2014 IEEE International Conference on High Performance Computing and Communications (HPCC), 2014 IEEE 6th International Symposium on Cyberspace Safety and Security (CSS) and 2014 IEEE 11th International Conference on Embedded Software and Systems (ICESS)

Snake: an End-to-End Encrypted Online Social Network

Alessandro Barenghi and Michele Beretta and Alessandro Di Federico and Gerardo Pelosi
Department of Electronics, Information and Bioengineering – DEIB
Politecnico di Milano, 20133 Milano, Italy
name.surname@polimi.it

***Abstract*—Online Social Networks (OSNs) have evolved into one of the prime means for social interactions, effectively generating significant amount of digital contents. In this context, providing proper privacy and confidentiality services for the generated contents is a crucial issue to foster their development. We propose Snake, an end-to-end encrypted social network, where the confidentiality and privacy-protecting logic is shouldered entirely by the clients, improving the guarantees provided by state of the art solutions. We designed Snake so that the storage provider performs only simple data access operations, and the inspection of the data at rest does not reveal the involved parties in a communication. We implemented Snake as a single page HTML5 JavaScript application employing W3C's WebCrypto API, and evaluated experimentally our approach proving its practical feasibility.**

***Index Terms*—Online Social Networks, OSN, End-to-End Encryption**

I. Introduction

Over the past decade, participatory information sharing and collaboration through Web 2.0 technologies has transformed many aspects of modern society and social interaction. One of the most significant developments connected to social media has been the advent of social network platforms, such as *LinkedIn*, *MySpace*, *Facebook*, *Twitter*, *Sina Weibo* and *Google+*. Nowadays, the most common and complete acceptation of Online Social Networks (OSN) defines them as communication platforms in which participants (i) have uniquely identifiable profiles that consist of user supplied content, content provided by other users, and/or system provided data; (ii) can publicly articulate connections that can be viewed and traversed by others; and (iii) can consume, produce, and/or interact with streams of user generated content provided by their connections on the site [1].

The rapid development of OSNs and the related business models (e.g., advertising) are receiving much attention in research communities due to the pressing privacy problems. Currently, most widespread social network platforms run on a logically centralized infrastructure for user and application data, while relying on content distribution networks to ensure responsiveness, scalability and availability of their services. Nevertheless, due to the public or semi-public nature of many social services and to the lack of any confidentiality preserving techniques, user content can be easily disclosed to a wider audience than the owner originally intended [2]. Moreover, a non negligible issue is the fact that such data can be easily accessed, manipulated or misused by the service provider itself [3]. In particular, it is commonplace for OSN service providers to employ extensive data mining techniques on the user-produced contents as a part of their business strategies, obtaining economic gains from targeted advertisements. In such a scenario, a prominent issue consists in providing a private and secure OSN approach where the end user is responsible for the disclosure of his own data. In addition to this, the issue of authenticating acquaintances and friends alike over a secure OSN should be tackled, as the creation of social links is one of the fundamental activities undertaken by leisure targeted OSNs such as Facebook and its likes. We note that, being inherently targeted mostly at non-technically oriented people, such services should be provided in a non obtrusive fashion [4].

Contribution. We propose Snake [5], an end-to-end encrypted OSN, where all the encryption and decryption actions are performed by client, while relying on a storage provider to perform only Create-Read-Update-Delete (CRUD) operations on the fully encrypted data hosted by it, an innovative approach to this area. To the purpose of providing authentication between users estabilishing a friendship, we provide a novel in-band, user friendly key-exchange mechanism, aiming at minimizing the amount of disclosed information. Snake also provides data integrity and prevents an adversary examining the whole corpus of data at rest residing on the storage provider from either identifying the endpoints of social interactions, or clustering exchanged data according to them. Shifting the whole burden of the encryption on the client side has been deemed cumbersome: we prove the practicality of our approach with respect to both the computational demand, and ease of cross device portability and deployment. In particular, we implemented the proposed OSN client as a single-page HTML5-JavaScript application. The novel W3C WebCrypto API [6] enabled us to employ efficient cryptographic primitives across multiple environments, through a standard interface.

Organization of the paper. Section II describes the security desiderata and key features of proposed architecture, Section III details the infrastructure and implementation choices of Snake, Section IV provides the results of the experimental evaluation of our proposal. Section V compares our approach with the existing literature on privacy-aware OSNs, and Section VI draws our conclusions.

II. System Structure and Security Objectives

System Architecture. Most successful Online Social Networks follow the client/server architecture that is common on the web. Assuming the same basic paradigm, the pivotal design choice of Snake relies on moving most of the domain logic on the client, in particular everything acting on un-encrypted data. Consequentially, the server side is assumed to be a simple storage provider, providing a CRUD interface on a database storing all user data and social graph in encrypted form. The client is a Single-Page Application, realized in HTML5 and JavaScript, providing immediate portability to a large number of platforms, as the required runtime is simply a modern web browser. Moreover, thanks to technologies such as PhoneGap [7], it is also possible to automatically package it as a mobile application for a wide range of devices. As depicted in Figure 1, this implementation choice allows the source code to be distributed from an *application provider* which can be fully independent from the *storage provider*. This responsibility split, along with an open-source license for all the components of the system, allows great versatility for the end-user in choosing whom to trust. Indeed, it is possible either to choose a common repository(marketplace)-based distribution strategy or, if it is desired by the user, to self-host the application distribution server with minimal effort. The latter solution provides the technology-savvy users with an application deployment platform relying on a shorter trust chain, as the application deployment server is fully trusted. The availability of both choices for the application distribution positively enhances the *Trust Agility* of the user, i.e, he has the option to

978-1-4799-6123-8/14 $31.00 © 2014 IEEE
DOI 10.1109/HPCC.2014.128

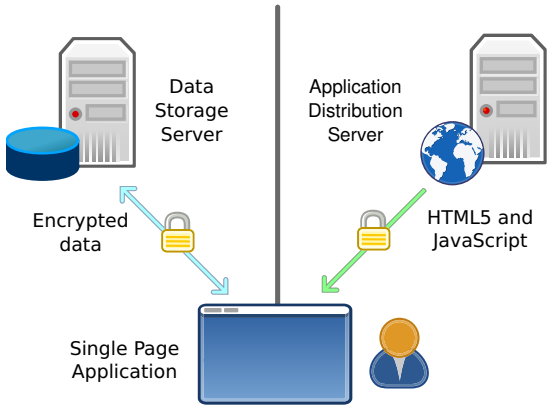

Figure 1. Overview of the System Architecture

choose who to trust and eventually move from one code provider to another without significant issues. In this work, we assume the network transport layer to be secure and trusted. Such a requirement is fulfilled by solutions such as the SSL/TLS protocols, employing a sound public key authentication infrastructure. Although the issues stemming from maintaining a PKI are out of scope in this work, we point to other works such as *Convergence*[1] or the emerging *Public Key Pinning Extension for HTTP* [8] standard for interesting solutions.

