher in epoch e_i, will become available to a subscriber in epoch e_{i+1}. The time between when a notification is sent and when it can be read is $|e| + t$, where t is the time taken by the notification to be routed through the mix network and $|e|$ denotes the server epoch length. Note, that this time t is dependent on the amount of traffic passing through the mix network, and the mix networks flushing mechanism. In the continuous AnoNotify, unlike the basic version, the latency is dependent solely on the mix network since notifications are added to the Bloom filter as soon as they are received by the server.

Refresh rate, epoch length, cost and privacy. In both the basic and continuous AnoNotify systems publishers and subscribers must decide on an epoch length, based on which their notification identifiers will change. There is a clear

trade-off: shorter epochs mean shorter wait times but result in the subscribers requesting more often.

In the basic AnoNotify, although all notifications received in an epoch are available at the start of the next epoch, if a publisher-subscriber epoch is much smaller than the server epoch, the subscriber will have to request many times to check if a notification was received in any of the possible publisher-subscriber epochs. For example, let the publisher-subscriber epoch length be denoted $|sr|$, and let $5|sr| = |e|$. During one server epoch, the publisher and subscriber generate five epoch specific identifiers $\{ID_0, ID_1, ID_2, ID_3, ID_4\}$. Then, at the start of the next epoch the subscriber will have to request each of the five identifiers. Clearly the smaller the publisher-subscriber epoch length, the more often a subscriber will have to request the system for notifications. In the continuous AnoNotify instead of a fixed server epoch there is a moving time window in which notifications are deleted and added. Publisher-subscriber epochs will be set depending on the granularity of activity they wish to capture.

Publisher-subscriber epoch lengths are entirely context dependent; a social network presence notification system will likely have much shorter publisher-subscriber epoch lengths than a storage system.

8 Experimental Evaluation

Three key advantages of AnoNotify over previous work [5] are efficiency, extremely low cost (even in large scale), and ease of implementation. In this section we describe a prototype implementation of AnoNotify, based on web technologies for the server components, and Tor as an anonymity system, and in particular, discuss design decisions to improve performance.

8.1 Distributing the Load

In epoch-based presence mechanisms, such as DP5 [5], we expect a high load of queries at the beginning of every epoch. At that time, clients both update their presence and retrieve the presence of their friends. To distribute the load, AnoNotify follows a different approach to distribute load without increasing average latency.

An AnoNotify client periodically sends a single request to both send a notification and perform a query. Assume epochs of $|e|$ minutes each. Every AnoNotify client uniformly chooses an offset time from $(0, |e|)$ from the beginning of the epoch and at that time sends the combined notification and query to the shard.

To implement the basic AnoNotify scheme, each AnoNotify server maintains two Bloom filters for each shard it controls, the *current* and the *next*. Upon receiving a notification the server always adds elements to the *next* Bloom filter and returns the *current* filter. At the end of every epoch, the *next* becomes the *current* and *next* is set to be an empty Bloom filter.

The data in the *current* Bloom filter that users receive is updated to $|e|$ minutes before the response is generated on average. This latency is the same,

on average, as for clients accessing the server at the beginning of the epoch (like in DP5): since the client sends the requests uniformly within each epoch, there is an average delay of $\frac{|e|}{2}$ minutes from the beginning of the current epoch. The subscriber is interested in notifications from a previous epoch, which were also posted at a uniform time within the previous epoch. Therefore, there is additional delay of $\frac{|e|}{2}$ minutes on average. Hence, the total delay is a single epoch ($|e|$ minutes) on average, as predicted by the theoretical latency analysis for the basic mechanism.

8.2 Implementation & Infrastructure

We implement AnoNotify as a web-server that clients can easily access through the most popular anonymity network today, Tor [12]. We are aware that Tor only provides anonymity properties against a local or limited passive adversary, and thus the experimental system inherits this limitation. Since we are concerned with performance we focus on supporting as many clients as possible, and decreasing the connection time between the client and the server.

