Usable Security as a Static-Analysis Problem

Modeling and Reasoning About User Permissions in Social-Sharing Systems

Hannah Quay-de la Vallee
James M. Walsh
William Zimrin
hannahqd@cs.brown.edu
Brown University
Computer Science Department
Providence, RI
USA

Kathi Fisler
kfisler@cs.wpi.edu
WPI
Department of Computer Science
Worcester, MA
USA

Shriram Krishnamurthi
sk@cs.brown.edu
Brown University
Computer Science Department
Providence, RI
USA

Abstract

The privacy policies of many websites, especially those designed for sharing data, are a product of many inputs. They are defined by the program underlying the website, by user configurations (such as privacy settings), and by the interactions that interfaces enable with the site. A website's security thus depends partly on users' ability to effectively use security mechanisms provided through the interface.

Questions about the effectiveness of an interface are typically left to manual evaluation by user-experience experts. However, interfaces are generated by programs and user input is received and processed by programs. This suggests that aspects of usable security could also be approached as a program-analysis problem.

This paper establishes a foundation on which to build formal analyses for usable security. We define a formal model for data-sharing websites. We adapt a set of design principles for usable security to modern websites and formalize them with respect to our model. In the formalization, we decompose each principle into two parts: one amenable to formal analysis, and another that requires manual evaluation by a designer. We demonstrate the potential of this approach through a preliminary analysis of models of actual sites.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.4 [*Software Engineering*]: Formal Methods; D.2.0 [*Software Engineering*]: Protection Mechanisms; H.1.2 [*Models and Principles*]: Human factors

Keywords usable security; static analysis; human factors

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509578.2509589

1. Motivation

Creating secure systems is one of the great computing challenges of our time. Thoughtful people recognize that an important part of security is the human in the system, who uses it and hence makes the decisions it enacts. Consequently, usable security—the application of human-computer interaction to security, especially to the design of interfaces—has become a significant area of study. Several influential papers [1, 15, 30, 31] have shown that this concern is not merely academic: interfaces really do affect the behavior of users, and thus play a key role in whether a system is *used* securely (and hence *behaves* securely).

Ultimately, however, these interfaces are (usually) implemented by *programs*. Therefore, it is reasonable to ask whether at least part of the task of analyzing these interfaces can be turned into the task of analyzing the programs that implement them. Whitten and Tygar's "Why Johnny Can't Encrypt" paper [31], one of the classics of usable security, laid down four principles for secure interface design. Quoting that paper:

> *Security software is usable if the people who are expected to use it are reliably made aware of the security tasks they need to perform, are able to figure out how to successfully perform those tasks, don't make dangerous errors, and are sufficiently comfortable with the interface to continue using it.*

This definition hints at the potential interplay between static analysis and usability analysis. Successful performance of a task occurs along program paths, which can be identified with static analysis; whether people take actions through the user interface (UI) that keep them on desirable paths requires

human-factors analysis. Reliable awareness depends both on which information is displayed to the user (static analysis) and whether people notice and understand that information (human-factors analysis). Generally speaking, static analysis can identify conditions that lead to secure outcomes, while human-factors analysis checks whether human users can satisfy those conditions.

This paper explores the potential for using static analysis to aid usability analysis. Our discussion focuses on web applications in which people directly or indirectly share potentially-sensitive data with other users. We look at several usable-security principles for this domain, decomposing each into two parts: one part amenable to automation and the other, residual, part still requiring user-interface evaluation expertise. The hope is that an analysis of the former part can be automated and routinely run during program design and development, leaving the (expensive and specialized) usability analysis of the latter part to be done less frequently and targeted to specific situations that the static analysis has identified as potentially leading to problems.

2. Usable Security in Data-Sharing Systems

Many modern applications—such as social networks, cloud-based filesystems, and conference managers—result in users sharing potentially-sensitive data. Often, end-users are responsible for configuring the sharing settings on data. These systems are strong candidates for usability analysis because users can leak their own or others' data if they misunderstand the security and privacy implications of their interactions with the system. Regardless of whether data sharing is a primary (social network) or secondary (conference manager) goal, applications must provide appropriate protections on data, while enabling users to complete intended (non-security) tasks.

End-user sharing configurations complicate the design process for developers and user-interface engineers. Developers must reason about the entire space of configurations that users *could* make, and when those configurations could change. Individual users' configurations interact with the application's overall sharing configurations. In the case of conference managers, for example, PC members could declare conflicts of interest after reviewing assignments have been made, at which point the software needs to restrict access to existing reviews accordingly. A static analysis could help developers reason about the corner cases and interactions in this complex system. Interface designers must ensure that users have the information they need to make well-reasoned decisions (without overwhelming them); this requires that designers understand the various ways in which sharing occurs in complex systems. Static analysis should be able to help designers identify which of these situations is critical.

Leaks arising from complex and multi-facted sharing policies are not just a theoretical concern: they happen in practice. Consider the widely-publicized incident in which Facebook revealed the sexual orientation of two university students who had taken strong measures to conceal this information [11]. The students friended the president of a homosexual chorus to which they belonged. The president added the students to the chorus' Facebook group (an action which did not require the students' permission because of the existing friends relationship). The owner had configured the group to make membership public, so Facebook published the students' group membership to their friends (including their parents). Our focus is not on whether Facebook's policy is reasonable. Rather, it is whether analysis tools could have assisted user-interface designers in ensuring that the users were aware of the potential effects of their settings (perhaps by informing the owner of the group that setting the group to public would inform its members' friends that they had joined the group).

Ideally, static-analysis support for usable security should not require developers to formulate properties that define usable security. In 2004, Ka-Ping Yee proposed ten principles of usable security [32], many of which focus on whether an application gives end-users sufficient information to make informed decisions about data access and sharing. This paper adapts several of Yee's principles to modern social-sharing sites, then looks at how static analysis can support developers and interface designers in assessing usable security through these principles.

3. Representative Websites

We frame our discussion of static analysis for usable security around five specific web-based sharing applications. Each of the five is an example of a modern data-sharing system, but collectively they represent systems with different broad sharing goals and different authority-granting features. We will refer to these examples throughout the paper to illustrate aspects of our program model, property formalizations, and proposed analyses.

GitHub, a version-control system: GitHub manages repositories of files. Each repository has an owner who can grant other users authority to take from ("pull") or add to ("push") the repository. The owner can also transfer her ownership to another user. GitHub represents systems with simple sharing models: an owner of a datum grants or revokes access to others through a single action.

Facebook, a social network: Facebook users share personal details, photos, and comments with other users. Users share individual data either publicly or with select groups of other users. Groups include Facebook's built-in options for "Friends" and "Friends-of-Friends", as well as user-defined groups (such as "Co-Workers" or "My Football Team").

Facebook's information-sharing model is interesting because it involves roles (through groups), limited delegation (through friends-of-friends), tags, and many different types

of content on which a user might want different sharing policies. The friends relation also induces a two-step authorization process, since certain authority is only granted after one user issues a friend request and another user accepts it.

Google Drive, a service for collaborative editing and file sharing: Drive users create, edit, and share files (such as spreadsheets, presentations, and documents) with self-defined groups of other users. Drive represents systems designed for fine-grained access control (separate policies are often defined per document), as well as systems in which many users may configure access to the same content. File-sharing systems are a common case study in capability-based access-control [17].

