A World Wide Web Without Walls

Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, and Michael Walfish (MIT CSAIL)

Abstract

Today's Web depends on a particular pact between sites and users: sites invest capital and labor to create and market a set of features, and users gain access to these features by giving up control of their data (photos, personal information, creative musings, etc.). This paper imagines a very different Web ecosystem, in which users retain control of their data and developers can justify their existence without hoarding that data.

1 Introduction

The set of companies chasing the Web 2.0 promise—acquire, control, and then "monetize" your users' data—continues to mushroom. Yet, users get *less* choice than they should. First, having entrusted her data to a Web application (e.g., Flickr for photo sharing), a user is generally "stuck": migrating to another application is hard, and incorporating third-party modules is impossible. Second, new applications must acquire a critical mass of data from scratch. This barrier to entry is high and diminishes the menu of choices for users. Third, users cannot choose what Web applications *actually* do with their data: the much-heralded "privacy settings" of certain Web applications do not come with an enforcement mechanism to prevent error, greed, or malice from leaking photographs, "friend lists", or private blogs. That such calamities will not happen is something that a user must trust—*for every Web application that she uses*.

While this arrangement benefits Web applications that control valuable data, we believe that the status quo is neither optimal nor fundamental. Indeed, our purpose in this paper is to propose a very different platform and concomitant ecosystem for the Web, called the *World Wide Web Without Walls* (W5). What should W5 look like? The above laments suggest the following desired properties:

Decouple applications from data . . . On the Web today, data are bound to applications. For example, as mentioned above, Flickr users are "stuck" with Flickr. As another example, to offer novel social networking features, a new application must acquire users, learn a rich set of connections among them, *and* develop the novel features. Moreover, sharing data among applications is hard.[1]

Ideally, Web applications would mirror the positive aspects of the desktop model. Specifically, new applications should be able to use existing data easily, if the owner of the data consents. For example, users should be able to select a photo cropping module from a set of contributions by independent developers, just as many people choose their text editor. Conversely, a single application should be able to work on commingled data (e.g., a user's photos, friend lists, blog, and bookmarks), each of which is today the province of distinct Web sites.[2]

. . . and give users control over their data. We mean two things here. First, continuing the desktop analogy, users should have the same control over their Web data that they do over local files. They should be able to do operations like "list all of my data", "delete this file", "move", "back up", etc. Second, users should be able to control exactly who or what sees their data. For example, they should be able to express arbitrary privacy preferences like, "don't sell my friend list".

Minimize the trust footprint. Today, to the extent that users *are* allowed to express privacy preferences, they must do so for each application anew (e.g., Flickr shouldn't expose what a user hides on Facebook). Ideally, a user could express her policies once, trust only one module, and have that module enforce her policies across all applications. One advantage of this "factorization" is that protecting users' data from other users and from external attack requires correctness from only a small number of components. Another is that users can run untrusted software on sensitive data—a key property, given our goal of allowing users to freely and safely experiment with alternative applications.

W5 achieves the above properties with *aggregates*. Internally, an aggregate is a single logical machine that hosts a large collection of applications and commingled data from many users. Each aggregate is supplied by a *W5 provider*. Applications are written by third-party developers, and they run inside the aggregate.

Externally, a user's interface to a W5 aggregate is HTTP. Users connect to their providers via Web browsers, and they see, for example, a `my.w5.com` page with a desktop-like display of their favorite applications and file folders. They use this interface much as they would a desktop PC, running applications, uploading new ones, or managing their files.[3] §2 discusses W5 in more detail.

[1] Facebook applications and "mashups" are steps in the right direction, but they do not meet the desired properties listed here; see §5.

[2] We do not expect today's Web applications to "open up" their databases. Our purpose here is to propose a new platform; its success does not depend on existing providers embracing it.

[3] The internal and external views of W5 are reminiscent of multi-user time-sharing operating systems (with terminals replaced by Web browsers). Indeed, the two face similar high-level challenges, but the details are different.

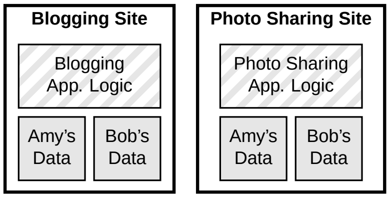

Figure 1: Today's Web site architecture.