Security Objectives and Threat Model. The proposed design deploys an encryption layer over all the data shared in the social network, providing data confidentiality and integrity against snooping and tampering attacks performed by either non-authorized users or the storage provider itself. In addition to this basic feature, Snake provides authentication of the stored data to prevent *impersonation attacks*. Finally Snake employs only anonymous metadata as indexes to perform remote database queries, striving to minimize the amount of information obtainable from a leakage of the whole set of data at rest. For instance, examining only a dump of the database contents, it is not possible to infer the identity in the system of the sender or the recipient. To this end, we consider a storage server honoring the service level agreement, i.e. providing the availability of the storage and not disclosing to the clients more information than those required by the protocol. We also assume that the server does not annotate the stored data with additional metadata (f.i., source IP address, web browser fingerprint or timestamps of when the communication took place). We note that the usefulness of the additional metadata annotations can be foiled, at the expense of an increased surfing latency of the clients, through the use of mixing networks and other tools performing browsing signatures removal. These measures can be employed even by non technically expert users through pre-built browser bundles such as the Tor Browser Bundle[2].

Tackling issues such as the one of protecting the information leakage from the mining of data access patterns is out of the scope of this work and data structures specifically designed to provide such guarantees can be integrated in our framework [9]–[11].

Along with the provided security services, one of our design requirements was the preservation of the usability of the system by non-technically aware users. This requirement, despite not being strictly related to security guarantees, has proven to be crucial for the adoption of the system, when the typical user does not receive explicit security training [12]. Indeed, in Snake the cryptographic protections applied to the social data of each user as well as to the information needed to enable any two accounts to privately interact, are protected by a single cryptographic key, derived upon login from the provided *password*. Subsequently, this key is employed to retrieve from the storage provider all the other cryptographic tokens needed for the functioning of the application as well as all the data the user is entitled to access. The usability requirement also drove the tailoring of a specific *authenticated key agreement protocol*, described in Section III-B, relying on implicitly shared secrets existing among users who already know each other.

Design choices. The data confidentiality on the remote storage server is enforced through end-to-end semantically secure symmetric-encryption. Practically, the remote storage server does not compute any cryptographic primitives, as they are all computed on the client. The resources shared within a group of users (f.i., files or messages) will be encrypted with a symmetric key generated, distributed and revoked by the group administrator making use of an efficient broadcast key-distribution protocols based on the design in [13]. However, in this paper we do not report the details of the group management protocol adopted by Snake in favor of the analysis of the security and performance figures of merit provided by the proposed social networking framework.

Each user is enabled to use the aforementioned keys upon signing-in the application and accessing his own keyring. The user stores its keyring, along with other private information, in a *private descriptor* on the storage provider, encrypted via symmetric block cipher with his password-derived key. Upon login, the client fetches the encrypted *private descriptor* from the database on the server, generates locally the decryption key from the password, and allows the user to access the cryptographic keys to communicate with his friends.

Data authentication is provided by means of digital signatures: all the data is signed before encrypting them with a symmetric key. Each user of the system is associated with a personal public/private keypair to perform signatures. The cryptographic keys employed for the encryption of messages destined to other users are the result of a new *authenticated key agreement* process between the involved parties taking place at the time of friendship establishment. To authenticate the user's public keys in our framework, we chose not to rely on a trusted third party vouching for their authenticity, instead we adopted two different methods: the first based on the Socialist Millionaire Protocol (SMP) and the second based on a Web of Trust (WoT). The SMP-based authentication was chosen due to its high usability, as it maps the trust relationship between a user identity and his public key to a real-world social trust between the two parties which are willing to mutually authenticate through matching a question/answer pair. We note that, this kind of implicit secret-shared authentication protocols have been proven to be a reliable form of authentication in [14], provided that the questions employed are not too generic. In alternative, the WoT-based authentication mechanism is supported by Snake to enhance the performance of the authentication subsystem, reducing the number of required interactions between the parties.

The *private descriptor* of each user includes, for each one of his contacts, their authenticated public key (for signature validation), and the symmetric key employed for private data exchange.

To the end of providing anonymous metadata, all the data stored on the server is indexed by pseudonyms chosen independently from the actual identity of the data owner.

A key point of our design is that it does not require the storage of metadata which could be useful to an attacker gaining access to a cold image of the entire data archive of the storage provider to function properly. Furthermore, our design minimizes the join paths on the storage provider database, so to reduce the amount of related information available.

The choice on which cryptographic primitives to employ in the

[1]Moxie Marlinspike, http://convergence.io/ (last accessed: May 2014)
[2]https://www.torproject.org/projects/ (last accessed: May 2014)

LOGIN TABLE

username	salt	privateDescriptor	editToken
$\mathcal{A}$	0x2152	$\text{Enc}_{MK_\mathcal{A}}(pri_\mathcal{A}, x, pri'_\mathcal{A}, fl_\mathcal{A})$	0x3FFA
$\mathcal{B}$	0x4FF2	$\text{Enc}_{MK_\mathcal{B}}(pri_\mathcal{B}, y, pri'_\mathcal{B}, fl_\mathcal{B})$	0x35A5
$\mathcal{C}$	0x6699	$\text{Enc}_{MK_\mathcal{C}}(pri_\mathcal{C}, z, pri'_\mathcal{C}, fl_\mathcal{C})$	0x4295
$\mathcal{D}$	0x5111	$\text{Enc}_{MK_\mathcal{D}}(pri_\mathcal{D}, w, pri'_\mathcal{D}, fl_\mathcal{D})$	0x1665

PROFILE TABLE

id	data	editToken
0x4F2F	$\text{Sign}_{pri'_\mathcal{A}}$ ({"Name" : "Alice"})	0x2525
0x3FA5	$\text{Sign}_{pri'_\mathcal{B}}$ ({"Name" : "Bob"})	0x0F58
0x458C	$\text{SignEnc}_{PK_\mathcal{A}, pri'_\mathcal{A}}$ ({"Age" : 25, "fl" : [...]})	0x00DE
0xAB2C	$\text{SignEnc}_{PK_\mathcal{B}, pri'_\mathcal{B}}$ ({"Age" : 18, "fl" : [...]})	0x17AB

FRIENDSHIP TABLE

username	friendshipAddress	staticPublicKey	ephemeralPublicKey	signingPublicKey	publicProfile	editToken
$\mathcal{A}$	0x3535	$pub_\mathcal{A}$	g^x	$pub'_\mathcal{A}$	0x4F2F	0x2F2F
$\mathcal{B}$	0x4F4F	$pub_\mathcal{B}$	g^y	$pub'_\mathcal{B}$	0x3FA5	0x4545

