AnonyLikes: Anonymous Quantitative Feedback on Social Networks

Pedro Alves and Paulo Ferreira
pedro.h.alves@gmail.com, paulo.ferreira@inesc-id.pt

INESC-ID / Instituto Superior Técnico / Technical University of Lisbon

Abstract. Social network applications (SNAs) can have a tremendous impact in raising awareness to important controversial topics such as religion or politics. Sharing and liking are powerful tools to make some of those topics emerge to a global scale, as already witnessed in the recent Tunisian and Egyptian revolutions. However, in several countries the simple act of liking an anti-government article or video can be (and has already been) used to pursue and detain activists. Therefore, it is of utmost relevance to allow anyone to anonymously ”like” any social network content (e.g. at Facebok) even in presence of malicious administrators managing the social network infrastructure.
We present anonyLikes, a protocol which allows SNAs users to ”like” a certain post (e.g., news, photo, video) without revealing their identity (even to the SNA itself) but still make their ”like” count to the total number of ”likes”. This is achieved using cryptographic techniques such as homomorphic encryption and shared threshold key pairs. In addition, the protocol ensures all other desirable properties such as preventing users from ”liking” a particular post more than once, while preserving anonymity.
The anonyLikes protocol is fully implemented using Facebook as an example and can be easily used by developers (e.g. Facebook itself or other social network applications and infrastructures) to provide an alternative ”like” button called ”anonyLike”.

1 Introduction

Social network applications (SNAs) have achieved massive popularity in recent years, with Facebook leading the way with its 900 million users[1], but also Twitter and Linkedin both with 200 million each.[2]

These applications allow people all over the world to connect each other at an unprecedented scale. Although these connections are primarily being established to share media and keep in touch with friends, family and colleagues, they are also being used to raise awareness and coordinate large communities around important topics, such as the political status of some countries. For example, Egyptian activist Wael Ghonim credited Facebook with the success of the Egyptian people’s uprising, in particular for its key

[1] http://www.statisticbrain.com/facebook-statistics/
[2] http://blog.linkedin.com/2013/01/09/linkedin-200-million/, http://mashable.com/2012/12/18/twitter-200-million-active-users/

role in organizing the most important protest on January 25th.[3] During the revolution, hundreds of thousands of Egyptians used Facebook to post, like and share news and videos, raising global awareness about what was going on in the country.

However, that didn't prevent Wael Ghonim from being arrested for 12 days, shortly after the protest.[4] Like Wael, many activists suffered from their activities on social networks - Egypt and several other countries have been reported to track down activists on social networks[5] to the point where bloggers died while in custody for their anti-government articles.[6]

We present AnonyLikes, a protocol which allows SNAs users to promote/raise awareness to certain content (news, links, photos or videos which we will refer generically as posts through the rest of the paper) without revealing their identity (even to social networks administrators). Even though social networks typically go to great length to protect the privacy of their users, they still have to abide by the legislation of each country and may be forced to reveal internal data to governmental agencies under a court order.[7] Even if that doesn't happen, there are still (less legal) ways to get access to data, either by employing hackers[8] or by tapping into internal disgruntled employees who have access to the social network database. Even under these potential attacks, AnonyLikes always guarantees the anonymity of the users who have "liked" some content. Our protocol uses cryptographic techniques (see below) to guarantee the privacy of the "like" activity (i.e., the information that an user "liked" a given post) even with access to the SNA's database.

Our interaction model is similar to the one already existent and to which users are accustomed - a "like button", which can be embedded into a blog or news site, associated with a given post along with the number of "likes" already submitted by other users. Note that we will use the term "like" for the rest of the paper as a generic verb to denote an action that increases awareness of a certain post (it could also be "share", "retweet", "reblog", etc.).

Despite similarities with the existing interaction models (in particular, Facebook) AnonyLikes is, in fact, a completely different protocol whereby messages exchanged with the social network server are encrypted in such a way that:

- it is not possible to ascertain which post each user "liked";
- it is possible to know how many people "liked" a certain post.

This is achieved using a combination of homomorphic encryption [23] and shared threshold key pairs: [11]:

- Users can "like" anything from an almost unlimited set of posts, i.e there is no limit to the set of posts a user can "like";

[3] http://www.huffingtonpost.com/2011/02/11/egypt-facebook-revolution-wael-ghonim_n_822078.html.

[4] http://www.huffingtonpost.com/2011/02/07/wael-ghonim-google-exec-egypt-protests_n_819438.html

[5] http://readwrite.com/2011/06/15/the_arab_spring_a_status_report

[6] http://readwrite.com/2011/04/12/bahraini_blogger_dies_in_custody

[7] http://www.theregister.co.uk/2012/06/11/woman_wins_landmark_trolling_case_against_facebook/

[8] http://www.guardian.co.uk/media/2013/jan/31/new-york-times-hacking-china-cybercrime

– The period during which users can ”like” is unlimited, i.e.there is no closing date after which a post can no longer be ”liked”.

In summary, the requirements that guide the design of the anonyLikes protocol are the following:

– **R1**: Users should be able to ”like” a given post;
– **R2**: Only authenticated users can ”like”;
– **R3**: It must not be possible for anyone (including the social network infrastructure, e.g., facebook.com) to know a certain user ”liked” a particular post;
– **R4**: ”Liking” a post must have an effect (even though not immediate) on the number of ”likes” (count) of that post and that effect must be visible to everyone;
– **R5**: It must not be possible for someone to ”like” a particular post more than once;
– **R6**: The user interaction must be similar to the one already existent in major social network applications such as Facebook and Twitter (in particular, concerning usability and performance).

Note that we cannot guarantee that the social network infrastructure servers (simply referred as SNA-server from now on) don't add fake ”likes” to a given post (although this already happens today), i.e., it doesn't artificially increment the ”likes” count. However, the SNA-server cannot ignore *true* ”likes” without raising suspicion since R4 mandates that the ”like” action must have an effect on ”likes” count.

