Keeping Information Safe from Social Networking Apps

Bimal Viswanath
MPI-SWS
Germany
bviswana@mpi-sws.org

Emre Kıcıman
Microsoft Research
Redmond, WA, USA
emrek@microsoft.com

Stefan Saroiu
Microsoft Research
Redmond, WA, USA
ssaroiu@microsoft.com

ABSTRACT

The ability of third-party applications to aggregate and repurpose personal data is a fundamental privacy weakness in today's social networking platforms. Prior work has proposed sandboxing in a hosted cloud infrastructure to prevent leakage of user information [22]. In this paper, we extend simple sandboxing to allow sharing of information among friends in a social network, and to help application developers securely aggregate user data according to differential privacy properties. Enabling these two key features requires preventing, among other subtleties, a new "Kevin Bacon" attack aimed at aggregating private data through a social network graph. We describe the significant architectural and security implications for the application framework in the Web (JavaScript) application, backend cloud, and user data handling.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls

Keywords

online social networks, social apps, data privacy, differential privacy, information flow control, Facebook

1. INTRODUCTION

Social network users trust online social network (OSN) sites, such as Facebook, with their most personal data. Many are unaware of the degree to which their privacy also depends on 3rd party applications and their developers. For example, even the simplest third-party applications on Facebook commonly require access to information such as message postings, interests, photo albums, and birthdays. Today's architectures not only allow this information to be replicated on third-party application servers, but some actually require it. These servers are outside of the OSN's control, and many are hosted in countries that lack privacy laws, exposing private information to misuse. To date, some applications have unintentionally leaked personal information to advertisers and exposed private profiles publicly [6, 7]. Other applications have purposefully sold private user data to marketers and advertisers [24, 29].

The issue is not that applications have access to private data, i.e., users expect that the application will access private data (only) as necessary to do its job. The fundamental problem is that applications can hoard, transfer to others and otherwise abuse this access in violation of user expectations.

One approach to limiting privacy leakage through 3rd party applications is to perform application sandboxing on both the server and client side. An existing framework using this technique is called xBook [22]. This framework assumes a trusted application-hosting platform that includes trusted storage components. The xBook platform confines untrusted application code on both the server and client side and explicitly mediates information flow within and from the app according to predefined security policies.

Inspired by the xBook framework, we began to develop a system that sandboxes social networking applications while it mediates their information flow to limit privacy leaks. In our journey to realize this system, we encountered two obstacles that led to serious modifications to our original design. This paper describes these varied challenges as they are likely common to other sandboxing mechanisms for social applications. While presenting the techniques we used to overcome the obstacles, we believe there is much room left to more deeply examine these problems and their solutions.

The first obstacle stems from the social network nature of these applications. Each instance of an application running on behalf of a user needs to communicate (i.e., share data) with another instance running on behalf of a friend. Any sandboxing architecture of social networking applications must therefore allow such basic channels of communication. Unfortunately, such systems are vulnerable to a new form of attack we call *the Kevin Bacon attack.* Developers can now stealthily siphon personal data through the social network itself. To stop such an attack, we need more sophisticated information flow control mechanisms on both server and client sides. This paper presents one such mechanism.

The second problem stems from the need of app developers to debug and extract large-scale analytics from their applications. How can the system distinguish personal data from debugging data to prevent information leaks of the former while allowing the latter? Our current design proposes leveraging differential privacy [4] to limit information leakage for such information flow. Our design also handles additional challenges: providing differential privacy to a dynamic stream of data (as is the case with debugging information), allowing prolonged queries on such dynamic data, and avoiding vulnerabilities due to dependences among multiple pieces of social network data.

We further discuss a simple prototype implementation of our new framework and the experience of building applications on our architecture. Finally to understand whether existing Facebook apps could fit into our framework, we analyze a sample of such apps as well as a benchmark quiz application that we built.

2. PRIVACY PROBLEMS AND EXISTING SOLUTIONS

This section presents background on how third-party social networking applications are built on Facebook, presents the threat model, and describes existing solutions.

There are several kinds of third-party Facebook applications: Web applications that run within Facebook's own site (embedded within an iframe HTML element), independent Web applications, and mobile and desktop applications that use non-Web front-ends. All of these applications access user data via Facebook's Web service APIs. The information that the applications can request is limited by user permissions, but, once retrieved, applications have no technical restrictions on how they store or use data.

