Jean-Sébastien Légaré*, Robert Sumi and William Aiello

Beeswax: a platform for private web apps

Abstract

Even if a web-based messaging service offered confidential channels, how would users know whether their keys, or indeed even their plaintext, was not being exfiltrated? What if a variety of applications offered confidentiality? How would a user gain trust in all of them?

In this paper we argue that a platform for private web applications is the only practical way for users to gain assurance about the confidentiality claims of a large number of full-featured web-services. We introduce Beeswax, a client-side platform that allows confidential data to be exchanged between users at the behest of an application, through a narrow set of APIs. Beeswax installs in a modern browser to deliver a complete practical solution, from key distribution to isolation of private data from the applications, thereby making an analysis of application code unnecessary. This focuses scrutiny and trust on the platform itself, rather than on all the applications using it.

DOI 10.1515/popets-2016-0014
Received 2015-11-30; revised 2016-03-01; accepted 2016-03-02.

1 Introduction

Recently, users are showing increased reticence in giving up their privacy, enough for some service providers to market their privacy-friendly data policies. For example, chat on Gmail [27], ChatStep [20], and Slack [32] claim that the data that users pass through their service will not be stored or will be stored but not monetized. Since such claims are enforced on the server side, the only assurance that users have is the knowledge that providers have a strong financial incentive to avoid a public breach of these claims. On the surface, web applications like Cryptocat [22] and Subrosa [33] provide more assurance of confidentiality as they use client-side cryptography. However, after having been lured to an application by promises of privacy, how would a user know that the client-side code was not exfiltrating private plaintext or keying material?

We posit here that any application that implements end-to-end cryptography must be considered by the user as being part of his/her trusted computing base: even if the code itself is not intentionally exfiltrating private data, any vulnerability in the code might be leveraged to extract private data. Ideally, the market would offer a large number and variety of privacy preserving applications: messaging apps, photo sharing apps, full-featured social networking apps, webmail clients, etc. If a consumer would like to benefit from the functionality of a handful of applications, he will have to implicitly put all of them in his TCB. To make the issue of trust even more problematic, many important safeguards (such as checking the fingerprint of an application version that has had a thorough open-source review) do not fit with the deployment model of modern web applications as HTML, CSS, and JavaScript are modified frequently and pages often contain dynamic content.

1.1 A Platform Approach

In current private web applications, it is all or nothing: either you don't use the application or there is no isolation of critical code and data. Because of this, every new application adds its entire code base to the TCB. The approach we take in this paper is based on the principle of least privilege: isolate highly privileged code and sensitive data and export a narrow interface to this code. We observe that once this is done, multiple applications can use the same sensitive code base via the APIs without adding to the TCB. We propose a platform approach to building confidential web applications. The goal of our approach is twofold: 1.) to allow for applications that have rich interactions with service-side functionality, and 2.) to reduce the assurance of an ecosystem of such web services to the assurance of the platform code.

In our proposed approach, the responsibility for the overall functionality of a service is split between our client-side platform, Beeswax, and the service provider's code (both client- and server-side). The platform is responsible for, among other things, managing and isolating keying material and for performing standard cryptographic operations on behalf of the application through an API. The application is responsible for the rest of the functionality, including its look and feel, sharing, and distribution of (encrypted) content. For example, the application is responsible for designating DOM el-

*Corresponding Author: Jean-Sébastien Légaré: University of British Columbia, E-mail: jslegare@cs.ubc.ca
Robert Sumi: *idem*, E-mail: rjsumi@cs.ubc.ca
William Aiello: *idem*, E-mail: aiello@cs.ubc.ca

ements as private, e.g., after user action. The platform takes care of accepting user input to a private DOM element and shielding that input from the application. The platform similarly displays data in private elements to the user but shields it from the application. The platform is responsible for providing an unspoofable indication to the user about whether a DOM element is private, and if so, the identities of other users with access to the content of the element.

As a result of this split, the technical community can focus its scrutiny regarding whether a given set of privacy properties are properly implemented onto a single code base. The intent is that focusing scrutiny on a single code base, as opposed to spreading the scrutiny across a large number of applications, will generate greater assurance in the claimed privacy and security properties, e.g., the impossibility of exfiltration of private user data or keying material to the application.