In summary, the contributions of this work are:

– A protocol (AnonyLikes) that allows social network users to provide quantitative feedback (e.g., ”like” on Facebook) about a certain post without revealing their identity, even to the SNA-server itself. Using this protocol, social networks are still able to calculate the sum of feedback (e.g., the number of people that ”liked” a given post on Facebook) while preventing multiple ”likes” from the same user.
– A reference implementation of the AnonyLikes protocol for Facebook ”likes” that can easily be used by Facebook itself or that can serve as the basis for other social network implementations.

The remainder of this paper is organized as follows. The next section presents the AnonyLikes protocol. After that, we describe the implementation of the prototype. Section 4 presents an evaluation of the protocol regarding three aspects: Usability, Probabilistic Duplicates Detection, and Performance. Finally, Sections 5 and 6, present relevant related work and draw some conclusions, respectively.

2 Protocol

The anonyLikes protocol can be applied to any social network that has the concept of quantitative feedback, i.e., that features an action that increments a number associated with a post. For example, in Facebook and Tumblr, that action is ”like”, in Twitter is ”retweet”, etc. The general idea (see Figure 1) is to encrypt the ”likes” with a key that is shared among the social network and a set of independent entities (e.g., NGOs) so that none of these entities alone can decrypt it. By using a special property of the encryption

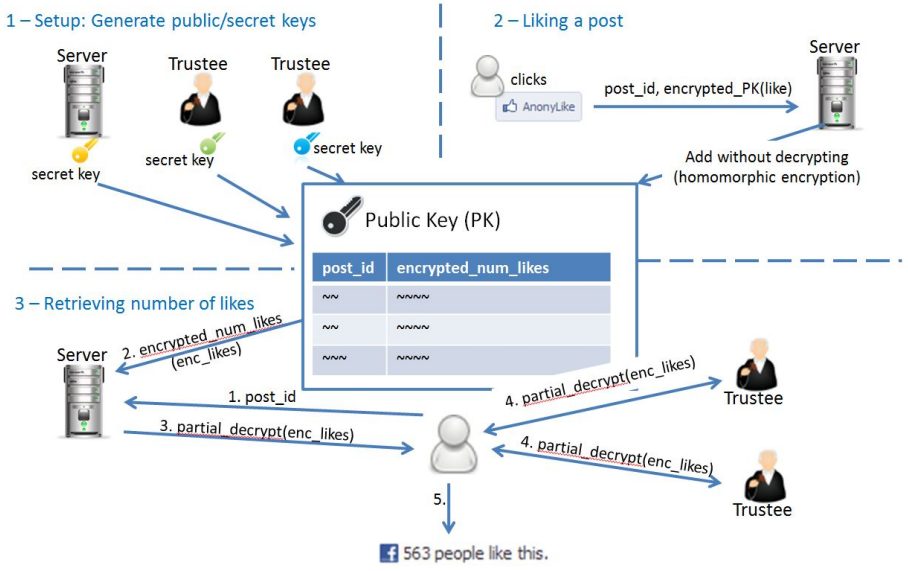

Fig. 1. The 3 steps of the AnonyLikes protocol.

algorithm, the SNA-server is able to sum the number of "likes" of a given post without having to decrypt them (i.e., the sum is also encrypted). When the user wants to know how many "likes" are associated with a given post, it coordinates all the entities to jointly decrypt the sum. We now present the details of this protocol.

The anonyLikes assumes a SNA-Server S (e.g., hosted at facebook.com) and a set of Trustees T_i (e.g., public national institutes, NGOs, etc.). Clients exchange encrypted messages with S and communicate with T_i to decrypt such messages (in particular, the number of "likes" of a given post).

There is an initial setup process where a shared threshold key pair is generated across S and all T_i. The generated public key will be published and used to encrypt all the "like" messages. Each entity (S and T_i) stores its part of the private key (also called shadow). To decrypt such messages (or, better said, the result of operations on those messages), a certain number of T_i (depending on the threshold) must collaborate, i.e., S alone is not be able to decrypt them.

After the setup stage, the social network is ready to accept "like" messages. A "like" message is a tuple $(post\_id, like)$, where $post\_id$ is a unique identifier of the post, and $like$ is an integer telling whether the user "liked" that post (value=1) or not (value=0). On each one of these messages, the $like$ value is encrypted with the previously generated public key and the tuple is then sent to S. Note that, since the $post\_id$ is sent in cleartext, the client cannot send only the posts the user "likes"; it has to also send posts the user didn't "like". So, every time a user "likes" a given post, the client sends not only that $post\_id$ but also n random other $post_id$s, so that S is not able to know which exact post the user "liked". S knows that the user "liked" one of those posts but is not

sure which one. The $post\_id$ has to be sent in cleartext to prevent duplicate "likes", as explained in the next paragraph.

Sending multiple $post_id$s each time the user "likes" a post is crucial to satisfy requirement R5 (it must not be possible for someone to "like" a particular post more than once - see Section 1). Since the "likes" are encrypted, S has no way to know if the user already "liked" a given post (therefore satisfying requirement R3). However, it knows that the user *potentially* "liked" a given post - every $post\_id$ sent to the SNA-server ("liked" or not "liked") is a potential "like" with probability $1/n$ (n is the number of $post_id$s sent for each "like"). Based on this, S applies a probabilistic detection of multiple "likes" for the same post - it refuses $post_id$s that has already received because there is a high probability that it is a "like" for a post the user previously "liked". Obviously, there may be false positives (i.e., S may refuse legitimate "likes") but the probability is low enough to guarantee the usability and usefulness of the system. In Section 4, we show that (being conservative) this probability is less than one collision per year for Facebook users (i.e., of all the "likes" the user does in Facebook during an year, one of them won't be successful).

