xBook: Redesigning Privacy Control in Social Networking Platforms

Kapil Singh[*]
School of Computer Science
Georgia Institute of Technology
ksingh@cc.gatech.edu

Sumeer Bhola[*]
Google
sumeer@acm.org

Wenke Lee
School of Computer Science
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract

Social networking websites have recently evolved from being service providers to platforms for running third party applications. Users have typically trusted the social networking sites with personal data, and assume that their privacy preferences are correctly enforced. However, they are now being asked to trust each third-party application they use in a similar manner. This has left the users' private information vulnerable to accidental or malicious leaks by these applications.

In this work, we present a novel framework for building privacy-preserving social networking applications that retains the functionality offered by the current social networks. We use information flow models to control what untrusted applications can do with the information they receive. We show the viability of our design by means of a platform prototype. The usability of the platform is further evaluated by developing sample applications using the platform APIs. We also discuss both security and non-security challenges in designing and implementing such a framework.

1 Introduction

Social networking sites have transformed the way people express themselves on the Internet and have become a door to the social life of many individuals. Users are contributing more and more content to these sites in order to express themselves as part of their profiles and to contribute to their social circles online. While this builds up the online identity for the user, it also leaves the data vulnerable to be misused, as an example, for targeted advertising and sale.

More private data online has lead to growing privacy concerns for the users, and some have faced extreme repercussions for sharing their private information on these networking sites. For example, students have been fined for their online social behavior [29]; a mayor was forced to resign because of a controversial Myspace picture [32]. There are numerous such cases, and these incidents clearly underline the importance of privacy control in social networks.

With the advent of Web 2.0 technologies, web application development has become much more distributed with a growing number of users acting as developers and source of online content. This trend has also influenced social networks that now act as platforms allowing developers to run third-party content on top of their framework. Facebook opened up for third-party application development by releasing its development APIs in May 2007 [22]. Since the release of the Facebook platform, several other sites have joined the trend by supporting Google's OpenSocial [10], a cross-site social network development platform.

These third-party applications further escalate the privacy concerns as user data is shared with these applications. Typically, there is no or minimal control over what user information these applications are allowed to access. In most cases, these applications are hosted on third party servers that are difficult to monitor. As a result, it is not feasible to police the data being leaked from the application *after the data is shared with the application*. There have been several reported cases where users' private information was leaked by the applications, either due to intentional leaks [21] or due to vulnerabilities in the application [26].

Most social networking platforms, such as Facebook, currently provide the applications with full access to user profile information. This permission is granted in Facebook when the user adds the application, which requires the user to make a trust decision. Setting fine-grained access control policies for an application, even if they were supported, would be a complex task. Furthermore, access control policies are not sufficient in enforcing the privacy of an individual: once an application is permitted by a user's access control policy, it has possession of the user's data and can freely leak this information anytime for personal gains. For example, a popular Facebook application, Compare Friends, that promised users' privacy in exchange for opinions on their friends later started selling this information [35].

In this paper, we are concerned with protecting the users' private information from leaks by third-party ap-

[*]Part of the work was done when the first author was an intern and the second author was an employee at IBM Research T.J. Watson.

plications. We present a mechanism that controls not only what the third-party applications can access, but also what these applications can do with the data that they are allowed to access. We propose and implement a new framework called `xBook` that provides a hosting service to the applications and enforces information flow control within the framework. xBook provides three types of enforcement that encapsulate the privacy requirements in a typical social network setting: (1) user-user access control (e.g., access to only friends) for data flowing within one application, (2) information sharing outside xBook with external parties; and (3) protection of the application's proprietary data. While (1) and (2) protects the privacy of a user from information leaks, (3) prevents the application's proprietary data or algorithm from being leaked to the application users.

The third-party applications are redesigned in such a way that they have access to all the data they require (allowing them to perform their functionality) and at the same time, not allowing these applications to pass this data to an external entity unless it is approved by the user. *Our framework enforces that the applications make these communications explicit to the user* so that he is more informed before approving an application.

There are several challenges associated with the design of our xBook framework:

Confinement. The execution of application code needs to be confined. This problem needs to be dealt with independently on the client side within the browser and on the server side in the web server. We use "the web server" as a conceptual entity to represent one or more servers.

Mediation. All communication from and within an application needs to be mediated by the xBook platform for permissible information flow. To this end, we developed a labeling model that enforces user-defined security policies. High-level policies specified by the user are converted to low-level labels enforced by xBook.

Programmability. The programming abstraction to the application writers should be practical and easy to use. xBook provides a set of simple APIs in line with the existing social networking platforms.

Portability. The requirements imposed by xBook on the application design should not break the existing applications. In other words, it should be feasible to port most functionality of typical applications to xBook with little effort.

We show the viability of our framework design by implementing a working prototype of our xBook system and porting some of the popular applications from existing social networks, such as Facebook, on top of the framework. We also demonstrate a practical deployment strategy of our system by porting our framework itself as an application on Facebook. Our system is available online [33]. We evaluate the security of our platform by illustrating some possible application scenarios, and how xBook ensures privacy control in such cases. We also create some synthetic attacks that attempt to exploit the platform to leak information. Our results illustrate that xBook can successfully prevent all such attacks. Our performance results further demonstrate that xBook's privacy control mechanism incurs negligible overhead for typical social networking applications.

The rest of the paper is organized as follows. Section 2 motivates our work by analyzing some privacy issues with the current social networking platforms. We present an overview of our xBook framework in Section 3. Section 4 and 5 discuss the implementation details of xBook's client-side and server-side components, respectively. Our labeling model is described in Section 6. Section 7 presents the evaluation results. We discuss the limitations of our work in Section 8, followed by related work in Section 9. Finally, Section 10 concludes the paper.

2 Background

2.1 Social Networking Platforms

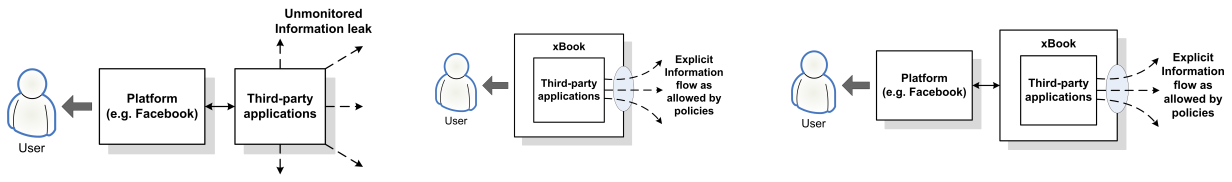
Social networks are the backbone of the online social life of many Internet users. These networks have expanded their development scope by allowing third-party developers to write their own applications, which in turn can be accessed and executed via the social network. An application is an entity that provides some value-added service to the user, and it requires user's profile data to perform its functionality. For example, a simple horoscope application generates daily horoscope based on user's birth information.

Facebook is one popular network that has pioneered the concept of the social network as a platform. The applications bring value both to the platform and its users in providing new features. Applications are deployed on their own servers and Facebook only acts as a proxy for integrating the applications' output to its own pages. The growing popularity of applications on Facebook has enticed other networks, such as Google's Orkut, to start supporting applications. The Orkut platform model is based on the OpenSocial framework [18]. OpenSocial provides a set of APIs for its partner sites (which it refers to as "containers") to implement. An application that is built for one container should be able to run with few modifications on other partner sites. The APIs allow third parties to have access to the social graph and personal user data.

For the rest of the paper, we use the Facebook case as an example; similar concepts apply to other social networking platforms.

2.2 Privacy Issues with Current Designs

Facebook supports customized policies for user-user access control, but currently provides no control on what



(a) (b) (c)

Figure 1: Application architecture for: (a) current platforms. (b) xBook platform. (c) xBook on Facebook.

user profile data can be accessed by third party applications. Applications run on their own servers that have no control administered by Facebook (Figure 1(a)). Applications need data to perform their functionality; they can request user data from the social platform and store it at their own servers. Facebook discourages storing user data on the application's own servers by barring it in their license agreement [5], but there is no way of enforcing it in Facebook's current architecture.

Application developers have access to a user's data even when they are not friends with the user. Unlike a regular friend relationship, this relationship is neither symmetric nor transparent: the application developer has access to the user's information, but the user does not necessarily know who the application developer is.

Before adding an application, the users are required to agree to a service agreement that allows the application to have access to their profile data. This general agreement is presented for every application, and no other specific information is provided about the application. Since a majority of the applications are known not to exploit users' personal data, the users tend to add any application, effectively defeating the purpose behind the service agreement. Additionally, second-degree permissions that allow applications to have access to the profiles of the users' friends add another layer of complexity.

There have been several reported incidents where users' information was leaked due to a vulnerability in the application [26]. The platform is trusting all third party developers, but the trust is misplaced since there is no restriction on who is allowed to develop an application. One of the most popular Facebook applications, TopFriends, had a vulnerability that allowed any user of TopFriends to see the profile of another user, even if they are not friends with each other [26]. Private information of some high profile users was leaked. Facebook's response to this controversy was that they *expect* third party applications to follow their policies, which is not acceptable considering that there is no effective way to police the application developers.