1.2 Adversarial Apps & Other Threats

Our goal is to ensure that users need not place applications that use our platform in their TCB. However, by placing applications outside the TCB we must assume them to be malicious. While it may seem counterintuitive to assume that an application dedicated to preserving privacy should be considered malicious, it would not be unthinkable for a service to lure users into using an application with promises of privacy via the use of a privacy platform, only to subvert the platform to retrieve private data for commercial or nefarious gains.

Thus, the platform must provide privacy guarantees even in the face of an application purposefully trying to circumvent its defensive mechanisms. We assume that an application may attempt to: 1.) exploit a vulnerability in the platform code in order to retrieve data in UI elements designated as private or to extract keying material in isolated storage or 2.) trick the user into entering private data into a UI element that the application has not designated to the platform as being private.

A top level goal is to maintain the integrity of the platform code (e.g., the mechanisms that isolate data in private UI elements from the application) even in the face of attacks against the platform. We discuss our software integrity defenses in detail in Section 3.2. We cannot, of course, guarantee code integrity completely. However, as discussed above, we believe the benefit of a platform approach is to focus the scrutiny of the community on the code base of the platform to maximize the chances of detecting and fixing vulnerabilities.

We assume that the application may attempt to mislead the user about the read/write permissions of a given UI widget via visual trickery, e.g., transparent overlays or rapid changes of focus. We take it as a given that users cannot achieve even the slightest of privacy guarantees without paying at least some attention. The goal of our platform is to provide defenses against active UI attacks assuming only moderate user attention. For example, Beeswax provides an indication to the user in the browser's toolbar about whether the UI element with the current focus is private, and if so, what other users have access to the data in that element. This is detailed in Section 3.3. Short of extremely sophisticated per-user behavioral modeling, as long as some users are sufficiently motivated to protect their privacy from UI spoofing, an application that occasionally mounts such attacks would be identified by the community.

Our platform exchanges messages over the network, via the application and a pub/sub service such as Twitter. Beeswax must thus also protect the confidentiality and authenticity of messages in flight. Beeswax uses well-known cryptographic protocols for such purposes. We do not defend against denial-of-service, but failure to deliver messages would only result in poor user experience, without negatively affecting security. Our platform is immediately deployable, but relies on the PKI trust rooted in the browser for HTTPS communications.

To bootstrap trust in the messaging, we assume users of the platform are also registered users of a pub/sub service. For this paper we use Twitter but other services could also be used. We assume that each user, via out-of-band mechanisms, makes a personal determination of the validity of the binding between the pub/sub (Twitter) account name and a person. We assume that when choosing to share, a user will only use account names that meet some personal threshold of trust. For example, an account name may be considered trusted when it has been shared via a number of channels and the stream of updates is completely consistent with what one presumes to know about the person supposedly bound to the account name. We cannot protect users from spoofed accounts if they put no effort into ascertaining the identities of account holders.

We do not combat attacks on the underlying OS or browser. We consider them as part the platform's TCB: any security improvements to them will be inherited by our platform. In any system implementing end-to-end security, the privacy of a user at one end may be compromised by a breach at the other end. Users of Beeswax must therefore trust the integrity of the systems of the users with whom they interact.

1.3 Beeswax

In this paper we present Beeswax, a platform for developing private web applications. In designing Beeswax, our goal is a platform that:

1. supports the development of rich interactive applications with custom look and feel,
2. provides transparent key management,
3. enables applications that give users control of the permissions of UI elements that the application has labeled as private.

Assuming that selected Twitter identities are trusted, and that the cryptographic libraries we use are implemented correctly and computationally secure, we claim the following, in parallel with the goals. Beeswax:

1. adds only minimal complexity and lines of code to an application and incurs minimal overhead,
2. prevents exfiltration of keying material, and spoofing of identities by an application,

3.a prevents an attentive user from reading (writing) private data from (into) a non-private UI element by providing an unspoofable indicator for the permissions of the UI element in focus,

3.b prevents data entered into a private UI element from being leaked to the application or unauthorized users.

To validate the first claim we build two applications: a privacy-enhanced version of the IRC web client Kiwi IRC [29] that allows encrypted messaging and an encrypted photo-sharing gallery called PicSure built from the ground up. We report on their construction and platform overheads in Section 4.

We cover the isolation of keying material with our platform's API in Section 2.1. We argue for the correctness of our identity and key management in Section 2.2. Isolating plaintext from the application (and unauthorized users) is covered by the implementation, Section 3.1 and Section 3.2. Finally, we describe our defenses against UI-redressing attacks in Section 3.3.