Finally, the protocol allows users to see the number of "likes" of a given post, without knowing who did each individual "like". This is possible thanks to the additive homomorphic properties of an ElGamal variant known as exponential ElGamal [9]. In exponential ElGamal, it is possible to add two encrypted messages, without having to decrypt them first, and the decryption of the encrypted sum equals the sum of the decrypted values. S applies this property by adding each encrypted "like" (remember that the "like" value is either 1 or 0) with the already existent encrypted sum of that $post\_id$ (or zero if it is the first). When a user retrieves the number of "likes" of a given post, it is actually retrieving the encrypted sum associated with that post. The client is responsible for coordinating with the necessary number of Trustees (based on a pre-defined threshold) in order to decrypt that sum using their part of the private key.

2.1 Phases of the Protocol

We now detail the three phases of the protocol (setup, "liking" a post, and retrieving the number of "likes"), starting with a brief summary of the cryptographic building blocks (ElGamal cryptosystem, homomorphic encryption, threshold ElGamal, and zero knowledge proof) which are used by the protocol.

Cryptographic Building Blocks

ElGamal The anonyLikes protocol relies on the ElGamal cryptosystem [14]. ElGamal works in the $\mathbb{Z}_p^*$ subgroup G_q of order q, where p and q are large primes such that $p = 2q + 1$. A secret key $x \in \mathbb{Z}_q$ is selected and the corresponding public key $y = g^x \bmod p$ is computed. A message $m \in G_q$ is encrypted by selecting a random integer value $r \in \mathbb{Z}_q$, and constructing the following pair $(\alpha, \beta) = (g^r \bmod p, my^r \bmod p)$. Decryption is computed as $m = \alpha^{-x}\beta$.

Fig. 2. Example of a post with the associated anonyLike button and the number of (anonymized) people that "liked" that post.

Homomorphic Encryption We say that ϵ is a $(\oplus, \otimes)$-homomorphic encryption scheme if for any instance E of the encryption scheme, given $c_1 = E_{r1}(m_1)$ and $c_2 = E_{r2}(m_2)$, there exists an r such that $c1 \otimes c2 = E_r(m1 \oplus m2)$. This property is crucial in the anonyLikes protocol to calculate the number of "likes" (sum) of a given post without having to decrypt individual "like" messages. The ElGamal cryptosystem satisfies this property for multiplication operations, so we need to use a variant known as exponential ElGamal [9] that satisfies this property for additive operations, i.e. $c1 * c2 = E_r(m1 + m2)$.

Threshold ElGamal The goal of a threshold scheme for public-key encryption is to share a private key among a set of receivers such that messages can only be decrypted when a substantial set of receivers cooperate [11]. In the anonyLikes protocol, the receivers are called *Trustees*. The main steps of a threshold system are: (i) a *key generation* step to generate the private key jointly by the receivers, and (ii) a *decryption* step to jointly decrypt a ciphertext without explicitly reconstructing the private key. More details about both steps applied to the ElGamal cryptosystem can be found in Cramer[9].

Zero Knowledge Proof of Validity In the anonyLikes protocol, each "like" message contains a set of tuples $(post\_id, E(like))$, where $like$ can be either 0 or 1. S needs to make sure that $like$ has indeed one of those two values without revealing the exact value. This is accomplished by attaching to each tuple a proof of validity (see Cramer[8] for more details).

Setup The setup phase occurs only once before the system is made available to the public in general. Its primary goal is to generate a keypair responsible for the encryptions/decryptions that occur in the other phases of the protocol.

S and all T_i create a shared threshold ElGamal key pair (ϵ_{pk}, ϵ_{sk1}, ϵ_{sk2}, ..., ϵ_{skn}).
ϵ_{pk} is published on S's site and S and each T_i hold its part of the secret key.

"Liking" a Post This phase occurs when the user clicks the "AnonyLike" button that is associated with a given post (see Figure 2). All "likes" have to be authenticated, so if the user has not previously authenticated himself on S he will be redirected to do so,

prior to submitting the "like" message. We now detail all the steps since the user clicks the "AnonyLike" button until the SNA-server responds with a successful message.

1. User U clicks the AnonyLike button associated with a given post identified by P_{like}, using the client software (e.g., browser).
2. The client software randomly chooses n other posts P_i that: (1) have occurred recently (w.r.t. the age of the post being "liked"); (2) are from the same topic (based on hashtags, labels and words within the post) and (3) are public (i.e., not confined to posts from friends). These three restrictions increase the difficulty for an attacker to guess P_{like} from the several P_i. If it simply chose posts regardless of their age or topic, the attacker could guess that the most recent or relevant P_i would be the one with the "like" (and this would be correct in the vast majority of cases). Also, choosing only posts from friends would hugely reduce the set of posts from which to draw the random ones so we use only public posts (which is usually the case for the political/ideological posts that motivated this article).
3. The client software creates a message M containing the tuple (P_{like}, $E_{\epsilon pk}(1)$) and a set of n tuples (P_i, $E_{\epsilon pk}(0)$), $1 <= i <= n$. This means the software encrypts "1" for the post the user "likes" and "0" for the others. The position of the P_{like} tuple in the message is random (its index is obtained from a random number generator).
4. The client software sends this message to S (remember that U has already been authenticated in S) plus a set of proofs $Proof_i$ (one for each tuple) that will be used to verify that they contain valid "like" values.
5. S uses $Proof_i$ to verify that the "like" value for each post is either 0 or 1 (without having to decrypt it).
6. S updates the vector of *potencial* "likes" of this user V_u = (P_0, ..., P_n). This vector contains all P_i that ever came in a message associated with this user. If any P_i contained in the message already exists in V_u, the SNA-server S returns an error. This is to probabilistically prevent duplicated "likes" from the same user for the same post.
7. S updates its *LikesStats* table (P_i, $E_{\epsilon pk}$(num_likes)) for every P_i in the message using the additive property of homomorphic encryption, without having to decrypt the tuples in the message. This table stores the total number of "likes" of every post in the system, encrypted by ϵ_{pk}. M can now be discarded.
8. S responds to the client software with a success message.