Our implementation of AnoNotify consists of two servers: a front-end with whom the clients communicate, and a back-end server that maintains the Bloom filters. We designed the basic AnoNotify such that queries are served as requests for a static resource: since those only need to retrieve the Bloom filter corresponding to a previous epoch. Leveraging this, the task of the front-end server is no harder than a simple web-server serving static large resources; caching and content distribution network may be used to speed this up. We expect the size of the Bloom filter served to be similar to the size of an image, between several kilobytes to a few megabytes.

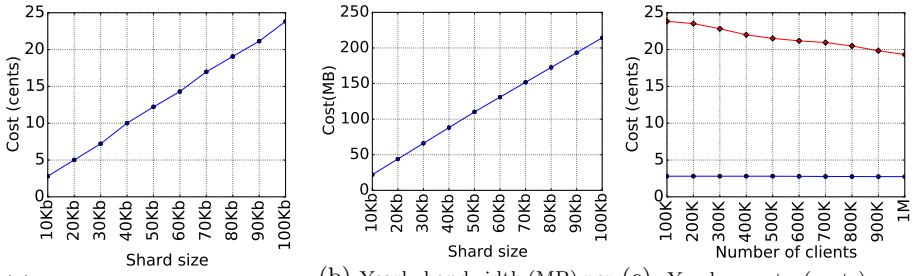
To perform a query and retrieve the Bloom filter, AnoNotify clients just send an HTTP GET request to the front-end server. To optionally register a notification, the clients can additionally send the notification identifier for the current epoch as a parameter to the HTTP request. The front-end server immediately responds with the relevant *current* Bloom filter, that is stored as a static file, and forwards the request to the back-end server to update the *next* filter. At the beginning of every epoch, the back-end server sends the *next* Bloom filters, one for each shard, to the front-end server, and the front-end server replaces the *current* Bloom filter with it.

We used Nginx[4] for the front-end server due to its high performance in serving static resources. We implemented the back-end server in Java, relying on Netty[5]netty, a non-blocking I/O (NIO) client-server framework. We relied on Google Guava's implementation of Bloom filter[6]. The front-end implementation simply consists of the Nginx configuration file, and the back-end is 300 lines of Java code.

[4] The NGINX Web Server`https://www.nginx.com/`

[5] The Netty Framework `http://netty.io/`

[6] Guava: Google Core Libraries for Java `https://github.com/google/guava`



(a) Yearly cost (cents) per client for different shard sizes

(b) Yearly bandwidth (MB) per client for different shard sizes

(c) Yearly cost (cents) per client using 10Kb (blue) and 100Kb (red) shards, for different numbers of clients

Fig. 3. AnoNotify's implementation evaluation summary. The system scales perfectly for the increasing number of clients. Larger shards implies higher bandwidth and cost per client. The cost evaluation was done according to Amazon EC2 m4.large instances.

8.3 Evaluation and Comparison to DP5

To evaluate AnoNotify, we ran an AnoNotify server on a single Windows7 OS, 8GB RAM machine. The back-end and the front-end ran as two processes. From another machine, we ran our client program from several processes to simulate 100K requests in epochs of 5 minutes. We tested the system for shards from 10 to 100Kb. Larger shards mean larger Bloom filters to retrieve and higher bandwidth.

A single machines served 100K clients for a shard size was up to 30Kb. For larger shards we encountered sporadic failures for some clients, and had to add additional servers to handle some shards. The design of AnoNotify allows distributing the shards among several machines without overhead. The yearly cost of an Amazon EC2 `m4.large` instance (in April 2016), which is equivalent to the machine we used, is $603 per year. Dividing the cost of additional machine to 100K clients implies minimal additional cost of less than a single cent per client. Our measurements indicate an additional server is required for each 30Kb increase of the shard size.

We calculated the cost of running AnoNotify in the Amazon cloud. The main factor in the cost calculation was the bandwidth that is increased linearly as a function of the shard size. However, the bandwidth cost per byte is decreased as the system consumes more bandwidth, e.g., for larger shards and for more clients.