1.4 Overlay versus Platform

An approach related to our own is embodied by the browser extensions Priv.ly [30] and ShadowCrypt [8], which we call overlays. Overlays do not require the cooperation of the web service to which they are applied and so work with existing services. Their aims are similar: a user may protect text by declaring input areas as private. Private posts are submitted as encrypted markers (inline, or external links) which are exposed as plain text by overlaying isolated elements. In its current implementation, ShadowCrypt does not meet this isolation goal. In Section 6 we show it has many vulnerabilities that allow us to craft attacks that retrieve plaintext.

While we believe that it is possible in principle to build an overlay that properly isolates keying material and user content using techniques similar to our own, the overlay approach, in general, leaves unsolved two significant impediments to security and deployment: UI spoofing and key management.

UI Spoofing. ShadowCrypt [8] explicitly excludes UI spoofing from its threat model. Indeed, without hooks into the application state, it is difficult, if not impossible, for a security layer to provide an unalterable indication to the user about the privacy characteristics of a UI element. For instance, the use of coloured borders and floating padlock icons around confidential elements in a page (as used in ShadowCrypt) are not sufficient to protect against malicious UI spoofing.

Key Management. Overlays remain entirely separate from applications, requiring that their key management be done out-of-band of the application. In practice this means that users of an overlay must, for example, email or text a symmetric key for each application stream and friend. Such *ad hoc* key management severely limits adoption of an overlay to all but a few motivated users. Our platform approach, on the other hand, allows keys to be exchanged securely in-band of the application.

In short, a plethora of stand-alone privacy applications are difficult to trust. An overlay will break some functionality, is susceptible to active UI attacks, and requires unrealistic user effort for key management. We posit that tight cooperation between developers and a security platform is necessary to achieve an ecosystem of easy-to-use applications that provide strong assurance of privacy.

2 Architecture

Beeswax is built as a Chrome browser extension. It therefore ties together multiple extension components, as per Figure 1. Below, we briefly describe these components and our usage of them in Beeswax. We also rely on another browser technology called ShadowDOM, which we also briefly describe below. We mostly mention in this paper the security properties of Chrome extensions

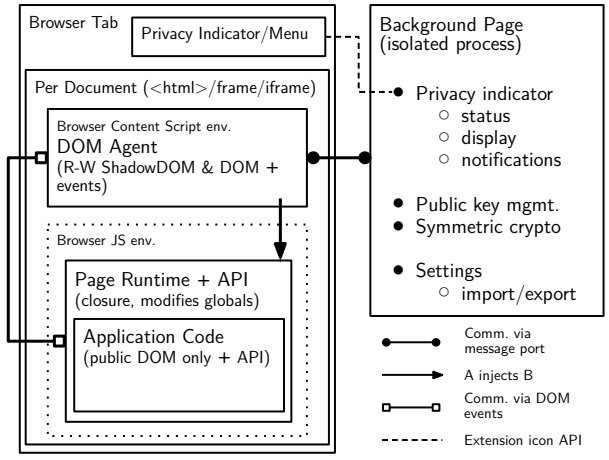

Fig. 1. Division of the Beeswax platform into components.

and ShadowDOM that are of high relevance, but they are otherwise well documented [3, 6, 24].

Background Page. A Chrome extension consists, in part, of a background page: a JavaScript program that the browser isolates from other pages and extensions. The browser provides the background page with its own isolated local storage and with access to several privileged browser APIs. A background page may also include HTML to be run in a tab. Beeswax relies on the background page's isolated context and its ability to run continuously for accomplishing a number of crucial tasks.
These tasks include:

- public key management: identity binding & validation, public key store, and key agreement;
- symmetric key cryptography: key generation & indexing, and cryptographic computations;
- control of the privacy indicator (described below): status, display, and user notifications.

DOM Agent. An extension may include a content script. Every time a page is loaded, the browser loads the content script in a separate JavaScript environment that is paired with the JavaScript environment of the page. The pair share access to the same DOM, but none of each other's JavaScript variables or functions [21]. We call the Beeswax content script the DOM Agent.

Page Runtime. By virtue of having access to the DOM, the DOM Agent can insert additional scripts in the page. These injected scripts share the same object space as the page. Extensions can be configured so that the browser runs the content script *before* the page is loaded. Beeswax uses this early-load capability to install our platform's API (covered in Section 2.1) and to create a modified runtime for the page which isolates the API implementation from the page code.