Continue, a conference paper manager: Continue is a conference manager (developed by one of the authors) with the usual features for handling conference submissions and reviews. Authors submit papers, reviewers declare conflicts, and chairs assign papers. Authors can delegate work to subreviewers. Continue represents systems in which authority can change according to phases: after submission authors can no longer modify their papers, and they get access to reviews only after the reviewing phase terminates (after which reviews can no longer be modified).

Resumé, a job-application manager: Resumé (a private system developed by one of the authors) manages submission and review of applications for faculty positions. Applicants and reference-letter writers upload materials to Resumé. An applicant can create an account and add materials to it before formally submitting her application. Following submission, members of the department can see the materials but neither the applicant nor the reference-letter writers can edit them further. Department members may submit comments on applications. The applicant never has access to the comments or the reference letters. Applicants and letter writers may email materials to a staff member who has authority to upload materials on behalf of others.

Resumé is typical of systems with rich access-control policies whose rules take effect at different points in a broader workflow; in this sense, it is similar to a conference manager, but the surrounding workflow raises more interesting security and privacy issues. The richness in the policy arises from rules that consult several user and system attributes. The delegation of tasks to support staff also raises questions about least authority and the trust boundaries within which software systems get deployed.

4. System Traits that Impact Analysis

Several traits of our sample systems have implications for usable-security analysis. Some traits are relevant for building models that help identify potential security and privacy flaws (the static-analysis side), while others affect usability decisions (the human-factors side). Thus, system characteristics affect how we state and formalize principles for usable security analysis. Enumerating them helps scope our work.

Articulating these characteristics also helps clarify the relationship between our versions of the principles and Yee's. When Yee developed his principles, massive-scale, cloud-based, web-based applications such as our benchmark systems were nascent or non-existent. His principles thus implicitly codify the expectations of closed, corporate systems; many of his underlying assumptions no longer directly apply to social sharing-based applications. Readers interested in seeing how our principles (presented in Section 7) differ from Yee's versions can align our summary descriptions of each principle with those from his paper [32].

4.1 Massive Scale in Users and Data

Github, Drive, and Facebook manage massive numbers of users and amounts of data; in Facebook, users and data are richly connected. At these scales, users cannot manage authorities at the level of individuals. All three systems support user-defined groups and multiple classes of data; users configure access at the level of groups, as in a role-based access-control (RBAC) policy. Simply seeing these systems as instances of RBAC, however, misses the more critical point that RBAC can be as much a problem as a solution in these systems. In an effort to control sharing, users might create more groups than they can manage at a cognitive (or temporal) level. To mitigate that, users often choose to violate least-privilege; they share with a superset of their desired audience to save the hassle of creating and managing yet another group. Attempts to reason about end-user sharing must account for the over-approximation of groups and other forms of labeling.

Implications: Massive scale in users and data does not significantly impact static analysis: only a handful of users and data should suffice to identify problems in program logic, so these scales should not induce state-space explosion. Massive scale in users and data does, however, have significant consequences for user-interface design. As connections between users and data grows, users are required to make decisions about more information than they can manage on a cognitive level. Interface designers must compress this information, but without sacrificing users' abilities to understand security and privacy consequences of their decisions.

4.2 Data Sharing versus Data Protection

Facebook and other social networks have different goals than many more traditional data-sharing systems in that they emphasize sharing data rather than protecting it. This is in part due to the type of data being shared. Users on social networks are often sharing data they consider non-sensitive, such as photos from public events. However, while most Facebook users want to share much of their data with a wide selection of users, they still expect some "reasonable" level

of privacy or protection. What constitutes a reasonable level of protection may depend on the perceived sensitivity of the data in question. For example, it may be acceptable for photos to be widely disseminated, whereas access to personal messages between users should be much more tightly controlled. Different social networks provide different mechanisms for users to control their privacy levels over various types of data; some provide more usable security than others. A useful analysis must therefore be able to encompass the range of sharing postures of social networks.

Implications: This characteristic suggests that usable security analysis cannot fall back on standard metrics such as "least privilege" to find sharing violations. Rather, developers need analyses to help identify what sharing does occur. Developers can then assess whether that sharing is "reasonable," and whether the interface makes that sharing apparent to the user. Analyses that triangulate how different forms of data are shared within the same system may also be important to help developers assess whether data that are more sensitive are indeed more restricted.

4.3 Delegation and Privacy

All five systems support some form of delegation (friends-of-friends in Facebook, granting administrative privilege in Github, etc). By design, users who delegated authority no longer have sole control of others' access to their data. In the absence of control, however, questions arise as to whether owners should have knowledge of how their data is accessed and managed. This rapidly gets into privacy considerations: in Facebook, for example, knowledge of how data propagates might effectively expose a user's list of friends. The naïve approach of always allowing a user to know about who has access to their data thus may well violate another user's privacy. This suggests that analyses will need tunable parameters about which situations constitute or violate privacy.

Implications: Delegation expands the space of users with authority over data; as such, static analysis should be particularly useful in identifying corner cases of authority that arise from delegation. On the human-factors side, interfaces must help users understand how delegations—which they should control—affect authority over their data. Static analysis can help developers compute the systems-level implications of delegation (such as the implicit delegation in the Facebook example in Section 2).

4.4 Privacy Across Organizational Boundaries

Resumé and conference-management systems raise privacy and confidentiality concerns. In general, some details of how an organization performs certain tasks should be withheld from users who have provided data, which in turn limits users' access to information about who sees their data. (In a conference, authors are expected to not know who reviewed their papers.) Other types of organizations may have different data-boundary requirements. Health-care policies, for example, are subject to transparency regulations that require releasing some of this information to users.

Implications: Information about organizational boundaries can often be codified such that static analysis could help check whether systems respect intended boundaries. A combination of organizational policy and human-factors analysis must determine how much of the boundary decisions should be accessible to users.

4.5 Workflow-Based Systems

In systems that support complex workflows, authority can vary with the phase of a workflow (e.g., conference reviewers may not modify reviews after decisions have been sent out), or change as a result of user actions (e.g., a reviewer can only read the reviews of others after they have submitted their own review). This raises questions about what access, control, or information users should retain across phases, and also what happens once they lose effective control of data such as job applications.

Implications: These questions affect modeling ownership of resources: does an owner retain rights to a resource even when the system no longer provides control over access, or should ownership somehow imply a degree of access? Regardless of how a given system approaches these issues, the user interface must make clear to users how their control over their data may change with time. Static analysis can help determine how permissions align with workflow boundaries, and help identify implicit workflow boundaries that may need to be communicated to users. Workflow systems are also interesting because people often circumvent security measures that interfere with completing needed tasks. Static analysis can identify whether there are different paths that complete tasks, but that have different security implications.

4.6 Ascribing Ownership

Interpretations of ownership of "a user's resources" must account for *tagging*, a social network feature in which one user (or an algorithm within the social network itself!) associates a resource or attribute with another user. In the interest of privacy, tagged users should have revocation access over the tag (i.e., the should be able to *untag* themselves) and be notified of tagging.

Implications: Tagging shows that the traditional notion of ownership is not sufficient for modeling or reasoning about data sharing. The interface should make clear the consequences of tagging to both the tagger and the tagged user. In addition, tagging should behave consistently across data types within a system. Static analysis could help a designer guarantee this consistency.

5. Requirements on a Program Model

The static-analysis component of usable security analysis will need to reason about how authority over data changes

within a running system. This reasoning requires a suitable program model. To explore what information such a model demands, consider the following simple usable data-sharing principle (from Yee): it checks whether data owners control other users' access to their data:

> *Access to a user-owned datum is granted or revoked only with permission of the owner.*

The challenge in formalizing this statement lies in defining "permission of the owner". In many systems, users do not grant permission through a single action with that express purpose. Assume that a Facebook user configures a photo album to be shared with her friends. If she issues a new friend request that is later accepted, the new friend will be able to see the album. The album owner did not directly edit a configuration such as an access-control policy; in fact, the access-control policy didn't even change in order to grant the new friend access to the album. This indicates that our program model must encompass more than just an access-control policy.