User data has a lot of commercial value to marketing companies, competing networking sites, and identity thieves. Therefore, it is not surprising that many applications have been observed to intentionally leak user data to external parties for profit [21]. Other surveys have also discovered similar violations based on an application's externally-visible behavior [19]. The situation could be even worse as it is not feasible to determine how many other applications violate the user's privacy with internal data collection.

Social networking sites have a responsibility to protect user data that has been entrusted to them. The current approach is to legally bind the third parties using a Terms of Service (TOS) agreement [4]. However, it is not possible to monitor the path of information once the information has been released to these parties. Therefore, social networks can not rely on untrusted third parties following their TOS agreements to protect user privacy. Instead, privacy policies should be enforced by the platform and applied to all data that has been entrusted to the social networking site. Our platform design, `xBook`, is one step forward in this direction.

Felt et al. [19] have proposed a solution to proxy the user information in the form of tags to the third-party applications. These applications do not have access to user data and instead use pre-defined tags to format their output being displayed to the user. Their solution limits the capability of some important and popular applications, such as the horoscope application, that perform processing on user data beyond just displaying it. Our work enforces no such restriction on the application behavior.

3 `xBook` Overview

xBook is an architectural framework for building social networks that prevents untrusted third-party applications from leaking users' private information. The applications are hosted on xBook's trusted platform (Figure 1(b)), and xBook provides complete mediation for all communication to and from these applications.

In a social network setting, an application might communicate with entities outside the xBook system, called *external entities*, to perform specific tasks. For example, the horoscope application may communicate with `www.tarot.com` to receive horoscopes for every sunsign. The application also encapsulates its own data or algorithm that needs to be protected from untrusted users.

In the xBook framework, applications are designed as a set of *components*; a component being the smallest granularity of application code monitored by xBook. A component is chosen based on what information the component has access to and what external entity it is allowed to communicate with. In the horoscope application, one component communicates with `www.tarot.com` and has no access to user data. Another component has access to user's birthday, but does not communicate with any external entity.

From an end user's perspective, the applications are monolithic as the user does not know about the components. At the time of adding a particular application, the user is presented with a manifest that states what user profile data is needed by the application and which external entity will it be sharing this data with. For example, horoscope's manifest would specify that it does not share any information with any external entity. Note that the horoscope application does not need to reveal that it communicates with `www.tarot.com` as no user information is being sent to `www.tarot.com`. The user can now make a more informed decision before adding the application. Admittedly, the user will need to make a trust decision with respect to the parties with which the application shares user data, but these external parties can be expected to be larger and better branded entities providing internet services, such as Google for ads, Yahoo for maps, etc.

Figure 2 shows a typical life cycle of an application. The developer of an application decides on the structure of the components for that application and during the application's deployment on xBook, he specifies the information required by each component and the external entity a particular component needs to communicate with. xBook uses this information to generate the manifest for the application. As shown in the figure, a manifest is basically a set that specifies all of the application's external communications (irrespective of the components) along with the user's profile data that is shared for each communication. Additionally, the xBook platform ensures that all of the application's components comply with the user's privacy policy and the manifest approved by the user. We discuss this further using the case study of an example application in Section 6.3.

The division of an application into multiple components allows the application writer to develop different functionality within an application that rely on different pieces of the user profile. For example, let us consider an application that requires a user's information to generate a customized profile for the user. It also requires his address information to be passed to Google to generate a map showing the address. In the application design of current social networks, the application would be able to pass all information about the user to Google. In the xBook framework, the application would be split into two components: the first component presents the customized profile of the user, has full access to the user's data and is not allowed to communicate with Google; the second component encapsulates the user's address (with no mapping to the user's profile) that is passed to Google to generate the map. We discuss some example applications in Section 7.1.

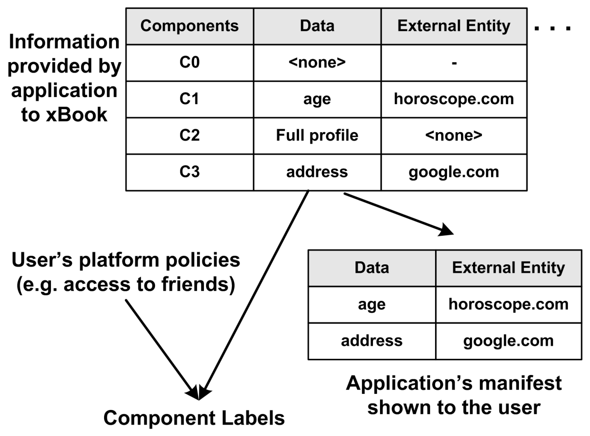

Information provided by application to xBook
Components | Data | External Entity
C0 | <none> | -
C1 | age | horoscope.com
C2 | Full profile | <none>
C3 | address | google.com
. . .
User's platform policies (e.g. access to friends)
Data | External Entity
age | horoscope.com
address | google.com
Application's manifest shown to the user
Component Labels

Figure 2: Typical life cycle of an application in xBook.

Figure 3 shows a high-level design of our xBook framework. There are two parts of the xBook platform, one that runs on the server-side and another that executes on the client-side in the user's browser (Figure 3). The application components, in turn, are also split into client-side and server-side components. The components are written in a safe subset of javascript, called ADsafe [1], which facilitates confinement of these components in our xBook implementation. Any communication to and from the components occurs by using xBook APIs, thereby allowing all such communication to be mediated by xBook. Each component is associated with a privilege level or label that is derived from the application's manifest. The platform mediates the information flow between the components based on these labels (Section 6).

Both client-side and server-side components communicate with server-side storage to retrieve data. There are two types of storage in xBook system: one for storing xBook data that includes user profiles, and second for the data stored by the application. While the structure of xBook data is known, the semantic of the application data is internal to the application and hence unknown to the platform. All data fields are labeled to control access by application components. These labels are assigned based on high-level user-defined policies, such as a policy allowing access to only the user's friends, and the manifest approved by the user (Figure 2).

To store application data with unknown structure and semantics, xBook contains a group of storage pools, where data is stored as a set of name-value pairs. An application can have multiple storage pools, which could be for each user or for user-independent data.

3.1 Leakage Prevention by xBook Design

In the current platform designs, a user's information can be leaked in three major ways: (1) applications can

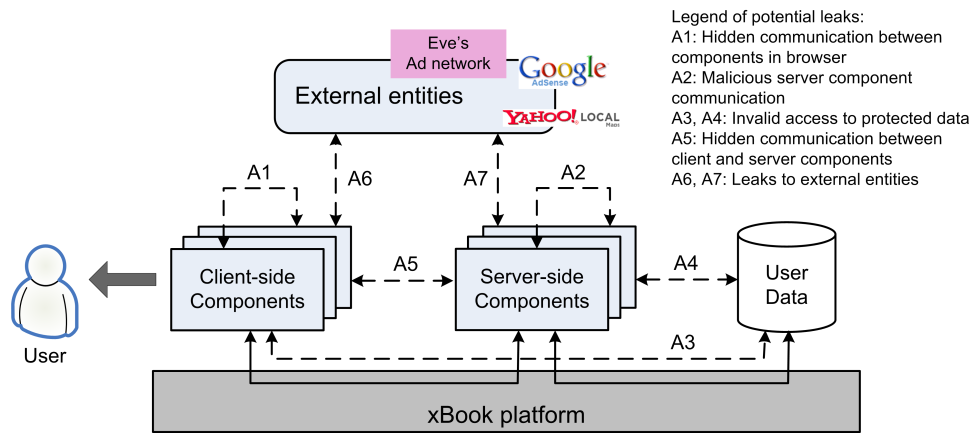

Legend of potential leaks:
A1: Hidden communication between components in browser
A2: Malicious server component communication
A3, A4: Invalid access to protected data
A5: Hidden communication between client and server components
A6, A7: Leaks to external entities
Eve's Ad network
External entities
Google AdSense
YAHOO! LOCAL
A1
A6
A7
A2
A5
A4
A3
Client-side Components
Server-side Components
User Data
User
xBook platform

Figure 3: `xBook` architecture shown along with sources of potential leaks.

share user's information with any third party, including advertisers, or fraudulent parties [21], and as shown in Figure 1(a), there is no way such a leak can be monitored in the current designs; (2) an application can pass information of one application user to another user, breaking free from the platform restriction that only friends can view a user's profile. The reported vulnerability in TopFriends allowed such a leak [26]; (3) the application can recreate the social graph of all its users by connecting common friends as edges in the graph.

xBook's design enforces complete mediation of all communication with the external entities (Figure 1(b)), thus preventing these applications from leaking information, effectively preventing (1) by design. A separate application instance is created for every user, and that instance only has a view of the data accessible to that user. Data access is restricted to allowed user policies, such as access to friends. We mediate any direct or indirect communication between the components of two application instances, thereby deterring (2). (3) is prevented as no single component of an application can have direct access to the data of all its users: a component can only access an anonymized view of this data set (Section 5.2).

xBook, by design, solves most of the leakage problems of the current platforms. However, there are still some potential mechanisms to leak information in our system. We enumerate these possible threats in our formal model and address these threats one by one throughout the paper.

