MIT-CSAIL-TR-2007-043 August 24, 2007

World Wide Web Without Walls

Micah Brodsky, Maxwell Krohn, Robert Morris, Michael Walfish, and Alexander Yip

World Wide Web Without Walls

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

1 Introduction

Although the Web is ever more interesting, it is still—despite the opinions of gushing commentators—fragmented and insufficient. The Web 2.0 ethos—to acquire, control, and "monetize" users' data—has companies scrambling for precious user data, a state-of-affairs reflected in users having to, for example, type in the same romantic, music, and food preferences to half a dozen social networking sites. Yet, despite the fragmentation, users get *less* choice than they should. First, new applications face a high barrier-to-entry: they must acquire from scratch a critical mass of user data (e.g., a new photo sharing application would require a user to retrieve her collection from an existing provider and upload it to the new one). Second, 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 those Web applications that control valuable user 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 eco-system, for the Web called the *World Wide Web Without Walls* (W5). What should W5 look like? The above litany of complaints suggests the following desired properties:

Decouple applications from data . . . On the Web today, data are bound to applications. For example, Flickr users must store their photographs on Flickr, must use only software modules from Flickr, and cannot easily migrate their photographs to a different provider. As another example, to offer novel social networking features, a new application must acquire a set of users, learn a rich set of connections between them, *and* develop the novel features.[1] Moreover, sharing data between applications is difficult (today's "mashups" combine data from multiple sites but are limited to the APIs exposed by the data-owning applications; see §4). Ideally, however, Web applications would mirror the positive aspects of the desktop model. Specifically, new applications should be able to use existing data easily (if the data's owner consents), and applications should be able to work on commingled sets of 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. Users should have fine-grained control over *which applications* process their data. Given the first two properties, a user could, for example, select his favorite photo cropping module from a set contributed by independent developers, just as many people exert choice over their text editor. Moreover, users should control *which policies* govern the use of their data. Today, users can express their privacy preferences only within the constraints allowed by the application (e.g., the policy "don't sell my friend list" can't even be expressed) and have to re-express their preferences for each application (e.g., Flickr shouldn't expose what a user has hidden on Facebook). Ideally, however, users would be able to express idiosyncratic policies and would be able to attach these policies to their data so that the policies applied *across* applications.

Separate data security from other functions. To actually enforce what users express about what applications may do with their data, the platform requires a mechanism that (a) controls applications' access to users' data and (b) is logically separate from the applications. This separation permits the same mechanism to work for many different applications, so protecting users' data requires proper functioning from only a very small number of components. Today, in contrast, the problems of protecting users' data from other users and from external attack must be solved by every application anew.

W5 achieves the above properties with *meta-applications* that host large collections of applications and user data. Internally, a meta-application is a single logical machine on which applications and data are segregated. We imagine there being only a small number of meta-applications, each supplied by a *W5 provider*. (We describe W5 in more detail in the next section.) A key component of this architecture is the mechanism that allows a single meta-application to protect users' data while commingling private data from many users and hosting a plethora of applications that all potentially have access to this data. For this function, our architecture relies on recent advances in Distributed Information Flow Control (see, e.g., [5, 11–13] and references therein).

Indeed, we are not creating technology but rather pirating it (which is ironic, given our goals) to imagine

[1] Facebook, in particular, lets new applications leverage its users, but their approach does not satisfy all of our desired properties; see §4.

[2] We do not expect today's Web applications to "open up" their databases, but our purpose here is to imagine a new platform. The platform's success does not depend on existing providers embracing it.

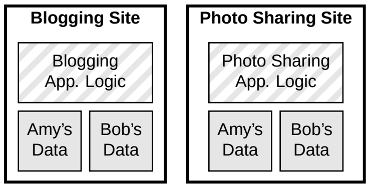

Figure 1: Today's Web sites.

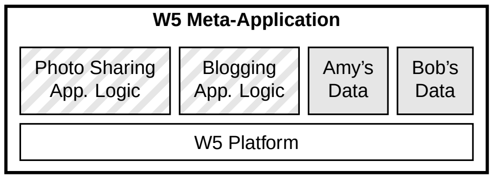

Figure 2: The proposed W5 architecture.

an alternate model for Web applications. Our innovations are the architecture itself (a general-purpose platform and eco-system for Web applications), the properties it upholds (e.g., that users control their data directly), and the new functions that it makes possible (e.g., any developer or user can customize a Web application—and run the customization on the server). However, we do not mean to imply that there are no hard questions, and §3 discusses challenges for W5.

We now comment on the relationship of W5 to the status quo, making two points. First, W5 is not a "clean sheet" design:[3] although W5 "servers" differ from today's, the clients are the same. 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 because W5 is consistent with today's trends. In particular, W5 takes to an extreme (1) commoditization of infrastructure (e.g., [1]) and (2) support for applications that leverage a site's existing data (e.g., on Facebook).

