# Securing Web Content

Joakim Koskela[†], Nicholas Weaver[‡], Andrei Gurtov[†], Mark Allman[‡]
[†]Helsinki Institute for Information Technology HIIT / Helsinki University of Technology TKK
[‡]International Computer Science Institute, Berkeley

## ABSTRACT

Security in the WWW architecture is based on authenticating the source server and securing the data during transport without considering the content itself. The traditional assumption is that a page is as secure as the server hosting it. However, modern web sites have often a composite structure where components of the web page are authored by different actors and one logical page contains components collected from disparate servers. Applying a single security policy to a whole page is inadequate. We introduce a new model to protect users from web-based malware. We have developed a new model that uses opportunistic personas to better secure web content by adding integrity and accountability to individual elements. In this paper we present the overall design of the mechanism, as well as details derived from a prototype of the system.

## Categories and Subject Descriptors

C.2.1 [Computer-Communication Networks]: Network Architecture and Design—Network communications; C.2.2 [Computer-Communication Networks]: Network Protocols—Applications

## General Terms

Design, Security, Management

## Keywords

Network architecture, HTTP design, Accountability

## 1. INTRODUCTION

The first web pages were simple: a bit of textual information with embedded hyperlinks coupled with small graphics delivered by a single server. However, modern web pages are complex and a user's experience of "the web" is often developed from myriad components from a variety of providers and systems. For instance, a simple blog post might include (*i*) content from the blogger, including the posts themselves, a set of thematic images and backgrounds, etc., (*ii*) content from the blog hosting services, which could include navigational aids, logos, etc., (*iii*) content from third-parties associated with the blogger or the hosting service (e.g., advertisements) and (*iv*) content from numerous readers who left comments to the given post. The number of actors contributing to a conceptually simple "web page" is potentially enormous. While this scenario points out the obvious flexibility inherent in "the web" it also points out a problem: how do we validate the content from various actors?

Without solid validation components of web pages—or even the entire page—is vulnerable to a number of issues. First, plain HTTP permits in-flight modifications to any content (HTML, images, etc.). A recent study [8] detected roughly 1.3% of clients from 50,000 distinct IP addresses received a modified version of the test web page. Although most changes were benign, there were malcode induced changes as well as injected advertisements. Even if TLS [6] is used to secure HTTP against in-flight tampering there is no way to carefully validate that (say) a blog comment purportedly from an acquaintance is legitimate or is from an imposter, or that a syndicated advertisement does not contain malicious content. Likewise, phishing scams count of replicating entire web pages by copying all the content of some well-known site in the hopes of luring users to provide their credentials—which are then fraudulently used. TLS can be used to validate a particular communication channel, but is not able to provide validation for the myriad of content from various locations.

In this paper, we attempt to secure composite web page using the general notion of opportunistic personas [3] and the flexibility of HTML's widely used `<div>` fields to secure content. The personas notion involves each actor in the system (or the system on their behalf) generating their own pair of public and private cryptographic keys and using that to sign data they send. While there is no fundamental trust in such keys they form the platform for trust to be built via a number of methods: direct knowledge, web-of-trust, track record, etc. Further, such personas can be used across applications. For instance, a user might sign their blog comments and emails with the same key and recipients can leverage two different track records to build trust in the actor. The `<div>` tag defines a division or a section in an HTML document. It is often used to group block-elements for formatting with

CSS styles [1]. The tag supports standard attributes such as `id` that can contains a unique identifier of the element and event-based attributes that define browser behavior (e.g., when the user moves a mouse pointer over a specific area of the page). Given these two building blocks we design a mechanism whereby web pages can be secured by signatures being applied to individual elements of a web page using a new attribute of the `<div>` field.

## 2. SECURING CONTENT

The objective of our research is to develop a framework for introducing accountability into the modern web, which will create a more secure environment. While some of our designs are quite general in nature our goal in this particular case is to introduce accountability without impinging on the way people currently communicate and collaborate using the web. As stated in the introduction, the nature of the web has changed significantly from its early days. Web sites are increasingly composed of content from different actors, and the site is only as safe as its least trusted component. For instance, [5] shows that using bogus types for content can be used to coax browsers into running code that at first blush should not be present (e.g., because the code is in the comments of a PDF file that the browser runs because it is mis-typed). In the absence of some mechanism for accountability the composition of web pages from many actors has opened a channel for malicious activity.