Threat model. Our primary motivation is to prevent application developers from purposefully or accidentally leaking information about individual users. Our main focus is on preventing large-scale leaks that could harm hundreds, thousands or millions of users. Our system does not protect against targeted attacks against individuals, such as social engineering attacks.

In our threat model, we treat application developers as our sole adversary. In addition to running application code on our hosted platform, developers are assumed to have additional compute resources not controlled by our system, and normal user access to the OSN (*e.g.*, they can create new user identities on the OSN). We assume that a users' friends, the OSN platform, and the hosting platform are **not** malicious. Given this threat model, we cannot trust compute resources not hosted on our platform, nor can we trust strangers in the OSN; therefore, we must prevent personal information from reaching them.

Existing systems already try to limit privacy leaks of social networking applications by decentralizing the social network and providing users more control over their data. Examples include decentralized social services on personal mobile devices [2] or personal servers [11]. Instead, our approach is based on a centralized OSN model (similar to Facebook's current model) and our initial strawman system design was inspired by a system called xBook [22]. In xBook, social applications are hosted on trusted centralized xBook servers which provides a privacy-preserving platform. xBook primarily relies on confining (or sandboxing) application execution and mediating information flow between sandboxes. Our initial strawman system design was inspired by the xBook approach.

To protect user data, our strawman system lets information flow from developers to users without restriction, but any information flow from users to developers is restricted. A user can send information to and receive information from friends (*i.e.*, a set of other users connected in a social network). These information flow control restrictions are enforced by requiring 3rd party applications to be hosted on a trusted infrastructure with the following components:

Sandboxed cloud compute and storage. The developer principal executes in a sandbox that can communicate with the open Internet and send information to users by writing to a per-application global database; it cannot directly receive information from users. The user principal executes in a sandbox that can read and write to a per-user database, which can be further shared with friends. The user principal also has read access to the per-application global database to fetch global, non-sensitive application data. The user sandbox responds to an individual user's HTTP requests.

Sandboxed Web layer. Often, significant parts of social networking application functionality execute as a Web application inside the client web browser. Like the design used in the cloud, this architecture relies on sandboxing on the client side or on the Web layer to enforce information flow control. The goal here is to prevent communication with untrusted servers. Several tools, such as Caja [9] and WebSandbox [16], let you setup sandboxes using HTML iframe elements to limit and control the functionality of JavaScript programs. The Web application interacts with our trusted hosting platform through the user sandbox in the cloud.

The following two sections define two key limitations of approach and describe how we overcame them.

3. PROBLEM #1: THE KEVIN BACON ATTACK

By analyzing the kind of sharing required by Facebook applications today, we observe that a majority of them require data sharing with friends of a user see Section 5.1). This opens the possibility for a new kind of attack that can circumvent restrictions on sharing with strangers and ultimately leak information to an application developer with a social network account. We call this new attack the *Kevin Bacon attack*.

3.1 Method of Attack

In the Kevin Bacon attack, an application silently shares personal information with a users' friends. When a friend runs the same application, that application silently receives this information and forwards it to a new set of friends. This process repeats, spreading the original personal information throughout the social network until it reaches the OSN account of the application developer. Because this sharing follows the social graph, it does not strictly violate our restriction of sharing information only with friends. But, because this sharing is done without the awareness of users, and given the relatively small diameter of social graphs [17], large amounts of personal data can be collected by the application developer without user knowledge or permission.

To protect against this *Kevin Bacon* attack, we refine our sharing restrictions to state that users cannot forward information shared by one friend to other friends. We discuss the architectural implications of this restriction in Section 3.2.

3.2 Our current solution

To prevent the Kevin Bacon attack, we need to restrict information flow to within the users' 1-hop social network. To

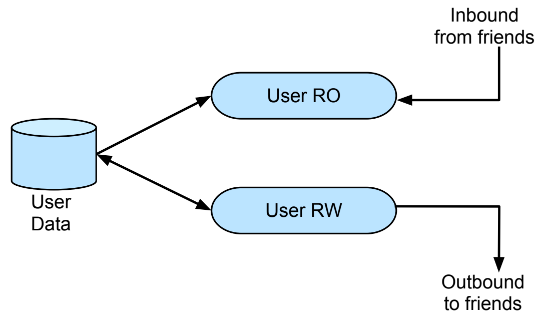

Figure 1: Information flow control at the server is enforced through two user sandboxes to prevent the Kevin Bacon attack.

enforce this, we need to make architectural changes to both the server and client sides. Instead of executing the user principal in a single sandbox, we split it into two sandboxes – *user read-only* and *user read-write*, as shown in Figure 1. To restrict sharing to 1-hop friends, the user read-only sandbox can read information shared from friends, but it cannot write to the user database or share to other friends. Further, the user read-write sandbox can write to the user database and share with friends, but it cannot read data shared from friends.