ShadowDOM. Another browser mechanism useful for our purposes is the recent ShadowDOM. The technology allows DOM subtrees (called "shadow trees") to be grafted to a host DOM element and override its rendering: the browser renders the contents of the shadow tree in lieu of its host. The technology became officially available in Chrome version 35 beta (May 2014). While the technology alone does not allow full isolation between a host and its shadow tree, mechanisms in Beeswax repurpose shadow trees to render private data in plain, but present it as ciphertext to the hosting application.

Private Areas. When an application designates a DOM element to host a private area, the background page, content script, injected script, and ShadowDOM mechanism work together to deny the application read and write access to its ShadowDOM. The injected script provides the APIs for an application to create a private area. When this API is invoked, the DOM Agent creates a shadow tree rooted at the chosen host element. User data is written inside the shadow tree and will be rendered by the browser. The injected script and DOM Agent relay user data to the background page and back for encryption and decryption. An application cannot read the shadow tree directly, but can extract ciphertext from a private area using the API. It can then embed the ciphertext into its various communication streams.

Privacy Indicator For each extension, Chrome reserves an area on the right side of the toolbar for drawing one icon. This icon can be drawn dynamically and can adorn a small amount of text called a "badge". The extension icons are unspoofable: a webpage cannot render UI elements outside the browser window, so it cannot cover up an extension icon. Similarly, an application could launch a popup window to redefine its boundaries, but the browser ensures that popup windows are always rendered on-screen and with an address bar. Moreover, the browser ensures address bars in popups cannot emulate the look of a browser tab toolbar.

A malicious application might trick a user into inputting private data into areas that are not private. We employ the Beeswax extension icon as a privacy indicator to defeat UI-redressing attacks. The DOM Agent informs the background page of references to DOM elements having received the last user event. The background page uses those references to update the icon and badge appropriately. Furthermore, if the current UI ele-

Table 1. Part of the Page Runtime API, detailed in text below.

Category	Methods
Friends	`get_friend(userid) → fr_chan`

Streams	`new_stream() → streamid`
	`invite(fr_chan, streamid) → invitation`
	`accept_invite(invitation) → streamid`

Private UI	`make_priv(dom_el, streamid) → bool`
	`is_priv(dom_el) → streamid | nil`
	`get_cipher(dom_el) → cipher`
	`put_plain(dom_el, cipher) → bool`

ment is part of a private area, the icon can be clicked to reveal the user names of those with access to the data.

2.1 Beeswax Operation and API

We present the operation of Beeswax and the APIs, listed in Table 1, that invoke those operations. The table omits appearance modifier calls (introduced below) and one call to enable the extension in the page. Our platform uses three standard asymmetric and symmetric cryptographic methods: trusted bindings between public keys and identities, authenticated key agreement using asymmetric cryptography, and communication channels protected by symmetric key cryptography. By design, the complexity of these operations is hidden from the application: the API is concerned with opaque handles to keys, rather than keying material.

Identities and Public Key Distribution. We could not find an existing public key binding and distribution system that fit easily with our deployment model. We sought to use Keybase.io [28], a promising approach to distribute keys across multiple social networks. However, it does not currently support multiple public keys per user. Moreover, formal documentation for their proofs of identity is lacking.

Another alternative is PGP-based tools. PGP's trust model is difficult to make fully programmatic, except with complex policy configuration. For example, even when a key with a given ID (e.g., an email account name) is pushed into the PGP key base by one party, it is trivial for another party to push another key into the key base with the same ID. In principle these can be differentiated by web-of-trust signatures by other parties. But for most users, deciding which public keys to trust would need to be done on an *ad hoc* basis depending on the web-of-trust of a each particular key/ID pair.

For Beeswax, we developed a key distribution mechanism which uses a pub/sub service both for identity and key distribution. We use Twitter, but other services would be suitable as well. Firstly, for a pub/sub service to be usable by Beeswax, it should be possible for Beeswax users to verify a binding between a person and their account ID on the service. For instance, the content on a Twitter feed should match what one would expect the feed's owner to post. This inferral can naturally be avoided if one already knows the account ID of the interlocutor. Secondly, the pub/sub service should authorize its users so that only the owner of a feed can post to it. This way, when certificates are retrieved from a feed, we can expect them to have been posted by the feed's owner. Thirdly, subscription to a feed must be done over an authenticated channel. Lastly, it should be suited to store certificate information, possibly using fragmentation or image steganography to bypass length or format limitations. The roles of identity provider and key distribution can be dissociated, but if and only if users on the key distribution service are authenticated using identities from the identity provider (e.g., distributing certificates with Twitter IDs on source code repositories would hypothetically work if the repositories could authenticate using Twitter accounts).