Note that, steps 5 and 6 prevent the submission of multiple "likes" for the same post (requirement R5). In addition, step 7 guarantees that the "like" has an effect on the number of "likes" of the associated post while still preventing S from knowing which particular post the user "liked".

Retrieving the Number of "Likes" This phase is responsible for retrieving the number of users that already "liked" a given post, as shown in Figure 2. Usually, this is calculated by the SNA-server but, in anonyLikes, the SNA-server has no way of knowing this number since it is encrypted with a shared key that is distributed among a set of trustees (generated on the previously mentioned setup phase). Therefore, the client software assumes the role of coordinating with the Trustees to decrypt that number.

1. The client software asks S for the number of "likes" of a given post P_i;
2. S searches its *LikesStats* table for P_i and gets the corresponding $E_{\epsilon pk}$(num_likes);
3. S partially decrypts the num_likes - $D_{\epsilon sk1}(E_{\epsilon pk}$(num_likes)) and returns this to the client;
4. The client software sends $D_{\epsilon sk1}(E_{\epsilon pk}$(num_likes)) to several T_i (previously chosen by the user or random) until it is fully decrypted. Supposing we had 2 Trustees (T_1 and T_2), the final decrypted num_likes would be the result of $D_{\epsilon sk_{T_2}}(D_{\epsilon sk_{T_1}}(D_{\epsilon sk_{SNA-Server}}(E_{\epsilon pk}$(num_likes))))
5. Finally, the client software shows the user the number of "likes".

3 Implementation

To evaluate the anonyLikes protocol, we developed three components: the SNA-server component, the trustee component and the client component. We used these components to implement a web application that provides an interface showing several random posts from Facebook (10 in the current implementation) along with an anonylikes button and the anonymized number of people that "liked" each one of such posts.

Due to the non-anonymous nature of Facebook, it is not possible to implement this protocol on top of their API.[9] For example, to use the API for submitting a "like", the application has to provide the user who is "liking", therefore preventing any kind of anonymization. Nothe that, using the same (fake) user for all "likes" doesn't work because Facebook prevents more than one "like" from the same user. Our SNA-server component is therefore a simplified replica of Facebook with some adaptations to allow the implementation of the anonyLikes protocol. This component can be the basis for adaptations implemented by Facebook itself, should it decide to use the anonyLikes protocol in the future.

Table 1 shows the responsibilities of each component. Note that the encryption/decryption operations are all performed by the client component. We now look into detail on each of such components.

3.1 SNA-Server Component

The SNA-server component is responsible for: (i) coordinating the distributed generation of the threshold shared key, (ii) receiving encrypted "like" messages, and (iii) providing the encrypted sum of "likes" that will be used to show how many people "liked" a given post. It is implemented as a Python/Django application and its source code is fully available at https://bitbucket.org/anonymousJoe/anonylikes. AnonyLikes reuses some code that was adapted from the Helios system [1] as it provides the implementation of some of the cryptography functions needed (and the code is open-source).[10].

The SNA-server connects to a MySQL database with 3 tables: (i) PublicKey - contains the public key used to encrypt "likes"; (ii) LikesStats - contains tuples ($post\_id$, $encrypted\_num\_likes$), and (iii) PotentialLikesUser - contains all the $post_id$s that each user has potentially "liked".

[9] http://developers.facebook.com/docs/reference/api/publishing/

[10] Available at https://github.com/benadida/helios-server

Comp.	Responsibilities	Impl.
SNA-Server	Generate public key	Django/
	Coordinate generation of shared secret key	Python
	Store secret key (SNA-server part)	
	Receive encrypted ”Likes”	
	Provide encrypted sum of ”Likes”	
Trustee	Store secret key (trustee part)	Bottle/
	Provide decrypting factor	Python
Client	Retrieve random posts	Browser/
	Encrypt ”likes”	Javascript
	Send encrypted ”Likes”	
	Receive encrypted sum of ”Likes”	
	Get decryption factors	
	Decrypt sum of ”Likes”	

Table 1. Responsibilities of each anonyLikes component

Several actions in the SNA-server must be previously authenticated. We delegate the authentication on Facebook using their OAuth implementation.[11] If the user is not already authenticated in our SNA-server, he is redirected to Facebook where he is asked if he wants to login into AnonyLikes and provide his basic information and email (see Figure 3).

Generation of the Shared Key The generation of the shared key among all the Trustees is partially executed in the browser and partially executed in the SNA-server. The SNA-server starts by generating the public key and its part of the secret key. Then, each Trustee opens the *Trustee setup* page using a browser to locally generate its part of the secret key. This generation is performed in the browser using a Javascript implementation of the ElGamal cryptosystem. Each Trustee stores locally the generated secret key which is then cleared from the browser's memory. Throughout the whole process, the secret key never leaves the Trustee's computer.

Receiving Encrypted ”Likes” The SNA-server provides a REST endpoint to receive encrypted ”like” messages. Although programmers can use this endpoint directly, we also provide an embeddable html/javascript that can easily be included on any site to add an ”AnonyLike” button (see Figure 2). In this case, all the encryption and communication is automatically handled by the embedded javascript code.