3.2 Formal Requirements

We present a formal model in this section that generalizes xBook's mediation of untrusted third party applications. We use this model to analyze possible attacks, in terms of potential data leaks, under an adversary that deploys an application for collecting users' private data. We also identify a list of requirements that our system should satisfy in order to defeat such attacks. These formal requirements drive the design and architecture of our system.

Consider an application A consisting of a set of client-side components and a set of server-side components. Let U be the set of all users of the platform and Y be the set of all external entities. Suppose the application is allowed to communicate to a set of external entities $X \subseteq Y$ and a set of users $F_u \subseteq U$ for a particular user $u \in U$ who is using the system. Now, we divide the set of all data items D into three categories. First, there is a set of proprietary data or code of the application represented as $d_A \subseteq D$. Second, the set of data items $d_{u \to x}$ belonging to the user $u \in U$ that the application can transfer to the external entity $x \in X$. This set could be in the form of user's age, interests, photos, etc. Third, for an application instance of user $u_i \in U$, the set of data items $d_{u_i \to u_j}$ is what the application can transfer to a user $u_j \in F_{u_i}$.

The platform wants to monitor the occurrence of a set of events E that can pass information outside an application component. Any event $e \in E$ is actively monitored by intercepting the information flow path between the point of the event occurring and the point where the event is handled. The platform monitors the content information I_e contained in the event. We express the response of the platform when the particular instance of the event has potential leaking information as $R(I_e)$, which may include filtering the content, blocking the communication, etc.

We can identify several sources of potential leaks in the xBook system (Figure 3). The first class of attacks (A1) bypasses the active monitoring by the xBook platform to leak private information from one client-side component to another, by creating a prohibited flow. Such attacks exploit some of the abstract features of the development language and the browser to leak information maliciously. In other words, A1 occurs if response $R(I_e)$ is not triggered even if the I_e contains private information content that is being leaked. Similar leaks (A2) are possible on the server-side where application components can break out of the sandbox to create a prohibited channel with other components. In addition, some attacks (A3 and A4) can occur during a component's access to data store, where the component gains access to restricted user or application data. Leaks (A5) can also occur in the communication between client-side and server-side components. Other attacks (A6 and A7) leak private information to entities outside the system. The leaks could be to an $x \in Y$ that is prohibited ($x \notin X$), or it could be leaking restricted piece of information $d \in D$ to an entity via communication that is allowed by the system, i.e., for $x \in X, d \notin d_{u \rightarrow x}$ for a user $u \in U$.

We completely forbid cross-application communication, effectively preventing leaks across applications. We also prevent direct communication between server-side components, only allowing them to communicate via storage, thereby preventing attacks of type A2. We mediate other communication paths based on the labels of the communicating parties (Section 6). We address all other identified classes of attacks in Section 7.3. The requirements of an ideal social networking platform that guides the xBook design are as follows:

- Response $R(I_e)$ is invoked if I_e contains prohibited private information. In other words, the platform should be able to monitor any event that might be potentially leaking information, and should take action to prevent such leaks.
- Applications can invoke an event e iff $e \in E$, i.e., applications are restricted to a limited set of events for passing information to external entities.
- Application component having access to user u's private data d can send information to an external entity $x \in Y$ iff $x \in X$ and $d \in d_{u \rightarrow x}$. In other words, the platform should enforce user policies by limiting the communication to only *allowed* external parties and passing only *allowed* information to these parties.
- Application component having access to user u_i's private data d can send information to another component acting for user u_j iff $u_j \in F_{u_i}$ and $d \in d_{u_i \rightarrow u_j}$. This means that the applications should inherit the user-user access control policies of the platform.
- Application component x can access d_A only if $x \in S$, i.e., only server-side component of the application should have access to application's proprietary data.

We do not cover attacks against the browser in this work and assume that the browser behaves non-maliciously. Although phishing attacks can entice the user in choosing policies that might leak user information, we do not consider such attacks here. This work enforces the policies specified by the user, and does not consider social engineering attacks against the user.

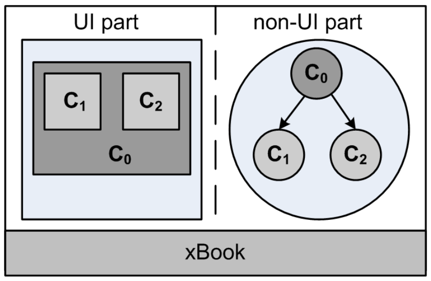


Figure 4: Client-side components in `xBook` design.

4 Client-side Components

The client-side of the xBook platform and the client components of the applications run within the web browser. The components are further divided into two parts: the user interface (UI) part that is visible as part of the page to the user, and the non-UI part that provides communication interfaces with the external parties and with the server side. There is a one-to-one mapping between the non-UI and the UI parts, i.e., for every non-UI part, there is a corresponding UI part visible to the user (Figure 4).

A component is allowed to create another component. Information can flow during the component creation and this opens up the possibility of an information leak. We prevent such leaks by allowing components to create other components that are at least as restricted as the creating component. This principle prevents the creating component from leaking information out of the system via a less restrictive component.

At the front end, the creating component needs to delegate some screen space to the created component. One challenge is to isolate the third-party application components within the Document Object Model (DOM) of the webpage. A DOM is a platform- and language-independent standard model for representing HTML or XML documents in a browser. We present our confinement approach in the next section.

4.1 Confinement Mechanism

The components of an application encapsulate different levels of private information for the users. Therefore, these components need to be isolated from each other in order to prevent information leaks. On the client side, the components form a part of the DOM of the web page. The web page's DOM may include multiple components from one or multiple applications, apart from the platform's DOM objects.

In the current browser specifications, any script in a page has intimate access to all of the information and relationships of the page. As a result, the components are free to access information about the DOM objects of other components. In order to confine the components within their own control domain, we limit the application code to be written in an object capability language called ADsafe [1]. In an object capability language, references are represented by capabilities and objects are accessed using these references. Other alternatives to ADsafe, such as Caja [25], are also available; we decided in favor of ADsafe due to its simpler design and easier feature addition and customization to meet our system needs.

ADsafe. ADsafe defines a subset of javascript that makes it safe to put guest code (such as third-party scripted advertising or widgets) on any web page. ADsafe removes features from javascript that are unsafe or grant uncontrolled access to browser elements. Some of the features that are removed from javascript are global variables and functions such as `this`, `eval` and `prototype`. It is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by static tools like JSLint [8].

ADsafe was initially developed to host untrusted advertising content safely on a webpage. xBook's isolation mechanism is designed with the code base taken from an earlier version of ADsafe. We customized ADsafe by adding code for our component confinement model and mediation based on our labeling model, to prevent information leaks from the "sandboxed" application components. A recent version of ADsafe have since implemented some of our features, but still would require changes to be useful for our system.

One such example is that ADsafe runtime supports only a single level of confinement: all subtrees of the untrusted guest applications exist as children of the trusted web page code. One guest application does not have another guest application as a child to its subtree. In contrast, xBook design requires *nested* DOM subtrees that need to be isolated from each other. Figure 4 shows an example of a nested subtree, where component C_3 is a child of component C_1, which in turn is a child of C_0.

Our requirement is to restrict an application component to within a set of connected DOM elements that form the component. In the current DOM specification, any DOM element can parse through the tree of the page via its parent, children or siblings. We enforce confinement by providing the component elements only with a partial view of the page's DOM and only indirect access to the DOM objects.

Confinement Rule 1. One DOM element belonging to an application component should only access another DOM element of the page (that includes accessing its properties, adding a new element to it, etc.) iff they both belong to the same component.

As part of the implementation, xBook associates each component with a unique *DOM wrapper* object at the time of creation. Figure 5 shows the partial code of our DOM wrapper implementation. Before deploying an application, xBook verifies that each component code is ADsafe compliant. The code must be wrapped in a `<div>` element having an identifier, which forms the root of the component. xBook ensures that this identifier is unique to the application page. The `ADSAFE.go` method gives the component code access to the `API` object that maps to our DOM wrapper object. The `ADSAFE` code ensures that the second parameter passed to the `createDOMWrapper` function is equal to the identifier of the encapsulating `<div>` element, effectively preventing the developer from faking the identity of the components. It also ensures that the DOM wrapper instance gets the right identity of the component's root node.