Initially, each user creates a self-signed cert embedding his/her Twitter ID. Users then publish these certs over their Twitter feeds. Users then pull certs from their own feeds and the feeds of others with whom they wish to communicate, and sanity-check these certs. Beeswax does not support anonymous keys – it provides confidentiality, but anonymity is beyond its objectives.

Normally, self-signed certs have no identity integrity on their own. The key idea here is for the identity in the cert to be the same as the name of the authenticated (i.e., HTTPS) channel over which the cert was obtained. The extension periodically compares certs installed locally to those on the key distribution service, and raises a flag if they differ. In this way, Twitter IDs and their associated authenticated pub/sub channels become roots of trust. We assume that users decide for themselves, based on out-of-band evidence, which Twitter username/identity pairs are sufficiently trusted for the purposes of a particular interaction. We analyze the security of this key distribution scheme in Section 2.2.

Applications must obtain Twitter usernames at registration to use Beeswax. As we will show, a user who registers a Twitter account to which it does not have access will not succeed in completing key agreement protocols with other users of that application. We support configuring multiple Twitter accounts, but only one may

be used at a time. For simplicity, this discussion assumes the extension has been configured with a single account.

Beeswax will make reads and writes to a user's Twitter account on behalf of the user. To do so, the user must first register his/her Twitter account name in the Beeswax settings. Subsequently, whenever the user is logged into his/her Twitter account, Beeswax will have the credentials necessary to post to the user's account.

To proceed, a user must install asymmetric keys into Beeswax to bind to the installed Twitter username. Beeswax provides two mechanisms for this. In the background page tab, Beeswax can generate an encryption key pair and a signature key pair and install them into the public key database using its `gen_pub_keys` routine. Beeswax can also import public keys the user owns with its `import_keys` routine.

Once the user's Twitter ID and public keys have been successfully installed, the background page invokes the `distribute` background daemon. This process periodically reposts self-signed certs of a user's public key pairs to the user's Twitter account. This daemon also periodically polls the user's feed for posts that claim to be self-signed certs of the user. If a cert is invalid, its public keys are not those installed into Beeswax, or its embedded Twitter ID cert is not the user's, a flag is raised. Details of this process, such as expiration and revocation, are described in Section 3.4.

The Beeswax background page also has a background process called `validate` that periodically fetches the recent posts of other users. When `validate` is first called, it scans a user's feed for the most recent self-signed cert. If the cert is valid, and the username in the cert matches the Twitter account name, it installs the username and public keys in the public key database, otherwise it raises a warning. On subsequent calls, `validate` tests for the validity of the certs, checks whether the embedded Twitter ID matches the feed, and checks whether the keys are the same as those already installed. If not, it raises a flag.

The background page operates as soon as, and as long as, at least one browser window is open. Periodic tasks are rescheduled immediately when the browser starts up or when connectivity is resumed. Operating 24/7 ideally minimizes problem detection delays, but is not strictly necessary.

Key Agreement and Friendship Channels. In order for a communication stream between two parties to be identified as private via the Privacy Indicator, an application must first enable a friendship channel, i.e., a secure bi-directional signalling channel between two parties. When a user selects a friend (identified, in part, by a Twitter username) inside an application, the application makes a call to `get_friend`. If the potential friend's public key is not yet in the public key database, the background page makes the `validate` call described earlier. Once a key is installed, the background page, with the help of the communication channels of the application, performs a key agreement protocol (Section 3.4). The net result of a successful protocol is a set of symmetric keys that enable secure communication between the two parties. If the key agreement protocol runs to completion, `get_friend` returns a handle to the keys for the secure friendship channel. This channel is used subsequently by friends for passing reference values, metadata, and symmetric keys for collections of private data that we call streams.

Stream. A stream is a series of media messages between one, two, or more users. The function `new_stream` creates new streams. During this call, Beeswax generates one symmetric AES key which will be used in authenticated encrypted mode to encrypt all messages from the stream. The creation operation returns the stream identifier `streamid` (a key handle) so that it may be stored by the application and retrieved subsequently. The user creating a stream is said to be that stream's *owner*.