This endpoint verifies that all the encrypted ”likes” contained in the message are valid ”likes” (i.e., contain the value 0 or 1) using a Zero Knowledge Proof Verification algorithm (already mentioned in the Section 2). Then, for each one, it adds it (using

[11] http://developers.facebook.com/docs/concepts/login/

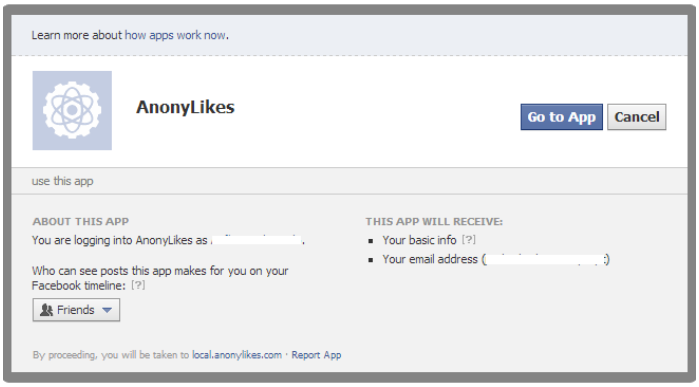

Fig. 3. Users login into AnonyLikes through Facebook

ELGamal homomorphic additive properties) to the existing number of ”likes” associated with that post using the LikesStats table. Finally, it updates the PotentialLikesUser table with all the $post_id$s contained in the message.

Providing the Encrypted Sum of ”Likes” The SNA-server provides a REST endpoint that responds with the number of ”likes” of a given post. This is a simple query to the LikesStats table. Since the SNA-server has one part of the shared secret key, it just partially decrypts the number of ”likes” before returning it to the client.

3.2 Trustee Component

The trustee component is responsible for storing its part of the shared secret key and, based on that, providing the decryption factor through a REST endpoint. It is implemented as a Bottle server (very lightweight application server that runs Python) that is easily installed on any hosting provider. This component should be installed by each Trustee on a server of his choice. The url of such servers must be made public, so that clients can choose among all the Trustees the ones that will be contacted to get the decryption factor.

3.3 Client Component

The client component is where the encryption/decryption takes place using a Javascript implementation of the ElGamal cryptosystem. Since the Javascript technology is too slow for certain heavy operations such as generating randomness and modular exponentiation, Java is utilized for such computation, using an applet running in the browser.

We now detail two important aspects of the Client component implementation: how to embed the ”anonyLike” button, and how to retrieve random posts to go along the *real* post (i.e the one ”liked”).

```
<script data-main="/anonyLikes.js" src="/static/require.js"></script>
```

Fig. 4. How to include the anonylikes javascript library and all its dependencies

```
<div class="anonylike-button" data-post-id="34435556_546456">
    <!-- anonyLikes.js will fill this space with the anonyLikes
         button and the number of people who liked it -->
</div>
```

Fig. 5. How to embed the anonylikes button

Embedding the "anonyLike" Button The client component consists of a javascript file (anonyLikes.js) and several cryptographic libraries (e.g., elgamal.js). For convenience, programmers only have to import anonyLikes.js as depicted in Figure 4 and all the required cryptographic libraries are automatically loaded. Then, programmers have to insert HTML code similar to Figure 5 in the place where they want the "anonyLike" button to show up. anonyLikes.js inserts html code inside that $< div >$ to show the button and the number of people that already "liked" that post. This is very similar to how the original Facebook "like" button is embedded, so it is easy to include anonyLikes on any site.

Retrieving Random Posts Another important aspect of the Client component implementation is obtaining n random recent posts to go along the "liked" post (recall that the message sent to the SNA-server contains the $post\_id$ the user "liked" plus n random $post_id$s the user didn't "like"). This operation is performed by the browser using AJAX calls to the Facebook's API. Since Facebook's API only provides a service to search for posts, we get random posts by issuing random queries and getting the first result. This operation (obtaining n random posts) is executed in the background as soon as the page loads to minimize the time spent after clicking the "anonyLike" button (see next section). The user usually clicks the "anonyLike" button after reading/watching the post, which takes sufficient time for the background job to finish retrieving n random posts.

4 Evaluation

In this section, we present the response to the following questions that are crucial to the success of anonyLikes:

- Usability - Can we satisfy requirement R6 (The user interaction must be similar to the one already existent in major social network applications such as facebook and twitter)?
- Duplicates Detection - Does our probabilistic duplicates detection mechanism (necessary for requirement R5) affect the user experience?
- Performance - Is the performance of our prototype adequate to user expectations (requirement R6)?

Total users	1.060 billion
Average daily ”likes”	2.7 billion
Average daily ”likes” per user	2.5 ”likes”
Number of ”likes” per year	912 ”likes”
Total number of location-tagged posts	17 billion

Table 2. Some statistics on Facebook usage

4.1 Usability

The AnonyLikes behavior is very similar to the original Facebook ”like” behavior from the user's perspective. As can be seen in Figure 2, the look of both the button and number of users who already ”liked” is very similar to the original one. From the programmer's perspective it is also very similar as programmers only need to import one javascript library (see Figure 4) and define the placeholder where the button will show up (see Figure 5).

So, from a usability point of view, the anonyLikes functionality is as easy as the original Facebook ”like” functionality (which is already used by millions).

4.2 Probabilistic Duplicates Detection

The anonyLikes protocol relies on the assumption that it is highly unlikely that a user will ”like” one of the random posts that is sent along the ”liked” one. This assumption is the basis for duplicate ”like” detection: if the user ”likes” a post that he has already *potentially* ”liked”, the SNA-server will not accept that action because it will assume that the user is ”liking” the same post twice.

The actual probability of such occurrence is calculated taking into account three variables: M being the total number of posts on Facebook (the universe from which to grab the random posts); n being the number of $post_id$s that are sent within each message ($n-1$ random posts plus the ”liked” one), and L being the number of ”like” messages the user sends per year. Thus, the probability of getting a collision per year is:

$$Pr_{collision} = \frac{Ln}{M} \tag{1}$$

In Table 2 we show some stats on Facebook.[12] Unfortunately, the total number of posts is not available (only the total number of location-tagged posts) so we will assume 17 billion as the total number of posts which is clearly a conservative assumption, given that many posts are still not location-tagged. We are also making the very conservative assumption the total number of posts is constant per year.