Figure 3 illustrates our costs estimation, extrapolated from measurements using our experimental setup, for a full year of operation in the Amazon cloud. The costs are illustrated in monetary values, on the basis of the cost of an Amazon EC2 `m4.large` instances. The results shows that AnoNotify is indeed very efficient, and extremely cheap to operate in the real world. Figure 3(a) shows that the yearly cost per client ranges from a few cents (shards of 10Kb)

to less than a quarter (shards of 100Kb). Figure 3(b) shows the linear growth in the yearly bandwidth used by AnoNotify client as a function of the shard size. However, as depicted by Figure 3(c), the AnoNotify scales perfectly in the number of clients, such that the cost per client even decreases as there are more clients in the system.

Compared to the thousands of C++ lines in the DP5 [5] implementation, AnoNotify was significantly easier to implement. Although implemented in Java, our implementation efficiently supports a hundred thousand clients, and can be scaled to millions of clients easily with significantly lower yearly cost of a few cents. Moreover, unlike DP5, the cost per client does not grow when more clients are using the system.

9 Applications

Notification-only Applications. The first application is a privacy-preserving version of event-notification services, such as the popular Yo application [31]. Yo and similar applications allow one user to send a content-free notification to peer(s). In Yo, the receiving applications notify the user by transmitting the word "Yo", in text and audio. Such event notification services can be used for social purposes, as well as to provide simple information about events, e.g., Yo was used to warn Israeli citizens of missile strikes [1].

The second application is *Anonymous Presence Services*. The goal of anonymous presence services is to allow users to indicate their 'presence', i.e., availability for online communication to their peers. It is one of the functionalities usually provided by social networks such as Skype and Facebook. A privacy-preserving presence protocol, providing presence indications to users while hiding their relationships, was presented in [5]. Their solution relies on cryptography and is rather complex to implement, whereas AnoNotify provides an easier-to-implement and more efficient solution.

The third application is privacy-preserving *blacklists*, e.g., of phishing domain names. The goal is to allow a relying party, e.g., a browser or email server, to check if a given domain name (or other identifier) is 'blacklisted', without exposing the identity of the domain being queried. In particular, all major browsers use some 'safe browsing' blacklist to protect users from phishing and malware websites. Google Safe Browsing (GSB) alone accounts for a billion users to date [21]. To protect users privacy, clients do not lookup the suspect URL or domain-name, instead the query is for a cryptographic hash of the domain-name or URL. However, as already observed [18], providers can still identify the query. AnoNotify provides an alternative which truly protects privacy, and with comparable overhead. We note that Bloom filters are already widely used to improve efficiency of blacklists, e.g., see [25,17].

In all applications, AnoNotify allows preserving the privacy of users, by hiding the relationships between users and the notifications they receive or send. The use of AnoNotify is easy, and has insignificant performance overhead in addition to the use of anonymous channels. However, notice that AnoNotify exposes the

total number of clients currently connected to the system. We believe this is not a concern in many applications. Indeed, many services publish an estimate of the number of online clients, e.g., see Tor metrics [26].

Privacy-Preserving Caching and Storage Services. A classical use for Bloom filters, is to improve the efficiency of caching and storage mechanisms, by allowing efficient detection when cached items were updated (or not). In particular, Bloom filters were used to improve the efficiency of web-caches [15,6].

AnoNotify can similarly improve the efficiency of caching and storage mechanisms - while also protecting privacy. This is especially important for privacy-preserving storage mechanisms such as Oblivious RAM [20,30] and PIR [8], where each access involves significant overhead - hence, avoiding unnecessary requests has a large impact on performance.

Due to its high efficiency, AnoNotify can also be used to improve the privacy of web and DNS caches. In particular, web-users may use AnoNotify to improve the efficiency of anonymous-browsing mechanisms such as Tor [26] and the use of AnoNotify seems to offer significant performance improvements compared to existing proposals for protecting privacy of DNS users, see [23,29,16]. However, to fully benefit from AnoNotify it would have to be extended to a group notification setting, ensuring only one publisher may post a notification, an multiple subscribers can receive it privately. This is a challenging area for future work.