There are many possible techniques for information flow control, including runtime systems, such as [10,27] and static compiler techniques [23,28]. However, information flow control technologies are an active area of research, and no state-of-the-art technology is yet ideal for all situations. In the cloud, we chose coarse-grained sandboxing to reduce the programming and runtime burden of fine-grained checks on information flows.

On the client side, we need to set up a more sophisticated system that allows communication back to our servers dependent on the provenance of the data being communicated. That is, whether a call back to a server to store or share data is allowed depends on whether the data originates from other friends or if it were originally the user's own data.

Our server-side implementation simply uses two separate sandboxes to achieve similar functionality; coarse-grained sandboxing on the client-side, however, is dependent on HTML iframe elements to enforce sandbox boundaries. For a web application to mix the display of data from different sources in the same HTML interface would have required a large number of iframe elements with their individual sandboxes throughout the HTML DOM tree, thus limiting the structure of the UI. Therefore, we elect to use static checking of information flow control in the client-side program. (Note that the server-side code is not restricted by such user interface considerations).

JavaScript as a dynamic language is known to be difficult to analyze [1]. Instead of verifying JavaScript directly, our system requires developers to write their client-side code in the Fine language [23] because Fine can express and verify information flow control policies. Once verified, the Fine language can be compiled into JavaScript and executed in browsers. The verifiable language can preserve our desired properties in the resulting JavaScript code.

Fine is a functional language and a variant of F# that supports static verification of various properties, including information flow control. Described in [23], Fine uses a type system that combines dependent and refinement types to express authorization policies and affine types to model changes in policy state. The Fine language has already been used to build different Web applications [8].

In addition to requiring that client-side code be verified to respect an information-flow control policy, our system must also ensure that other resources sent to the client, such as HTML and images do not enable an information leak. For this reason, HTML and other content are served statically as a global resource. All user-specific or social data must be retrieved from the server-side sandboxes. This communication between client-side code and server-side code takes place over two separate channels. One channel connects the client with the user read-only sandbox, and the other connects the client with the user read-write sandbox. The information flow control policies ensure that data from the read-only sandbox (*i.e.*, data from friends) is not sent to the user read-write sandbox where it could be shared to other friends.

4. PROBLEM #2: GRANTING DEVELOPERS AGGREGATE ACCESS

Developers require access to aggregate app data for both app monitoring or debugging and for creating application functionality. To monitor applications, developers often gather aggregate statistics. Current OSNs (e.g., Facebook) offer an application usage dashboard (with statistics such as the number of new app installs, active users, etc...) and an application performance dashboard (with statistics such as the number of daily API calls, average API call time etc...). Aggregate data is also needed to build new application functionality such as recommendation features. For example, a movie recommendation application may want to provide average movie rating scores to each user query for a movie. This computation requires computing over all users' records.

This aggregate data information flow from users to developers should be privacy preserving. To achieve this, our architecture includes an additional component – called the *secure data aggregator* – that uses differential privacy techniques to give developers access to aggregate data. Developers can issue queries to the secure data aggregator to fetch aggregate user data statistics.

4.1 Our current solution

4.1.1 Primer on differential privacy

While the math of differential privacy is complex [4, 5, 15], the concept is straightforward. Differential privacy provides an intuitive formalization of privacy – it measures the loss of information when answering a given query over a given dataset with a given privacy budget. In particular, it measures the degree by which an attacker is more likely to guess any row in the dataset knowing the result of the query. Differential privacy injects "noise" in the answer. The use of noise is a control knob: if set to "high", the query answer has little information loss, but it is also less precise (i.e., more inaccurate). When noise is set to "low", a query answer is more accurate but has more information loss; if the information loss is higher than the privacy budget allocated to the dataset, we abort the query.