With these numbers, we can now calculate the $Pr_{collision}$ (per year), using $n = 20$:

$$Pr_{collision} = \frac{912 * 20}{17.000.000} = 0,1\% \tag{2}$$

[12] http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/

Type	n	Avg. Time (s)
Sequential	4	6.5
	6	8.5
	12	18.3
	18	26.9
Parallel	4	2.0
	6	2.0
	12	3.8
	18	5.4

Table 3. Average time spent on getting random posts from Facebook

We can see that, even with our very conservative assumptions (we believe that this probability is much lower), the user will have approx. one collision per year (0,1% of 912 posts). We believe that this is an acceptable collision rate; obviously, developers can use higher n values to further decrease the number of collisions, although doing so will also impact the performance of the application as explained in the next section.

4.3 Performance

There are three steps in the anonyLikes protocol that may take long enough to affect the usability: (i) retrieving n random posts; (ii) encrypting the message with the "likes", and (iii) decrypting the number of "likes". Their duration is related to CPU consumption and time spent on network calls. We measured each of these steps.

Retrieving n Random Posts To retrieve n random posts we issue n AJAX calls to Facebook's search API. We experimented with two different approaches: (i) sequential - we make (synchronously) AJAX calls sequentially, and (ii) parallel - we make all AJAX calls in parallel and wait until all of them finish. Using the Chrome browser connected to the Internet through a regular domestic 6 Mb/s Cable connection (a common scenario), we achieved the results shown in Table 3.

As expected, the parallel approach is much faster as the browser establishes multiple connections with Facebook. However, per the HTTP specification, browsers put a cap on the maximum number of connections to the same host (this cap is browser dependent). In Chrome (the browser that was used for these tests) the cap is 6 connections and that is the reason why the time to get 4 random posts is equal to the time needed to get 6 random posts.

Even if we set n to be 18, and given the fact that we start fetching random posts as soon as a page loads, we find 5.4 seconds to be a reasonable time that doesn't affect the usability because the user will generally take longer to read the page content before clicking the "anonyLike" button.

Encrypting "Like" Messages To measure the time spent on the ElGamal encryption algorithm, we used both Chrome and Firefox browsers running on a Intel Core i7 Processor (4 cores) at 2.2Ghz (Windows 8). We experimented several values for n (number

Operation	Chrome	Firefox
Encrypt "likes" (n = 6)	1366	1324
Encrypt "likes" (n = 12)	2742	2461
Encrypt "likes" (n = 20)	4584	3968

Table 4. Average time spent encrypting the number of "likes" (ms)

Operation	Avg. time (ms)
Getting encrypted num of "likes" from SNA-server	19
Getting decryption factor from one trustee	79
Decryption in the browser (Javascript)	30
Total (with three trustees)	286

Table 5. Average time spent on getting and decrypting the number of "likes"

of random posts that go along the "liked" post) since this affects directly the encryption time.

We can see in Table 4 that, although the ElGamal encryption is a heavy operation, it consumes an acceptable amount of time. We can also see that Firefox is slightly faster than Chrome but the difference is very small. To improve the user experience, we use a similar technique to what we used for retrieving random posts (see previous subsection): start encrypting the "like" as soon as the page loads so that when the user clicks the button it is (hopefully) already encrypted and the user only has to wait for the SNA-server to respond. If the user takes longer than 10 secs to read the page, it is long enough to *hide* the 5.4 secs retrieving random posts (as shown in previous section) plus 4 secs encrypting; thus, the "like" will have already been encrypted before the user reads the page; otherwise he will have to wait a few seconds (during which a progress dialog is shown).

Decrypting the Number of "Likes" Decrypting the number of "likes" involves three steps: (i) getting the encrypted number of "likes" from the SNA-server; (ii) getting the decryption factors from each trustee; (iii) use the decryption factors to decrypt the number of "likes".

We can see that the time spent is split between network calls and CPU consumption. To measure this operation, we used a Chrome browser running on a Intel Core i7 Processor (4 cores) at 2.2Ghz (Windows 8). We used three Trustees and both the SNA-Server and the Trustees were in the same machine as the browser (localhost). Since we cannot control the real network bandwidth that will be available to anonyLikes clients, we measured this operation on the localhost. This way, we could understand how long the SNA-server takes to respond with the encrypted number of "likes" (step i), and how long each trustee takes to respond with the decryption factor (step ii). Table 5 shows the results. We can see that the whole operation is very fast (almost unnoticeable by the

user) with most time spent in getting the decryption factors from each trustee. Obviously, the more trustees needed to decrypt the message, the longer it will take.

We believe that three trustees is a reasonable number of trustees in a real-world scenario (regarding the risk of collusion) but there is nothing in anonyLikes preventing any other number of trustees. We have performed the same performance tests (as above) with five and seven trustees and the whole operation is very fast as well (424 ms and 552 ms, respectively).

5 Related Work

AnonyLikes is motivated by the recent use of SNAs as a vehicle for activism and can be related to previous work done in the area of electronic voting (considering a vote to be similar to a "like"). Therefore, we present related work pertaining both areas.

5.1 Activism and Privacy on Social Networks

The role of social networks during major political transitions such as the ones witnessed in Tunisia, Egypt and Iran have raised interest on the research community in recent years [20,2,24]. For example, the series of uprisings transforming the Arab world (the so-called "Arab Spring") following the Tunisian revolution in January 2011 made extensive use of SNAs coupled with mobile technology enabling citizens to create communities bounded together by a common goal [22]. Also, some studies have shown the impact of SNAs on civic activism (activism not targeted towards a government) such as the Mexican Drug War [21]. These forms of activism all share concerns regarding the privacy of the people involved, since the targeted entity (e.g., government, drug cartel) can repress them by arresting, torturing and even killing.

To prevent these forms of repression, activists have to take special measures to protect themselves. They use pseudonyms instead of real names on their SNA accounts [24]. However, using pseudonyms has several disadvantages: (1) people don't trust pseudonyms as much as real names; (2) pseudonyms can be used by the government to post misleading activist content; (3) given enough history, it is possible to associate pseudonyms with real identities [4].