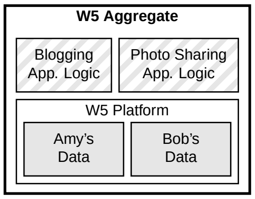

Figure 2: The proposed W5 architecture.

W5 faces a number of challenges, including: How can a W5 aggregate simultaneously protect data from different users, commingle it, and host a bevy of applications that each have access to it? (Isolated virtual machines cannot help because W5 must support multi-user applications, like social networks.) How will users choose from what will ideally be a much larger set of applications and modules? How can W5 support multiple providers? And what economic incentives will draw providers, developers, and users? §3 discusses these questions and others.

We now comment on the relationship of W5 to the status quo, making two points. First, although W5 applications run on a different server infrastructure compared to current applications, the clients are unmodified Web browsers. Thus, W5 can be deployed gradually; the world need not switch Webs suddenly.

Second, one corollary of the W5 architecture is that, if it is even partially successful, the barrier to entry for new applications will be lower than it is today. For W5 not only solves some technical problems for new applications (e.g., protecting users' data), it also solves a marketing problem. Today, for a new application to acquire a user, the user must visit the new site and input data from scratch. Under W5, a prospective user can sign up simply by checking a box or "accepting an invitation". We conjecture that these changes—together with fine-grained competition among software modules and users' ability to run *any* code while still having a protective backstop—will lead to a burgeoning set of Web applications, thereby transforming the market for Web services.

Of course, such changes cannot benefit everyone: existing Web applications do not benefit, and it is *possible* that, by lowering barriers-to-entry, W5 diminishes incentive to innovate. A large-scale cost-benefit analysis is beyond our pay grade (and requires predicting the future). Instead, we simply observe that *W5 yields new options*. It is up to the market whether W5 will supplant the current model, coexist with it, or fail. Nevertheless, we are hopeful, for two reasons. First, W5 is consistent with today's trends: it takes to an extreme (a) commoditization of infrastructure (e.g., [1]) and (b) letting new applications gain access to existing data (e.g., as Facebook does today). Second, in the days and weeks after we first drafted this paper, others made similar observations about the status quo and issued calls for new Web platforms; see §5.

2 The W5 Architecture

Figure 2 depicts the architecture of W5 relative to today's Web (Figure 1). In W5, the underlying platform is factored out, so that different applications can operate on a common platform, sharing data within the same administrative boundary. This architecture yields a Web ecosystem with three entities: *providers*, who supply the platform (i.e., low-level plumbing); *developers*, who write the applications; and *end-users*, who read and write data on the W5 platform through a Web interface. We first discuss these players and then show how W5 yields the desired properties in §1.

2.1 Players

End-Users. End-users interact with W5 sites through Web browsers. When establishing an account, logging on, or configuring her security preferences, she interacts with provider-written code. Otherwise, developer-written code handles her data and requests. For example, a developer-written "home screen" W5 application presents the user with the "desktop" view described in §1, in analogy with today's Web portals (my.yahoo, iGoogle, etc.).

Providers. A provider's job is to supply hardware infrastructure (machine clusters, routers, etc.) and the standard W5 platform. The provider's responsibilities are to secure the infrastructure (physically and against remote exploits) and to maintain it.

The W5 platform is a runtime environment that provides many services commonly used by Web applications. W5 applications run as Unix-like processes on top of the platform and have access to common Unix services such as file I/O and inter-process communication, as well as to W5-specific system calls. The platform provides CPU resources, a file system, a database, and a user login system. Like other time-sharing systems, the W5 platform must enforce per-user CPU, memory, network and storage quotas. The platform and API should be standard, allowing W5 applications to run on any provider's infrastructure.

Developers. Developers get access to the utilities and programming languages supported by the platform. Developers upload binaries, libraries, and scripts to W5 aggregates, and can chain these components to make Web applications. Like today's Unix systems, W5 allows developers considerable latitude in how to engineer their applications. They can be closed or open source; they can run as short-lived helper processes, long-lived server processes, Unix-style pipelines, or plugins for preexisting applications.

Any individual or organization can become a W5 developer, with privileges to run code inside the aggregate.