10 Conclusions

AnoNotify provides efficient and private notifications in a scalable manner, unlike previous approaches like DP5 [5] that could not scale past 1 million users. In contrast AnoNotify benefits from many users and the number of shards and size of the underlying anonymity system used may be tuned to provide meaningful, but not perfect, privacy protection. Using Bloom filters allows for more efficient downloads and notification lookups, leading to a system that can support a large number of users at a lower cost than PIR. AnoNotify lowers the quality of protection to achieve scalability, but does so in a controlled and well understood manner: the concrete security theorems indicate the advantage of the adversary, and the differentially private lemmas may also be used in the future to estimate the loss due to repeated queries.

References

1. BBC. “Yo app warns Israeli citizens of missile strikes”. Online, July 2014.
2. G. Bianchi, L. Bracciale, and P. Loreti. Better than nothing. privacy with Bloom filters: To what extent? In *Privacy in Statistical Databases - PSD 2012, Palermo, Italy, September 26-28, 2012.*, pages 348–363, 2012.
3. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

4. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *Algorithms – ESA*, UK, 2006.
5. N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. *PoPETs*, 2015(2):4–24, 2015.
6. A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.
7. D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.
8. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.
9. S. Cohen and Y. Matias. Spectral Bloom filters. In *Conference on Management of Data (SIGMOD)*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM.
10. C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX Security 12*, pages 269–283, 2012.
11. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, Sept. 2006.
12. R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *USENIX Security*, pages 303–320. USENIX, 2004.
13. C. Dwork. Differential privacy. *ICALP*, 2006.
14. U. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, USA, 2014. ACM.
15. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.
16. H. Federrath, K. Fuchs, D. Herrmann, and C. Piosecny. Privacy-preserving DNS: Analysis of broadcast, range queries and mix-based protection methods. In *Computer Security - ESORICS 2011, Leuven, Belgium, September 12-14, 2011.*, 2011.
17. S. Geravand and M. Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.
18. T. Gerbet, A. Kumar, and C. Lauradoux. A privacy analysis of google and yandex safe browsing. Technical Report Research Report RR-8686, INRIA, Sept. 2015.
19. I. Goldberg. Improving the robustness of private information retrieval. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 131–148. IEEE, 2007.
20. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
21. Google. Google transparency report - making the web safer, June 2014.
22. B. Laurie. Apres-a system for anonymous presence, 2004.
23. Y. Lu and G. Tsudik. Towards plugging privacy leaks in the domain name system. In *Peer-to-Peer Computing*, pages 1–10. IEEE, 2010.
24. D. Mazières and M. F. Kaashoek. The design, implementation and operation of an email pseudonym server. In *Conference on Computer and Communications Security*, CCS '98, pages 27–36, 1998.
25. S. D. Paola and D. Lombardo. Protecting against DNS reflection attacks with Bloom filters. In T. Holz and H. Bos, editors, *DIMVA*, 2011.
26. T. T. project. Tor Metrics. `https://metrics.torproject.org/`, April 2016.
27. C. E. Rothenberg, C. Macapuna, F. L. Verdi, and M. F. Magalhães. The deletable Bloom filter: A new member of the Bloom family. *CoRR*, abs/1005.0352, 2010.
28. L. Sassaman, B. Cohen, and N. Mathewson. The pynchon gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 1–9, 2005.

29. H. Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *Workshop on Privacy in the Electronic Society, WPES 2014*, 2015.
30. E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Computer & communications security (ACM CCS)*, pages 299–310. ACM, 2013.
31. Wikipedia. Yo (app). `https://en.wikipedia.org/wiki/Yo_(app)`, April 2016.

A Proof of Main Theorem

Proof. To prove the main security theorem, and ultimately show that AnoNotify is Δ-private, we need to show that the adversary can only win the game in Definition 1, with an advantage Δ over a random guess. We do so by first arguing that the adversary learns nothing from rounds preceding the challenge round, and then computing the advantage given the information in this final round.