More experienced users also hide their IP address (which can be used to identify them by their location) when making connections with SNAs using proxy servers or public hotspots [24]. However, this technique relies on technological know-how that the majority of the users don't have. One interesting (and effective) measure taken by the authorities to suppress and obstruct the information flow between local internet activists is to reduce the speed of data transfer on the local ISP (Internet Service Provider) thus increasing the time to upload video materials to Facebook accounts. Activists get around this restriction by sending the material abroad via email in low resolution to be uploaded from there.

The use of pseudonyms raises a fundamental question: "can I trust this person?". For example, Monroy-Hernandez [21] presents the difficulty on asserting the credibility of information as one of the main conclusions of his work on the role of SNAs in Mexican Drug War. One important technique outlined in his work is to test reproducibility - if

the same fact is present in posts coming from multiple disparate sources, than it must be true. Based on this, some people have created special well-known Facebook accounts to which everyone can send messages. If a lot of messages contain the same fact, that fact is published on this account.

The anonyLikes protocol is designed to protect the privacy of content ”likers” and not so much the content creators. However, it can also help anonymized (through pseudonyms) content creators to get credibility through the amount of quantitative feedback their posts receive. Although not guaranteed, a post ”liked” by thousands of users is more credible than a post ”liked” by only a few. By protecting the privacy of the ”likers”, this effect may be more visible on these environments.

5.2 Electronic Voting

Electronic voting has been a research topic for over 25 years [3,7]. Today, there is a consensus on the minimum set of properties that these systems should satisfy [16]:

- **Accuracy**: (1) it is not possible for a vote to be altered, (2) it is not possible for a validated vote to be eliminated from the final tally, and (3) it is not possible for an invalid vote to be counted in the final tally.
- **Democracy**: (1) it allows only eligible voters to vote, and (2) it ensures that eligible voters vote only once.
- **Privacy**: (1) neither authorities nor anyone else can link any ballot to the voter who cast it, and (2) no voter can prove that he voted in a particular way.
- **Verifiability**: anyone can independently verify that all votes have been counted correctly.

Note that when compared to our scenario of supporting anonymous ”likes” in social networks, there are some relevant differences: i) users can ”like” anything from an almost unlimited set of posts while in elections there is a small limited set of candidates from which to choose from, and ii) the period during which users can ”like” is unlimited, while in an election there is not a point in time when the election is closed and the number of votes revealed; this is also related to the fact that, while on elections it is not possible to know intermediary results (i.e., to know the current vote count before the voting period ends) in SNAs it is possible (and desirable) to know the current number of ”likes” at any time.

Regarding the accuracy property, the anonyLikes protocol cannot ensure that it is not possible to inject fake ”likes”. However, it ensures that a ”like” cannot be transformed into a ”non-like” (since what is stored is the number of ”likes” and not the individual ”like”). AnonyLikes also satisfies the democracy and privacy properties but does not satisfy the verifiability property. However, since users expect to see an immediate increase on the number of ”likes” after their action, it would be very difficult to alter (adding or decreasing) the ”likes” count without raising suspicions. This derives from the fact that in anonyLikes, the effect of a ”like” must be visible to everyone (requirement R4).

Voting protocols can be categorized into three main categories accordingly to their cryptographic primitives: mixnets [5,17], blind signatures [15,16] and homomorphic encryption [3,9].

Mixnets create a robust anonymous channel by having encrypted votes going through a collection of servers whose task is to shuffle them. To ensure that mix-servers do not drop or replace votes, the servers must provide proofs of correct operation. One example of mixnets applied to elections is the scheme proposed by Lee [18] consisting on four steps. First, each voter prepares a first ballot by encrypting his vote. The ballot is then sent to a tamper-resistant randomizer (TRR) for randomization. Second, the TRR randomizes the first ballot with re-encryption to produce a final ballot. Third, the TRR also produces a Designated Verifier Re-encryption Proof (DVRP) to prove the correctness of re-encryption to the voter. The final ballot and the DVRP are then sent to the voter. Finally, the voter checks the DVRP, then signs and submits the final ballot if the check is accepted. A general criticism of mixnets is that the proofs of correction can be complex, cumbersome and inefficient [15].

Blind signatures [6] are a class of digital signatures where a message gets digitally signed without giving any knowledge about the message to the signer. This is similar to putting a document and a sheet of carbon paper in a sealed envelope that somebody signs on the outside. After removing the envelope we get the signed document. Applying this technique to electronic elections, voters obtain a blind signature on their ballot from an administrator which is then submitted to a voting bulletin board. The bulletin board will only accept votes signed by the administrator. This protocol has the advantages of simplicity and low computational cost. The problem is that the submission of votes to the bulletin board must use an anonymous channel which is hard to achieve. Frequently, this anonymous channel is implemented using mixnets but if a mixnet is available a blind signature is not required anymore.

Regarding SNAs, blind signatures have been used to design a privacy-enhanced variant of Twitter, where the content, hashtags and follower interests are encrypted and thus not visible to the Twitter server [10]. However, this mechanism doesn't provide anonymous quantitative feedback (e.g., number of retweets).

In a homomorphic encryption-based [23] voting scheme, votes are added while encrypted, so no individual vote ever needs to be revealed. In order to ensure that the private decryption key of the election is not used to decrypt an individual vote, a threshold encryption scheme must be applied to distribute the key among several authorities in such a way that multiple authorities have to combine their shares in order to use it. Although computationally expensive, this scheme has the big advantage of not requiring an anonymous channel. In fact, voters may openly authenticate themselves to the voting servers. Since the anonyLikes protocol is to be used in large-scale through the Internet channel (increasing the complexity of creating an anonymous channel), we chose homomorphic encryption as the underlying scheme for the anonyLikes protocol.

Homomorphic encryption has already been proposed as privacy-preserving mechanism for SNAs. For example, Domingo-Ferrer [12] proposes a system that uses homomorphic encryption to preserve the privacy of social network relationships when accessing a resource. It has also been used to preserve privacy when finding friends within a certain geographical distance [13] and matching personal profiles [19] among others.