Differential privacy queries are accompanied by a privacy parameter. This parameter, ϵ, provides a tradeoff between the accuracy of the output and the likelihood of leaking information about an individual record. For example, a low value of ϵ shows that the issuer does not want to consume a large portion of the privacy budget on answering this query, i.e., the query answer will have stronger privacy guarantees in exchange for weaker accuracy. In our experiments, we set ϵ to 0.1, 1.0, or 10.0, values used in previous work as well [13].

4.1.2 Querying aggregate user data

We must solve two challenges when using differential privacy to query user social networking data. First, new data is always being generated in the system, and we need to recast information loss in the context of a continuous stream of new data. Second, differential privacy rests on a crucial assumption, namely that the rows in the database are independent (i.e., there is no information that spans multiple rows). The remainder of this section describes how we address these challenges.

In our scenario, new data is generated each time an application is used. Our user sandbox stores users' data as a collection of (key,value) pairs. This means that the underlying set of data records for a given key is not static, as it is in previous systems that use differential privacy for querying data [13,14,20]. To address this issue, we split the input data records for a given user into independent chunks, called *epochs*, and assign each epoch a separate privacy budget predetermined by our framework. The data is split into epochs based on the timestamps associated with each data item. By the parallel composition property of differential privacy [12], querying these disjoint epochs preserves the privacy properties of differential privacy: the overall information loss of all queries is proportional to the maximum loss of a single query, not to the sum of losses for all queries.

Epochs help us address an important problem that faces all systems that use differential privacy: what happens when the privacy budget is exhausted? Because the system operates on a stream of data, we can always issue a query on a new epoch whose privacy budget is still available. Our system could help identify eligible epochs for a given query, i.e., which epochs have a big enough privacy budget to answer the query. Once eligible epochs are identified, the application can then specify the epoch it wants to run the query. Identifying eligible epochs does not consume any privacy budget because no query answers are returned. Also, we could cache previous queries' results (in case the application issues the same query multiple times); caching query answers does not lead to additional information loss.

A query issued by the developer sandbox specifies what key should be selected from each user's data. This query is run in two phases. First, for each user, the system creates one row from all (key,value) pairs selected from that particular user's data. For example, if the key is 'login time', then the values are all the different login times stored in each user's datastore. Our system allows further processing of these values depending on the analyst's needs.

In the second phase, we perform a view transformation of the user records in the database before applying the analyst-specified differential privacy operators. Differential privacy assumes that each data record is independent of the rest of the records [13]. For example, this assumption excludes the possibility of a single user's data appearing multiple times in the queried set of records. Such a scenario could defeat the guarantees of differential privacy because one user's data could occupy the majority of records in a dataset and overcome the effects of noise injected by differential privacy. Instead, our system maps each user's data to a single record using an arbitrary analyst-specified function, removing the possibility of a single user's data dominating the database. We then apply the analyst-specified differential privacy operators [12] across all the transformed user records. This way of handling a query is critical to security because of the assumptions differential privacy makes about the data being queried.

5. PROTOTYPE IMPLEMENTATION

This section describes a prototype implementation of the proposed system and evaluates the privacy-versus-accuracy tradeoff of using differential privacy in the context of social networking applications. Our implementation does not require tight integration with the host OSN (e.g., Facebook). It does require API access to fetch user profile information (e.g., lists of friends) and the ability to validate a user's credentials. These features are available via the APIs of the most popular social networks today.

Information flow control at server and client. In the cloud, our system is implemented as an ASP.NET program running within an IIS HTTP server. To implement server-side sandboxing, we use .NET AppDomains [18], a sub-process isolation mechanism that provides an isolated secure boundary for executing managed code. Other lightweight sandboxing technologies would also have provided acceptable implementations, including pico-processes [3, 19] and lightweight virtual machine technologies such as Denali [25]. Azure cloud storage [26] was used for storage of user and application global data in semi-structured tables. The server-side implementation mediates access to these storage systems, verifying that the various principals have appropriate read and write privileges before granting access.

For static verification of Web application code, we use Fine, a functional language and variant of F#, that has support for static verification of various properties, including information flow control [23]. Once verified, Fine code is translated into JavaScript and sent to the client browser for execution.

Secure data aggregator. Our secure aggregator is built with the PINQ library [12], a language-integrated query API for computing on sensitive data sets. In addition to the core differential privacy functionality provided by PINQ, our secure aggregator adds epoch management and integration with storage of user data.