Because of the disparate nature of web content we take the approach of adding accountability to the content. Furthermore, we are simply advocating an accountability framework and leaving policy decisions about how to handle content to users. In other words, we provide a solid foundation for a web browser to understand which objects are coming from which actors and which actors are expected to play a part in the current page's composition. However, how each object or the overall collection of objects is treated is a *policy decision* for the user or their browser. We do not dictate policy, but consider it a tussle-space for users and content providers to deal with.

### 2.1 Page structure

The first aspect of securing content concerns protecting the integrity of the overall web page, which in our framework is done by using a site's private key to cryptographically sign the markup document. This signature is included in a new HTTP header, along with the web site's public key (which becomes its identity). Protecting the integrity of the site structure is not crucial for the securing the individual content components (discussed below), but only for conveying the overall intended composition of those components. Providing over-arching page integrity is similar to the integrity protection provided by existing technologies such as TLS. However, integrity protection of the overall page is only the first step of our mechanism as illustrated in the subsequent sections.

By using an overall page layout that is concretely tied to a given persona the web browser can act more confidently. For instance, a browser could warn users upon entering a new site for the first time and if this is not the first time a user has visited the intended site it may be cause for alarm (e.g., because the site is actually an imposter). Additionally, a browser's password cache could be tied to the key of the web site and therefore the browser would not be fooled into placing default passwords into a form that the user would then submit.

### 2.2 Content components

The basis of our approach to securing individual content components is to add cryptographic signatures to content blocks within the web page. Different blocks can therefore be authored by different personas with integrity and accountability information that directly map the content to the author. This gives browsers and end users the ability to enforce policy decisions on what components to render, omitting possibly malicious components based on their knowledge of the authors.

As explained in Section 1, our framework uses HTML `<div>` elements as containers for these components. The signatures are included in custom attributes of the `<div>` elements, making them renderable by nearly all legacy browsers (which will ignore unknown attributes). (Note, these attributes do break strict HTML conformance.)

We offer two approaches for using `<div>` elements. First, the signature can be interpreted literally, meaning that the signature is made from the actual markup within the `<div>`. This ensures that the content of the `<div>` is precisely what the author intended and allows for no possibility of the site maintainer customizing the look or formatting of the content. This tradeoff may well be useful in some circumstances. The second approach is to add the signed content as a custom attribute to the `<div>`, and use it for *decorating* the `<div>` when rendering the page. The data is provided in a safely encoded (e.g., base64) format, which is decoded, validated and inserted into marked locations within the `<div>`.

Although technologies such as Cascading Style Sheets (CSS) provide a degree of customization without changing the markup of a page, the second approach above gives the maintainer a more freedom in designing the site's look and feel because it divorces the content from the formatting. However, in some cases the formatting of the content is fundamental to the content. For instance, consider an HTML table. In this case, the content author may well wish to concretely impose the formatting to ensure that rows and columns are constructed properly. Our framework treats both options as valid and leaves the decision about which to use to an author's policy.

Finally, we note that signatures may or may not be included in the main markup for external objects. In some cases it may be trivial for the main markup to also include a signature for an object. For instance, signatures could be easily attached to static images that come from the same actor as the markup. Or, content that is signed and given to the actor serving the markup could clearly include the signatures. As noted above, such signatures given integrity and accountability benefits. In addition, such signatures allow for a de-coupling of where the given content comes from trust placed in that content. For instance, content that comes from proxies or content distribution networks can be readily validated. While including signatures for components is clearly useful it is not always possible. For instance, a page might want to include a small weather graphic from some external service, but will not be able to include a signature of the current graphic because the actual content is outside the main site's control. Therefore, there must be an allowance made for including components that do not have signatures. See below for a longer discussion of "partner" sites.

## 2.3 Collecting content

Next we need a method for users to produce signed content suitable for the framework described above. For this, we add browser functionality that cryptographically signs user-submitted data to a web page (for instance via a form) with the user's private key (their persona). This enables the web site to receive signed blocks of data, which it can then re-use as content that is accountable to the originating persona. Such a system requires careful planning in the web site construction so that the submitted data will be readily usable as content (if appropriate—which of course all form input is not). The structure of forms and the names of input elements need to be chosen to match the `<div>`s they will ultimately reside within.