Figure 2. Logic schema of the database on the storage server. The FRIENDSHIP TABLE contains for each user $\mathcal{A}$, her the destination address for friendship requests, her long term public key $pub_\mathcal{A}$, her ephemeral public key g^x, her signing public key $pub'_\mathcal{A}$ and a reference to her public profile. The LOGIN TABLE contains for an user $\mathcal{A}$, the salt employed to derive her password-derived master key $MK_\mathcal{A}$ and her *private descriptor*. The PROFILE TABLE contains both public and private profiles. Public profiles are stored in signed cleartext, while private profiles are also encrypted with a dedicated symmetric key $PK_\mathcal{A}$. The client is able to automatically detect whether the record is encrypted or not depending on its format.

ciphersuite was driven by the need to adopt efficient and state-of-the-art algorithms as well as by the availability of such primitives within the WebCrypto API [6], the emerging W3C standard for in-browser cryptography allowing to use high performance primitives in a web browser scripting environment. Willing to provide the data confidentiality over a long timeframe, we opted for AES as the symmetric block cipher of choice, employing a 256-bit key. The data integrity is obtained encrypting the data in Galois Counter Mode (GCM). The key derivation function employed to obtain the *private descriptor* encryption key is the IETF-standard PBKDF2 primitive (RFC 2898 [15]), based on 1024 iterations of the SHA-2-256 algorithm. As digital signature scheme to provide data authentication we chose the well established Elliptic Curve Digital Signature Algorithm (ECDSA) employing the NIST standardized P-256 curve.

III. THE SNAKE OSN

In this section we will describe the structure of Snake, detailing how common OSN interactions are implemented.

By design, the interaction between the client and the storage provider is fully stateless, and can thus be realized over a classical RESTful interface. However, due to performance reasons, we chose to adopt a WebSocket-based communication layer, employing the `Socket.IO` library[3]. This reduces the communication overhead, avoiding the need to employ HTTP headers. The exchanged data is encoded as JSON messages, which are validated against a formal description of the message language itself, as specified in the IETF JSON-Schema proposed in [16]. The validation is strict, and messages failing to pass it are discarded, effectively preventing malformed malicious inputs from being employed.

A. Details of common OSN interactions

We provide a description of the management of the typical user interactions with the OSN describing the inner workings of Snake. Figures 2 and 3 provide an instance of the remotely stored data.

Registration and Login. Upon registration, the user $\mathcal{A}$ picks an `username`, a password, and optionally provides public profile information. A symmetric AES-256 master key (e.g. $MK_\mathcal{A}$ in Figure 2) is derived from the user-provided password using PBKDF2 with a random `salt` value, and is employed to encipher the `privateDescriptor`. The `privateDescriptor` contains the list of friends $fl_\mathcal{A}$ along with all the information needed to communicate with them, i.e., a copy of their public keys and the secret keys (as sketched in Section II) agreed upon friendship establishment. Following the symmetric key derivation, $\mathcal{A}$ generates three asymmetric keypairs. Given a finite algebraic cyclic group $(\mathbb{G}, \cdot)$ generated by g, $\mathcal{A}$ generates two keypairs of a discrete logarithm based cryptosystem (e.g., ECDSA), $\langle x, g^x \rangle$ and $\langle pri_\mathcal{A}, pub_\mathcal{A} \rangle$, picking as private keys $x, pri_\mathcal{A}$ two integers $x, pri_\mathcal{A} \in \{2, \ldots, |\mathbb{G}| - 1\}$, and deriving the corresponding public keys as g^x and $pub_\mathcal{A} = g^{pri_\mathcal{A}}$. The last keypair, employed for message-signing purposes, $\langle pri'_\mathcal{A}, pub'_\mathcal{A} \rangle$ can be generated from any asymmetric cryptosystem allowing to perform message signatures. The private keys are then stored in the `privateDescriptor`. Once the key generation process is completed, the PBKDF2 salt, the username, and the `privateDescriptor` are inserted into the LOGIN TABLE employing the username as the primary key. Following this, the public key triple $\langle pub_\mathcal{A}, g^x, pub'_\mathcal{A} \rangle$, the username, and the identifier of the public profile associated with the user are stored in the FRIENDSHIP TABLE. This table also contains the `friendshipAddress` field, which we will detail better later.

To provide in-depth protection for the password derived key, we employ the so-called *unextractable keys* feature from the WebCrypto API. An unextractable key cannot be exported by JavaScript code, i.e., once it has been initialized it is not possible to access its value, preventing an attacker from gaining direct access to it in case of compromission after application initialization.

To access his account, $\mathcal{A}$ provides the client his username and password. The client application retrieves from the server the record corresponding to the user from the LOGIN TABLE, derives $MK_\mathcal{A}$ via PBKDF2 using the password and the `salt` value and decrypts $\mathcal{A}$'s *private descriptor*. In this way $\mathcal{A}$ gains access to all the secret information using a single password.

Send a private message. User $\mathcal{A}$ picks $\mathcal{B}$ from his friend list $fl_\mathcal{A}$ as the recipient. The message content is signed with the sender's private key $pri'_\mathcal{A}$, encrypted with the symmetric key shared between $\mathcal{A}$ and $\mathcal{B}$ (agreed upon at friendship establishment time). The encrypted message is then sent to the storage provider along with a destination `address` (agreed upon for the first time at friendship establishment time, and refreshed at each message), identifying a directed sender→recipient pair, to be stored in the MESSAGE TABLE (Figure 3). Each user stores in its *private descriptor* two addresses for each friend: one to receive messages and one to send them. Since everything but the randomly generated `address` is stored in encrypted form, it is not possible to identify the sender or the receiver from an inspection of the data at rest. The `address`, together with a progressive integer, is used as the primary key of the message in the remote database.

Fetch new messages. To fetch all her unread messages, $\mathcal{A}$ asks for all the messages whose `address` identifies her as the recipient, and where `index` is greater than the last seen one. The last seen `index` is stored in the *private descriptor* of the user. The client decrypts each

[3]http://socket.io/ (last accessed: May 2014)

MESSAGE TABLE

address	index	data	editToken
0x4F4F	10	g^x, $I_\mathcal{A}$, $t_\mathcal{A}$, $\text{SignEnc}_{pri'_\mathcal{A},K_S}$ (0x35DF)	0x3555
0x35DF	1	$t_\mathcal{B}$, $\text{SignEnc}_{pri'_\mathcal{B},K_S}$ (0x2E2E)	0x5DDF
0x2E2E	1	$\text{SignEnc}_{pri'_\mathcal{A},K_S}$ (0x458C, $PK_\mathcal{A}$)	0x3F8F
0x35DF	2	$\text{SignEnc}_{pri'_\mathcal{B},K_S}$ ("Hello!")	0x42EF

Figure 3. The MESSAGE TABLE contains both system messages, such as friendship requests, and ordinary text messages. Each message is identified by a destination `address`, and a progressive integer. Its content structure varies depending on the type of message. This example shows the messages exchanged during friendship establishment. K_S represents the session key derived through FHMQV-C, 0x458C is a reference to the private profile of $\mathcal{A}$ and $PK_\mathcal{A}$ is the key to use to decrypt it

message using the correct key depending on the *address*, checks for their authenticity and presents it to the user.