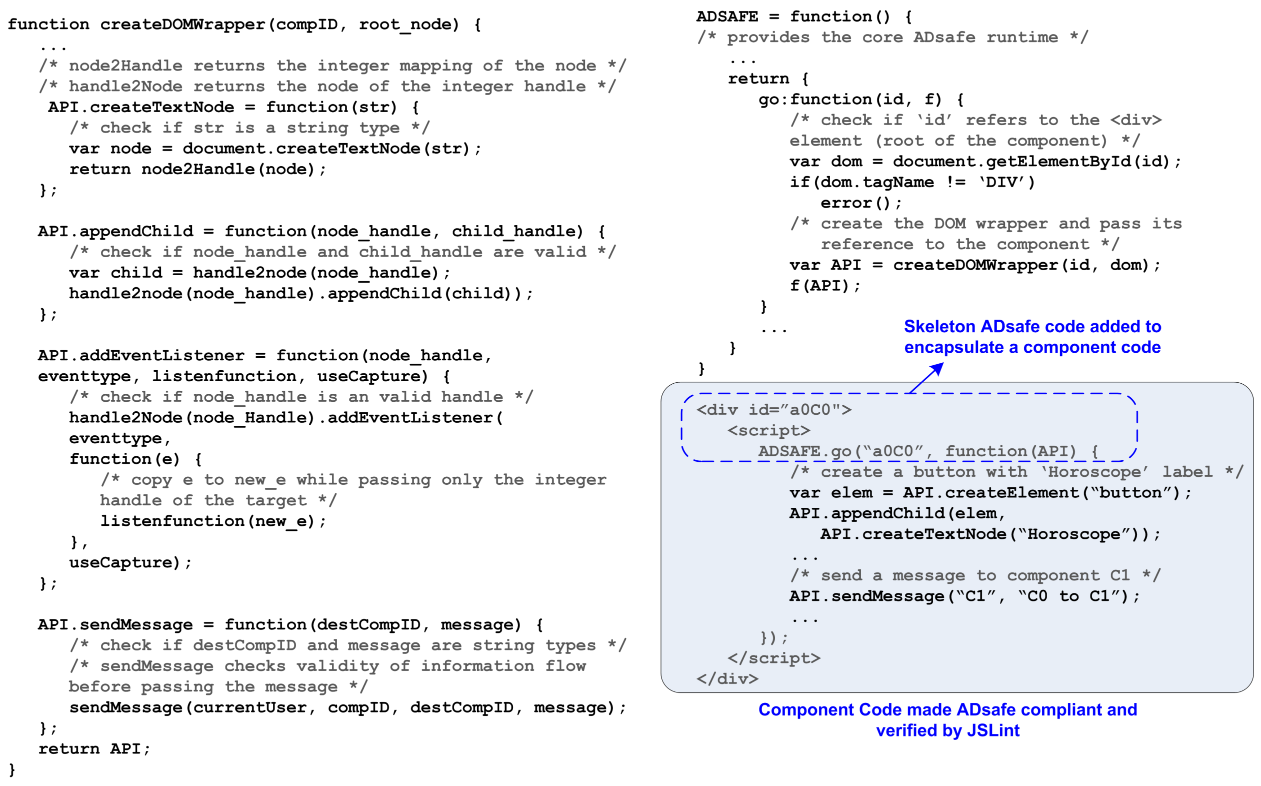
The wrapper allows an untrusted component to view DOM nodes simply as integer handles; the component has no direct access to the real DOM. To read or modify the DOM, the component code passes the appropriate handles to the wrapper DOM object using the xBook APIs, which in turn interacts with the real DOM. Additionally, element creation and modification are administered using this component-specific wrapper object. For example, `createTextNode` method in Figure 5 would return an integer handle. Since a wrapper instance is identified by its root element `<div>` that is unique, the DOM wrapper object restricts the untrusted component code to interacting only with the portion of the document tree that belongs to that component. All direct accesses to any real DOM elements are forbidden: the wrapper is the only interface for accessing the elements and it is mediated by the xBook platform.

4.1.1 Event Handling

Another possibility of an application breaking the confinement mechanism originates from the way event handling is designed in the current DOM specification.

Every event has a target, i.e., the XML or HTML element most closely associated with the event. An event handler is a piece of executable code or markup that responds to a particular event. Any element of the DOM can register an event handler to receive a particular event type. Since an event generated from within a component can be received outside the component, the flow of events within a DOM needs to be controlled by the xBook platform for any potential leaks.

In the current DOM implementation, it is possible to assign multiple handlers for a given event. It allows a DOM element to capture events during either of the two phases in the event flow. The event flows down from the root of the document tree to the target element in the first phase called *capture*, then it bubbles back up to the root in the *bubbling* phase. An element can receive the event only if it lies in the path between the document root and



```
function createDOMWrapper(compID, root_node) {
   ...
   /* node2Handle returns the integer mapping of the node */
   /* handle2Node returns the node of the integer handle */
    API.createTextNode = function(str) {
      /* check if str is a string type */
      var node = document.createTextNode(str);
      return node2Handle(node);
   };

   API.appendChild = function(node_handle, child_handle) {
      /* check if node_handle and child_handle are valid */
      var child = handle2node(node_handle);
      handle2node(node_handle).appendChild(child));
   };

   API.addEventListener = function(node_handle,
   eventtype, listenfunction, useCapture) {
      /* check if node_handle is an valid handle */
      handle2Node(node_Handle).addEventListener(
      eventtype,
      function(e) {
         /* copy e to new_e while passing only the integer
         handle of the target */
         listenfunction(new_e);
      },
      useCapture);
   };

   API.sendMessage = function(destCompID, message) {
      /* check if destCompID and message are string types */
      /* sendMessage checks validity of information flow
      before passing the message */
      sendMessage(currentUser, compID, destCompID, message);
   };
   return API;
}
```

```
ADSAFE = function() {
/* provides the core ADsafe runtime */
   ...
   return {
      go:function(id, f) {
         /* check if 'id' refers to the <div>
         element (root of the component) */
         var dom = document.getElementById(id);
         if(dom.tagName != 'DIV')
            error();
         /* create the DOM wrapper and pass its
            reference to the component */
         var API = createDOMWrapper(id, dom);
         f(API);
      }
      ...
   }
}
```

```
<div id="a0C0">
   <script>
      ADSAFE.go("a0C0", function(API) {
         /* create a button with 'Horoscope' label */
         var elem = API.createElement("button");
         API.appendChild(elem,
            API.createTextNode("Horoscope"));
         ...
         /* send a message to component C1 */
         API.sendMessage("C1", "C0 to C1");
         ...
      });
   </script>
</div>
```

Skeleton ADsafe code added to encapsulate a component code
Component Code made ADsafe compliant and verified by JSLint

Figure 5: DOM wrapper implementation with sample functions.

the event target.

One of the goals of our event handling model is to keep the functionality of the current DOM model (including preserving the concept of the two stages). Therefore, we specify our event flow model as follows: for any application component, an element can receive an event iff it lies in the path between the *root of the component* and the target element for the event. We still need to restrict this access to a single component so that no outside component can receive the event; we provide such a restriction by the following confinement rule:

Confinement Rule 2. A DOM element belonging to an application component can receive an event iff the event target belongs to the same component.

We implemented our event handling model using the DOM wrapper object introduced in the previous section. As shown in Figure 5, the object makes a wrapper to the event handling interface available to applications. The wrapper receives the event from the browser's DOM implementation and filters the information presented in the received event object before passing the event to the applications. Any information about the real DOM elements, such as the handler to the target element, is filtered; this prevents application's component code from breaking the confinement. The `addEventListener` method copies the received event `e` into `new_e` while transforming the real DOM element references to wrapped integer values. The xBook platform mediates the event delivery and as a result, ensures that an event can only be received by elements that belong to the same component that contains the target, thereby enforcing the second confinement rule.

4.2 Communication with External Entities

It is common for the applications to communicate with external parties to perform specific tasks. One typical example is the use of Google map APIs to generate maps of some address known to the application [9]. In other cases, a user's date of birth is used by applications to contact external providers to generate horoscopes [3]. What we achieve in our architecture as compared to the existing social networking platforms is that *we enforce the applications to make these communications explicit* so that more informed decisions can be made. The user or the platform can decide on the policies regarding which external entities are allowed to receive what piece of the user's private information. These policies could be coarse-grained for all applications of a user or fine-grained specific to each application. xBook ensures that the information flows from a specific application component to an external entity according to the defined policies.

There are two kinds of communication flows that can happen in our system:

Symmetric communication in which the response is received by the requesting component. This is a typical case for most client-server communication in which there is a two-way exchange of information between the two parties.

Asymmetric communication in which the response is not received by the component that made the request, but is handled by another component of the application. Our motivation for supporting this type of communication is to enable some specific application scenarios. One motivating example is the advertising scenario where advertisements are generated by external parties based on the information passed to them: Google generating advertisements based on the address passed to it. These external party advertisements are typically in the form of links that users click to access the related site. If we design this scenario using symmetric communication, these advertising links would not work, since the receiving component has been restricted to communicate only with Google and not any other party. In order to solve this problem, we can create another application component that is considered part of Google's trust domain; since Google servers are unconfined or public from xBook's point of view, the created component is also unconfined. We do not allow any other application component to peek into this new component or disrupt its integrity. Since we are only showing Google's view in this component and the application is not allowed to change this component, this component maintains the trust level of Google. The new component is placed in an `iframe` with its own DOM and hence cannot communicate with any other component. However, since the component is unconstrained, it is allowed to communicate with any external entity and as a result, the advertising links would work.

4.3 Communication between Components: Message Passing Interface

xBook exposes a one-way message passing API that the components use to pass messages to other components. We implement this interface using the DOM wrapper object as shown in Figure 5. The platform mediates this communication and ensures that the information flow model is enforced. Since each component is associated with a unique wrapper object that is used to send the message (Section 4.1), the sending component of the message can not fake its identify to fraudulently pass the information flow checks: as seen in Figure 5, the value of `currentUser` and sender's `compID` are implicitly provided by the wrapper object to xBook's `sendMessage` function. A component can register a message listener with the platform through the xBook API. Any message intended for a particular component is delivered to its message listener. Since the platform knows the identity of each component, it makes sure that the message is delivered to the right component.