## 2.4 Partnerships

Web sites commonly have a number of *partners*, either for purely economic reasons (advertisements) or for enhancing the browsing experience through sharing of content. When rendering a page which relies on these partnerships, the main site will not be able to provide digital signatures of all the expected content (as discussed above). Therefore, another aspect of our approach is to be able to convey to the web browser the identity of a partner that is expected to contribute components to a given web page. These identified partnerships are indications that within a well-scoped context (the current web page) the web browser should expect some content from some given cryptographic key. How to treat such keys that are previously unknown is a matter of policy. That said, our vision is that these actors are treated independently (to a large extent). For instance, the web browser might want to use identified partnerships as part of their local trust assessment (but likely not to the point of full-blown transitive trust). Finally, we note that partnerships could include annotations that indicate fine-grained expectations of the partner. For instance, that the partner should be serving static graphics and not JavaScript. This can aid the browser in understanding how to treat or quarantine the content.

## 2.5 Trust providers

The use of opportunistic keys in our framework merely provides a foundation on which trust can be built without specifying any particular mechanism for building trust. There are several methods that can be used to develop trust:

- **Track Record:** Using a peer's track record of directly observed good behavior can be a useful way to develop trust. The downside of this mechanism is that bootstrapping can be difficult and compromised peers that were once trustworthy, but have turned malicious can be difficult to detect.
- **Peer Review:** A system for reviewing peers could be developed that let a community vouch for a particular actor may be useful. This is akin to developing individual track records except the community develops a communal track record. This can help individuals bootstrap, but cedes some control to the community, as well.
- **Web-of-Trust:** A global review of peers may be useful, but a further scoping to web-of-trust (a la PGP) may be useful, as well. This retains some aspect of individual control in terms of who "reviews" are coming from, but also may not have the reach of a global pool of reviewers.
- **Trust Databases:** It is possible that security and anti-virus companies may track confirmed malicious actors in some manner.

While we do not dictate a mechanism for developing trust we believe the pros and cons of each of the above approaches suggest that several will be needed.

## 2.6 Security policies

So far, we have only discussed different methods for gaining an understanding of a content provider's trustworthiness, without considering how this information is used, and how it affects the web browsing. The trustworthiness of content can obviously run the gamut from fully trusted to completely untrusted. Trusted content is fairly easy to handle. However, there are different approaches to dealing with untrusted content. The harshest option is to block untrusted content completely, not rendering or loading it at all. Although effective against potential threats, this easily cripples the browsing experience, rendering the web useless. A better option is to *sanitize* the content by turning all or part of the markup into plain text. As for the presentation, we could prevent certain elements which can result in disturbing the layout of the site, such as positioned `<div>`s, while allowing changes in text color or other minor modifications. Further, functional components—such as embedded JavaScript or external Flash applications—could be either disabled completely or allowed to run in a restricted environment. Sandboxing such content is effective but requires close cooperation from the run-time environment. JavaScript, due to its dynamic nature, allows partial restrictions to be applied within the browser. We could for instance disable popups, network access ("Ajax") or browser redirects. Obviously, when content is received with an invalid signature great precaution should be taken, preferably not rendered at all. The particulars of how which of these restrictions are imposed and how they are applied are *policy decisions* and should be set according to the browser preferences (either set by the user or an administrator). We stress that these policies are *local* and not setable by remote web pages or components thereof.

## 3. IMPLEMENTATION

As part of our research, we created a prototype implementation to gain a better understanding of the feasibility, technical challenges and usability of our model. The prototype was implemented as a Mozilla Firefox plugin for Linux, which leverages a *persona daemon*, a system service providing the track-record database and cryptographic operations.

Our prototype model for using track records to manage trust was simplistic. We use a single *trust* value recorded for each persona and the length of the track record. As a Firefox plugin, the development and deployment was considerably easier than actually modifying the browser, but this model did also impose restrictions which meant that all features could not be completed. The prototype has, however, provided a solid initial understanding of how the model we sketched could be implemented and the technical challenges.