Edit a message. To edit an existing message, the client simply sends the updated version of the message to the server. Tampering with the messages contents or impersonating another user is detected, as all of them are digitally signed. To add a further layer of protection against unauthorized message editing, a user willing to perform an edit action must provide an `editToken`. The `editToken` is a 256-bit value generated by the user and sent upon message creation to the storage server, which never returns it to the clients. The `editToken` is employed as a proof of authorship: the storage server will update the message only if the `editToken` of the update request matches the stored one. For the sake of practicality, the client may use as an `editToken` the hash of the message, encrypted with a dedicated symmetric key. Snake employs `editToken`s to provide additional protection to all table entries.

B. Friendship establishment

Snake presents the authenticated key agreement on a symmetric key to be used for all communications with another user, as a *friendship establishment*, a familiar concept in the OSN environment. To this end, we chose, and adapted to the needs of our scenario, FHMQV-C as the key agreement protocol, relying on two strategies for public key authentication: the Socialist Millionaire Protocol or the Web of Trust infrastructure.

Key agreement. The FHMQV-C [17] protocol is an authenticated Diffie-Hellman key exchange protocol, which has been proven resistant in the Universal Composability model to: simple impersonation, Unknown Key Share (UKS) (where the two parties derive the same key but one is tricked into believing to be talking with an attacker), Key Compromise Impersonation (KCI) (where an attacker recovering the long term private key of an entity can impersonate any user at his eyes) and exposure attacks (i.e., as long as an attacker does not have both the long term and the ephemeral key of one of the two parties, the shared key is not compromised). In addition, FHMQV-C provides *perfect forward secrecy* and ensures both parties that the other end actively participated in the protocol and computed the same key.

Taking into account the long delays in the asynchronous interactions between users of an OSN, and our requirement to obtain anonymity of data at rest, we modified the FHMQV-C protocol both to reduce the number of messages to be sent interactively, and to keep private the identities of the participants in the key agreement. We will now provide the details of our tailoring, while describing the key exchange phases reported in Figure 4, where the modified actions of FHMQV-C are highlighted in bold. We define $\text{H}(m)$ as the result of a unforgeable hash function over m, $\text{H}'_K(m)$ as the result of an unforgeable keyed hash function over the bit string m using K as key, $\text{Enc}_K(m)$ and $\text{Dec}_K(s)$ the result of, respectively, the encryption and decryption of s using a symmetric block cipher with key K, $m|n$ as the concatentation of the bit strings m and n, $\text{Sig}_{pri'_\mathcal{A}}(m)$ as the digital signature of the bit string m using private key $pri'_\mathcal{A}$, and $\text{SignEnc}_{pri'_\mathcal{A},K}(m)$ as $\text{Enc}_K(Sign_{pri'_\mathcal{A}}(m)|m)$.

FHMQV-C involves two Diffie-Hellman keypairs for a given party $\mathcal{A}$ participating to the key exchange: a static keypair $\langle pri_\mathcal{A}, pub_\mathcal{A}\rangle$ and an ephemeral one x, g^x. We will denote as $\langle pri_\mathcal{B}, pub_\mathcal{B}\rangle$ and $\langle y, g^y\rangle$ the static and ephemeral keypairs for $\mathcal{B}$, and, with a small notation abuse, we will use $\mathcal{A}$ and $\mathcal{B}$ to indicate both the parties and the unique identifiers representing their identities. As a first step of our tailored protocol, we assume all the parties generate a static and an ephemeral keypair, and publish them in the FRIENDSHIP TABLE on the storage server. Exploiting the availability of the aforementioned keys, the key agreement initiator $\mathcal{A}$ (who is asking $\mathcal{B}$ to be her friend) is able to obtain $\mathcal{B}$'s ephemeral and static public keys, g^y and $pub_\mathcal{B}$ respectively. This allows $\mathcal{A}$ both to skip an interaction with $\mathcal{B}$, and to derive an ephemeral shared secret key $K_{ID} = (g^y)^x$, with a freshly generated ephemeral keypair x, g^x for this session, thus providing her the means to encrypt her identity before sending it to $\mathcal{B}$. Note that $\mathcal{A}$ does not use the ephemeral keypair she published on the storage server, but generates a new one.

After computing the encrypted identity $I_\mathcal{A}$, the session initiator $\mathcal{A}$ uses his own private keys, x and $pri_\mathcal{A}$, and the public keys of $\mathcal{B}$, g^y and $pub_\mathcal{B}$, to derive both the shared session key of the FHMQV-C protocol, K_s, and the one used to compute a confirmation tag, K_t. Subsequently, she computes the confirmation tag $t_\mathcal{A}$ as the keyed hash over her identifier $\mathcal{A}$ and her ephemeral public key g^x, and sends to $\mathcal{B}$ the encrypted identity $I_\mathcal{A}$, her ephemeral public key g^x, the confirmation tag $t_\mathcal{A}$ and a new `address` to contact her. $\mathcal{A}$ uses as message destination the `friendshipAddress` of $\mathcal{B}$, which is dedicated to friendship requests for him and is available in the FRIENDSHIP TABLE.

The vanilla FHMQV-C protocol does not encrypt the identity of the involved parties during key exchange, as opposed to our tailored version. Therefore, if we employed it, the messages concerning a friendship request would leak the initiatior identity. Such a knowledge, together with the knowledge of the destination `friendshipAddress` would allow the detection of a friendship request from $\mathcal{A}$ to $\mathcal{B}$, effectively leaking significant information. Note that this defense is not effective against an active attacker, since K_{ID} is derived using an unauthenticated public key g^y; however an active attack is detected by the friendship request recipient.

Once $\mathcal{B}$ receives the message, he is able to decrypt $I_\mathcal{A}$ employing g^x and his own ephemeral private key to derive $K_{ID} = (g^x)^y$, and decide whether or not accept $\mathcal{A}$'s friendship request. In case the request is accepted, $\mathcal{B}$ fetches $\mathcal{A}$'s static public key $pub_\mathcal{A}$, and uses his own private keys, y and $pri_\mathcal{B}$, and the public keys of $\mathcal{A}$, g^x and $pub_\mathcal{A}$, to derive the values of K_s and K_t, storing K_s as the symmetric communication key with $\mathcal{A}$ and employing K_t both to check the confirmation tag $t_\mathcal{A}$ and to compute his confirmation tag $t_\mathcal{B}$. $\mathcal{B}$ then generates a new ephemeral keypair $\langle y', g^{y'}\rangle$, replacing the one published on the storage server. Finally, $\mathcal{B}$ sends to $\mathcal{A}$ the confirmation tag together with a new `address` $\mathcal{A}$ will use from now on to contact him. We note that, in case multiple friendship establishment protocols towards $\mathcal{B}$ are initiated with the same ephemeral key, e.g., in case $\mathcal{B}$ is offline and cannot promptly refresh it, no effective protocol session advances to completion until $\mathcal{B}$ refreshes the keypair. This allows $\mathcal{B}$ to choose whether to enforce a per-friendship establishment rekeying, simply dropping all the requests made with the same ephemeral key, or to tolerate the fact that his ephemeral key was used more than once. Note that, whichever the choice it is not possible for $\mathcal{A}$ to rely on $\mathcal{B}$'s interactive answer to mount an adaptive attack where the ephemeral keypair of $\mathcal{B}$ is the same and her one changes. The last action of $\mathcal{B}$ is to send back the confirmation tag $t_\mathcal{B}$ to $\mathcal{A}$ which in turn checks it for correctness, confirming that $\mathcal{B}$ has performed the same computation