Sharing. Sharing in an application is achieved by distributing stream handles. The owner of a stream can use the `invite` method with two key handles: one designates a friendship channel and the other one designates a stream. The operation is asynchronous, which allows the platform to ask for user permission (e.g., "The application is inviting `@Carl` to a stream in which `[@Alice, @Bob]` already take part, do you accept?"). The call generates an invitation message over that friendship channel. One invitation is sent over each friendship channel between the owner of the stream and its participating users. That is, the friendship channels are one-to-one, but multiple friends can be invited to the same stream.

The invitation messages are authenticated and contain the encrypted stream key, the stream ID, and metadata indicating their provenance. Calling `accept_invite` on the invitee's computer reconstructs the stream handle. The provenance allows our UI to determine and list a stream's participants.

Private UI. Once streams have been created, the application may choose elements of the page to host stream content for display. First, `make_priv` transforms a regular DOM element (`dom_el`) to "host" private content. The platform clones the host's subtree into a new private

area and inserts the area as the children of a shadow tree on the host – which hides the original subtree from view. The platform associates the host element with that stream for the rest of the element's lifetime. The application keeps references to the host element (it is part of the regular DOM), but our mechanisms prevent the application to query the DOM to access the private content. To query whether an element hosts private content or not (i.e., if it has been passed to `make_priv` precedently), we provide `is_priv` as a convenience.

To read the user's encrypted input, the application calls `get_cipher` with the host element `dom_el`, which will cause the extension to read the contents of elements, and encrypt them with the associated stream key. To display it, ciphertext information is passed back into `put_plain`, during which the extension replaces the visible contents of the element with those of the decrypted message.

Beeswax supports secure input and output (display) of both text and images. The element passed to `mark_priv` determines a private area's type: e.g., `<input>`, `<p>` host text and `<img>`, images. To provide the application more control over the private area content, which it cannot access directly, the platform ships with "appearance modifiers", functions part of our TCB (omitted from Table 1), that the application can call to change an area's look and feel. We list limitations in Section 5.

Beeswax fully controls the flow and contents of DOM events falling into (and outside of) any private area. Our privacy indicator monitors these events: at any given time, it can confine events to only certain private areas, and block others. This interposing of events protects users against UI-spoofing.

2.2 Security of Beeswax Key Distribution

We distribute keys over the pub/sub service, Twitter. For this discussion, we assume that if a Twitter account is hacked and the true owner loses access to the account, the account can be disabled by contacting support. Such a scenario is outside of our threat model.

Unauthorized Posts. Consider a malicious party other than Twitter. Such a party may craft key pairs for which it knows the private keys and attempt to post the cert for those keys to a user's Twitter feed. If successful, several other users may install the malicious public key into their Beeswax keyring for the victim. We note that such an attack can only be successful if three things are true: the party gains access to the victim's Twitter account, the victim does not notice the bogus cert in its twitter stream, and the malicious party has gained access to the victim's Beeswax extension and installed its bogus key pair. The latter is necessary because the distribution protocol requires the background page of a user to ensure the user's published certs and those installed locally have matching public keys. We consider it unlikely that all three conditions above will be true.

Although Twitter itself has write access to every Twitter account, it is not in a position to install key pairs in Beeswax extensions. Thus, Twitter cannot simply try to post a crafted cert to a user's feed without the user noticing. However, Twitter could perform a fork attack on a user's account. That is, it could present the correct replay of a user's posts to the user, but present an incorrect feed of posts to the user's followers. This fork could trick that user's followers into binding a bogus public key to the victim's Twitter ID, a public key for which Twitter had the corresponding private key.

We do not attempt to provide protection against such a strategy by Twitter. Nonetheless, we posit that maintaining fork-consistency after having lied (i.e., serve veridical tweets to the key owner, but lie to all others) would be prohibitive. Amongst applications in which the target user and friends were already enrolled, and Twitter's indisposition to man-in-the-middle, some evidence of the subterfuge would surface.

Trusting Strangers. A malicious party may attempt to register a Twitter handle designed to deceive other users about the identity of the account's owner. A user who asks her extension to subscribe for key broadcasts on a given Twitter ID should perform due diligence, up to her desired level of assurance, on the account owner's true identity (e.g., using out-of-band channels). Our scheme does not attempt to protect unvigilant users.

Limitations. Twitter does not guarantee that tweets will be searchable beyond about a week after their posting. Not all tweets are indexed in the same way, and what determines that a tweet will remain searchable past this window is not made clear [34]. This API behavior forces extensions to continually re-publish their identities to prove possession of the signing private key and permit discovery by other users of Beeswax applications. This API limitation, in fact, encourages good practice. It is for this reason that we expect a working validity period to be just shy of a week.