The purpose of our message passing interface is to allow xBook-mediated communication among untrusted components of an application, while still preventing creation of any hidden channels. To this end, we needed to evaluate some of the features of javascript that gives application writers alternatives to pass hidden information in the messages.

Javascript is a weakly typed language and allows any property to be added to any object. For example, an object `message` can take a property `foo` using `message.foo = value;` where `value` could be a number, string or any other object type. Since all application components run in the same scope, a component can pass information to another component if it has access to an object of that component. Let us assume that a component C_1 is allowed to talk to another component C_2 as per the information flow policies, but C_2 can not communicate to C_1. Effectively, we have a one-way communication channel from C_1 to C_2. If C_1 passes the object `message` to C_2, the platform can observe `message`, but cannot identify the object handler `foo` being passed. C_2 can pass information to C_1 by writing to this handler.

We counter such leaks by limiting the message passing to being a JSON container [7], that is pure data. A javascript JSON container is a collection of key/value pairs or an array of values. These key/values are limited to pure data types such as string or numbers. We make a copy of the JSON object and pass the copy to guarantee that there are no additional properties in the passed object. This solution is also effective against attacks by a message sender that use getters and setters.

The simplest way of designing the message passing interface is to pass messages from a source to a destination in a single thread of execution. This option opens up the possibility of a covert communication channel from a more restricted to a less restricted component. For example, let us consider that a less secret component C_0 is passing multiple messages to a more secret component C_1. Because of the single-threaded non-preemptive nature of javascript, C_1 will complete processing the first message before the control goes back to C_0. This creates a covert timing channel from C_1 to C_0. The amount of time taken by C_1 can be observed by C_0 and C_1 can change this time to pass the desired information bits to C_0.

We reduce the effect of this timing channel by making the message passing interface asynchronous. We achieve asynchronous behavior by implementing a global queue for message passing that is shared among all the components of an application. The receiving components register listeners with the platform in order to receive messages. A timer event dequeues an available message and delivers it to the message listener of the target component of the message. Note that addressing all covert channels in our system is beyond the scope of this paper; we discuss this further in Section 8.

5 Server-side Components

The server-side of the application contains the main functionality for typical applications. It follows a familiar web server model where a server-side component is instantiated for every client request.

Besides the regular user-specific components on the server side, there are certain components that are user independent and works on non-user data or user public data. These components perform two tasks: First, they communicate with external parties to provide functionality independent of the user data. Second, they handle statistical aggregation on user data sets. We discuss declassification based on data anonymization in Section 5.2.

The server components also protect application proprietary data that needs to be declassified before sending it to the client. The threat model is reversed in this case: the applications do not trust the user for their data, so they protect their internal data from being leaked to the users. For example, an application might be giving horoscope predictions to users based on their birth date, but it wants to protect the data or algorithm used for such predictions.

There is no direct communication between the server-side components: all such communication happens via application-specific storage. The platform ensures that the information flow is enforced while accessing the database. The platform also administers the communication with external parties and client-side as allowed by the labeling system.

5.1 Component Confinement

The server-side components need to be isolated from each other. The server-side of xBook mediates all communication flowing in and out from these components. There are several options available for server-side isolation. Operating system isolation mechanisms [12, 30] can be used to sandbox the application components. Another option is a language level confinement similar to the client-side isolation with options like Caja (Javascript) [25], ADsafe (javascript) [1] and JoeE (Java) [20] available. We use ADsafe on the server-side in order to have the same language for developing application components for both client and server.

To the best of our knowledge, we are the first ones to port ADsafe to the server side. We had to make some modification to the ADsafe object to implement our server-side xBook APIs and to perform checking of the information flow labels. Each server-side component holds a unique handle to the modified ADsafe object, and access is restricted to the set of APIs provided by the modified ADsafe object. The modified ADsafe object is conceptually similar to the DOM wrapper object on the client side, but is customized to work in the server-side environment. The platform verifies the validity of the information flow before any access is granted. The javascript execution environment is provided by Helma [6], a popular open source web application framework.

5.2 Anonymized Statistics

xBook ensures that no user data is leaked against the user's policies. A particular instance of an application can only have access to profile data that belongs to the user and only his friends. Different instances of the applications cannot share data due to the restrictions posed by xBook's labeling system.

It is desirable for some applications to have a view of all its users so that some statistical results can be published for the whole application. In other words, a component of the application needs to receive data of all the application users and still should be able to share these statistics as output to all users, crossing the boundary of friends.

In order to facilitate this case, we are exploring a three-step anonymization algorithm that provides conservative access to data for the applications. Currently, case 1 and 3 have been implemented, case 2 will be explored as part of our future work.

Case 1. If an application component requests a single field of user information for all application users, it is given access to the requested set in an unmodified form, but in a random order of sequence.

Case 2. If an application component requests multiple fields of user information for all application users, it is given access to the requested set in a form generated by anonymizing the original dataset and then randomizing the resulting tuples' order of sequence. We plan to leverage some of the existing work [15, 24, 31] to generate the anonymized statistics. We acknowledge that providing security in anonymity and statistical queries is a challenging problem and has its own limitations [13, 24]. Addressing these limitations is orthogonal to our work and is not the focus of this paper.

Case 3. Applications can also request the xBook platform for statistics on unanonymized data. This gives the applications more accurate statistics as compared to case 2, where some fields might be filtered or altered to preserve anonymity. xBook provides a limited list of such operations, including aggregation, maximum and minimum value over one or multiple fields.

Discussion. Anonymizing the data might limit some applications that rely on the original data for their functionality. One such example is an application that plots the location of a user's friends on Google maps, and would need to pass names and addresses of the user's friends to Google. The application also makes subsequent queries to Google (for example, to build a Google calendar of friends' birthdays). If the data is anonymized, the application might not produce completely accurate results.

On the other hand, if Google is provided with unanonymized data, it can use the data to cross-reference and identify the friends. This is a conflict between privacy and functionality. If functionality is preferred and unanonymized information is passed to external entities, user's personal information can be leaked. In such a case, our xBook design, at the minimum, enforces the applications to explicitly declare all external communication (including the data that will be transferred). Based on such information, the user can make a much more informed decision about adding the application.

6 Labeling Model

The xBook platform tracks and enforces information flow using a labeling system defined based on existing models [17,23,27,36]. All system abstractions are layered on top of two types of entities – active and passive. Application components represent active entities that actively participate in label compatibility checks; database entries are passive entities. Every active entity corresponds to a principal and a label; passive entities only have a label.

We do not enforce information flow at the language level [27], but instead at the level of application components and database entries. There are multiple reasons for this choice: (1) it is simpler for the application programmers as they do not need to learn a new language or perform fine-grained code annotations, (2) information flow on a language like javascript with dynamically created source code may not be feasible, and (3) run-time information flow at fine-grained language level would probably be expensive as compared to a much coarser level of components.

The label specifies the secrecy level of an entity. It represents what information is contained in a passive entity and what information the active entity currently has or will read. The entity's principal defines whether the entity has declassification privileges over the label. xBook labels originated along the lines of the language based labels in Jif [27]. Labels represent the confidentiality or secrecy level of an entity in the system. Integrity labeling is not the focus of this work since we are focusing on privacy.

A label L is represented as a set of tags, with each tag having one principal as owner o and another set of principals called readers $R(L, o)$. The owner is the principal whose data was observed in order to construct the data value. The readers represent principals to whom the owner is willing to release the information. An example of a typical label is $L = \{o_1 : r_1, r_2; o_2 : r_2, r_3\}$, where $O(L) = \{o_1, o_2\}$ denote the owner set for the label and readers sets are $R(L, o_1) = \{r_1, r_2\}$ and $R(L, o_2) = \{r_2, r_3\}$.

In the xBook system, principals represent the identities of various entities in the labeling model. There are five types of principals in our system:

- $C(a_i, u_j)$ and $S(a_i, u_j)$ represents the client-side and server-side components for an application a_i specific to a user u_j.
- $C(a_i)$ and $S(a_i)$ represents user-independent client-side and server-side components for an application a_i.
- u_j represents the entities that the user u_j is in complete control of. Once the user u_j is logged into the xBook system, the user's browser is assigned the principal u_j.
- $\top$, $\bot$ where $\top$ is highest priority principal in the system and is allotted to the xBook platform. For the sake of completeness, $\bot$ is the least privileged principal.
- External entities also have principal names that contain the hostname and optionally the scheme and port (like in URLs). For example, `https://www.example.com:8888` represents one such principal.

Our model assumes static labels for the entities and information flows from one entity to another if allowed by the label comparison of the end points. Information can flow from one label L_1 to another label L_2 only if L_2 is more *restricted* than L_1 denoted as $L_1 \preceq L_2$.