2 Overview of W5 Architecture

Figure 2 depicts the architecture of W5 relative to today's Web architecture (Figure 1). At a high level, the entities in W5 are *providers*, who supply a platform that achieves the properties in §1; *developers*, who write software—applications, modules, modifications to other developers' work, etc.—that run on this platform; and *end-users*, who store their data on the platform and who choose which software interacts with their data, what that software is allowed to do with their data, and under what circumstances the software can reveal the data (to other software or to external clients).

One aspect of this architecture is that, unlike today, safeguarding users' data is not under the control of a bevy of different applications. Instead, the platform protects users' data from other users, from external attack, and from applications. One might wonder what assurance a user has that providers will implement these functions 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 strictly less trust than the status quo. Moreover, "protection" and "non-interference" would presumably be encoded in a contract, just as today's companies that provide storage services do not try to control or profit from the contents of their customers' files.

We now discuss the above entities in more detail.

Providers. We assume a single provider and relax this assumption in §3.3. The provider's job is to supply hardware infrastructure (machine clusters, routers, etc.) and a general-purpose software platform (i.e., an operating system) for which developers write software using common development tools. In building this platform, the provider's only requirements are that the infrastructure be secured (physically and against remote exploits) and that the software platform enforce users' policies.

Because these policies concern, in part, what data may be revealed to other users (e.g., a user may wish to express that the output of a new photo processing application may be viewed only by his roommates and certainly not, say, emailed to the application's author), the provider must establish a logical *security perimeter* that excludes external clients and that allows only "authorized" data to exit. Within this perimeter, the provider's software must track data as it moves inside of a machine, between machines, or to and from persistent storage. Implementing this requirement is possible with recent advances in Distributed Information Flow Control; see §3.1 for more detail.

We imagine that providers would allow users to configure their policies via front-ends like Web forms. In-

[3]It is, however, a paper design; building a prototype is future work.

deed, all of W5 should have DNS and HTTP front-ends so that users can interact with a W5 application with today's Web clients. When an HTTP request arrives at the provider, the provider would read incoming cookies or HTTP data fields to authenticate the user; identify the requested application; and launch the application, perhaps granting it some privileges over the user's data (depending on the policies configured by the user).

Developers. Under W5, developers have much flexibility. First, while they must code to the API exposed by the W5 platform, we expect that API to enable a wide range of functions, including file I/O, communication with other modules, etc. The Unix system call API, for instance, fits the bill and would allow existing software to run on W5. Second, various development models are possible. For example, developers can write closed-source software, in which case they upload binaries to the server which are "executable" but not "readable" (we discuss developers' incentives in §3.4).

Developers can also release source code for their modules, which permits operations not possible today. For example, the platform itself can guarantee that the code with which a user is interacting is exactly the code that the user has audited. As another example, *any* developer—not just the application owner—can customize an existing application by simply "forking" the existing code. At that point, the customizing developer has a pool of users (who need only check a box on a form to begin using the modified application).

End-Users. An end-user interacts with a W5 site much as he would any other. When establishing an account, logging on, or choosing which applications to grant authority to, he is interacting with code written by the provider. Otherwise, developer-written code handles his data and requests on the server side.

Consistent with the desired properties in §1, users can express choice about all aspects of the server-based applications that are interacting with their data. For example, users can choose particular modules from different developers (e.g., "Use developer A's photo cropping module and developer B's labeling module") or even particular versions of software (e.g., "I want to use version X.Y of that Web application, not the latest version").

The user would express preferences like these either via configuration options on Web forms or else by navigating to particular URLs (e.g., developer A's cropper at http://w5.org/devA/crop, or developer B's version at http://w5.org/devB/crop).

Examples Besides the examples inline, we can imagine new applications that W5 enables. For instance, W5 enables arbitrary recommendation engines over private data: Bob can deploy an application that sends him daily e-mail with the 5 most "relevant" photos and blog entries posted by his friends. For an online-dating application, Bob can upload a custom compatibility metric. Bob can also create a "chameleon" profile display that adjusts its output based on the viewer (for instance, to hide his penchant for Sci-Fi novels from love interests). We believe that bona fide Web innovators (unlike ourselves) will find ways to use W5 to greatly improve today's Web features.

3 CHALLENGES FOR W5

W5 raises a number of questions. In this section, we first list the most salient of these, then give preliminary answers (in §3.1–§3.4), and then, in §3.5, quickly mention other challenges that we will need to address. Having not yet begun a prototype, we caution that our initial estimates of the difficulty of these problems may be optimistic, and of course new issues may arise.

Securing data. Bad developers might upload applications designed to steal data, maliciously delete it, vandalize it, or misrepresent it. Moreover, W5 needs to ship private data across the Internet (to Web clients), and write data read from those clients to local files. The W5 platform must distinguish between authorized data transfers and those that compromise a user's security aims.