We proceed through "game hopping" with slight modifications over the initial security Definition 1, including the IndNotExp experiment (Game0). We first note that in the concrete protocol $\mathcal{N}$.Notify and $\mathcal{N}$.Query act on notification IDs generated using a Pseudo-random function (PRF) keyed with an unknown key to the adversary and the epoch number ($ID^t = PRF_{ck}(t)$). Thus, we can replace all instances of the first invocation of the PRF by true random functions (Game1). The adversary can only distinguish between the original experiment Game0 and Game1 with negligible advantage due to the properties of secure PRFs. In Game1 the information within each epoch is statistically independent, and thus an adversary cannot learn anything about the final round from previous ones. Thus, we define Game2 that consists only of the final challenge round, and the advantage of the adversary winning this game is equal to winning Game1.

In the final round the security definition restricts the adversary to not issue notifications for a number of queries seen. Thus we define Game3 in which we provide the random ID^t for all notifications that have been requested in the final round. We argue that this additional information increases the adversary advantage in winning Game3 over Game2.

Finally, Game3 consists of the final challenge epoch and the only uncertainty of the adversary are the exact ID^t of non observed but queried notifications, and also whether user $n-1$ or user n has acted (depending on the challenge bit b). All ID^t of non-target users are random, and the ID^t of the two potential target users are known. Thus the adversary now has to decide on the basis of the volume X_0 and X_1 observed in the shard s_0, s_1 corresponding to $n-1$ and n respectively, what the challenge b was.

We compute the adversary advantage in Game3 directly. We denote S_0, S_1 the events that the target user queried shards s_0, s_1 corresponding to users $n-1, n$ with notification identifiers ID^t_{n-1}, ID^t_n. Lemma 1 then shows that given two known shards and $u-1$ random shards we can find ϵ, δ such that for user $n-1$ and n and all query volumes observed by the adversary: $\Pr[X_0, X_1|n-1] \leq e^{\epsilon} \Pr[X_0, X_1|n]$ with probability at least $1-\delta$. Lemma 2 then concludes the proof by showing this differentially private property can be translated to a concrete adversary advantage Δ, and the proof concludes.

B Proof of Lemma 1

Proof. We define as S_0, S_1 the events that either shard s_0 or s_1 was queried. For a mapping function F, which maps the users identifiers to the storing shards, such that $F(n-1) = s_0$ we have

$$\Pr[X_0 = x_0, X_1 = x_1 | n-1] = \Pr[X_0 = x_0, X_1 = x_1 | n-1, S_0]. \tag{1}$$

Using the properties of the conditional probability[7] we obtain the following equality (proof in Lemma 3 in Appendix B), and using (1) we have:

$$\Pr[X_0 = x_0, X_1 = x_1 | n-1, S_0] = \Pr[X_0 = x_0, X_1 = x_1 | S_0] \Rightarrow \tag{2}$$
$$\Pr[X_0 = x_0, X_1 = x_1 | n-1] = \Pr[X_0 = x_0, X_1 = x_1 | S_0]. \tag{3}$$

Now, we can prove that the events when either subscriber $n-1$ or n is requesting in the challenge is (ϵ, δ) - *differentially private*, so the adversary who wants to infer which subscriber is querying is not able to distinguish this two events with significant probability.
The probability density function of the binomial distribution $\text{Bin}\left(n, \frac{1}{p}\right)$ is $\Pr[X = k] = \binom{n}{k}\left(\frac{1}{p}\right)^k\left(1-\frac{1}{p}\right)^{n-k}$. The probabilities that either subscriber $n-1$ or n request for identifier ID_{n_1} and ID_n are denoted as

$$\Pr[X_0 = x_0, X_1 = x_1 | S_0] = \binom{u-1}{k}\left(\frac{2}{S}\right)^k\left(\frac{S-2}{S}\right)^{u-k-1}\binom{k}{x_0-1}\left(\frac{1}{2}\right)^{x_0-1}\left(\frac{1}{2}\right)^{k-x_0+1}$$
$$\Pr[X_0 = x_0, X_1 = x_1 | S_1] = \binom{u-1}{k}\left(\frac{2}{S}\right)^k\left(\frac{S-2}{S}\right)^{u-k-1}\binom{k}{x_1-1}\left(\frac{1}{2}\right)^{x_1-1}\left(\frac{1}{2}\right)^{k-x_1+1}.$$