Restriction. $L_1 \preceq L_2 \iff O(L_1) \subseteq O(L_2)$ and $\forall o \in O(L_1), R(L_1, o) \supseteq R(L_2, o)$

6.1 acts-for Hierarchy

To facilitate easier conversion of user policies to low-level labels, system entities are statically labeled. We decided on immutable labels since it improves usability of the application programming model from the perspective of the application programmer. Unexpected runtime failures can occur when labels of components change at runtime [23]. With immutable labels one can statically verify that all the communication dependencies with respect to other components, external entities, storage will be satisfied.

Some principals have the right to act for other principals and assume their power. The acts-for relation is transitive, defining a hierarchy or partial order of principals [17]. The right of one principal to act for another is predefined by the platform. Figure 6 presents the acts-for relationship within the xBook system. This hierarchy defines the priority of different principles in the system. The reasoning behind the defined hierarchy is as follows:

- $\top$ defines the xbook platform and has the highest security label. As a result, it can declassify any label.
- Any data sink or source that is not explicitly defined by xBook is modeled as an unprivileged entity with label $\bot$.
- The client-side components are given lower priority than server-side components, because intuitively server-side components residing on xBook servers

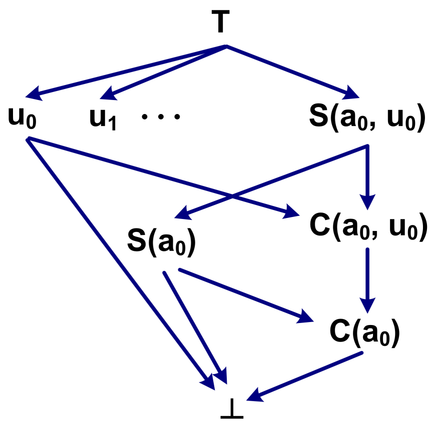


Figure 6: Label hierarchy in xbook.

Algorithm 1 Label Compatibility Check Algorithm.

```
$eL_1$ = ($entity_1$ is a database) ? $L_1$ : maxDeclassify($L_1$, $P_1$)
$eL_2$ = ($entity_2$ is a database) ? $L_2$ : maxRestrict($L_2$, $P_2$)
if $eL_1 \preceq eL_2$ then
    ALLOW flow from $entity_1$ to $entity_2$
else
    DENY flow
end if
```

Figure 7: Algorithm to check if the information flow from $entity_1$ to $entity_2$ is allowed.

are more trustworthy than client-side components. For example, $S(a_0, u_0)$ has higher priority over $C(a_0, u_0)$ for application a_0 and user u_0. The server-side components can declassify an application's proprietary data, which has been labeled in a manner such that it cannot be directly read by client-side components.

- User-independent principals are at a lower priority than any user-specific principal. This allows user-specific components to read user-independent data generated by an application, also effectively allowing users to read statistical data generated for the whole application.
- Principals representing the end user are higher than the corresponding client-side principals since the user controls the client.

6.2 Flow Enforcement

Information flows within the xBook system if the label of source is less restricted than that of destination. Such flow restrictions have been proposed earlier in classical information flow control models [14]. We introduce the concept of endpoints similar to the Flume model [23]. Instead of changing the labels of the entities, for every communication the source and the destination create an endpoint each to facilitate the flow. The entity, based on its principal, can restrict or declassify its label and allocate it to an endpoint for communication. While restricting a label means adding more owners and removing readers, declassification either adds some readers for an owner o or removes the owner o. This relabeling can be done only if the principal of the entity is higher than an owner o in the hierarchy.

Figure 7 shows our flow enforcement algorithm, where maxRestrict and maxDeclassify are defined as:

- ***maxRestrict(L, P).*** $O(L) = O(L) \cup descendent(P)$; $\forall o \in descendent(P)$: $R(L, o) = \{\}$
- ***maxDeclassify(L, P).*** $\forall o \in O(L)$: if $(o \in descendent(P))$ then $O(L) = O(L) - \{o\}$

where $descendent(P)$ represents all descendents of a principal P in the acts-for hierarchy, $O(L)$ is the set of owners for label L and $R(L, o)$ represents a set of readers in label L for owner o. Intuitively, the communicating end points support the communication with the sender declassifying its label to the maximum possible using $maxDeclassify$ and the receiver restricting its label using $maxRestrict$. Since the information can only flow from a less restricted to a more restricted component, these functions facilitate the flow of information.

Some typical flows in the xBook system are depicted in Figure 8. To demonstrate the validity of our algorithm, let us consider the example of the flow between the client-side component C_1 and the server-side component S_1. For the flow from S_1 to C_1,

$$\begin{aligned} eL_1 &= maxDeclassify(\{S(a_0) :; \top : C(a_0, u_0)\}, \\ &\qquad S(a_0, u_0)) = \{\top : C(a_0, u_0)\} \\ eL_2 &= maxRestrict(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ &= \{C(a_0, u_0) :; C(a_0) :; \top : C(a_0, u_0)\} \end{aligned}$$

Recollecting the definition of restriction, we can see that $eL_1 \preceq eL_2$, therefore S_1 can send data to C_1. Considering the reverse flow from C_1 to S_1,

$$\begin{aligned} eL_1 &= maxDeclassify(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ &= \{\top : C(a_0, u_0)\} \\ eL_2 &= maxRestrict(\{S(a_0) :; \top : C(a_0, u_0)\}, S(a_0, u_0)) \\ &= \{S(a_0, u_0) :; S(a_0) :; C(a_0, u_0) :; (a_0) :; \\ &\quad \top : C(a_0, u_0)\} \end{aligned}$$

We can see that $eL_1 \preceq eL_2$, i.e., C_1 can send data to S_1. Effectively, there is a two-way communication between C_1 and S_1.

6.3 Case Study: Horoscope Application Lifecycle

An application's lifecycle consists of three steps: the application being hosted by xBook, a user adding the ap-

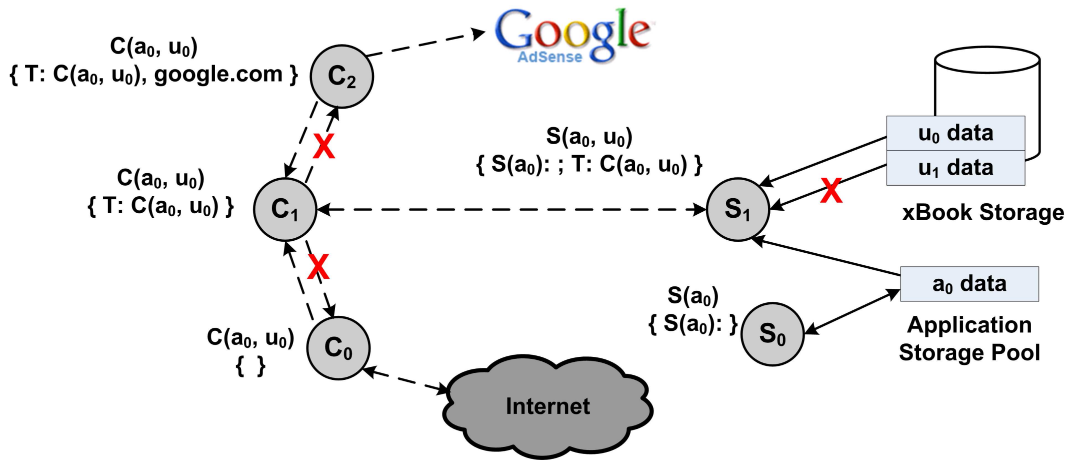


Figure 8: Typical Flows in `xBook` system with the corresponding labels. For every component, the first parameter is the principal and the second is the label associated with the component.

plication and then the user accessing it.

Hosting. Before xBook accepts a new application, the developer needs to provide the following information:

- The application provides the components to be deployed, in each case specifying if the component is client-side or server-side and if it is user-dependent or not, what user data would the component require and which external entities and other components will it communicate with. In our horoscope example, there are three components: S_0 communicates with `www.tarot.com` and requires no user data; S_1 requires user's birthday; C_1 is on the client-side and also requires user's birthday.
- The application also states that there are user-independent or user-dependent storage pools and each is named declaratively by the application. This ensures that the storage pool names do not leak any user information, as the application has no user information at this time. For example, horoscope application declares a storage pool for storing its application data generated by S_0.

Based on the label of the user data, xBook derives the labels and the principals of the components. The birthday field has a label $\{\top : C(a_i, u_j)\}$, therefore the following labels are allocated to the horoscope components:

- S_0 *Principal:* $S(a_i)$, *Label:* $\{S(a_i) : \}$
- S_1 *Principal:* $S(a_i, u_j)$, *Label:* $\{S(a_i) : ;$ $\top : C(a_i, u_j)\}$
- C_1 *Principal:* $C(a_i, u_j)$, *Label:* $\{\top : C(a_i, u_j)\}$

The principals define if the component is server-side or client-side, and if it is user-dependent or not. The labels allow S_1 and C_1 to read the birthday field. S_0's label allows it to declassify itself to be public to communicate with `www.tarot.com`, and write to the storage pool that is given S_0's label. The storage pool label prevents any of the client-side components (C_1) from viewing this data, thereby protecting application data from untrusted users. S_1 is allowed to read from the storage pool. The labels of S_1 and C_1 correspond to the labels of S_1 and C_1 respectively in Figure 8, where $i = 0$ and $j = 0$. As we have observed in the last section, the labels of S_1 and C_1 effectively allow a two-way communication channel. Thus, S_1 can pass the results to C_1 that, in turn, can present a formatted form of the horoscope to the user's browser.