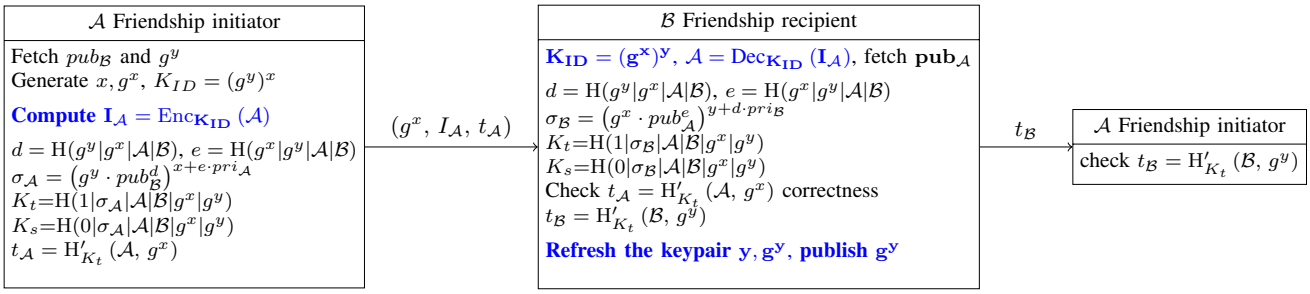

Figure 4. Friendship establishment protocol between $\mathcal{A}$ and $\mathcal{B}$: g is the generator of a finite cyclic group of order n, $2<x<n$, H is an unforgeable hash function, H$'$ a keyed unforgeable hash. The protocol derives K_s the key for the communication between $\mathcal{A}$ and $\mathcal{B}$ and K_t the key employed for the confirmation tags.

and effectively validating the friendship establishment. Our variation on the FHMQV-C protocol reduces the number of asynchronous interactions between the friendship establishment parties from 3 to 2, the same of an unencrypted friendship request, while providing the aforementioned security guarantees and eliminating the information leakage about the identities of the participants.

Long Term Public key authentication. The FHMQV-C Protocol requires both parties to have a long term keypair $\langle pri_{\mathcal{A}}, pub_{\mathcal{A}}\rangle$, which should be securely bound to their identity. In Snake we chose to provide such a long term keypair authentication either through the use of two instances of the Socialist Millionaire Protocol (SMP) [18], or via a Web of Trust, in case the two parties share common friends.

In particular, the SMP protocol allows us to bind the identities of $\mathcal{A}$ and $\mathcal{B}$ to their long term public keys, exploiting the knowledge of a pre-shared secret among them (e.g., the place where the two people met physically the first time) without relying on an explicit agreement on which secret should be used. Among the possible approaches to do so, we adopted the well established *question and answer* mechanism, in a fashion similar to the Off-The-Record (OTR) protocol [19]. From the end user's point of view, one party inputs both a question and the relative answer, which should be known only to the other party and himself. The SMP employs this question-answer pair to bind together the identity and the public key of the party providing the answer. As proven in [18], in case the answer provided by the second party does not coincide with the one provided by the asking one, the SMP does not leak any information, save for the fact that the two answers don't match. In our authenticated key agreement scheme, the SMP protocol is run concurrently with the tailored FHMQV-C. Thanks to this, it is possible to encipher all the SMP protocol messages employing the symmetric key K_s derived by the friendship establishment initiator since the beginning of the interaction.

In order to reduce the number of actions required in the authenticated key exchange, we provide a second authentication mechanism based on a Web of Trust (WoT) approach. Employing such an approach, it is possible to perform the authenticated key exchange sending only two messages, thus reaching optimal performances (i.e., the same communication requirements of an unencrypted friendship establishment). The WoT is a directed graph where the nodes are the user/public keys pairs, and the edges represent the belief by the source node that the destination node public key is authentic. A user $\mathcal{A}$, willing to authenticate the public key of user $\mathcal{B}$, will count for all the distinct paths, shorter than a certain value, from $\mathcal{A}$ to $\mathcal{B}$ on the Web of Trust: if the count exceeds a user determined threshold, the public key of $\mathcal{B}$ is deemed authentic.

We note that a straightforward implementation of the WoT, such as the one present in PGP/GnuPG [20] requires the complete WoT graph to be public. In our scenario, such a requirement cannot be met, lest any malicious user provider learns all the friendship relations among others. To this end, in Snake each user is able to access only the portion of the WoT graph comprising nodes which are at most two edges afar from him. This is implemented allowing a user to see only his friends' friend lists, effectively limiting the scope of his knowledge of the WoT graph. To allow a user to decide whether to trust her friend's WoT authentication, the friends list are annotated with the method employed to authenticate each public key. The list of friends of a user is stored in his private profile, which is made available by its owner to (a portion of) his friends through a message containing its unique identifier and the key to decrypt it (see message with `address` 0x2E2E and `index` 1 in Figure 3).

C. Privacy of Data at Rest

On the storage provider the system data is split in four entities: the LOGIN TABLE, the FRIENDSHIP TABLE, the PROFILE TABLE and the MESSAGE TABLE. To minimize the leakage of information on the database, we identified the essential fields that have to be in a form intelligible by the storage provider to keep the service working without major performance drops. In practice, these fields consists in the primary indexes as marked in 2 and 3, which are used to uniquely identify a record. In particular we use a primary key formed only by a random identifier, the destination `address`, and a progressive `index` for the MESSAGE TABLE. As shown in Figures 2 and 3, the result of these choices is that all the candidate join paths are suppressed, with the exception of Friendship.friendshipAddress $\rightarrow$ Message.address and Friendship.publicProfile $\rightarrow$ Profile.id. The former path associates a user with an *address* which has to be used to send new friendship requests to him. Note that if this value is refreshed at each friendship request, in practice the join path only reveals the pending friendship requests the user received. The latter path does not reveal anything, except for the information the user intentionally wanted to be public.