3 Implementation

Our prototype Beeswax platform is implemented against a stable version of Google Chrome (initially version 40.0.2214.95) and can be installed in only a few clicks. This allows our work to be immediately deployable on the Web. It is composed of about 5000 lines of code, including comments and HTML, and excluding third-party library code. The code for our prototype and applications is open-source and is available online on our project page [19].

3.1 Secure Display of Private Data

An intricate aspect of the implementation follows from the need for private areas to be hidden from the application, but visible to the user. It also aims to allow the application to preserve its desired look and feel.

Creating a private area. Our Page Runtime modifies the application's environment to prevent Beeswax-enabled applications from creating new shadow DOMs and retrieving the root of existing ones. As a result, transforming host elements into private areas can only be done via the content script, with the use of the `make_priv` API call. We support two types of private areas: generic text and images, but this could be extended to other media (canvas, file attachments, etc.).

We use HTML5's `Element.createShadowRoot()` to create a protected ShadowDOM subtree. The created shadow root is made available in a single getter property called `elt.shadowRoot`, and that property only. Nullifying that property suffices to hide the private area from the application DOM. The application cannot recover nodes from shadow trees by traversal nor with typical selectors (e.g., `document.querySelector`). However, the shadow trees remain accessible to the content script.

Browsers render what is called the "composed" DOM tree, a stitched-up global view of the main DOM tree and shadow roots. Mangling the `shadowRoot` property of nodes in the application environment does not affect this composition, as the browser still uses internal references to form the correct tree. The composition rules are such that the content of the shadow root (which will contain the private data) is rendered in place of the content of the host node in the composed tree. The rules are complicated by the nesting and relocation of multiple roots, but we eliminate this concern by preventing applications from creating new roots outside `make_priv`.

Trapping Events. Due to the way events propagate in the DOM, events targeting private elements may leak private information, such as keycodes, selection of text in the page, or data input changes, into the application's event handlers. One challenge of our implementation is to prevent leaks without breaking the event system applications depend upon. The platform traps events that carry sensitive information. Not all event types can target private elements, so we apply a first round of filtering to move certain events back on the fast path. For instance, events that only target the `window` or `document` (e.g., `DOMContentLoaded`) propagate unmodified.

To ensure our filters are effective, we arrange for the platform to inspect every event dispatched *ahead* of the application. We rely on two properties enforced in the browser: first, that the order in which event listeners are registered is honored during dispatch, and second, that event listeners cannot be removed without a reference to the identical object given at registration. The rules of event dispatch are intricate and are omitted here for brevity. We make use of the former property when our Page Runtime is injected before the application, and we make use of the latter by hiding the Runtime's event listeners in a closure, thus preventing the application from de-registering our listeners. Although the order of invocation of listeners on a given object (e.g., `EventTarget`) is unspecified by DOM Level 2 standards, it is predictable within a particular browser. This is rectified in the newer DOM Level 3 [26], where orders of invocation and registration of listeners must match.

Filtering only the event types for which the application has handlers registered would be most efficient, but is hard in practice. It is straightforward to obtain a complete view of DOM Level 2 listener (de-)registration because it is centered around a single prototype method (i.e., `addEventListener`) that can be interposed. However, legacy DOM Level 0 events can be registered independently on all objects, either inline in the HTML or with a script (e.g., `img.onload = my_func;`). We are not aware of ways to disable this older event model, and intercepting all accesses to all `on*` properties of all objects (and along their prototype chain) is impractical. Our Page Runtime remains conservative, pre-registering all event types for which there exists an `on*` method before the application does (around 100 different types of events are registered this way, e.g., `onfocus`, `onblur`, etc.).

Sanitizing Events. Our sanitizer functions will erase properties that might reveal information about the contents of the private area, such as keycodes or selection information. When an event is dispatched on a node in-

side a shadow tree, a private area in our case, standard event dispatch undergoes a process called event retargeting. In this process, the effective target of an event is updated when it crosses a shadow tree boundary. This is so that the target of an event (as in `event.target`) always points to a node within the same subtree as the node on which the current handler is installed. This retargeting process is in line with ShadowDOM's original intent, i.e., to hide the complexity of a sub-layout from the parent layout. Consequently, if our `window` filter detects an event targeting (or relating to) the host element of a private area, the actual node targeted is either the host element itself or an element within the private area. In either case, the event is flagged for sanitization.