2.2 Properties

W5's delegation of responsibilities lets it achieve the properties discussed in §1:

Data divorced from applications. As end-users interact with a W5 site, they deposit data in the aggregate, either in the form of regular files or rows in a database. Once inside the aggregate, the data are available to all applications (see below for how data is secured). Any developer can now upload an application or a modification to an existing application that manipulates end-users' data in new and interesting ways.

Untrusted applications. W5's modus operandi is to let large quantities of untrusted code interact with large quantities of sensitive data. Yet, recall that W5 imposes few *internal* limitations on how developers can chain processes together to form applications. Thus, to provide security guarantees, the platform does not rely on fine-grained access control but rather on a *security perimeter* that strictly controls which data leaves the aggregate. This perimeter excludes end-users' clients (e.g., browsers). It includes end-users' data and application code that runs inside the aggregate. To make correct decisions at the perimeter, a given W5 aggregate must track the movement of sensitive data through an arbitrarily complex chain of processes so that the ultimate disclosure decision at the perimeter accurately reflects the data's origin, owner, and destination. We discuss how a W5 aggregate does so in §3.1.

Users control their data. As mentioned earlier, under W5 a user's data lives in one place, so the user should be able to list her data, delete it, etc.

Users also get exact control over how their data is exported (and therefore sold). By default, a W5 security perimeter conservatively allows Bob's data to exit only if destined for Bob's browser. To allow more interesting applications, such as photo sharing with friends, the W5 provider allows end-users to customize their perimeter policies. For example, a user might allow certain types of data (say, vacation pictures) to flow to his friends' browsers but not to his family's browsers.

One might wonder what assurance a user has that providers will offer flexible policy configuration and implement the policy correctly. Our answer is that the providers' *entire purpose* and business is to get these functions right; that, because of the factorization in the architecture, only a small number of components must be correct; and that this factorization requires less trust than the status quo. Moreover, protection and non-interference would presumably be encoded in a contract between providers and users, just as today's online storage service providers do not try to control or profit from the contents of their customers' files.

3 Design Challenges

To realize the W5 platform and its benefits, we must address a number of challenges. We now list the most salient of these, then discuss how we plan to address them (§3.1–§3.5), and then briefly mention other challenges (§4).

Securing data. Any developer can write W5 applications. A malicious developer could publish a W5 application designed to steal, delete, vandalize, or misrepresent users' data. W5 must protect users' data, despite such developers.

Identifying suitable software. Because W5 hosts a large menagerie of applications and modules, users need a way to select for function and trustworthiness (the latter is necessary because while users need not trust much of the software that they use, they may occasionally need to trust small modules not developed by the provider; see §3.1). Such identification mechanisms would also help users avoid *anti-social* applications—those that are not malicious but are still against the spirit of W5 (e.g., an application that stores its output in a proprietary format).

Multiple W5 providers. To ensure that W5 providers have an incentive to give good service, W5 must support multiple competing providers, but what are the trust relationships between different providers, and how can they be enforced? Can applications running on one provider gain access to data residing on another provider?

Client-side information flow. Preventing privacy leaks at the perimeter of the aggregate is not sufficient to protect users' privacy. As in cross-site scripting attacks, malicious applications could leak private data out of W5 via users' browsers. W5 must prevent such leaks.

Incentives. Hardware, bandwidth, and development will make running a W5 aggregate costly. Similarly, developers must invest in writing applications, and users must move their data from other sites. These entities need a reason to bother.

3.1 Securing Data

In §2, we described which properties W5 requires of its underlying platform. An overarching theme is that while untrusted developer-written processes can read and traffic in sensitive data, they cannot freely export it beyond the security perimeter. The questions that we must now answer are: how does the W5 platform implement the security perimeter, and how do users express their policies?

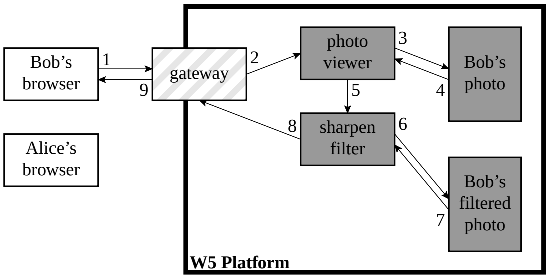

Figure 3: Data flow under default policy. Dark-shaded regions represent "Bob's data" or those processes or files influenced by "Bob's data." The striped region is the provider's application gateway.