IV. EXPERIMENTAL RESULTS

In this section we will provide the experimental validation supporting the fact that shifting the burden of the OSN logic and encryption fully on the client is technically feasible, and does not hinder the usability of the technological solution. This achievement is possible thanks to both a significant increase in the computing power of the clients, especially in the mobile area, and the availability of a unified cryptographic API, the WebCrypto API. Currently, it is possible to adopt three different implementation strategies for the WebCrypto API: a native binary implementation in the browser, a native binary implementation in a browser plugin, and a full JavaScript *polyfill* (i.e., a JavaScript implementation fallback). Clearly, native browser implementations are the ones providing the best performance figures, and will be the prime way to offer WebCrypto API support in the future. Willing to provide a quantitative evaluation of the current state-of-the-art, we performed our benchmarks with three different implementations: the native implementation of the WebCrypto API

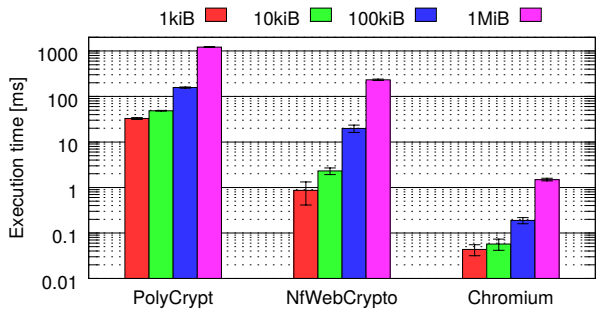

Figure 5. Time required to encrypt messages of different sizes with the available WebCrypto API implementation strategies

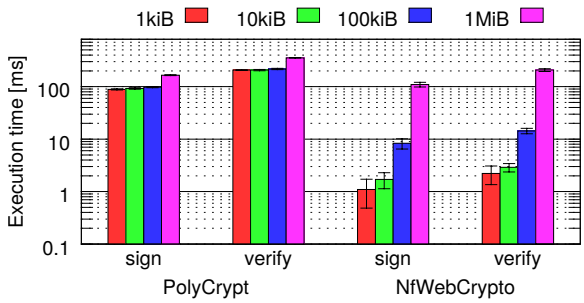

Figure 6. Execution times for the ECDSA signature and validation primitives using the P-256 curve

released with the Chromium web browser, the native implementation developed by Netflix as Chromium plugin, NfWebCrypto[4], and the JavaScript-based polyfill, PolyCrypt[5]. We extended NfWebCrypto and PolyCrypt by adding the support to the ECC primitives that our application requires (generation of ECDSA keys, creation and verification of ECDSA signatures). The implementation of the protocols which are not available in the WebCrypto API, namely the modified FHMQV-C and the SMP have been implemented in pure JavaScript. We note that, employing this approach, the bulk of the computationally demanding primitives is delegated to the WebCrypto API. All the execution times were measured using the High Resolution Time API[6], which provides sub-millisecond resolution and highly accurate measurements. The tests were executed using a nightly build of Chromium (35.0.1868.0 r254428) running on a Linux x86-64 3.12.9 platform equipped with a Intel Core i5-2520M processor. The NfWebCrypto plugin was compiled using GCC 4.8.2 with OpenSSL 1.0.1g. Each test has been run 100 times to capture the variance of execution timings.

A. API Performance Microbenchmarks

As a first step in the evaluation, we performed API microbenchmarks, to determine the computational overhead introduced by encryption and signature actions for different message sizes. The results of the symmetric encryption benchmark are shown (in logarithmic scale) in Figure 5 for the GCM mode of operation. The native Chromium implementation is able to encrypt hundreds of megabytes per second, which is way more than enough for our purposes. NfWebCrypto is about one order of magnitude slower, but from 10 to 100 time faster than PolyCrypt.

The reason behind the significant performance drop of NfWebCrypto with respect to the native implementation relies in the fact that the communication between the browser and the plugin is performed through JSON messages and that their entire content is encoded in the Base64-encoded. This provides a significant overhead, although not as high as employing the pure JavaScript implementation. However, considering the typical size of an encrypted payload in our use cases, even the JavaScript polyfill yields a sufficient throughput.

Concerning asymmetric primitives, we ran micro-benchmarks on the generation and verification of digital signatures with ECDSA on the P-256 curve. We note that the native WebCrypto API implementation in Chromium is still lacking the support for ECDSA, and elliptic curve-based cryptosystems in general, which is scheduled for implementation in the near future. The results of the benchmark are shown in Figure 6, and show a speedup for NfWebCrypto against the pure JavaScript implementation similar to the one seen with symmetric encryption. We thus expect the native implementation of the signature and verification algorithms to achieve speedups coherent with those achieved for symmetric encryption. We note that signing messages up to 100kiB requires less than 20ms, an acceptable overhead, given the network latency bound nature of our application.

B. Application overhead measures

Willing to measure the effective overhead introduced by the cryptoschemes on the user interaction with the OSN, we measured the time needed to perform typical interactions with both the actual WebCrypto API and a stub implementation of the same API which does not perform any cryptographic operations. This strategy was chosen due to the asynchronous nature of the API, as a lot of operation can be launched at the same time and the results are collected with callbacks. This in turn results in an effective parallelization in the execution of some calls, which is now possible even on mobile devices thanks to them sporting more than a single computing core. The net effect is that the perceived overhead may be different from the sum of all the time needed to compute cryptographic primitives. In the following experiments, we used the NfWebCrypto implementation of the WebCrypto API as the native implementation in Chromium lacks elliptic curve-based cryptographic primitives. The data storage server is implemented by the event-driven HTTP server platform Node.js[7] (v0.10.26), with a MySQL (v5.5.35) backend for data storage.

We simulated 3 different types of network through the Linux `netem` network simulation tools: (i) a direct connection through the machine local loopback interface to provide an ideal case, (ii) a network with a RTT of 40ms and a bandwidth capped to 10Mbps simulating a home DSL connection, (iii) a network with a RTT of 100ms and a maximum bandwidth of 1Mbps mimicking a mobile data connection. The entire database was placed in RAM to minimize as much as possible the server's delays, thus providing an effective worst case estimation for the overhead imposed by the cryptographic primitives. Each test was run 100 times to compute the variance in the timings.

Users exchange contents between each other through sending and receiving interactive messages. Thus we measured the perceived overhead in sending 10 messages, each one 10KiB large. The results are shown in Figure 7, reporting a 50ms overhead due to the encryption and signing of the messages. The overhead introduced is slightly below 20% considering an ideal network and goes down considering more realistic network simulations. We deem such an overhead tolerable for common user interaction, and note that the overhead to receive the message is substantially the same, in fact signature verification and block cipher decryption take about the same amount of time as signature computation and block cipher encryption.