Three aspects of this example are particularly interesting when we consider how to reason about users granting authority to one another:

- Multiple actions from multiple users were required to grant the new friend access to the album: issuing the friend invitation (by the album owner), and accepting the invitation (by the new friend). Thus, **enabling new permissions is a multi-step process**. In the model, this will manifest in access-control rules with conditions beyond simple RBAC-style role/action/resource triples.
- The action that corresponds to the owner granting permission (here, issuing the invitation) is not necessarily the last user action that occurs before the new permission is active on the website. Thus, **permission-granting actions may be temporally separated from activation of permissions**. Our model therefore needs to capture the sequences of actions taken in the program, as well as information on how users' actions update the conditions referenced in access-control rules.
- User actions may have implicit consequences for data sharing. The link or button to invite a friend says nothing about sharing the album: sharing is an implicit consequence of adding a friend. Users are responsible for understanding what information new friends will be able to access. Thus, **some aspect of the model and analyses must connect user actions with data-sharing consequences**. The model must capture the consequences of actions and perhaps information about how the interface presents these to users; analyses will need to consider whether users understand actions' consequences.

How permissions are granted *across program actions* lies at the heart of all of these issues. This suggests that an effective program model for our analyses must allow ready access to the permissions in a system, the conditions under which permissions are granted, and the program actions that set those conditions. We propose a model that captures these interactions through rich access-control policies and the programs that use them. This model resembles one we developed in prior work [8], but is adapted to capture elements of client/server applications that are relevant to usable-security analysis.

6. Our Program Model

Intuitively, we base our analysis on a model of client/server systems in which users access shared data stored on the server, and initiate operations through elements on (web) pages. At its core, our model views a web application as a transition system augmented with access-control policies and multiple web pages per client. An informal description of the model suffices to understand how to apply static analysis to usable security. This section therefore describes the model components carefully but informally, with formal versions of each component following the informal descriptions. The informal version should suffice to understand the analyses in Section 7.

6.1 Access-Control Policies

Section 5 argued that usable-security analysis requires reasoning about how permissions on data evolve through system execution. Access-control policies capture rules under which users may act on various data. For our purposes, we need a policy model that captures not only which users may act on which data, but also the conditions that must be met to enable such access.

As an example, consider two sample rules from conference managers. Suppose that during the review phase, a user can read a review only if (a) the user is assigned the paper, and (b) the user has already submitted their own review for it. Such a rule would be:

$$\begin{aligned}&\text{Permit}(u, r, readRev()) \textbf{ if}\\ &\quad \exists r_2, forp : r \neq r_2 \wedge Review(r) \wedge Paper(forp) \wedge\\ &\qquad\qquad Assigned(u, forp) \wedge\\ &\qquad\qquad SubmittedReview(u, r_2, forp) \wedge\\ &\qquad\qquad \exists u_2 \neq u : SubmittedReview(u_2, r, forp)\end{aligned}$$

The predicates *Assigned* and *SubmittedReview* in the rule body capture the conditions for granting permission to read the review. The contents of each of these predicates are set through user interactions with the application: when a reviewer submits a review, for example, the application stores a tuple in the *SubmittedReview* relation. Looking ahead to static analysis, the policy cleanly captures conditions for access, while a program analysis would identify paths that satisfy those conditions.

As a second example, the following rule allows a program chair to upload a review on behalf of a reviewer. The action (submitting a review on behalf of a reviewer) requires aux-

iliary data as arguments: who the review is being submitted on behalf of (pc) and what paper it is reviewing ($forp$).

$$\text{Permit}(u, r, submitRevFor(pc, forp)) \textbf{ if } Chair(u) \wedge reviewer(pc) \wedge Paper(forp) \wedge Assigned(pc, forp)$$

We assume that each web application has an access-control policy comprising rules of the style shown here.

Formal Model Formalizing access-control policies requires formalizing the data domains and relations referenced in policy rules. These data and relations capture the informational core of a website. Formally, we bundle the data, relations, and policy into a model of applications as follows:

DEFINITION 1. *An* Application *contains:*

- *A set D of* Domains*, representing system-specific information of relevance to the application.*
- *A distinguished element* Users $\in D$*, representing potential users of the system.*
- *A set $R_1, \ldots, R_n$ of* Relations *maintained by the application. Each relation has a type $D_1, \ldots, D_k$ where each $D_i \in D$.*
- *A set* Data $= D_1 \cup \ldots \cup D_j$ *where each $D_i \in D$. This distinguishes domains corresponding to securable data.*
- *A set Act of* Actions*. Each action has a type $D_0, \ldots, D_n$ where each $D_i \in D$ and $n \geq 0$.*
- *A list of* Access-Control Rules *of the form*
  Permit$(u, d_0, act(d_1, \ldots, d_n))$ **if** $\exists \overline{v} : \phi$
  where act is an action of type $D_0, \ldots, D_n$ and ϕ is a conjunction of terms over u, $d_0, \ldots, d_n$, $\overline{v}$, and the domains and relations in A.

Our definition of actions and the form of access rules are closely linked. Permissions typically capture actions of a single user on a single resource, but actions may require multiple inputs. Our action model assumes that the primary resource is the first argument to the action; the access rules separate this argument from the others. Rules may need to reference data on the server that are not inputs to the action; the existentially-quantified variables capture these.

6.2 Servers

Servers store data and relations on data (including attributes of data, captured as unary relations). We assume that servers have a subset of data called Users which captures the set of users of the corresponding application. For instance, in the case of GitHub, the server would store sets of Files and Folders, as well as a relation such as Owns (on Files $\times$ Users). Servers contain sufficient information to evaluate access-control rules. Intuitively, a user has the authority to perform an action if the body of some rule permitting them to do that action evaluates to true under the domain and relation contents in the server.

Formal Model The server for an application contains actual data elements and tuples corresponding to the domains and relations defined in the application.

DEFINITION 2. A *Server for Application A* consists of:

- A set *KnownUsers* $\subseteq$ *Users*
- A distinguished element *UnknownUser* $\in$ *Users* that is not in *KnownUsers*
- A set *KnownData* $\subseteq Data$
- For each domain D_i in A, a set DE_i of elements of D_i
- For each relation $R \in D_0 \times \ldots \times D_n$ in A, a set of tuples over $DE_0 \times \ldots \times DE_n$

The data in the server is adequate to evaluate the access-control rules laid out in the application.

DEFINITION 3. *Given a server S, user u, action act, and data $d_0, \ldots, d_n$ from S that respect the type signature of act, S* permits $\langle u, d_0, act(d_1, \ldots, d_n)\rangle$ *if there exists an access-control rule such that Permit$(u, d_0, act(d_1, \ldots, d_n))$ returns true when the rule's formula ϕ is interpreted under the domain and relation contents of S.*

6.3 Clients and Pages

Clients correspond to website users; each user has a set of active *pages*, which intuitively correspond to different tabs or windows in a web-browser. In an actual website, pages contain descriptive text, information stored in the server, and ways to execute actions. Our model of pages abstracts away descriptive text, leaving only bits of server data and summaries of current permissions that would otherwise be embedded within free-form text. Our model also uses a generic concept of Links to cover links, buttons, and other UI elements that execute actions against the server. Links are distinct from informational content (Contents) and summaries of permissions (Permissions) that can appear on a page.