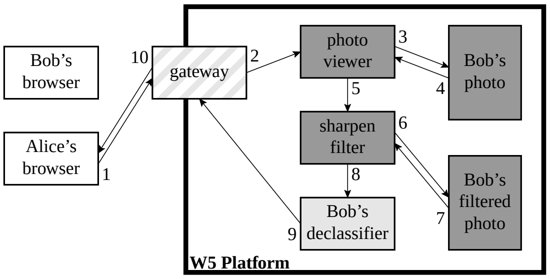

Figure 4: Data flow under a declassification policy. Bob's declassifier, shown as a light-shaded box, allows export of Bob's data to Alice's browser.

To our knowledge, today's popular operating systems do not provide the needed primitives. As a simple counter-example, imagine that Bob runs a new W5 application that processes his sensitive photos. The application performs its advertised feature, with a silent side effect of copying his photos to a hidden yet publicly-readable directory. Meanwhile, the malicious application author runs another module that exports those hidden files to his browser. The platform must prevent this leakage—but cannot do so with popular operating systems technology.

Yet, decentralized information flow control (DIFC) technology [6, 8, 13, 14, 16] can, in a practical way, handle this scenario and, more generally, implement the security perimeter needed for W5. We therefore propose DIFC technology for the W5 platform. One can implement DIFC either within a new operating system [8, 16] or as a modification to an existing one [13].

We now spend some time working through an example that illustrates one application of DIFC to W5.

Privacy protection. In Figure 3, Bob stores a private photograph inside a W5 aggregate and attempts to view the result of passing it through a "sharpen" filter. The "photo viewer" and "sharpen" applications were both contributed by developers whom Bob does not trust. Our goal is to show how DIFC allows Bob to see the result while hiding it from other end-users and developers.

At a high level, all processes (the photo viewer, the filter, etc.) and all files (e.g., Bob's photo) lie inside of the provider's security perimeter. Within this perimeter, the provider computes the transitive closure of all processes and files influenced by any secret data (e.g., Bob's photo). This influence can occur by local file I/O, interprocess communication, or local network communication. The only way for data to enter or exit the perimeter is through a *gateway*. When a process influenced by Bob's secret data attempts to export information, the gateway allows such a transfer only if it is destined for Bob's browser.

In more detail, Bob's browser in Step 1 sends Bob's request to the gateway, with authentication materials (e.g., an HTTP cookie) that prove his identity. In Step 2, the gateway forwards Bob's request to the photo viewer. When the viewer receives Bob's request, it reads Bob's photo from storage in Steps 3 and 4, and invokes the filter process in Step 5. The filter caches Bob's filtered photo in Steps 6 and 7, then sends it to the gateway in Step 8, which sends it to Bob's browser in Step 9.

We assume that the application that originally stored Bob's photo inside the aggregate labeled it, "Bob's secret data." Because the photo viewer reads Bob's photo and later communicates with the filter, the platform regards both as influenced by Bob's secret data. Similarly, because the filter writes a file after coming under the influence of Bob's private data, the platform labels that file equivalently. The gateway allows the transfer in Step 9 because a process influenced by Bob's secret data can send data to Bob.

How might an attacker, Eve, try to steal Bob's photo? Issuing the same request as Bob would not work; the gateway would thwart her in Step 9. Or she could try to upload code that reads Bob's photo (filtered or original) from the file system, but that would not work either: her code, having been influenced by Bob's private data, would be barred from sending messages to her browser.

Declassification. The default privacy policy is too restrictive for Web applications that share data among multiple users. Thus, the W5 architecture allows end-users to make surgical adjustments to the default security policy. First, developers upload applications called *declassifiers* that intelligently disclose private data to end-users other than the owner. By default, declassifiers have no special privileges, but the *provider* supplies a simple Web-based interface that allows end-users to *authorize* declassifiers to act on their behalf. For instance, a developer might upload a "friends-of-friends" declassifier that allows a user's friends and their friends to see the user's data. A user then enables this declassifier via the provider's interface.