Application Addition. When the user is adding the application, he is provided with a manifest that declares what information is passed to which external entity. xBook derives the manifest from the component information provided by the application developer. For example, since none of the components of the horoscope application share any user information with any external entity, horoscope's manifest would specify that it does not pass any information to any external entity. Since the user's birthday is not shared with any external entity, the application does not need to declare its need to access the birthday information.

Application Access. When the user is accessing an application, all user-specific components are instantiated for that user, replacing the user wildcard in the template of labels and principals with the user identifier. This enforces access control across multiple users: access is only granted if it is aligned with the user's privacy policy, for example, access is granted to only user's friends.

7 Evaluation

7.1 Prototype System and Example Applications

We developed a working prototype of the xBook system, which includes platform code and APIs for developing third-party applications. We also implemented the labeling model that enforces information flow control for

Attack Step	Attack Type	Prevented by xBook?
One client component accessing another component's DOM object	A1	$\surd$
Leaks via the message passing interface	A1	$\surd$
A component creates or destroys a less restricted component leaking information	A1	$\surd$
Retrieve information of another user not in the friend list	A3/A4	$\surd$
Client component retrieves more restricted information from the server	A5	$\surd$
Leaks to an unknown external entity	A6/A7	$\surd$
Leaking restricted information to an allowed external entity	A6/A7	$\surd$

Table 1: Prevention of information leaks against various kinds of synthetic attacks.

the data flowing through the system and prevents any information leaks. Our xBook platform consists of about 4300 lines of javascript code.

We developed two sample applications using the xBook APIs to show the ease and viability of application development in xBook. These applications are similar in functionality to two popular Facebook applications: Horoscope [3] and TopFriends [11].The horoscope application produces a user's daily horoscope based on his birthday information. The utility application based on TopFriends produces a customized profile for the user based on his complete profile information. It also generates a Google map showing the user's home location on the map. The applications are written in javascript using xBook APIs, with the horoscope application having about 180 lines and the application based on TopFriends having around 480 lines of code. We tested these applications against a series of synthetic scenarios, where these applications tried to leak the user's private information. Our tests showed that the xBook system was successful in detecting and preventing all such leaks.

7.2 Porting xBook on Facebook

In order to show the practical viability of the system and to demonstrate that our system can be incrementally deployed, we ported the xBook platform as an application on Facebook. Since Facebook allows any application to have access to user data, including their friends' data, of any user adding the application, xBook as an "application" is able to receive the data of the users agreeing to use the xBook platform. Applications developed using xBook APIs can execute on top of xBook, while still running on xBook servers. Since xBook act as an application for Facebook, xBook's response would be rendered as part of Facebook's web page. Since the third party applications are encapsulated in the page forming xBook's response, the output of these applications would also be effectively rendered on Facebook (Figure 1(c)). Facebook provides the data feed to xBook, which then enables access to this data for xBook applications in a controlled manner through xBook APIs. Facebook's user identity is maintained within xBook. Our running system is available online on Facebook [33].

We envision xBook to be assimilated into the Facebook platform with Facebook providing two levels of application service. First, the current applications based on current Facebook design would be supported. Second, applications that are developed using xBook APIs are supported, with added privacy protection advantage. Users can be given the discretion to choose between the two options, and the users' choice can drive new application development on xBook.

7.3 Security Analysis

Our analysis shows that xBook prevents the applications from leaking any user information. All of the documented leaks in the current social networks are prevented in the xBook system. For example, the TopFriends leak [26] cannot happen in our system because a separate application instance is created for every user. Each instance only has view of the data accessible to that user and xBook mediates all cross user data accesses.

We evaluated the privacy protection ability of our system in three steps. First, we analyzed the security of the xBook design in view of the potential leaks specified in the formal model (Section 3.2). Second, we developed a set of synthetic attacks targeting the xBook framework and performed experiments to show that our prototype successfully prevents these attacks. Finally, we prove that xBook's information flow model ensures that information leaks cannot happen in the xBook design.

We first analyze the security of our prototype and show that all the attacks discussed in Section 3.2 will not succeed against our design. Attack type A1 is prevented due to the various mechanisms developed in our system for client-side confinement (Section 4.1), such as component isolation, event handling, etc. A2 is prevented by server-side confinement of application components, only allowing them to communicate via storage. Leaks via A3 and A4 are inherently prevented by mediating the information flow from the database to application components with label enforcement based on user-defined policies, and also by anonymizing data for statistical purposes (Section 5.2). A5 is also prevented by label enforcement before the client-side request is passed to the server-side component and before response is returned. Enforcing the confinement model to mediate the external communication, both in synchronous and asynchronous communica-

Application	User latency	Server processing time	Time for label checks (Number of checks)	Overhead
Horoscope	183.1ms	128.8ms	7.7ms (6)	4.2%
Map utility	111.4ms	51.2ms	3.5ms (2)	3.1%

Table 2: Performance results of various operations in typical xBook applications.

tion scenarios, prevents A6 leaks (Section 4.2). Following the same lines, A7 is prevented on the server-side.

Second, we tested the ability of our prototype by creating synthetic exploits that try to break out of xBook's information flow control model to leak user information. We developed a sample application to launch these attacks against our prototype; if successful, these attacks allow the application to leak information to entities outside the system. Table 1 contains the results of testing our prototype against a wide range of these synthetic attacks. In all our experimental tests, xBook successfully prevented the leaks before the information could be passed outside the system.

We can also prove that if xBook's confinement mechanism is correctly enforced, the information model ensures that no user information is leaked to external entities (Theorem 1) and to any other user (Theorem 2) outside the user-defined policies.

Theorem 1. *Given a set of policies $P = D \times X$, where the application can pass user's information field $d \in D$ to external entity $x \in X$, and assuming that the intended confinement is enforced, the information flow model ensures that there is no possible leak outside the xBook system. In other words, if $(d, x) \notin P$ then $\forall C_i : C_i \nrightarrow^d x$, where C_i are application components and $C_i \nrightarrow^d x$ shows that C_i can not pass data item d to x.*

Proof. Let $C^0, C^1, \cdots C^k$ represents the information flow path of a data element d from the xBook database to external entity x.

We present the proof by contradiction. Let us assume that C^i can pass any information (represented by $*$) to x, illustrated as $C^i \xrightarrow{*} x$. This communication is monitored by our xBook platform, but the platform does not know the semantics of the information being passed.

Also, $\forall i \in [0, k] : C^{i-1} \xrightarrow{*} C^i \implies L^{i-1} \preceq L^i$ (flow is a restriction)

$C^i \xrightarrow{*} x \implies L^i \preceq L^x$

Therefore, $L^{i-1} \preceq L^x \implies C^{i-1} \xrightarrow{*} x$

Continuing this by induction, $C^0 \xrightarrow{*} x$

In our labeling model, the computational granularity is at the component level. Therefore, we consider that $\forall C_i$: $Output(C_i) = \digamma(Input(C_i))$ for any computation $\digamma$.

For component C^0, $Input(C^0) = d$, $Output(C^0) = *$ $\implies * = \digamma(d)$

Since the input to C^0 is supplied by the xBook platform, and since $(d, x) \notin \mathbb{P}$, $C^0 \nrightarrow^* x$.

This is a contradiction. Therefore, $C^i \nrightarrow^* x$.

By definition, $*$ represents any information (including d). Therefore, $C^i \nrightarrow^d x$.

Theorem 2. *Given a set of user policies $P(x) = D \times U$, where the application can pass user $x \in U$'s information field $d \in D$ to another user $y \in U$, and assuming that the intended confinement is enforced, the information flow model ensures that user-user access control is enforced in the xBook system. In other words, if $(d, y) \notin P(x)$ then $\forall C_i(x), C_j(y) : C_i(x) \nrightarrow^d C_j(y)$, where $C_i(x)$ and $C_j(y)$ are components of application instance for user x and y, respectively.*

Proof. Similar to Theorem 1.

7.4 Performance Estimates

xBook does not impose a substantial burden on the performance of the third party applications. With an architectural framework of developing applications, it is difficult to accurately predict the impact of our design on the performance of these applications as perceived by the user. To get a rough estimate of the cost of supporting the xBook design and the overhead involved in our system, we conducted some experiments with our sample applications, measuring latency at the user end and overhead imposed by the mediating design of xBook.

The xBook server side is hosted on a 2.4GHz Pentium 4 machine with 512MB of RAM. The requests are made from Firefox 3.0 browser on a 2.33GHz, 2GB RAM, Pentium Core Duo laptop. Each test was run 10 times and values were averaged. We define user latency as the difference in the time when the request is made at the browser and the time at which the response is received by the browser. Table 2 shows the time required by xBook's information flow control in comparison to the user's overall latency. Server processing includes the application's logic, database access to retrieve required user data, and xBook flow checks, and is independent of the network latency experienced by the application. We instrumented our code to derive the time for performing label checks in the system, and measured overhead as a function of the label checking time over the total latency experienced by the user. Our results show that the overhead introduced by xBook's label checks is considerably small: about 4% for the horoscope application and 3% for the map utility marking user's hometown location on Google maps.