The distinctions among Links, Contents, and Permissions are key to our model. Links represent actions that the user viewing the page is permitted to take (or was, at the time the page was generated). Permissions, in contrast, purely convey information about the state of authority in the system (as it was when the page was generated). In the context of Facebook, the ability to invite a friend would be a Link; that another user, Susie, is allowed to view the page-viewer's vacation album would be a Permission. If a page chose to embed Alice's photo in the user's page, that photo would lie in Contents. In the context of GitHub, a repository would be available to a user as Contents; indications of what kinds of access the user has to that repository, in contrast, would appear as Permissions.

Formal Model Pages containing links, contents, and permissions capture the formal details of clients.

DEFINITION 4. *Let S be a server for an Application. A* page for S *contains*

- *A set of* Links*, each corresponding to an action*

$$act(d_0, \ldots, d_k)$$

  and supplying a concrete value for each d_i from the corresponding domain D_i as given in the type for act,
- *A set* Contents $\subseteq S$.Data*, and*
- *A set of* Permissions *of the form $\langle u, d_0, act(d_1, \ldots, d_n)\rangle$ for $u \in$* Users*, act in Act, and d_i elements that respect the type signature of act.*

As the state of the server may change while a page is open on a client, some of the links on a page may become invalid (meaning that the corresponding client user no longer has permission to execute the corresponding actions). Definition 7 addresses this issue at the point when a user attempts to take an action through a page.

6.4 Transition Function and Page Generation

The state of an application consists of a server and a set of clients (each with their own set of pages). Applications generate pages dynamically in response to users' requests to execute actions. Executing an action yields both a new page to display to the client and an updated server. The generated page must reflect the authority of the user who took the action (and hence receives the new page). Authority is determined by the rules of the access-control policy: a page should only contain Links, Contents, and Permissions that the policy permits for the user receiving the page.

A transition between application states occurs when a user takes a permitted action by clicking a link on a page. The resulting next state replaces the old page with a new one and updates the server state according to the action performed. When multiple users interact with a site simultaneously, the actions of one user can affect the permissions of another: in particular, a user action can invalidate a link that is currently displayed on another user's page. In Facebook, for example, Alice might have a link to view her friend Bob's photo album. If Bob changes the settings on the album to exclude Alice, her subsequent attempt to use her link should fail to access the album. Our model of transitions accounts for such TOCTOU (time-of-check versus time-of-use) violations by checking, at the time a user clicks a link, that they are still eligible to execute that action.

Formal Model Transitions occur when users click on links on their client pages. When a page is generated for a user, every link should correspond to an action that the user is authorized to perform: we refer to such pages as *valid*. Links may become invalid due to changes in the server state between when the page was generated and the action was attempted. Our formal model must capture valid changes, still-authorized actions, and the impact of actions on server state. The following definitions capture these concepts.

DEFINITION 5. *Page P is* valid *for user u and server S iff $P.Contents \subseteq S.KnownData$, and for every $act(d_0, \ldots, d_k)$ in $P.Links$, S permits $\langle u, d_0, act(d_1, \ldots, d_k)\rangle$.*

DEFINITION 6. *Let S be a server, P be a page on client C for user u, and $act(d_0, \ldots, d_k)$ be in $P.Links$. act is* authorized *for C in S iff S permits $\langle u, d_0, act(d_1, \ldots, d_k)\rangle$.*

The model captures the effects of actions in two functions: Act_{pg} maps each user u, action invocation, and server S to a valid page for u and S. Act_{op} maps each user, action invocation, and server to a new server, reflecting how actions change the server state (e.g., adding friends or documents).

Given Act_{pg} and Act_{op}, we can define the transition system for a website. The website state reflects the contents or pages and the server state. A transition occurs when a user takes a permitted action by following a link on a page. The resulting next state replaces the old page with a new one and updates the server state according to Act_{op}.

DEFINITION 7. *An* Application State *consists of a server and a set of clients. Let $\langle S, \{C_1, \ldots, C_j\}\rangle$ be an Application State and act be an action linked to some page P in some C_i. If act is authorized for C_i in S, then the* next Application State *is $\langle Act_{op}(act, S), \{C_1, \ldots, C_j, C_i'\} - C_i\rangle$ where C_i' is identical to C_i except P has been replaced with $Act_{pg}(act, S)$. If act is not authorized for C_i in S, then the next Application State is the same as the given Application State.*

7. Properties and Analysis for Usable Data Sharing

With a model in hand, we now propose and formalize several properties for usable end-user data-sharing over the model. These properties derive from usable security principles proposed by Yee [32], who in turn distilled them from several real systems he had built or studied, some of which are now landmarks in usably-secure system-design. These properties are interesting because they emphasize different ways that a website might help end-users make data-access decisions.

We present a high-level description of each principle, discuss how the system traits from Section 4 affect its interpretation, and formalize the principle to a level suitable for analysis. We then decompose the formalization into two parts: one part that will be computed automatically, and the other requiring manual analysis. From the perspective of this paper, we will call the latter *human-factors residuals*. These residuals are an important part of of the analysis, because any work with respect to usability will be incomplete without consideration of human factors. The residuals thus help designers focus on the aspect of their design that have a bearing on users' ability to operate the system securely. In practice, we would expect the static analysis to occur over the program model from Section 6, while the residual analysis occurs over the actual system implementation.

The formalizations depend on certain definitions about how authority over data evolves in a running system. These definitions are relative to the transition system model described in Section 6. The informal description should suffice to understand the spirit of each definition.

Several of our properties reference user actions that contribute to the granting or revocation of data-access authority. Thus, we begin by defining how actions in the model can affect authority. First, we define what it means for authority to change on a single transition, and on a path:

DEFINITION 8. *A transition from state s to s'* grants permission p *if s does not permit p but s' permits p. Similarly, a transition* revokes permission p *if s permits p but s' does not permit p. Action act is* authority editing *from state s if a transition from s on act grants or revokes some permission. A path of transitions is* authority preserving *if no permissions are granted or revoked along the path.*

Next, we define what it means for a transition to make progress towards granting permission. The computation underlying this definition is local to the access-control policy within the model; had our model captured access controls without explicit policy rules, this definition would have been more complicated. Once we identify actions that make progress towards permissions, we can examine paths to states that eventually grant permissions to determine whether an action effectively constitutes a user's consent. Some of the properties in this section will address whether the effective consent is intentional or sufficiently informed.

DEFINITION 9. *Let s be a state with server S, p be a permission that does not hold in S, act be a valid action for user u in S, and s' be the state (with server S') that would result if u took act from s.*

- act advances p in S *if for some access rule r for p, more conjuncts of r are satisfied in S' than in S.*
- act *is* consent granting for u and p in S *if act advances p in S and there exists a path from s' on which p is granted without further actions from u.*

We now discuss candidate properties and their formalizations against our model. Each of the following subsections starts with an informal description of a property, then formalizes its concepts. Within each formalization, underlined terms represent parameters that must be instantiated relative to data and relations in a particular website, while boldface terms represent human-factors components to the property.

7.1 Change Authority Only with Consent

Grant or revoke authority to others in accordance with user actions indicating consent.

The key concepts to formalize here are "who can consent" and "what constitutes consent". Our definition of consent-granting actions (Definition 9) captures the latter. For the former, we assume that each site has a notion of which users administer or manage each datum: Github tracks repository owners, Google Drive tracks document owners, Facebook users own data that they uploaded. To use this property, a developer would instantiate the *administers* term in the formalization below with references to whichever relations in the data schema reflect this concept. For simplicity, we formalize the property in terms of granting authority; the version handling revocation is similar.