Consider Figure 4. Here, Bob authorizes a declassifier to reveal his private data to his friends, Alice being one. Alice authenticates herself to the provider's gateway and issues a request to see Bob's photo in Step 1. Then, Steps 3 through 7 are as in Figure 3. However, in Step 8, the filter routes the photo through Bob's declassifier. The declassifier checks that Bob has authorized Alice as a friend, then removes the "Bob's private data" moniker and applies “Alice’s private data” instead. In Step 9, the gateway sees Alice’s private data, destined for Alice’s browser, which is permitted, and it forwards the data in Step 10.

W5 declassifiers have two appealing characteristics. First, they are agnostic to the structure of the data (e.g., pictures or blog entries) that they are declassifying. Thus an end-user can use the same declassifier for multiple applications. Moreover, users can select which declassifiers they will use, such as a static access control list policy or an application-specific policy based on the application’s notion of friends.

We envision that casual W5 users will authorize only a handful of reputable declassifiers (see §3.2). Such a user’s data security is then vulnerable only to bugs in the provider’s infrastructure and in these declassifiers. While it would be reassuring to eliminate declassifiers and the associated trust, we believe that they are required to support application-specific privacy policies. To establish declassifiers’ trustworthiness, W5 can require them to be open source, thereby allowing users to audit them. Furthermore, the W5 platform can ensure that the audited code is identical to the actual code running as the declassifier agent.

Finally, note that the examples in this section are simplified so that Bob has only one category of private data. Of course, a real system would allow Bob to label his data along many dimensions (e.g., “Bob’s private family data”, “for Bob and his friends only”) and to apply specific declassification policies accordingly.

Write protection. Apart from protecting the *privacy* of its users’ data, a W5 aggregate protects the *integrity* of that data. By default, all data in a W5 aggregate are *write-protected*: the data cannot be overwritten or deleted except by an application with explicit write privileges. A user can delegate the write privilege for some or all of his data, and trusts the delegate to write faithful representations (as opposed to vandalizing his files). W5 can also use a rollback storage system to recover old data in case of accidental or malicious corruption.

3.2 Identifying Suitable Software

One of W5’s primary goals is to give users many options, both for the applications that process their data and the modules employed by those applications. Given the choices, users need some guidance as to which applications and modules they should invoke and, more important, which software they should trust with their export and write privileges. We now propose several techniques by which users can select applications.

Users can establish trust in code based on a code audit or on the developer’s reputation. One can also imagine the emergence of W5 editors, who collect, audit and vet software collections that are compatible and dependable. These editors can establish reputations based on various popularity metrics mined from users’ preferences.

Also, W5 can infer code quality by considering dependencies between modules. This notion is inspired by the PageRank algorithm for Web pages [5]: where PageRank uses the structure of the Web’s hyperlink graph to infer a page’s suitability, a W5 code ranking engine could use the structure of the *dependency graph among modules* to infer a module’s suitability. In the context of W5, code fragment A can depend on code fragment B in two ways. First, A is an application that renders HTML for Web browsers, and the HTML that A outputs embeds a URL that points to an application that uses B’s code. Second, A imports B as a library. Collecting such dependencies over a W5 aggregate will likely yield information about which developers and libraries are widely trusted. Highly ranked applications would receive top placement when users search for new features.

These editorial policies are clearly fallible, but we argue that they are at least as good as those in effect today. Desktop users and Web application builders alike install (and therefore trust) software either because they trust the code’s developers, because the software has achieved some level of popularity, because they audited the code, or because it was endorsed by an editor (such as a trade journalist or a package maintainer for Linux-based systems), or some combination of the four. The W5 platform captures all of these approaches.

We now address *anti-social applications*. These applications do not engage in thievery but might artificially constrain the user for the developer’s benefit. One can imagine applications, in an attempt to entrench themselves, writing out users’ data in a proprietary format, or in a corrupted format to crash other (honest) applications. Nothing in W5 prevents such behavior, but W5 editorial controls can discourage it, just as their analogues do for antisocial software on today’s desktops.

Moreover, we see an encouraging trend toward modularity and interoperability in today’s software landscape. On the Web, many sites syndicate content via RSS and expose simple APIs via XML-RPC. On the desktop, the adoption by many desktop applications (e.g., Microsoft Office) of XML data formats shows that previously isolationist developers are opening up, because users are demanding it. We are optimistic that W5 could tap this trend and that popular W5 applications would conform to convention when storing and transporting data.