Thus, we have

$$\Pr[X_0 = x_0, X_1 = x_1 | S_0] = \frac{x_0}{x_1}\Pr[X_0 = x_0, X_1 = x_1 | S_1].$$

So we would like to ensure, that $\frac{x_0}{x_1} \leq e^{\epsilon}$, which implies $x_1 \geq e^{-\epsilon}x_0$.
Let $C = \frac{2(u-1)}{S} - \gamma$, where $\gamma = \tau \cdot \frac{2(u-1)}{S}$ and $\tau \in (0,1)$. We define the value of δ (related to the events when it is easy to distinguish the two observations in our challenge) as below

$$\delta = \underbrace{\Pr[X_0 + X_1 \leq C]}_{\text{Bin}\left(u-1, \frac{2}{S}\right]} + \underbrace{\Pr[X_1 \leq e^{-\epsilon}X_0 | X_0 + X_1 \geq C]}_{\text{Bin}\left(C, \frac{1}{2}\right)}. \tag{4}$$

From the Chernoff bound for a random variable X, which is a sum of independent variables with Bernoulli distribution, we have

$$\Pr[X \leq (1-d)E[X]] \leq e^{\frac{-E[X]d^2}{2}}.$$

[7] $\Pr[A|C,B] \cdot \Pr[C|B] = \Pr[A,C|B] \implies \sum_C (\Pr[A|C,B] \cdot \Pr[C|B]) = \Pr[A|B]$, for the events A, B, C, such that $\Pr[C,B] > 0, \Pr[B] > 0$.

First, we estimate the part I of equation (4)

$$\Pr\left[X_0 + X_1 \leq C\right] \overset{X=X_0+X_1}{=} \Pr\left[X \leq C\right] = \Pr\left[X \leq \frac{2(u-1)}{S} - \gamma\right]$$
$$= \Pr\left[X \leq (1-\tau)\frac{2(u-1)}{S}\right] \leq \exp\left(-\frac{2(u-1)\tau^2}{2S}\right) = \exp\left(-\frac{(u-1)\tau^2}{S}\right).$$

Now, we estimate part II

$$\Pr[X_1 \leq e^{\epsilon} X_0 | X_0 + X_1 \geq C] \leq \sum_{i=C}^{u} \Pr[X_1 \leq e^{\epsilon} X_0 | X_0 + X_1 = i] \Pr[X_0 + X_1 = i]$$
$$\leq \sum_{i=C}^{u} \Pr[X_1 \leq e^{-\epsilon} X_0 | X_0 + X_1 = C] \Pr[X_0 + X_1 = i] \leq \Pr[X_1 \leq e^{-\epsilon} X_0 | X_0 + X_1 = C].$$

Thus, $X_1 \leq e^{-\epsilon}(C - X_1)$, which implies $X_1 \leq \frac{e^{-\epsilon}C}{1+e^{-\epsilon}}$.

Applying this to the upper equation we have

$$\Pr\left[X_1 \leq \frac{e^{-\epsilon}C}{1+e^{-\epsilon}}\right] = \Pr\left[X_1 \leq \frac{C}{2} \cdot \frac{2e^{-\epsilon}}{1+e^{-\epsilon}}\right] = \Pr\left[X_1 \leq \frac{C}{2} \cdot \left(1 - \frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)\right]$$
$$\leq \exp\left(-\frac{C}{2} \cdot \frac{1}{2}\left(\frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)^2\right) = \exp\left(-\frac{(u-1)}{2S}\left(\frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)^2\right).$$

Taking these two equations together, we obtain

$$\delta \leq \exp\left(-\frac{(u-1)\tau^2}{S}\right) + \exp\left(-\frac{(u-1)}{2S}\left(\frac{1-e^{-\epsilon}}{1+e^{-\epsilon}}\right)^2\right).$$

The above bound gives us the estimation of the value of δ, which bounds the probability of very rare events which can destroy our differential privacy guarantee. Note, that we have a dependency between δ and ϵ in this equation, so we can select both values to work the best for us. In the next sections, we present the experimental evaluations of the security parameters ϵ, δ.