To further understand if executing differential privacy queries over dynamic data by breaking data into epochs works well in practice, we perform an analysis over real-world social app data. We aim to evaluate the privacy-versus-accuracy tradeoff of this technique.

We use the publicly available dataset of a real Facebook application [21] called "The Streets", a Facebook game in which users can pick virtual fights with each other and ask for the support of their friends. We use the PINQ [12] library to run differential privacy queries on the application data. PINQ uses a Laplace distribution to generate noise.

The dataset consists of all API calls made by the application to the Facebook portal to retrieve users' data. Each call is timestamped, has a call type (there are only four types in the entire dataset), and has the timestamp of the reply data received from Facebook. A total of 83,067 calls were made between November 8th, 2008 and January 1st, 2009.

We present the results for the performance statistic that measures the number of API calls made over time. This statistic is provided by using the noisy count operator in differential privacy, and counting is representative of most aggregate statistics operations.

	Relative error		
Epoch size	**t=30min** 2,962 epochs	**t=1hour** 1,481 epochs	**t=1day** 63 epochs
ϵ=0.1	69%	24%	0.7%
ϵ=1.0	7.3%	2.6%	0.09%
ϵ=10.0	0.6%	0.2%	0.007%

Table 1: Relative error in count queries.

Table 1 presents the results of computing the daily number of API calls using three epoch sizes (1 day, 1 hour, and 30 minutes) for three ϵ values: 0.1, 1.0 and 10.0. We compute the average relative error in each bucket between the differential privacy counters and the true ones. For a strict value of the privacy parameter ($\epsilon = 1.0$), we notice that the error is 7.3% for small epoch sizes (30min) and drops to 0.09% for larger epoch sizes. As the epoch sizes decrease, each epoch contains less data amplifying the effects of noise, and, thus, reducing accuracy. However, for an epoch size of 1 day, the relative error is lower than 0.7% for all values of ϵ; interestingly, the current Facebook app framework updates app statistics mostly at a day's granularity to provide developers insight into the app's health and popularity.

5.1 Do existing apps fit our model?

To determine if we could support existing Facebook apps in our framework, we analyzed 50 Facebook applications (as ranked by `www.appdata.com` based on the number of monthly active users), including the top 25 apps and a sample of 25 random apps from the list of the remaining top 1000 applications. For each application, we examined whether it shares the data of a user with others. While some apps share no data (e.g., horoscope apps or daily quotes), others share it with third-party services (worst in terms of privacy), with people other than just the user's friends, or with the user's friends only (best in terms of privacy). Table 2 shows the results.

During this experiment, we discovered that four applications could not be analyzed. One was not working properly and three were just wrappers to a third-party website and not "true apps". Looking at the results, we find 17% of the apps require no data sharing. However, the majority of applications (52%) require share data with friends only, the best behavior in terms of privacy for data sharing. In our framework, these apps would stop further propagation of a user's data beyond the set of friends.

We found that 24% of apps share data with strangers; inspecting the results a little closer, we discovered that 7% were using the data as input to recommendation and aggregation engines, which is arguably less of a privacy violation. The remaining 17% share the user's data directly with other strangers.

Overall, 76% of the sampled applications can run within our framework. The remaining 24% share data with strangers in a direct manner (e.g., a dating app) or use 3rd party Web services and widgets (e.g., map services), operations that our framework explicitly disallows. The sharing model used by these apps explicitly puts a user's data in the hands of strangers making it impossible to offer any privacy guarantees.

Sharing	**Examples**	**Apps %**
None	Daily quotes and horoscopes	17
w/friends	Games and quizzes sharing scores with friends	52
Indirectly w/strangers	Aggregation and recommendation services such as Flixster	7
Directly w/strangers	Dating applications	17
w/3rd-party services	Apps that embed Google Maps or other Web widgets	7

Table 2: The kind and distribution of data sharing found in Facebook applications.

5.2 Sample application

To demonstrate practicality, we built a benchmark quiz application in our framework that exercises the Web layer, the server layer, and the sharing mechanisms. Our quiz asks users to answer various questions and, once a response is entered, it shows the friends' answers for comparison (Figure 2). Since this app needs data to be shared between friends, we need to coordinate the answers entered by users in their read-write sandbox with the answers entered by their friends received in the user's read-only sandbox. Although Fine, due to its functional language nature, does require developers to use a different programming model than JavaScript, an imperative language, our experience coding in Fine was positive.