3.3 Multiple W5 Providers

Different people may use the same W5 application on different providers, and may need to share data across providers. How does an application that is running on one W5 provider safely read data from another? One approach is for all providers to agree on a single database of users, and to communicate ownership information (e.g., “Alice’s data”) when sending data between providers. Such transmissions require correctness from *both* of the communicating providers. For example, the recipient provider must enforce the same privacy policies as the origin provider. Thus, users must have some control over this process—they must be able to express to their providers which other providers they approve for data exchange.

3.4 Client-side Information Flow

Malicious W5 applications might try to make Web browsers leak data. In this attack, which resembles a cross-site scripting attack, the W5 application returns HTML or JavaScript to the browser that causes it to request, say, an image from a non-W5 Web server. Meanwhile, the *contents of the request* reveal secret data.

To prevent such leaks, the W5 gateway (see §3.1) examines the HTML in outbound Web pages, applying three rules. First, for all embedded hyperlinks, the target must be a W5 application hosted at a known W5 provider. Second, if the hyperlink contains secret data, the gateway verifies that the data’s owner trusts the target provider (see §3.3). Third, the target application must be permitted to receive the data according to the user’s privacy policy.

The gateway must also prevent outbound JavaScript from causing data leaks. Such leaks could happen if the JavaScript, once running in the browser, modified HTML (to induce image requests, as above) or initiated HTTP requests directly. One solution is for the W5 platform to provide a restricted language that the gateway translates to JavaScript. Programs written in the restricted language would be able to create only “legal” hyperlinks and issue only “legal” HTTP requests. An alternate approach is to augment the browser with information flow tracking.

3.5 Incentives

W5 is “backward compatible” with the current Web. However, we must ask why providers, developers, and end-users would adopt it, particularly since many of today’s Web applications derive their value from the data that they control, and, under W5, this asset would not be theirs. In answering this question, we first focus on the “steady state” incentives and then on bootstrapping.

We do not claim to know all of the possible economic models so here just speculate on a few. We think that being a W5 *provider* could be profitable. Commoditized Web services (Web hosting companies, Amazon’s S3 and EC2, and others) are already successful, and if developers attract users to W5, then a W5 provider could charge for hosting users, developers, or, perhaps, for advertising space on pages. *End-users* would presumably be attracted to the privacy, control, and new applications.

Developers might be attracted to the large supply of users (who would allow the developers to profit from advertising on their pages). Also, under W5, developers could contribute free software, just as some developers do today. These incentives mirror those of today’s third-party Facebook developers (see §5). Of course, as discussed in §1 and just above, developers might receive lower *returns* than they do today, but their *costs* and risks would also be lower (because they would have to invest far less in user acquisition; see §2.2). We do not claim to know which model is the better investment for developers; our purpose is to present new options.

For bootstrapping, the requirements are not onerous. A commercial W5 provider could evolve from a research prototype. A developer could—out of conviction, curiosity, or wish to avoid managing and securing his user’s data—build a “killer app” for W5 that does not exist on the old Web. Once the platform began attracting users, a kind of “network effect” could develop (as more users and developers move to the platform, more features arise, thus attracting more users). This development would in turn attract other W5 providers.

4 Next Steps

We have a minimal prototype that uses the Flume [13] DIFC system. We plan to expand the prototype with the solutions described above, and address these additional challenges:

Performance and resource allocation. Processes’ disk, network, memory and CPU usage must be limited, lest rogue applications degrade the performance of a W5 aggregate. Many systems have experimented with resource allocation locally [3, 7] and over a network cluster [10], and perhaps techniques from the VM (virtual machine) literature will be helpful. A more difficult issue is that all W5 applications are allowed to issue database queries, but none should be able to tie up a database. Today’s sites have dedicated “performance tuners” on staff, but no obvious analogue exists for W5: under W5, many authors contribute code, and, besides, even collecting traces for tuning could violate users’ privacy policies.

Debugging. If the W5 platform were to send core dumps to developers, it could wrongly expose users’ data to them. Yet developers need to get some information when their applications malfunction.

Covert channels. Covert channels are a way to leak data without the system’s consent. For example, today’s SQL interface to databases can leak information implicitly [8] and thus needs to be modified under W5.