[4]https://github.com/Netflix/NfWebCrypto (last accessed: May 2014)

[5]http://polycrypt.net/ (last accessed: May 2014)

[6]Jatinder Mann, "High Resolution Time", http://www.w3.org/TR/2012/REC-hr-time-20121217/ (last accessed: May 2014)

[7]http://nodejs.org/ (last accessed: May 2014)

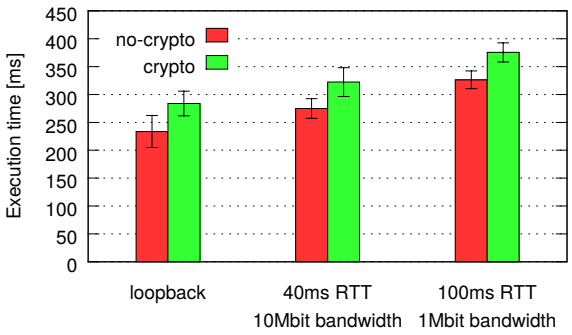

Figure 7. Execution times for sending messages

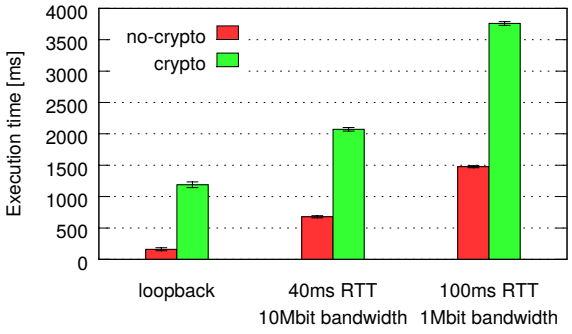

Figure 8. Execution time required to complete a friendship establishment protocol instance

The friendship establishment protocol was the object of our second experiment as it involves the exchange of a sequence of messages with a well-determined order as described in Section III. We note that the non-secure reference implementation of the friendship establishment, which is used as a baseline for the comparison, simply settles the friendship exchanging two messages, while our protocol employs 5 messages. Figure 8 reports the results of the experiment, showing that, despite a significant overhead, the whole friendship establishment protocol runs in less than 4 seconds even on the most constrained network settings. We deem this acceptable for everyday use, especially since direct user interaction (for both formulating the SMP question and answering) is required during the protocol execution, thus lowering the perceived longest latency in the action. Moreover, the friendship establishment is performed significantly less often than simple communication, thus further suggesting that a higher overhead may be tolerable.

Finally, we want to remark that all the timings obtained in the application overhead measurements were obtained with the NfWebCrypto plugin implementation of the WebCrypto API: we thus expect such figures to improve as soon as native implementations are available in web browsers.

V. Related Work

A significant amount of work in open literature has been directed to providing security and privacy in OSNs. We now provide a brief survey of the existing approaches, highlight their focus and features, and compare them with the ones offered by Snake.

Vis-a-Vis [21] and *Confidant* [22] tackle the issue of providing data security and privacy making the user own his data storage, using an Amazon EC2 instance where the data is stored. Both approaches do not rely on any cryptographic technique claiming security through data isolation. We note that, besides the lack of protection from the data storage provider, for most of the typical users of OSN such a solution is problematic as they lack the technical knowledge to maintain the required infrastructure.

NOYB [23] aims at providing privacy through scattering the data of a user profile across other ones, among the users of NOYB. NOYB is implemented as a browser plugin scrambling Facebook profile data: the data is not encrypted, and the privacy guarantees simply rely on the server's inability to match the correct data with the inspected profile. We note that this approach may lead to misinterpretation of the scrambled data as true from users which are not, using NOYB, resulting in undesirable side effects.

FlyByNight [24] is a Facebook overlay, encrypting all the profile data, implemented as a Facebook application. Consequentially, FlyByNight employs the underlying OSN as both a provider for secure and untampered code, and as a certification authority for public key authentication. The authors maintain that it is possible to counteract eventual malicious actions by the OSN provider resorting to legal action, as there is no technical mean to do so.

FaceCloak [25] stores the encrypted profile data of a Facebook profile on a separate data storage provider, while feeding Facebook with fake profile information. The aim of FaceCloak is to avoid the detection of the use of data encryption techniques by the OSN owner, while exploiting it to provide the social graph. FaceCloak assumes a direct, off-band key distribution, and suffers from the same fake information misinterpretation issue as NOYB.

Scramble [26] allows the encryption of single web-page elements, and is implemented as a browser plugin, calling system resident binaries. The key management is based on a leap-of-faith authentication, which the authors adopted for the sake of usability. We note, however, that besides the possible man-in-the-middle attacks allowed by the lack of key authentication, the direct action required by a user to authenticate the contents, as opposed to an encrypted-by-default policy may represent a usability issue.

Hummingbird [27] is a Firefox extension which manages the encrypted interaction with Twitter, providing the user with the ability of both posting tweets and subscribing to hashtags without the server learning her interests or the posted data. Hummingbird relies on an honest-but-curious attacker model for the OSN provider, and thus does not employ authentication techniques for the user public keys.

Snake improves on the schemes such as [21], [22] and [23] enforcing data confidentiality against the storage provider through symmetric encryption, and over [24]–[26] and [27] providing public key authentication which does not rely on any trusted third party, exploiting implicitly shared secrets among acquainted users [14].

Persona [28] tackles the privacy and confidentiality of user data on an OSN employing Attribute Based Encryption to enforce minimum information disclosure policies among them. Persona is implemented as a Firefox extension targeting Facebook, as the authors state that a gradual migration starting from the public data of a current OSN is the most viable approach to foster adoption.

DECENT [29] improves over Persona in terms of efficiency in key revocation strategies, and proposes a key authentication mechanism based on self signature and redundant key fetching, relying on the underlying distributed storage. The approaches employed in [28] and [29] tackle issues orthogonal to the ones tackled by Snake. An interesting direction for future research their integration into Snake.

Frientegrity [30] aims at providing a centralized storage OSN, where the OSN provider is prevented from isolating some clients through enforcing is fork*-consistency. Their approach is based on a standalone implementation of the system in Java, and employs RPC as a communication strategy. The authors note that interesting extensions to [30] would be the efficient handling of multi-administrator groups and friend recommendation. Due to Frientegrity's design, the multi-

Table I
SUMMARY OF PRIVACY AWARE OSN SOLUTIONS

Proposal	Key Authentication	In-band Key Distribution	Data Conf. vs Server	Pseudonymous social graph	Not relying on other OSNs	Standalone Multiplatform
Vis-a-Vis [21]	✘	✘	✘	✘	✓	✓
Confidant [22]	✘	✘	✘	✘	✓	✓
NOYB [23]	✘	✘	✓/✘	✘	✘	✘
FlyByNight [24]	✘	✓	✘	✘	✘	✘
FaceCloak [25]	✘	✘	✓	✘	✘	✘
Scramble [26]	✘	✓	✓/✘	✓/✘	✓/✘	✘
Hummingbird [27]	✘	✓	✓	✓	✘	✘
Persona [28]	✓	✓	✓	✓	✓/✘	✘
DECENT [29]	✓	✓	✓	✓	✓/✘	✘
Frientegrity [30]	✓	✓	✓	✓	✓	✓
Snake [Our]	✓	✓	✓	✓	✓	✓

administrator group management requires an amount of traffic which grows proportionally with their number, and suffers from a complex friend recommendation scheme. Snake on the other hand provides the ground for a multi-administrator group management, with a log-cost key derivation strategy (e.g., [13]), and allows a recommendation mechanism examining the friends list of the user's friends.