On a cluster of commercial servers with much better computational capacity, these values will be even smaller. Although it is not possible to precisely determine the cost of our approach without a large scale experiment, both the details of our design and the results from these experiments, support the conclusion that xBook design would not substantially increase the latency experienced by users.

8 Discussion

In this section, we discuss the limitations of the application design in xBook and address some of the challenges arising from the new requirements imposed by our design.

Our xBook design imposes no limitations on applications that follow a "pull model", i.e., xBook would preserve the functionality of applications that only receive data from external entities without passing any private information to these entities. Our horoscope application is an example of such as application: one public component of horoscope pulls horoscope data from `www.tarot.com` and does not pass any of the user's profile information. Note that the xBook platform does not need to sanitize the request parameters (in both GET and POST requests), as the component making such requests has no user information that can be leaked. Another component, which has access to the user's birthday information, uses the data to calculate the daily horoscope corresponding to the particular user. This component has no communication with any external entity.

On the other hand, our design might limit some of the applications that require data to be sent to external entities for receiving user-specific information. One typical example is the use of Google APIs to generate maps: it requires a location to be passed to Google before the map is generated. In many cases, we expect these external entities to be larger and well branded entities, such as Google, Yahoo, etc. Such cases could be whitelisted after explicit approval from the user. Note that xBook makes no recommendation about which websites can be trusted, including Google and Yahoo; such trust decisions are made by an individual user from his own knowledge and experiences. Our xBook system can keep track of these approvals across applications for every user, so the users need to approve an interaction only once.

Any social networking application would follow either the pull model or the push model to get data from external entities. In both cases, our platform enforces the applications to make all such interactions explicit and allows the user to make a more informed decision based on the information available. We argue that an application using the pull model would be more acceptable to the users as it requires minimal trust decisions from a user's perspective. It is possible to transform many of the current social networking applications that use the push model to start using the pull model. We acknowledge that such a transformation would require some changes to the application design, and in some cases, such transformations might not be practical due to large download size of the required data. However, if enough users decide not to use the application in view of privacy concerns, it would motivate the developers to consider such a transition.

Our system also suffers from classical covert channels, e.g. timing, memory, process, etc. However, in general these channels have limited bandwidth and viable approaches such as randomizing the time (for example, the delivery time of our message queue discussed in Section 4.3) can further limit their utilities. We plan to study some of these channels as part of our future work.

Scalability of the applications is not a concern in our system: applications hosted on clusters outside xBook would now be hosted on clusters inside the xBook platform. The application developers are already paying for hosting their applications, in most cases to third-parties or cloud owners like Amazon EC2 [2]. Thus, instead of the developers paying to these parties, they would be paying to xBook for the hosting service. xBook, in turn, can outsource the hosting to third-parties, still assuming control of the hosted applications.

We also propose a hybrid model where only the application components that require access to xBook's private data needs to be hosted at the xBook servers. Other public components can be controlled by the application developers on their own servers. Such an approach is useful for many applications as research has shown that a large number of applications do not use any private data to perform their functionality [19].

9 Related Work

Information flow control at the language level has been well studied [16,27]. Jif is a Java-based programming language that enforces decentralized information flow control within a program, providing finer grained control than xBook [27]. In comparison to these language level techniques that require the applications to be rewritten, the xBook platform provides a simpler interface to the application programmers: they do not need to learn a new language or perform any fine-grained code annotations. Additionally, information flow on a language like javascript with dynamically created source code may not be feasible. Cong et al. [16] presented a technique of writing secure web applications, which generates javascript code on the client side and java code on the server side. However, the applications are still written in the Jif language.

There are other systems [23, 36] that have utilized the information flow concept to control data flow at the operating systems (OS) level. Information flows are tracked at low-level OS object types such as threads, processes, etc. xBook works at a much coarser level at the applications, with smallest unit of information being an application component. As a result, run-time information flow in xBook would probably be less expensive as compared to a much finer granularity level used in these systems. In order to make these systems useful for a typical social networking environment, it would require the systems to be installed at a user's computer because leaks can also happen at the browser, which might not be feasible. In comparison, xBook runs on a typical web server without any changes to the OS environment.

Similar to the ADsafe environment, other safe subsets of programming languages, such as JoeE [20] (for java) and Caja [25] (for javascript), allow third-party applications to provide active content safely and flexibility within the existing web standards. While we used ADsafe for its simplicity and suitability to meet our system needs, we expect that it would be similarly possible to develop xBook using these alternatives.

10 Conclusions

We presented a novel architecture for a social networking framework, called xBook, that substantially improves privacy control in the presence of untrusted third-party application. Our design allows the applications to have access to user data to preserve their functionality, but at the same time preventing them from leaking users' private information.

We developed a working prototype of the system that is available as an application on Facebook [33]. We showed the viability of our system by developing sample applications using the xBook APIs: these applications are similar in functionality to the applications on existing social networks.

Our system shows promise in designing potentially valuable future applications, that would require user data to provide more customized service to the user. The growing popularity of social networks would attract increasing attention from attackers because of the value of user information available in these networks. This user information not only has commercial value, but when combined with some anonymized public data such as medical records, might leak more sensitive information [28, 34]. The current design of social networking applications poses a serious threat to the privacy of individuals that needs to be mitigated; the xBook platform is a major step in protecting user privacy in social networking applications.

Acknowledgement

This material is based upon work supported in part by the NSF under grants no. 0716570 and 0831300 and the Department of Homeland Security under contract no. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the Department of Homeland Security. We would also like to thank Monirul Sharif, Roberto Perdisci and the anonymous reviewers for their helpful comments and our shepherd George Danezis for his valuable suggestions.

References

[1] ADsafe. `http://adsafe.org`. Last accessed Feb. 1, 2009.

[2] Amazon elastic computing cloud. `http://aws.amazon.com/ec2/`. Last accessed Feb. 1, 2009.

[3] Daily horoscopes. `http://apps.facebook.com/daily-horoscope`. Last accessed Feb. 1, 2009.

[4] Facebook developers: Developer terms of service. `http://developers.facebook.com/terms.php`. Last accessed Feb. 1, 2009.

[5] Facebook's privacy policy. `http://www.facebook.com/policy.php`. Last accessed Feb. 1, 2009.

[6] Helma javascript web application framework. `http://www.helma.org`.

[7] Javascript object notation (JSON). `http://www.json.org`. Last accessed Feb. 1, 2009.

[8] JSLint: The javascript verifier. `http://www.jslint.com`. Last accessed Feb. 1, 2009.

[9] Map your friends. `http://apps.facebook.com/mapyourfriends`. Last accessed Feb. 1, 2009.

[10] Opensocial. `http://www.opensocial.org/`. Last accessed Feb. 1, 2009.

[11] Topfriends. `http://apps.facebook.com/topfriends`. Last accessed Feb. 1, 2009.

[12] A. Acharya and M. Raje. MAPbox: using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.

[13] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, Banff, Canada, May 2007.

[14] D. E. Bell and L. J. Lapadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976.

[15] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Baltimore, MD, 2005.

[16] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[17] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[18] D. Farber. Google to open orkut opensocial developer sandbox tonight, Nov. 2007. `http://blogs.zdnet.com/BTL/?p=6856`. Last accessed Feb. 1, 2009.

[19] A. Felt and D. Evans. Privacy protection for social networking platforms. In *Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2008.

[20] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, Oct. 2008.

[21] S. Hacking. More advertising issues on facebook (updated), 2008. http://theharmonyguy.com/2008/06/20/more-advertising-issues-on-facebook/. Last accessed Feb. 1, 2009.

[22] R. Konrad. Facebook opens to third-party developers, May 2007. http://www.msnbc.msn.com/id/18899269/. Last accessed Feb. 1, 2009.

[23] 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 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[24] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Transactions of Knowledge Discovery from Data*, 1(1):3, 2007.

[25] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: safe active content in sanitized javascript, Oct. 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[26] E. Mills. Facebook suspends app that permitted peephole, 2008. http://news.cnet.com/8301-10784_3-9977762-7.html. Last accessed Feb. 1, 2009.

[27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.

[28] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[29] T. Panja. Oxford using Facebook to snoop. http://www.msnbc.msn.com/id/19813092/. Last accessed Feb. 1, 2009.

[30] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, San Franscisco, CA, Aug. 2002.

[31] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[32] D. Sciba. Mayor in myspace photo flap asked to resign. http://www.katu.com/news/13670287.html. Last accessed Feb. 1, 2009.

[33] K. Singh, S. Bhola, and W. Lee. xBook on Facebook. http://apps.facebook.com/myxbook. Last accessed Feb. 1, 2009.

[34] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *Journal of Law, Medicine and Ethics*, 25:98–110, 1997.

[35] C. Williams. Facebook application hawks your personal opinions for cash, Sept. 2007. http://www.theregister.co.uk/2007/09/12/facebook_compare_people/. Last accessed Feb. 1, 2009.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.