Formalization: *For every state s, every permission $p = \langle u, d, act \rangle$ that does not hold in s, and every path Π from s to a state s_2 in which p first holds: Π contains a transition t by a user u_a such that u_a administers d and either (1) the action on t is consent-granting for p by u_a, or (2) the action on t gave administrative privilege for d to a user u_d, who subsequently took a consent-granting action for p within Π. In either case, u_a* **understood** *that the action on t would grant consent or delegate authority, respectively.*

This first formalization illustrates our notions of property parameters and human-factors components. Parameters should be instantiated in terms of data and relations in the model. Whenever a formalization raises a human-factors component, that component could be ignored (by omitting a clause of the property) during a traditional static analysis, then handled as discussed in Section 9. In this case, the analysis would not attempt to determine if the interface makes clear to the user that they are granting authority, but could tag the granting action as a place that must be carefully considered by a developer or UI designer.

Static Analysis Component: Compute the consent-granting actions for each permission.

Human-Factors Residual Component: For each consent-granting action a for each permission p, assess whether users understand that a would grant p.

7.2 Allow Reduction of Authority

Offer the user ways to revoke others' authority to access the user's resources unless the user previously took an action he understood would (eventually) relinquish that authority.

The previous property checked that revocation of access only happens with the consent of an appropriate user; it did not mandate that users have access to actions that revoke permissions over their data. This property requires the latter. Exceptions to this property can arise in workflow-based systems, as users take actions that are expected to relinquish authority. In Resumé, for example, once a candidate submits a job application, the candidate no longer controls its propagation (though the candidate may retain rights to withdraw the application). Our formalization therefore requires users to maintain access to actions that revoke authority unless they

took actions specifically intended to relinquish their administrative control.[1]

One other subtlety arises in this property: users should have final say as to whether access to their data is revoked. With multi-step permissions, users may consent to an permission, but actions of other users are required to realize the permission. With revocation, an action by an administering user should suffice to revoke another's permission (otherwise, a user might be in a position to prevent having his authority revoked). When access-policy rules all specify permits (rather than permit/deny/not-applicable) and have only conjunctions in their rule-bodies, this requirement simply needs a revoking action to falsify a conjunct in the policy rules. Our formalization, however, uses a more general statement that applies to a wider range of policies.

Formalization: *In every state s, if user u_a administers datum d and s has permission $p = \langle u, d, act \rangle$, then there exists an authority-preserving path from s on which u_a can revoke p unless u_a previously took an action that he* **understood** *would eventually revoke p and that revoking action has occurred. Furthermore, on all paths from s, u_a can reach an action to revoke p, u_a stops administering d, or the action that revokes p occurs.*

Here, the analysis can determine if a consent-revoking path exists, but the designer must ensure that that path is discoverable by the user and that it is clear that taking that path revokes consent.

Static Analysis Component: Compute paths that a user can take to reduce authority, as well as combinations of permissions and states for which no such paths exist.

Human-Factors Residual Component: Confirm that users find their way to paths that reduce authority. Assess whether unreachable authority-reductions were intentional by developers and are acceptable to end-users.

7.3 Summarize Others' Authority

Maintain accurate awareness of others' authority at a granularity relevant to user decisions, without violating others' privacy.

Imagine that a Google Drive user has several folders, each of which is shared with a different group of other users. She needs to share a somewhat sensitive document with some other users. In order to decide whether to upload the new document to an existing folder or to a new one, she needs to review who can access each of her shared folders. This property checks that the user can view this information. Formally, we capture ability to view as having access to a web page that contains the information (through the Permissions component of a page from Definition 4); the path to such a page should not be guarded by the actions of other users.

In systems with massive numbers of users and data, direct presentation of others' permissions could be overwhelming. Furthermore, relevant information might infringe on another user's privacy (such as if a Facebook user wants to know who her friend's friends are before using the "friends-of-friends" setting). Our formalization must account for these nuances.

The key challenge in formalizing this property lies in defining when an existing permission is "relevant to a user's decision". We restrict "user decisions" as "choosing to take a consent-granting action". A narrow interpretation of "relevance" to a consent-granting action for a specific permission would look at the conditions on that permission in the access-control policy: information that affects those conditions would be deemed relevant. Such a definition would not, however, capture concerns about higher-level connections between system data. For example, a user might be willing to share anonymized medical data in a corporate file-sharing system (a private version of Google Drive), as long as the viewers did not also have access to the file with the mapping from anonymous tags to names. Given the challenge of a suitable general definition, we leave "relevance" as a parameter for developers to tailor relative to the data and relations in a particular website.

Formalization: *In every state s, if user u_c can take a consent-granting action for permission $p = \langle u, d, act \rangle$ in s, then for every related permission p, there exists an authority-preserving path to a page for u_c that* **summarizes** *p, unless doing so would* **violate u's privacy**. *Furthermore, on all paths from s, either u_c can reach a page that summarizes p or some action revokes p.*

The formal analysis can determine that all related permissions are reachable by the user. However, these permissions may be numerous or difficult to understand, or they may reveal information about other users. Designers must then ensure that this information is presented to the user in a comprehensible way, and confirm that the privacy of other users is maintained.

Static Analysis Component: Compute the consent-granting actions for each permission.

Human-Factors Residual Component: For each action that grants consent to some permission, assess which of those permissions a user needs to be warned about prior to taking the action.

7.4 Explain Consequences

Indicate clearly the consequences of decisions that the user is expected to make, without violating others' privacy.

As with the principle on others' authority, we focus on users' decisions to execute actions within the website. User

[1] We distinguish between users *relinquishing* their own authority from *revoking* the authority of others, as a system might take different steps to educate users about how these two losses of authority occur.

actions can have one of three kinds of consequences relative to data access: an action can change a permission, an action can consent to changing a permission, or an action can advance a permission. In each case, the effect could be either on the user executing the action or on some other user. Under even moderate numbers of users and data, the set of all potentially-affected permissions for an action could be overwhelming. As a result, our formalization focuses on changes and consent. The formalization is easily adapted, however, to include advances of particular permissions.

As when summarizing others' authority, naïvely showing permissions has the potential to violate others' privacy. Our formalization leaves privacy violation as a parameter, as in the case for others' authority.

Formalization: *For every state s and action act available to user u, u has an authority-preserving path to a page displaying all of the permissions that will change between s and the next state s' of s on act and that do not* **violate another user's privacy**. *If act is consent-granting for u and permission p, then u has an authority-preserving path to a page displaying p.*

Again, a static analysis could determine that a page displaying all the permissions and potential changes exists and is reachable. This page (or set of pages) would then be reviewed by a UI designer to ensure that the information is presented in a comprehensible way and that any consequences of actions that a user might take are sufficiently emphasized.

Static Analysis Component: Compute the permissions that change by virtue of taking each consent-granting action.

Human-Factors Residual Component: Assess whether end-users understand how each consent-granting action will affect the permissions. Assess which of these impacts the interface needs to explicitly convey to users versus which can be left implicit.