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 do not trust much of the software that they use, they do 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 nonetheless antithetical to the ethos of W5 (e.g., an application that stores its data in a proprietary format).

Multiple W5 providers. What are the trust relationships between different providers and how can they be enforced? Can users "link" accounts on different W5 platforms, so that their data is mirrored across provider boundaries?

Incentives. Hardware, bandwidth, and development will make running a W5 cluster 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

The W5 provider configures the cluster to enforce basic security policies, forcing all developers and end-users to conform to them. In this way, the provider can safeguard end-users from nefarious developers. The two most important policies, *privacy protection* and *write protection* are discussed below.

Privacy Protection The key security insight behind the W5 proposal (also discussed in other work [5, 7, 11–13]) is that today's Web-based systems cannot separately grant privilege for *reading* of data vs. *exporting* it past a security perimeter. In a system that separates these privileges, however, untrusted software (such as developer-contributed code in W5) can read private data, manipulate it, write it to disk, pass it to other applications, but can neither export it from the system nor enlist another untrusted application to do so on its behalf. The boilerplate privacy policy on the W5 platform, which the provider assigns to all data by default, is that Bob's data can only leave the security perimeter if destined for Bob's browser. Thus, Bob can let applications of any pedigree or provenance read and manipulate his data; the boilerplate policy protects his data from theft, regardless of its movement inside the perimeter.

However, to provide interesting features, applications need leeway to "poke holes" through the security perimeter. For instance, a social networking application should be able to show Bob's profile to Alice but not to Charlie. Unfortunately, the provider cannot supply this logic, since application data (like a list of Bob's friends) is opaque to the provider. However, the provider does give Bob a mechanism for granting export privileges to developer-contributed applications in the form of small *declassifier* [12] agents, and their job is to selectively export end-user data across the security perimeter. If Bob wants to use W5 social networking, he must grant an appropriate declassifier his data export privileges. A correct declassifier in this context will send Bob's profile to users on Bob's friend list and not to others.

Declassifiers are what enable many of the security features of the architecture. In W5 they have two defining characteristics. First, they are agnostic to the structure of the data (e.g., pictures or blog entries) they are declassifying. Thus an end-user can use the same declassifier for multiple applications. Moreover, they are "pluggable" and factored out of larger applications. Modularity gives users users latitude in selecting their security policies, which can range from standard to "idiosyncratic." And because declassifiers are typically much smaller than entire applications, they are easier to audit.

We envision that casual W5 users will authorize only a small handful of reputable declassifiers (see §3.2). Such a user's data security is then vulnerable only to bugs in the provider's architecture, and bugs in his declassifiers. While it would be reassuring to remove declassifiers as a point of trust (and therefore failure), we believe that they are required as described to make the W5 vision feasible.

Write Protection All user data on an W5 cluster is by default *write-protected*, meaning applications running without explicit write privileges cannot overwrite (or delete) user data. A user can delegate the write privilege for his data as he sees fit, but must trust the delegate to write faithful representations of his data (as opposed to vandalizing his files).

Other interesting policies complement these two, such as *read protection*, in which only authorized software can read Bob's secrets in the first place, or *integrity protection*, in which Bob can authorize an application to act on his behalf only if all of its components (such as its libraries and configuration files) are meritorious. They are covered in other work [11].

The W5 architecture is agnostic to the underlying operating system and Web application platform, so long as they accommodate and enforce the described properties. Several recent decentralized information flow control (DIFC) systems suffice: the Asbestos [7] and HiStar [13] operating systems, and the Flume [11] system running on standard Linux. An alternate architecture built with language-level support [5, 12] is also possible.

3.2 Identifying Suitable Software

One of W5's primary goals is to give users ample choice, both for applications that process their data and for the modules employed by those applications. Of course, too much choice can be a bad thing, and users need some guidance as to which code they should invoke and, more important, which code 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 [4]: where PageRank uses the structure of the Web's hyperlink graph to infer a page's suitability, a W5 "code search" 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 cluster can yield information about which developers are widely trusted. Applications written by top-ranked developers would receive top placement in searches by users for new features.

Though these editorial policies are clearly fallible, we argue they are at least as good as those in effect on today's desktops and servers. 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 on Linux-based system), or some combination of the four. The W5 platform captures all of these approaches.

Before continuing, we want to address *anti-social applications*. These applications, though not engaging in thievery, might artificially constrain the user for the developer's benefit. One can imagine applications, in an attempt to entrench themselves, writing out user data in 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 programmer APIs via XML-RPC. On the desktop, browser plug-ins for Mozilla, ActiveX plugins for Internet Explorer, and Microsoft Office's adoption of simplified XML data formats show that previously isolationist developers are opening up, because users are demanding it. We hope that W5 can tap this trend and that the popular W5 applications will conform to convention when storing data and transporting data.