VI. CONCLUDING REMARKS

In this work we presented Snake, an end-to-end encrypted social network, providing secure user interactions with a single sign-on key management, a sound in-band key authentication protocol relying on the real world social network to establish trust, and minimal information disclosure from data at rest.

We proved the feasibility of shifting all the cryptographic primitive computations on the client, together with most of the OSN logic, through a practical implementation of Snake as a single page HTML5-JavaScript application.

The resulting application is easily portable to several different platforms, introduces a negligible overhead in terms of time and offers a user experience similar to classical OSNs such as Facebook and Google+.

REFERENCES

[1] N. Ellison and D. Boyd, "Sociality through Social Network Sites," in *The Oxford Handbook of Internet Studies*, W. H. Dutton, Ed. Oxford University Press, 2013.

[2] B. Krishnamurthy and C. E. Wills, "On the Leakage of Personally Identifiable Information via Online Social Networks," *Computer Comm. Review*, vol. 40, no. 1, pp. 112–117, 2010.

[3] C. Tucker, "Social Networks, Personalized Advertising, and Perceptions of Privacy Control," Technical Report of the MIT Sloan School of Management, Tech. Rep., 2013. [Online]. Available: http://www.twcresearchprogram.com/pdf/TWC_Tucker_v3a.pdf

[4] A. Whitten and J. D. Tygar, "Why Johnny Can'T Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, ser. SSYM'99. Berkeley, CA, USA: USENIX Association, 1999, pp. 14–14.

[5] M. Beretta and A. Di Federico, "Snake: the privacy-aware communication platform," https://snake.li/, 2014.

[6] M. Watson and R. Sleevi (W3C Editors), "Web Cryptography API - Working Draft 25 June 2013," Tech. Rep., 2013. [Online]. Available: http://www.w3.org/TR/WebCryptoAPI/

[7] J. Bowser, M. Brooks, R. Ellis, D. Johnson, A. Kadri, B. Leroux, J. MacFadyen, F. Maj, E. Oesterle, B. Whitten, H. Wong, S. Abdullah, and A. Systems, "PhoneGap," http://phonegap.com/.

[8] C. Evans, C. Palmer, and R. Sleevi, "Public Key Pinning Extension for HTTP," Tech. Rep., 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-websec-key-pinning-11

[9] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, "Efficient and Private Access to Outsourced Data," in *ICDCS*. IEEE Computer Society, 2011, pp. 710–719.

[10] ——, "Distributed shuffling for preserving access confidentiality," in *ESORICS*, ser. Lecture Notes in Computer Science, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134. Springer, 2013, pp. 628–645.

[11] ——, "Supporting concurrency and multiple indexes in private access to outsourced data," *Journal of Computer Security*, vol. 21, no. 3, pp. 425–461, 2013.

[12] A. Whitten and J. D. Tygar, "Why Johnny Can'T Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, ser. SSYM'99. Berkeley, CA, USA: USENIX Association, 1999, pp. 14–14.

[13] H. Lu, "A Novel High-Order Tree for Secure Multicast Key Management," *IEEE Trans. Computers*, vol. 54, no. 2, pp. 214–224, 2005.

[14] R. Baden, N. Spring, and B. Bhattacharjee, "Identifying Close Friends on the Internet," in *HotNets*, L. Subramanian, W. E. Leland, and R. Mahajan, Eds. ACM SIGCOMM, 2009.

[15] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification," Tech. Rep., 2000. [Online]. Available: http://tools.ietf.org/html/rfc2898

[16] F. Galiegue, K. Zyp, and G. Court, "JSON Schema: Core Definitions and Terminology," http://tools.ietf.org/html/draft-zyp-json-schema-04, 2013.

[17] A. P. Sarr, P. Elbaz-Vincent, and J.-C. Bajard, "A Secure and Efficient Authenticated Diffie-Hellman Protocol," in *EuroPKI*, ser. LNCS, F. Martinelli and B. Preneel, Eds., vol. 6391. Springer, 2009, pp. 83–98.

[18] F. Boudot, B. Schoenmakers, and J. Traoré, "A Fair and Efficient Solution to the Socialist Millionaires' Problem," *Discrete Applied Mathematics*, vol. 111, no. 1-2, pp. 23–36, 2001.

[19] C. Alexander and I. Goldberg, "Improved User Authentication in Off-the-Record Messaging," in *WPES*, P. Ning, Ed. ACM, 2007, pp. 41–47.

[20] The Free Software Foundation, *The GNU Privacy Handbook*, 1999. [Online]. Available: https://www.gnupg.org/gph/en/manual.html

[21] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. A. Li, D. Liu, and A. Varshavsky, "Vis-à-Vis: Privacy-preserving Online Social Networking via Virtual Individual Servers," in *COMSNETS*, D. B. Johnson and A. Kumar, Eds. IEEE, 2011, pp. 1–10.

[22] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox, "Confidant: Protecting OSN Data without Locking It Up," in *Middleware*, ser. LNCS, F. Kon, Ed., vol. 7049. Springer, 2011, pp. 61–80.

[23] S. Guha, K. Tang, and P. Francis, "NOYB: Privacy in Online Social Networks," in *Proceedings of the First Workshop on Online Social Networks*, ser. WOSN '08. ACM, 2008, pp. 49–54.

[24] M. M. Lucas and N. Borisov, "flyByNight: Mitigating the Privacy Risks of Social Networking," in *SOUPS*, ser. ACM International Conference Proceeding Series, L. F. Cranor, Ed. ACM, 2009.

[25] W. Luo, Q. Xie, and U. Hengartner, "FaceCloak: An Architecture for User Privacy on Social Networking Sites," in *CSE (3)*. IEEE Computer Society, 2009, pp. 26–33.

[26] F. Beato, M. Kohlweiss, and K. Wouters, "Scramble! Your Social Network Data," in *PETS*, ser. LNCS, S. Fischer-Hübner and N. Hopper, Eds., vol. 6794. Springer, 2011, pp. 211–225.

[27] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams, "Hummingbird: Privacy at the Time of Twitter," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012, pp. 285–299.

[28] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: an Online Social Network with User-Defined Privacy," in *SIGCOMM*, P. Rodriguez, E. W. Biersack, K. Papagiannaki, and L. Rizzo, Eds. ACM, 2009, pp. 135–146.

[29] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia, "DECENT: A Decentralized Architecture for Enforcing Privacy in Online Social Networks," in *PerCom Workshops*. IEEE, 2012, pp. 326–332.

[30] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten, "Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider," in *Presented as part of the 21st USENIX Security Symposium*. Berkeley, CA: USENIX, 2012, pp. 647–662.