7.5 Encourage Least-Privilege

> *Match the most comfortable way to do security-oriented tasks with the least granting of permissions. Match comfortable ways to do sharing-oriented tasks with acceptable granting of permissions.*

A "task" corresponds to a user's goal, such as creating a new document or sharing specific photos with certain friends. While some tasks correspond to atomic actions in a system (such as "create a document"), others correspond to sequences of actions ("upload photos and share them"). We therefore view tasks as paths to goal states within a system. Accordingly, we interpret this principle as asking whether paths that grant the least authority are also sufficiently easy that users will take them (a site that made it difficult to share with individuals instead of everyone would, for example, violate this property).

This principle is interesting for two reasons: first, the human-factors issues are harder to separate out from a core information-based principle; second, massive scale of users and data often makes least-privilege an impractical standard. One could certainly approximate this principle by weighing the number of permissions granted along a path against the number of actions required of a user along that path. In practice, we expect developers will need more nuanced interpretations of "comfort".

Finer-grained formalizations of comfort would also require finer-grained models of pages. Our page model (Definition 4) could be expanded to include link styling information (bold fonts, colors, position on the page, etc). Given that modern web systems codify many of their design choices in CSS stylesheets, models that at least partly draw on UI decisions seem feasible, and worthy of additional investigation.

DEFINITION 10. *A path's* authority-weight *is the total number of permissions granted along its transitions.*

Formalization: *Given an end-user task, the* **commonly followed** *paths that accomplish the task should be least-authority-weight paths. Furthermore, every* **comfortable path** *to a goal state for a task yields* **acceptable authority** *for that task.*

The static analysis can determine the least-authority-weight paths for completing a task. Designers must then determine that these paths are comfortable. The tool can also present alternate paths, and designers can assess that there are no comfortable paths that grant an excess of authority.

Static Analysis Component: For each end-user task that the application supports, compute the set of paths that accomplish the task from various states, including the collection of permissions that change along each path.

Human-Factors Residual Component: When an application offers users multiple paths to accomplish a task and those paths impact permissions differently, assess whether the UI guides users to take those paths that have the least impact on authority changes.

8. Applicability to the Representative Systems

An important measure of this work is whether it would detect actual errors in real systems. This question has two components: whether our model can reflect errors that occur in real systems, and whether one can obtain instances of our model from real systems. The latter problem is standard in model-based analysis research; we offer no new ideas or insights on it, and thus do not discuss it further. This section focuses on the former question about the suitability of our model. Violations of the principles from Section 7 exist even in widely used and, presumably, well-tested and -designed

systems. Take, for example, two of our representative systems: GitHub and Google Drive.

GitHub offers both public and private repositories. In private repositories, the repository owner must explicitly grant other users the ability to push and pull the repository. The principle stated in Section 7.3 requires that any user with push access be able to view the list of users with pull access. However, this is not the case in GitHub. The repository owner has access to this list, but other users with push access do not. This particular violation could be addressed purely by static analysis, because it is a reachability problem. Either no page with such a list exists, or it is not reachable through links available to users with push access.

In Google Drive, document owners may grant several levels of access to other users: read, comment, or edit. However, edit access has two potential meanings: edit the document, or edit the document and share it with others. The current meaning of edit for a given document is presented in small grey text at the bottom of the document's share page, below the button the owner must click to confirm changes. We argue that this is a violation of the principle presented in Section 7.4, which states that the interface must clearly explain the consequences of their actions to users, because the term "edit" is misleading and inconsistent. Static analysis alone cannot address this issue because, technically, the user interface presents all the consequences of the sharing action to the user. However, the way that information is presented does not ensure that the user sees or registers the consequences of granting another user edit permissions. A static analysis could assist a designer in addressing this issue by recognizing that sharing the document with edit permissions is a granting action for a set of privileges, and flag this action for evaluation by a user experience engineer. The engineer could then easily review the action and the compare privileges it grants against the information presented in the user interface and, hopefully, realize that the interface does not make the consequences sufficiently clear and apparent. They may then choose to rename one or both of the edit levels to distinguish them, or to display the current meaning of edit more prominently.

Examples like these make clear the need for static analysis for secure user-interface development. Furthermore, they reinforce the necessary interplay between formal analyses and interface evaluation. Improving the usable security of interfaces requires providing designers with tools that accommodate both these modes of analysis.

9. Building Tools on This Work

This work strives to provide foundations for useful analysis tools for developers. We want to see our models and properties used to create tools that help developers identify subtle bugs in the complex interactions that underlie modern web-based data-sharing systems. The combination of our model (Section 6) and preliminary properties (Section 7) suggest two broad categories of tools.

First, one might build a verification tool that could verify properties (ours or others') against instances of our model. We have begun building such a tool ourselves within the Alloy analyzer [14]. To date, we have a model of GitHub with repositories as data, basic push and pull operations as allowed accesses, and various actions that affect users' authority to act on repositories. The model differs a bit from that in Section 6, most notably in embedding the access-control policy in specifications of actions (rather than model a standalone policy); this simplifies the state space, though we cannot yet quantify the impact of this modification. Against the embedded access-control model, we have been able to check the "authority with consent" property against this model on traces up to five states in length, with each check requiring a few minutes in real time. Work on this tool is ongoing.

Second, one could build tools that compute data-sharing consequences of model instances without performing verification. For example, one could compute all sequences of user-level actions under which a (kind of) datum gets shared with a (kind of) user. Such an computation could start from the access-control rules, compute the various requirements for permitting an access under those rules, then compute the program and user actions required to satisfy those requirements. Using such an approach against a model of Facebook, for example, one might determine that "A user u will be able to see a photo owned by user o if o sends a friend request to u, u accepts the request, o uploads the photo, and o sets the sharing permissions on the photo to her friends". (Some of these steps could also have been permuted, and the analysis would report these alternatives as well.) Presenting these data-sharing scenarios to a developer might help her identify cases in which data-sharing is not guarded by anticipated user actions, without burdening the developer with stating formal properties. This is in the same spirit of property-free analysis provided in policy-analysis tools such as Margrave [10].

In all these cases, we believe the split between automated and residual components is essential, because they represent precisely complementary strengths. Humans are weak at reasoning about, e.g., *all* paths through a system. However, aspects such as comfort, awareness, etc., are best done through human-subject studies and other mechanisms that involve human intervention (if only for interpretation). Thus, we believe the long-term prospects of designing usably secure systems will depend on a careful division of labor between these complementary strengths of humans and computers.

10. Related Work

Much research has been done on the interplay between usability and security in computer systems. Several researchers have shown that user actions can unwittingly circumvent seemingly sensible security policies and mechanisms. Whit-

ten and Tygar's classic case study [31], as well as other usable-security projects [1, 15, 30], illustrate the challenge to designing security features that people use properly.

Other researchers have proposed design guidelines for usable security or privacy. Lederer, et al. [18] present five design pitfalls when designing interactive systems with privacy implications. Many of their principles resonate with Yee's, and could be similarly supported with static-analysis tools that target residuals. One of their pitfalls explicitly raises the need for security solutions to integrate into, rather than compete with, established workflows. At Microsoft, Reeder, Kowalczyk and Shostack [23] have developed guidelines for developers on how to present decisions to users in a way that encourages them to make secure choices. These guidelines contain more human-factors advice than Lederer's or Yee's, but also hint at avenues for formal tool support.

Yee's original principles [32] lacked the nuances of granularity, delegation, groups of users, privacy, and workflows that appear in our informal property statements. An interested reader should have no problem identifying which of Yee's principles corresponds to each of our informal properties. Our updating of Yee's statements to modern systems is a minor contribution of this work.