## 3.1 The persona daemon

The persona daemon acts as the proxy for the user, maintaining a record of all the *personas*, identity keys, and user encounters, as well as the user's own persona. It was implemented as a stand-alone Python-based daemon accessed by a simple language independent interface through the D-Bus session bus.

The functions offered by the key daemon are:

**Sign ():** Employs the user's persona key to sign the given data.

**Verify ():** Verifies a signature. Although this can be done independently of the key daemon, this function streamlines the use of different types of data normalization methods.

**GetStatement ():** This function provides the application with a *statement* about a persona which consists of a summary of the experiences with the given persona.

**Add ():** Records an event or experience with a persona. This is used to build the track record on which the user and applications base their trust decisions in our prototype. Each addition is accompanied by information regarding the context of the event and how it should affect future encounters. For instance, simply reading a blog post does not indicate we know or trust the author, only that we have seen something authored by that person. Furthermore, we might want to add remarks to the database to explicitly indicate mistrust towards a person. Our prototype uses three parameters to codify our experiences. The *context* parameter indicates the environment in which a persona was encountered (such as web page or blog post). A *trust* parameter indicates whether the experience increased, decreased or had no effect on the trust for a given persona. Finally we encode whether the event affected our overall knowledge of the persona, i.e., whether it increased our *history* with it (the amount of content encountered that was produced by that persona). This does not directly affect trust, but hints of consistency and what we can expect.

## 3.2 Browser implementation

Our prototype was implemented as a plugin for the Firefox browser. The plugin consists of three parts; an XPCOM[1]component that processes the raw HTTP streams, a JavaScript application that alters the rendering and a small user interface for displaying security information and for controlling the security policies of the prototype.

[1]XPCOM (Cross Platform Component Object Model) is the component model used by Firefox to expose much of the functionality (as object components) of the browser to plugins, and allow custom components to be added or replace existing ones.

## 3.3 Page signing

We use custom HTTP headers for both indicating support for the overall mechanism and for carrying page signatures. Support for the scheme is indicated (both by the client and server) with the *X-OP-Supports: true* HTTP header. Page signatures are inserted in the *X-OP-Signature* HTTP header (SHA-1-based RSA signatures in our prototype). Finally, the *X-OP-Key* HTTP header contains the user's public key, base64 encoded.

The stream processor inserts these headers in every request made by the browser, signaling to supporting servers that pages should be signed. The browser plugin, acting as a Firefox *stream decoder*, verifies the page signatures before passing the content to the renderer. The result of this verification is stored in the instance variables of the page window, accessible for the other components of the plugin.

The stream processor also captures and signs the data of HTTP POSTs made by the browser to supporting sites. These signatures are carried in the same headers as used by the server for the page signatures.

## 3.4 Content processing

Our plugin alters the rendering process by executing a JavaScript application after constructing the initial DOM tree. This approach has flaws (allowing possible malicious JavaScript or content to be loaded), but allows post-processing of the page, and alterations similar to what a proper implementation would do.

Our prototype supports only the second of the two methods of securing `<div>`s discussed in Section 2.2—by including both the data and signatures in element attributes. Three attributes are used: *op_key* contains the public, *op_signature* contains the signature, and *op_data* for the actual signed data. Both the signature and key are (as in the HTTP headers) base64 encoded. The data attribute contains the data as URL-encoded key-value pairs as this is the format in which we encode HTTP POSTs. Therefore, data posted by users can be used unmodified in these blocks.

These `<div>`s were rendered by *decorating* the content instead of completely replacing it. After verifying the signature, the plugin uses the key-value data to complete fields within the `<div>`. The target elements are located by matching the keys of the source data to the element identifiers (*ids*). As these are page-unique, we use the parent `<div>`'s ID as a prefix to create an unique namespace within the `<div>`. For instance, a `<div>` with an ID of *msg01* and a data key *title* caused the renderer to replace the content of the child element with the id *msg01_title*. To support better customization of the appearance, the rendering processor adds the result of the verification as an attribute *op_status* to the element. This is used to select a suitable style when rendering. Figures 1 and 2 illustrate this process. Figure 1 displays (truncated for readability) a signed `<div>`, which is rendered (when judged as *trusted*) as shown in Figure 2.