Lemma 3. *For random variables X_0, X_1 and events $n-1, n, S_0, S_1$ defined as in Section 6 we have the following dependency*

$$\Pr[X_0, X_1 | n-1, S_0] = \Pr[X_0, X_1 | S_0].$$

Proof. From conditional probability properties and the fact, that $\Pr[S_0] = \frac{1}{2}$, $\Pr[n-1, S_0] > 0$ we can write that

$$\sum_i \left(\Pr[X_0, X_1 | i, S_0] \cdot \Pr[i | S_0]\right) = \Pr[X_0, X_1 | S_0].$$

The sum of the probabilities over requesting subscribers can be considered as a sum of the probabilities for the users i which map to shard s_0 and those who do

not. As N_0 we denote a set of users whose identifiers map to shard s_0. Following this, we can present the previous equation as

$$\sum_{i \in N_0} \Pr[X_0, X_1 | i, S_0] \cdot \Pr[i|S_0] + \sum_{i \notin N_0} \Pr[X_0, X_1 | i, S_0] \cdot \Pr[i|S_0] = \Pr[X_0, X_1 | S_0].$$

Because $\Pr[i|S_0] = 0$ for each $i \notin N_0$ we have

$$\Pr[X_0, X_1 | n-1, S_0] \cdot \sum_{i \in N_0} \frac{1}{|N_0|} = \Pr[X_0, X_1 | S_0] \Rightarrow$$
$$\Pr[X_0, X_1 | n-1, S_0] = \Pr[X_0, X_1 | S_0].$$

C Proof of Lemma 2

In this section, we present the proof of Lemma 2. From Lemma 1 we know, that a differentially private bound holds for the that probability an adversary observes volumes of the shards resulting from events S_0, S_1 (with some probability $1-\delta$). On the basis that since $\Pr[S_0] = \Pr[S_1] = \frac{1}{2}$

$$\Pr[X_0, X_1 | S_0] \leq e^{\epsilon} \Pr[X_0, X_1 | S_1]$$
$$\Pr[X_0, X_1 | S_0] \Pr[S_0] \leq e^{\epsilon} \Pr[X_0, X_1 | S_1] \Pr[S_1]$$
$$\Pr[S_0 | X_0, X_1] \leq e^{\epsilon} \Pr[S_1 | X_0, X_1].$$

Since the probabilities of all possible outcomes of an event must sum up to 1, $\Pr[S_0|X_0, X_1] = (1/2 + \Delta_\epsilon) = 1 - \Pr[S_1|X_0, X_1]$. Following this, we obtain

$$\frac{1}{2} + \Delta_\epsilon \leq e^{\epsilon} \left(\frac{1}{2} - \Delta_\epsilon \right) \Rightarrow \Delta_\epsilon \leq \frac{e^{\epsilon} - 1}{2\left(e^{\epsilon} + 1\right)} = \frac{1}{2} \tanh\left(\frac{\epsilon}{2}\right).$$

The total value of Δ, which is the probability that the adversary guesses successfully is bounded as

$$\Delta \leq (1-\delta) \cdot \frac{e^{\epsilon} - 1}{2\left(e^{\epsilon} + 1\right)} + \delta.$$

since the differential privacy holds with probability $1-\delta$ and otherwise we can even assume that she automatically can guess correctly.

D Known Theorems

Theorem 1. *False positive probability for a Bloom filter of size n constructed using k hash functions after inserting m elements is defined as $fp(n,k,m) = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^m \approx \left(1 - e^{-\frac{mk}{n}}\right)^m$.*

Theorem 2. *The number of hash functions, which minimizes the probability of false positive error is $k = \frac{n}{m} \log 2$.*