5 Related Work

Building extensibility into the Web is not a new idea. Among others, the Semantic Web project has long advocated for services to understand each other’s data [4]. More recently, the explosion in “mashups” (sites combining data from other sites) has led to creative Web services. Also, LiveJournal permits its users to customize the site by uploading PHP-like scripts. And Facebook, to the delight of Web commentators and venture capitalists, now allows third-party programmers to run applications "inside" Facebook's service. Finally, Ning lets developers build new social networks on top of common data storage. These developments are innovative and exciting (and make us think that W5 may not be far-fetched). However, as we now describe, none of them provides a general-purpose Web platform that satisfies the properties in §1. Indeed, in these cases, data remains the province of Web services, not users.

Mashups are limited, first, by the API that the "mashee" happens to expose. This API may be narrow as a result of privacy considerations, corporate policy, or simple caprice. Under W5, in contrast, users set policies for their data and decide with whom to share it. Second, mashups lack dependable security for private data so traffic primarily in *public* data. For example, consider a mashup that combines a private address book from MyYahoo with a map from Google. Under the status quo, such a mashup would reveal the address book (both names and addresses) to Google. The recent MashupOS proposal [15] can hide *names* from Google. However, the application uses the Google API to place markers on the map so cannot stop Google's servers from getting the *addresses*. The same application on W5 could generate an annotated map *inside* a W5 aggregate, disallowing export of the address data to the map developers.

LiveJournal's users can customize data presentation but not contribute features. In contrast, Facebook has been billed as a platform that welcomes external contributions. However, Facebook, not the user, is in control of data. Moreover, Facebook applications run on third party developers' servers, which is a vulnerability (the developers could expose users' profiles). In contrast, a W5 user controls exactly the set of clients to whom his data is exported. Like W5, Ning allows third-party developers to create social networks from existing users' profiles, but it does not address the challenges in §3. For example, Ning developers can read and leak users' private data just as Facebook application developers can.

Recently, others have called for Web platforms in which users' data is not proprietary to applications [9, 11, 12]. Though geared mainly to social networks, these authors' motivations resemble ours. However, they do not address the security issues that we do; in particular, they suggest linking together existing databases with HTTP, rather than housing many applications within a security perimeter. Finally, Andreesen issues a like-minded call for general Web platforms [2].

As §3.1 describes, W5 relies on DIFC technology (see [6, 8, 13, 14, 16] and citations therein). Some of these papers use simple Web sites as examples [8, 13], but they do not call for—or address the particular challenges associated with—a new Web platform.

6 Conclusion

Even as Web services expose APIs, they continue to hoard users' data, for protection if not profit. Indeed, it is often assumed that safeguarding data requires *isolation*, either strict (e.g., virtual machines on a server) or loose (e.g., narrow APIs). A noteworthy tension exhibited by W5 is that, in contrast to these trends, it calls for *aggregation* over isolation—yet offers the Web security properties and functional possibilities that are unavailable today.

Acknowledgments

This paper was improved by helpful comments from Jakob Eriksson, Frans Kaashoek, Eddie Kohler, Mythili Vutukuru, and the anonymous reviewers. This work was supported in part by Nokia.

References

[1] Amazon Web Services. `http://aws.amazon.com`.

[2] M. Andreesen. The three kinds of platforms you meet on the Internet, Sept. 2007. `http://blog.pmarca.com/2007/09/the-three-kinds.html`.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, Feb. 1999.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, May 2001.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW*, 1998.

[6] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in Web applications. In *USENIX Security Symposium*, Aug. 2007.

[7] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. *IEEE Annals of the History of Computing*, 14(1):31–32, 1992.

[8] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, Oct. 2005.

[9] B. Fitz. Thoughts on the social graph, Aug. 2007. `http://bradfitz.com/social-graph-problem/`.

[10] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: an architecture for secure resource peering. In *SOSP*, Oct. 2003.

[11] S. Gilbertson. Slap in the facebook: It's time for social networks to open up. *Wired*, Aug. 2007. `http://www.wired.com/software/webservices/news/2007/08/open_social_net`.

[12] Google group on social network portability. `http://groups.google.com/group/social-network-portability`.

[13] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, Oct. 2007.

[14] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, Oct. 1997.

[15] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for Web browsers in MashupOS. In *SOSP*, Oct. 2007.

[16] N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, Nov. 2006.