To the best of our knowledge, anonyLikes is the first protocol to apply homomorphic encryption (or any cryptographic technique whatsoever) to quantitative feedback in social networks.

6 Conclusions

Quantitative feedback on SNAs has been shown to have profound impact on several forms of activism, by raising awareness and giving voice to important and sensitive topics in environments that otherwise constrain freedom of speech. Examples of such environments are countries with authoritarian governments and cities controlled by drug cartels. In these environments, people are afraid to use SNAs to promote their causes as they can be subject to retaliations from the targeted entities.

In this paper, we propose anonyLikes, a protocol that enables SNAs users to give quantitative feedback without revealing their identity, even to the social network infrastructure itself. This protocol employs strong cryptographic techniques (homomorphic encryption, shared threshold key pairs) to guarantee the privacy of the users. In particular, what is stored on the SNA-server is not individual ”likes” but rather the count of ”likes” on any given post. Moreover, that count is encrypted in such a way that several entities (trustees) must collaborate to decrypt it (i.e., it is not possible for a single entity to decrypt it).

We are able to *probabilistically* prevent duplicated ”likes” without breaking user's privacy by mixing the real ”like” with several fake ”likes” (i.e. by sending several post_ids for which the SNA-server doesn't know if they have been ”liked” or not). This effectively prevents duplicated ”likes” but can wrongly prevent a legitimate ”like” although this happens only once per year on average. We believe we have achieved an acceptable tradeoff between privacy and usability.

We have implemented the anonyLikes protocol within a Facebook replica, using an interaction model very similar to Facebook: developers can easily embed an ”anonyLike” button next to any content (blog post, video, etc.). The same mechanism also displays the current number of ”likes” of that post. This implementation is publicly available and can be used by any SNA developer wishing to support privacy-preserving quantitative feedback.

Finally, we evaluated the performance of the system and found that it can be implemented today without breaking user expectations for this kind of applications. Moreover, since the performance is tied to CPU speed, it will tend to improve in the upcoming years with advances in processor technology.

Acknowledgments

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects Pest-OE/EEI/LA0021/2013 and PTDC/EIA-EIA/113993/2009.

References

1. B. Adida. Helios: Web-based open-audit voting. *USENIX Security Symposium*, pages 335–348, 2008.
2. B. Al-Ani, G. Mark, J. Chung, and J. Jones. The Egyptian blogosphere: a counter-narrative of the revolution. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, 2012.
3. J. Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.
4. A. Beresford and F. Stajano. Location privacy in pervasive computing. *Pervasive Computing, IEEE*, 2005.
5. D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.
6. D. Chaum. Blind signatures for untraceable payments. *Advances in Cryptology: Proceedings of Crypto*, 1982.
7. D. Chaum. Elections with unconditionally-secret ballots and disruption equivalent to breaking RSA. *Advances in Cryptology—EUROCRYPT'88*, 1988.
8. R. Cramer, I. Damgå rd, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. *Advances in Cryptology - Crypto 94*, 1994.
9. R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European transactions on Telecommunications*, (8.5):481–490, 1997.
10. E. D. Cristofaro and C. Soriente. Hummingbird: Privacy at the time of twitter. In *2012 IEEE Symposium on Security and Privacy (SP)*, volume 1692, pages 285–299, 2012.
11. Y. Desmedt and Y. Frankel. Threshold cryptosystems. *Advances in Cryptology—CRYPTO'89*, 1990.
12. J. Domingo-Ferrer, A. Viejo, F. Sebé, and U. González-Nicolás. Privacy homomorphisms for social networks with private relationships. *Computer Networks*, 52(15):3007–3016, Oct. 2008.
13. W. Dong, V. Dave, L. Qiu, and Y. Zhang. Secure friend discovery in mobile social networks. *2011 Proceedings IEEE INFOCOM*, pages 1647–1655, Apr. 2011.
14. T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO 84 on Advances in cryptology*, pages 10–18, Aug. 1985.
15. A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. *Advances in Cryptology—AUSCRYPT'92*, 1993.
16. R. Joaquim, P. Ferreira, and C. Ribeiro. EVIV: An end-to-end verifiable Internet voting system. *Computers & Security*, 32:170–191, Feb. 2013.
17. A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pages 61–70, 2005.
18. B. Lee, C. Boyd, E. Dawson, and K. Kim. Providing receipt-freeness in mixnet-based voting protocols. *Information Security and Cryptology-ICISC 2003*, pages 1–14, 2004.
19. M. Li, N. Cao, S. Yu, and W. Lou. FindU: Privacy-preserving personal profile matching in mobile social networks. *2011 Proceedings IEEE INFOCOM*, pages 2435–2443, Apr. 2011.
20. G. Lotan, E. Graeff, M. Ananny, D. Gaffney, I. Pearce, and D. Boyd. The Revolutions Were Tweeted: Information Flows During the 2011 Tunisian and Egyptian Revolutions. *International Journal of Communication*, 5:1375–1406, 2011.
21. A. Monroy-Hernández. The new war correspondents: he rise of civic media curation in urban warfare. In *Proceedings of the 2013 ACM conference on Computer Supported Cooperative Work*, pages 1443–1452, 2013.
22. O. Olaore. Politexting: Using Mobile Technology to Connect the Unconnected and Expanding the Scope of Political Communication. *Information Systems Educators Conference 2011 ISECON Proceedings*, pages 1–8, 2011.

23. R. Rivest. On data banks and privacy homomorphisms. *Foundations of secure computation 4.11*, 1978.
24. V. Wulf, K. Misaki, and M. Atam. 'On the ground'in Sidi Bouzid: investigating social media use during the tunisian revolution. In *Proceedings of the 2013 ACM conference on Computer Supported Cooperative Work*, pages 1409–1418, 2013.