Usable security is often an underlying design goal in capability-based systems. In particular, CapDesk [27], Polaris [26], and ScoopFS [17] co-evolved with or were influenced by Yee's principles. In capability-based security, a user's access is embodied in references to objects, rather than in access-control policies. The recent PubShare [25] system demonstrates how tracking transfer of capabilities provides users with fine-grained information about and control over data, in the spirit of Yee's principles. Our system model assumes an explicit access-control policy, rather than capabilities. Discussions of capability- versus access-control-based models frequently contrast the former with access-control lists, which are less expressive than our policies, as ours can reference arbitrary domains and relations.

One could instantiate the relations and policy rules in our model with the details that underlie user-driven delegation in capability-based systems (using a relation to record, for example, that Alice delegated to Bob, Bob delegated to Carol, and that Carol can access a resource until Alice or Bob revokes the access). However, our program model cannot capture the fine-grained propagation of references that occurs through function calls and system-level operations in a capability-based system. Our model updates relations (and hence propagates permissions) on transitions between Application States, each of which is driven by a user-action on a client (Definition 7). Capability-based systems would require a much finer-grained notion of transitions.

At a high-level, the choice between a capability-based program model and an access-control-based model is orthogonal to our main point that usable security should be able to leverage static analysis. That said, we believe a program model based on access-control is better suited to our work. In our prior research on reasoning about policies [10] and their interactions with programs [8], an explicit access-control policy is an essential artifact for formal analysis: many interesting analysis questions can be answered (at least in part) on the policy alone, leaving lighter-weight analyses for the full system model. In contrast, in a capability system reasoning about the policy reduces to the harder problem of reasoning about dynamic program behavior. We suspect the best way to reconcile capability-based systems with our work is to view capabilities as a way to implement a separately-declared policy (at the modeling level).

Cognitive walkthroughs enable usability evaluation without expensive and time-consuming user studies. Walkthrough frameworks propose questions for developers about their systems' expectations of users' mental models and goals. Rieman et al. [24] and Blackmon et al. [3] propose automated tools to support walkthroughs, but their tools automate only the process of conducting a walkthrough. They do not automate the artifact analysis within the walkthrough, which is arguably the promise and purview of static analysis.

Automated formal analysis tools have been applied to specific usability concerns. Both Leveson et al. [19] and Butler et al. [4] proposed design metrics and tool support for detecting mode confusion in interfaces. Curzon and Blandford [7] analyzed usability design criteria themselves against a formal model of human cognitive processes. Other tools formalize user task models for systems [16] or log all user interactions with the system and check whether these correspond to expected usability patterns. Ivory and Hearst [13] provide a detailed summary of projects on the latter.

These earlier techniques focus mainly on the users' interactions with the interface, not the users' interactions *with the underlying system model* via the interface. Our work tackles the latter. A system model is essential for reasoning about usable security: the system controls how permissions are used, while the user ideally controls how they are granted and revoked. Our formalization can be used to check that (a) users have the expected control over how permissions are granted and revoked, and (b) the system applies those permissions in ways that the user expects. Formal modeling and analysis of interface components alone is likely a useful complement, but not a substitute, for our work.

User-Managed Access (UMA) [21] is an OAuth-based protocol that provides users with centralized control of web-accessible resources such as their online personal data. UMA targets usable security in providing users with a single point of specification for their sharing preferences. In a UMA-based system, a web application provides access to information that is hosted on a third-party server. End-users provide centralized policies that regulate release of information from the third-party server. UMA-based applications thus effectively have two servers–the host server and one more inside the web application (storing application-specific data

and state information)—as well as a central authorization manager that stores and manages users' policies.

Extending our program model to support externally-hosted policies and data should be straightforward: Definition 3, which defines which actions are permitted, could consult an external service with a similar type signature on access requests. Extending our model, however, is not the interesting issue. The interesting questions lie in how UMA changes the permissions workflows in applications, what impact that has on program-policy interactions (which are at the heart of our static analyses), and whether UMA architectures demand different principles for usable security than those presented here. For example, if one argues that users can always revoke others' permissions by editing the centrally-managed policy (a task outside of the application's core workflow), then some other principles must govern whether users are able to edit policies to achieve desired behavior in the UMA-based application (whose details the user does not understand, as in our Facebook privacy leak from Section 2). UMA thus poses interesting questions for future work.

Garg, et al. [12] analyze audit logs for compliance with security and privacy policies. Their work uses formal analysis to discharge objective components of the compliance checks, leaving a subjective residual for manual analysis. Our residual tools are in the same spirit, but with a focus on residuals about human-factors decisions.

Several projects have used formal analysis to explore the impact of access-policy settings on data sharing. Besmer et al. formalize access-control policies of social networks with a specific eye towards the permissions of third party applications on those networks [2]. They focus on limiting the permissions of applications so that those permissions align more closely with users' privacy settings. Carminati et al. [5] and Mika [22] use ontologies in the semantic web to design such a formalization. Those works represents user actions as well as user data and relationships between users. However, while they model which actions are available to each user, they do not investigate the effects of those actions or tie them to usability analysis.

Several social-network researchers have developed tools to help end-users manage access to their resources. Cheek and Shehab's tool leverages users' existing social-network contacts to inform policies for other contacts [6]. Wang et al. also incorporate user relationships in order to suggest appropriate privacy settings, as well as offering users fine-grained controls [29]; Fang and LeFevre use limited user input to craft potential privacy settings [9]. These tools are designed to help users navigate the existing privacy controls of social networks. Our work focuses on helping developers and designers ensure that the access-control policy and privacy controls interact as expected.

Wang and Jin propose a system for minimizing the leakage caused by user error in cloud collaboration [28]. Their work relies heavily on the ability to instate company-wide mandatory access controls and to require company employees to adhere to file tagging standards. These restrictions would be infeasible in the context of a social network, where the data is owned and administrated by arbitrary users.

Targeting end-users' control of data-access seems reminiscent of ARBAC [20], a formalism for administrative access control. Including an ARBAC policy in our model would not address the core problem in this paper, which is the connection between user-facing actions and actual permission changes. ARBAC also says nothing about the human-factors issues discussed in this paper.

11. Discussion and Future Work

As this paper has shown, data-sharing systems raise interesting opportunities to apply static analysis to usable security. Usable security is a fascinating and difficult problem because its two goals can sometimes seem at odds, though, as authors like Yee point out [32], sometimes this is an artifact of our interface designs and methods for designating authority. When these two are brought closer together, seemingly conflicting requirements can actually become harmonious.

Our paper has shown how usable-security principles can be formalized against a model of data-sharing applications. The nature of the formalization decomposes these principles into a computation that is amenable to static analysis and a residual question for traditional human-factors analysis. Our formal principles are parameterized over application-specific notions of security and privacy. This enables creation of analysis tools that help designers identify important cases for human-factors analysis without the burden of developing their own formal properties.

Reasoning effectively about usable security demands system models that readily reveal the connections between user actions and authority. This paper argues for a separate access-control policy as part of the system model: the policy rules provide a clear starting point for computing information about authority within a system. Policies for mainstream web applications, however, consult information stored in the system state as well as in the request for access. Accordingly, we propose a fairly rich system model, in which individual states are effectively first-order relational models. We believe such models are essential in order to capture multi-step permissions and other complexities of authority in web applications. The richness of this model also supports our claim that static analysis can benefit usable security analysis, by identifying corner cases that need particular attention to user-interface design.

Our discussions hint at several areas for ongoing work:

- *Security versus privacy:* Security and privacy are widely accepted as different concerns with different norms, but the line between them can be blurry. Principles about "accurate awareness" clearly raise privacy issues. Our current formalizations of the principles leave developers to

flag privacy violations. Extending the system model with a privacy policy could better support this task. Such a policy would affect our definition of valid pages (Definition 5) to not leak private information.