As only the content of certain elements is modified, it allows the site maintainer to control the visual appearance. Using the *op_status* attribute, the site maintainer can provide a style sheet (such as the one illustrated in Figure 3) highlighting in a site-specific manner the trustworthiness of the content blocks.

```
<div id="sdiv5" class="entry"
     op_data="header=Hi&message=Testing+123"
     op_signature="OyjONQTCAR6Mv/sBjRaF.."
     op_key="LS0tLS1CRUdJTiBQVUJMSUMgS0..">
   <div>Posted 11:43:51</div>
   <div id="sdiv5_header"></div>
   <div id="sdiv5_message"></div>
</div>
```

**Figure 1: The HTML source of a signed block.**

```
<div id="sdiv5" class="entry"
     op_status="trusted">
   <div>Posted 11:43:51</div>
   <div id="sdiv5_header">Hi</div>
   <div id="sdiv5_message">Testing 123</div>
</div>
```

**Figure 2: The signed block after processing.**

Before rendering the data values, the plugin *sanitizes* the data according to security policies. In our current prototype, we support only full sanitation (escaping all markup) which normally is applied to all content, although the user can choose to bypass this from trusted personas (i.e., that have a good trust score).

In future versions we plan a more fine-grained sanitation process, where different elements could be neutralized depending on the threat or disturbance they might cause. In addition, embedded JavaScript could be partially sandboxed as explained in Section 2.6.

## 3.5 Control interface

The control interface of our prototype is used to display information about the current page and the signed `<div>`s, as well as to control how the content is rendered. The interface is implemented as a small popup-menu located at the bottom status panel, showing the trust status of the current page, similar to HTTPS indicators.

The prototype uses the key database's statements to classify the page as being either *trusted*, *untrusted* or *invalid*. In addition to this classification, the interface can display a simple human-readable description of the track record, such as *You trust this person, knowing him well (through browsing)*. Although currently discrete and rudimentary, we expect to improve the interface as well as include more advanced trust rating based on the track record. In addition, we plan to include methods for user's to express rich policies (perhaps through additional code).

The controls allows the user to block or completely hide untrusted and invalid data blocks, and choose whether to sanitize content from trusted personas.

## 3.6 External objects

Our approach to external objects (such as media files or JavaScript source code) embedded in a page mirrors our handling of `<div>`s, by adding *op_signature* and *op_key* attributes. Due to how the page rendering in Firefox is structured, we are not able to intercept or prevent these external objects from being loaded. However, we experimented with post-processing images, changing how they are displayed

```
div.entry[op_status="trusted"] {
     background: green; font-size: large;
} div.entry[op_status="invalid"] {
     display: none;
} div.entry[op_status="untrusted"] {
     background: red; font-size: small;
}
```

**Figure 3: A CSS style sheet declaration highlighting the trustworthiness of the content.**

based on the track record of the keys. As with `<div>`s, images can be hidden completely, blocked or displayed with a warning.

## 3.7 Partnering

The partnering scheme was implemented with a custom HTTP header in server responses. The server indicates the partners by adding their persona keys in *X-OP-Partner-n* (where n $\geq 0$) HTTP headers.

As discussed in Section 2.4, how users should react to partnerships is not straightforward. Trust should not be treated as transitive, and partnerships should be seen more as *suggestions*, *recommendations* and *expectations*. In our prototype we use a policy where the partner keys are added to the key database when encountered, but without raising the level of trust in them (affecting only the history length). However, within the scope of the current page, the browser plugin gives these partners the same privileges as the page provider, unless the partners are already mis-trusted.

This model does not provide ideal safety, but it is both feasible to implement and likely to provide a fairly high degree of safety. In future revisions, we will explore finer-grained security policies which would allow us to indicate what to expect from partners and assign privileges accordingly.

## 3.8 Server library

As our mechanism only affects the HTTP headers and HTML content, it can be implemented on servers using server-side scripting (without changes to the HTTP server itself).

We created a PHP library for Apache's HTTP server which automates the page signing process. Applications need only initialize the library with their private keys, and call *flush()* at the end of each page transmission. The mechanism enables the key to be application-specific, instead of site-specific (as in TLS).

## 4. RELATED WORK

Our approach is in some ways similar in spirit to data-oriented architectural notions [4, 7]. However, while previous work has considered such a concept in general our goal is to build better security within the data delivery architecture that is currently deployed.