Our simple quiz app that coordinates across the user read-only and read-write sandboxes was implemented in 328 lines of C# code on the server side and just 123 lines of Fine code on the client side. For comparison, we re-implemented our quiz app without the sandboxing mechanisms of our framework. We were able to replicate the behavior of the app in 193 lines of C# code on the server side and only 67 lines of JavaScript code on the client side. This experiment shows that the programming cost of adding privacy protection could be significant.

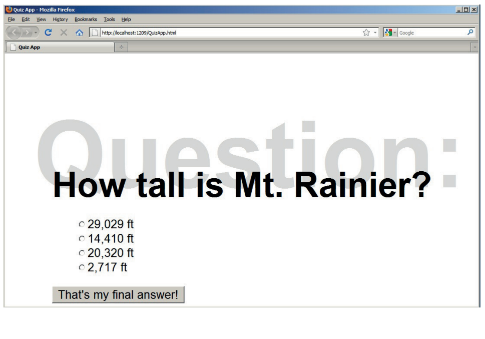
Quiz App - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:1209/QuizApp.html
Quiz App
Question:
How tall is Mt. Rainier?
29,029 ft
14,410 ft
20,320 ft
2,717 ft
That's my final answer!

Figure 2: Screenshot of a quiz application.

6. CONCLUSION

This paper identified two problems with current mechanisms for sandboxing third-party social network applications to stop privacy leakage. We presented our solutions to these problems in an effort to initiate a discussion around these problems and the "right" ways to overcome them.

In this context, we proposed a new architecture that was designed to be resilient to these problems by leveraging varied mechanisms, including functional languages, static verification, and differential privacy. We discussed a prototype implementation and an example application built on our new framework. Our analysis of a sample of Facebook apps revealed that our new architecture can support a majority of existing Facebook apps.

7. REFERENCES

[1] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for Javascript. In *Proceedings of the Programming Languages Design and Implementation (PLDI)*, 2009.

[2] B. Dodson, I. Vo, T. Purtell, A. Cannon, and M. Lam. Musubi: Disintermediated Interactive Social Feeds for Mobile Devices. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2012.

[3] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[4] C. Dwork. Differential Privacy. In *Proceedings of the International Colloquium on Automata, Language and Programming (ICALP)*, 2006.

[5] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2006.

[6] Facebook leaks access tokens to advertisers. `http://tinyurl.com/3z7g3g7`.

[7] Facebook privacy hole due to buggy app. `http://news.cnet.com/8301-10784_3-9977762-7.html`.

[8] Fine Project. `http://research.microsoft.com/en-us/projects/fine/`.

[9] Google Caja. `http://code.google.com/p/google-caja/`.

[10] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007.

[11] D. Liu, A. Shakimov, Ramón Cáceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN Data without Locking it Up. In *Proceedings of the International Middleware Conference (Middleware)*, 2011.

[12] F. McSherry. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

[13] F. McSherry and R. Mahajan. Differentially-private Network Trace Analysis. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[14] F. McSherry and I. Mironov. Differentially Private Recommender Systems: Building Privacy into the Netflix Prize Contenders. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2009.

[15] F. McSherry and K. Talwar. Mechanism Design via Differential Privacy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2007.

[16] Microsoft Web Sandbox. `http://www.websandbox.org/Default.aspx`.

[17] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[18] .NET AppDomain. `http://msdn.microsoft.com/en-us/library/system.appdomain(v=vs.71).aspx`.

[19] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[20] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation(NSDI)*, 2010.

[21] RUBINET Data Sets. `http://www.ece.ucdavis.edu/rubinet/data.html`.

[22] K. Singh, S. Bhola, and W. Lee. xBook: Redesigning Privacy Control in Social Networking Platforms. In *Proceedings of the Conference on USENIX Security Symposium (USENIX Security)*, 2009.

[23] N. Swamy, J. Chen, and R. Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *Proceedings of the European Symposium on Programming (ESOP)*, 2010.

[24] Third-party apps sell private data. `http://tinyurl.com/27fnslp`.

[25] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[26] Windows Azure Storage. `http://www.microsoft.com/windowsazure/storage/`.

[27] N. Zeldovich, S. Boyd-wickizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[28] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 2007.

[29] Zynga Hit With Lawsuit Over Facebook Privacy Breach. `http://tinyurl.com/3qjcbll`.