- *Principles based on cognitive factors:* Yee's principles strive to give users information about and control over all of their data. Massive scale, delegation, organizational boundaries, and workflows—common features in systems built since Yee's work—all increase the amount and complexity of such information. Interfaces cannot reasonably present all of this information directly; rather, information gets summarized, abstracted, or staged. This in turn calls for usable-security principles inspired from the user's cognitive perspective, rather than just the permissions perspective. We expect a deeper study of the human-factors literature could suggest interesting principles about issues such as maintaining consistency when compressing presentation of permissions.
- *Administrative Permissions:* The notion of administrative control is our model is coarse: it does not distinguish between different operations on a datum. In practice, administrative controls are finer-grained: someone may have control over who edits, but not who deletes a file. Administrative access-control policies [20] capture these nuances in separate policies from the access controls. We consciously chose to interpret administrative control coarsely in this paper, to allow us to focus on a coherent set of principles. A more realistic model, however, might include an administrative access policy (which would in turn affect the formalizations of the presented principles).
- *Usability Over Time:* Time introduces subtle complications in the management of information. It is tempting to look primarily at the availability and impact of actions that change authority in individual system states. Optimizing for temporally local usable security, however, can make the system less usable over time. In a system for small-scale collaboration within a large set of users, creating many groups (to maintain local security) increases the effort needed later (as users have to search for the right group for their task). Furthermore, there exist studies showing how too many security choices lead users to use systems insecurely [30]. We are, however, not aware of proposed principles that balance immediate and long-term usable security.
- *Accounting for Human Processes and Behavior:* Software tools are often used within rich systems of interactions between people and software. In social sharing systems, many aspects of security lie in the human processes rather than in the software: operating policies, incentives, and trust relationships are good examples. The principles we've discussed focus on usable security within the software. How would the residuals or our proposed tools change if we extended our model with process models of surrounding workflows, or task models of users with different security expertise?
- *Formalizing Presentation:* Web-based applications contain layout and formatting specifications through CSS stylesheets. If our model of pages included CSS-like styling tags on individual data, automated analyses might be able to flag certain formatting concerns, such as a more secure action being presented in smaller, lower-contrast, text than a less secure one. Given the plethora of usability design rules in the literature, it is worth exploring which affect security and which, if any, could be analyzed against the tools and languages used to actually build production web applications.

Acknowledgements

We thank Alan Karp, Marc Stiegler, Mark Miller, and the anonymous reviewers for detailed and thought-provoking comments on the ideas on this paper. The US NSF and Brown University's undergraduate research program partially supported this work.

References

[1] A. Adams and M. A. Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, Dec. 1999.

[2] A. Besmer, H. R. Lipford, M. Shehab, and G. Cheek. Social applications: exploring a more secure framework. In *Proceedings of the 5th Symposium on Usable Privacy and Security (SOUPS)*, 2009.

[3] M. H. Blackmon, P. G. Polson, M. Kitajima, and C. Lewis. Cognitive walkthrough for the web. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 463–470, 2002.

[4] R. W. Butler, S. P. Miller, J. N. Potts, and V. A. Carreño. A formal methods approach to the analysis of mode confusion. In *17th AIAA/IEEE Digital Avionics Systems Conference*, Oct. 1998.

[5] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. A semantic web based framework for social network access control. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 177–186, 2009.

[6] G. P. Cheek and M. Shehab. Policy-by-example for online social networks. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[7] P. Curzon and A. Blandford. Justifying usability design rules based on a formal cognitive model. Technical Report RR-05-07, Queen Mary University of London, Dept of Computer Science, Dec. 2005.

[8] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access control policies. In *International Joint Conference on Automated Reasoning*, 2006.

[9] L. Fang and K. LeFevre. Privacy wizards for social networking sites. In *Proceedings of the 19th International Conference on the World Wide Web (WWW)*, pages 351–360, 2010.

[10] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering*, 2005.

[11] G. A. Fowler. When the most personal secrets get outed on Facebook. `http://online.wsj.com/article_email/SB10000872396390444165804578008740578200224-lMyQjAxMTAyMDEwMjAxODI3Wj.html`, Oct. 2012. Accessed: 08/11/2013.

[12] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *Proceedings of CCS*, 2011.

[13] M. Ivory and M. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4), Dec. 2001.

[14] D. Jackson. *Software abstractions: logic, language, and analysis - revised edition*. MIT Press, 2012.

[15] M. L. Johnson, R. W. Reeder, S. M. Bellovin, and S. E. Schechter. Laissez-faire file sharing access control designed for individuals at the endpoints. In *Proceedings of the New security paradigms workshop*, pages 1–10, 2009.

[16] N. Kamel and Y. Ait Ameur. A formal model for CARE usability properties verification in multimodal HCI. In *IEEE International Conference on Pervasive Services*, pages 341–348, July 2007.

[17] A. Karp, M. Stiegler, and T. Close. Not one click for security. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, SOUPS '09, 2009.

[18] S. Lederer, J. I. Hong, A. K. Dey, and J. A. Landy. Personal privacy through understanding and action: five pitfalls for designers. *Personal and Ubiquitous Computing*, 8(6):440–454, 2004.

[19] N. Leveson, L. Pinnel, S. Sandys, S. Koga, and J. Reese. Analyzing software specifications for mode confusion potential. In C. Johnson, editor, *Proceedings of the Workshop on Human Error and System Development*, pages 132–146, Mar. 1997.

[20] N. Li and Z. Mao. Administration in role-based access control. In *Proceedings of ASIACCS*, 2007.

[21] M. P. Machulak, E. L. Maler, D. Catalano, and A. van Moorsel. User-managed access to web resources. Technical Report CS-TR-1196, Newcastle University, Mar. 2010.

[22] P. Mika. Ontologies are us: A unified model of social networks and semantics. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1), 2007.

[23] R. Reeder, E. Cram Kowalczyk, and A. Shostack. Helping engineers design NEAT security warnings. In *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*, July 2011.

[24] J. Rieman, S. Davies, D. C. Hair, M. Esemplare, P. Polson, and C. Lewis. An automated cognitive walkthrough. In *Proceedings of the SIGCHI conference on human factors in computing systems*, 1991.

[25] M. Stiegler. PubShare: An example of rich sharing (YouTube demo). `http://www.youtube.com/watch?v=LJ8FPM1_uTA`. Uploaded: 02/21/2012; Accessed: 08/11/2013.

[26] M. Stiegler, A. H. Karp, K.-P. Yee, and M. Miller. Polaris: virus safe computing for windows XP. Technical report, HP Laboratories Palo Alto, December 2004.

[27] M. Stiegler and M. Miller. A capability based client: The DarpaBrowser. Technical report, Combex Inc., June 2002.

[28] Q. Wang and H. Jin. Data leakage mitigation for discretionary access control in collaboration clouds. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 103–112, 2011.

[29] T. Wang, M. Srivatsa, and L. Liu. Fine-grained access control of personal data. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 145–156, 2012.

[30] T. Whalen, D. K. Smetters, and E. F. Churchill. User experiences with sharing and access control. In *CHI Extended Abstracts*, pages 1517–1522, 2006.

[31] A. Whitten and J. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th Usenix Security Symposium*, Usenix, 1999.

[32] K.-P. Yee. Guidelines and strategies for secure interaction design. In L. F. Cranor and S. Garfinkel, editors, *Designing secure systems that people can use*. O'Reilly Media, 2005.