In addition, our mechanism is similar to the scheme used by OpenPGP to sign either whole pages or individual sections [9]. The OpenPGP mechanism presents syntax similar to our's for validating the integrity of external objects. However, the focus of OpenPGP is only on validating the source and data integrity, without considering how the content is handled. The scheme appears also too rigid (relying on the exact HTML formatting of the data) to be easily adapted by web applications.

Sandboxing individual HTML elements is discussed in [2] and shares ideas with our approach. In particular, the idea of downgrading the privileges of individual content components. However, [2] considers only how elements could be protected from each other and does not consider protecting the user from the content.

## 5. DISCUSSION

The framework we present in this paper has the potential to offer a more secure web environment for users, but there are also downsides, including the additional processing required to sign and verify web pages, increases the size of web pages and the resulting increased load on both servers and clients. Also, the scheme may not work well with certain types of content, such as streaming media.

On the other hand, the mechanism works well with existing protocols, making it possible for servers to simultaneously serve clients that support the scheme and those that do not. Using the indicators in the HTTP request headers, servers can select which clients are sent the secured version, while omitting the signed version—and resulting overhead—for clients that will not use it. Even if legacy clients do get the secure content, the syntax is backwards-compatible and clients will still be able to display a *default*, server-sanitized, version.

Furthermore, giving the client control of the sanitation has an interesting side-effect. Currently user input (such as blog posts) is heavily filtered, removing elements that the site does not explicitly understand. With our mechanism the responsibility of sanitizing and sandboxing can be shifted to the client and therefore we can support much richer interaction. Users can submit content with custom layouts, media components and even embedded applications and as long as the consumer trusts the content author this will be relatively safe. A hosting provider (e.g., blog hosting service) may still wish to sanitize content for consumers who do not advertise the ability to understand our mechanisms.

Another feature is that as the scheme provides both integrity and accountability of the actual content, it enables the content to be distributed in different ways. We do not need to trust the party hosting the content, but are able to use a wide range of distribution mechanisms from public, untrusted, servers to peer-to-peer networks without compromising either the users or content.

Overall, we believe the framework has much to offer. There are still a number of issues that need improvement, especially in the prototype implementation. Fine-tuning the use of the track record and the setting of security policies needs additional work. In addition, handling of the persona keys is inefficient and could be improved with an OpenPGP-like KeyId scheme. The mechanism should also be implemented at a lower level in the browser to fully support all the intended features.

## 6. CONCLUSIONS

Modern websites have a composite structure, often combining content from multiple HTTP servers and many content authors. This opens the door to a number of threats—such as phishing and impersonation attacks—with no easy way to secure the content. We propose a refinement of WWW architecture by applying the notion of opportunistic personas with accumulated reputation tracking for securing web page components. By permitting the content providers and users to sign fragments of web pages with their private key and include the public key as an attribute of `<div>` tag, we can enable the browser to present trustworthy and malicious content differently. In addition, we provide a platform for identifying partners and developing custom security policy. We have sketched the system architecture, as well as implemented a prototype as a plugin to Firefox browser. However, our mechanisms require refinement which we are undertaking as part of our future work.

## Acknowledgments

This research is supported by TEKES as part of the Future Internet program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT). In addition, this work was funded in part by NSF grants CNS-0831780, CNS-0831535 and CNS-0433702.

## 7. REFERENCES

[1] Html div tag. http://www.w3schools.com/tags/tag_DIV.asp.

[2] The <module> tag. http://json.org/module.html.

[3] M. Allman, C. Kreibich, V. Paxson, R. Sommer, and N. Weaver. The strengths of weaker identities: opportunistic personas. In *HOTSEC'07: Proceedings of the 2nd USENIX workshop on Hot topics in security*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[4] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *ACM SIGCOMM 2004*, Portland, OR, September 2004.

[5] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proc. of IEEE Security and Privacy Symposium*, May 2009.

[6] T. Dierks and E. Rescorla. Rfc 5246: The transport layer security (tls) protocol. http://tools.ietf.org/html/rfc5246.

[7] T. Koponen, M. Chawla, B. G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(4):181–192, 2007.

[8] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, Berkeley, CA, USA, 2008. USENIX Association.

[9] J. Willingham. The pgp how'd you do that? page. http://jim.willingham.com/pgphow4.htm.