3.3 Multiple W5 Providers

While it is convenient to envision a single W5 provider, competing providers would be best for users and developers, but how might providers peer and share data? One approach is to create import/export declassifiers that synchronize user data between two W5 providers. If an end-user deemed such applications trustworthy, it would give its privileges to data transfer applications on both platforms A and B. Then, whenever the user updated his data on one platform, the changes would propagate to the other. One can imagine more elaborate systems, wherein providers have explicit peering arrangements with other providers. We leave the specifics for future work.

3.4 Incentives

W5 is "backward compatible" with the current Web but 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 user 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 the platform.

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 §4). 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). We do not claim to know which model is the better investment for developers; our purpose is to present new options to developers and users.

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, and users could follow. Once the platform began attracting users, a kind of "network effect" could develop (as more users and developers use the platform, more features arise, thus attracting more users). This development would in turn attract other W5 providers.

3.5 Further Challenges and Future Work

We plan to build a prototype, expand the preliminary solutions above, and address these additional challenges:

Performance and resource allocation. Processes must be limited to reasonable amounts of disk, network, memory and CPU usage, lest rogue applications degrade the performance of the W5 cluster. Many systems have experimented with resource allocation locally [2, 6] and over a network cluster [9], and perhaps techniques from the VM literature can be brought to bear. Similarly, though most Web sites employ database administrators to audit SQL for adequate performance, a W5 cluster would need to welcome SQL from all developers, and therefore must prevent malicious queries from locking the database for all other applications.

Debugging. If the platform were to send core dumps to developers, it could wrongly expose users' data to developers. 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, the SQL interface to databases can leak information implicitly [7] and thus needs to be replaced under W5.

Client-side support. JavaScript is an important Web feature, as well as a source of many security problems, such as cross-site scripting attacks. W5 exacerbates these problems, allowing developers to upload arbitrary JavaScript. W5 could disable JavaScript entirely by filtering it out at the security perimeter, but recent ideas described in MashupOS [10] could extend W5 policies to the client's Web browser.

4 Related Work

The idea of building extensibility into the Web is not new. Among others, the Semantic Web [3] project has long advocated for a Web in which services understand each other's data. The recent explosion in "mashups" (sites combining data from other sites) has led to creative combinations of Web services. Also, LiveJournal permits its users to customize the site by uploading PHP-like scripts. And the popular Facebook site recently introduced a feature allowing third-party programmers to develop applications that run as part of the Facebook service.

These developments are innovative and exciting (and make us think that W5 may not be far-fetched), but none of them seeks a general-purpose Web platform that satisfies all of the properties in §1. In the models above, data remains the province of Web services, not users. LiveJournal's S1 and S2 interfaces are mainly concerned with data presentation and do not allow users to contribute new features. Facebook applications are certainly a notable development in Web services. However, in this case, it is Facebook, not the user, that controls the data. Moreover, these third-party applications run on Web servers external to Facebook, thereby revealing users' profile information to third party developers, creating a vulnerability (being exposed to the users' data, the developers could in turn expose it). In contrast, under W5, a user controls exactly the set of clients to whom his data is exported beyond the security perimeter; the user may wish to exclude from this set the very software developers who implemented the modules that he has invoked.

Mashups lack dependable security for private data and therefore primarily traffic in *public* data. For example, consider a mashup that combines a page of a private address book from MyYahoo with map from Google. Under the status quo, such a mashup would reveal the page of the address book (both names and addresses) to Google. The recent MashupOS proposal [10] can improve security in this example, hiding *names* from Google. However, the application still uses the Google API to place markers on the map, and therefore cannot stop the transmission of the *addresses* back to Google's servers. The same application on W5 could generate the annotated map on the server side, disallowing export of the address data to the map developers. Current mashups are also limited by the API that happens to be exposed by the "mashee", which may be narrow as a result of privacy considerations, corporate policy, or simple caprice. Indeed, the mashee may hide a user's data even when the user wants to share it with a "masher." W5 removes this limitation by allowing *users* to reveal their data to the masher.

Another recent proposal hardens browsers against cross-site-scripting and code-injection attacks [8]. This technique is complementary to the W5 architecture, and can help it address JavaScript (see §3.5).

To establish a server-side platform within which data is both protected and under users' control, W5 providers use Distributed Information Flow Control (DIFC) technology (see [5,11–13] and citations therein). Some of this literature illustrates how simple Web sites can achieve privacy and integrity [7, 11]. These results give us hope that the privacy requirements of W5 can be met, but any realization of W5 must extend this work.

5 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.

References

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

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

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

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

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

[6] 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.

[7] 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*, October 2005.

[8] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *HotOS*, May 2007.

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

[10] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *HotOS*, May 2007.

[11] 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*, October 2007.

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

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