The platform is configured with a static list of properties that should be erased for the various event types. Depending on the nature of the property to hide, we either use `delete` to remove the property from the event object or rely on non-reconfigurable getters (that return a constant instead of the true value) along the prototype chain. During event propagation, each event listener will receive the same event object, so our modifications persist across the event's entire lifetime.

We considered it appropriate to only erase the sensitive content from the event object, rather than stopping its propagation altogether. This is useful, for instance, to determine if a user has started typing. In the future, we could also base these decisions, i.e., to sanitize or stop, on a configurable application policy. We could also delay the events, for instance to prevent a timing attack that would infer keystrokes, but this is outside the scope of this paper.

Revealing private content. We assume the application running in the page cannot simulate keypresses on behalf of the user. This ensures all private data entered comes from users. Applications can indeed synthesize UI events, but in Chrome we can differentiate those from true user events. The system clipboard could constitute an exfiltration path, but modern browsers cannot read its contents without the user's permission. The page could access regions of the page that are selected, i.e., with `window.getSelection()`), but those can be interposed. HTML5 comes with a Web Clipboard API with a programmatic RW capability, but it cannot amass private content without user action.

Styling rules in the page can affect the look of private areas, and may shift content around. While we are not currently aware of attacks that would infer precise contents of a box by observing its response to repeated resizing, such an attack is conceivable (e.g., by prescribing crafted fonts, or forcing text to wrap in different ways). A related attack exists where links are styled differently based on their "visited" status (a CSS `a:visited` property exploit [1]), but this has since been fixed by limiting the capabilities of CSS in certain selectors. We imagine similar patches could be applied if other such side-channels were to surface. Another remedy could be to impose fixed dimensions on private areas.

3.2 Integrity of the Page Runtime

The security of the platform, and the safety of the user's private data, hinges on the inability of the application to modify the Page Runtime's behavior. If application code were to compromise the Runtime, it could exfiltrate private content. The Runtime constitutes a self-protecting reference monitor to the privileged objects and functions it defines. It is a *lightweight* modification to the browser (as defined in previous work on self-protecting JavaScript [14]): it is trusted and does not parse or modify the code of the application. To the application, the Page Runtime appears as a built-in browser API. We use language-based techniques, such as closures and function wrappers, to reach the following goals of protection:

1. Globals, methods, and object properties used in the Runtime are those original implementations provided by the browser.
2. The application cannot tamper with objects and functions of the Runtime.
3. Functions wrapped by the Runtime do not reveal the unwrapped implementation to the application.

We have not directly used tools from previous work [10, 11] to generate code for our Runtime, but our construction defends against the tampering attacks they describe: prototype poisoning (when functions, getters, and setters are rewritten by the application), abusing the callchain (e.g., reflection over `arguments.caller`), unsafe accesses on application-provided objects (safeguards against objects which can change appearance between time-of-check and time-of-use), unsafe casts (e.g., implicit string casts), etc. We explain how we achieve the first goal next. Goals 2 and 3 are handled with the same care, but we omit their descriptions for brevity.

First, before the application starts, a (pristine) copy of each needed global is passed as an argument to an all-enclosing anonymous function – in other words, we save global objects and functions into local scopes. These locals are used instead of accesses via the `window` object. Verifying that only symbols local to the Runtime are

referenced is tedious, but this task could be automated with a static analysis tool designed for this purpose.

Second, accessing properties with the dot operator or brackets, e.g., `x.y` or `x["y"]`, might invoke getters and setters that have been defined by the application. Where needed, we rely on built-in object reflection mechanisms to read application objects, for instance with `Object.getOwnPropertyDescriptor`. In cases where the Runtime must invoke a prototype method on an application object, e.g., `obj.hasOwnProperty('x')`, we avoid the dot completely by first wrapping the method's `apply`:

```
function wrap(m){ return m.apply.bind(m); }
var HOP = wrap(Object.prototype.hasOwnProperty);
```

Assuming `HOP` is saved to a local in the Runtime, the example can then be safely re-written as `HOP(obj, ['x'])`. This is a refinement over the approach presented in [11].

Third, access to still-undefined object properties in the Runtime cause a lookup in the prototype chain, which could find a hit on a property of `Object.prototype`. For instance, if the application installed a setter property called `key` on `Object.prototype`, statements in the Runtime code as seemingly innocuous as `var data = {}; data.key = "X";` would leak the value to the setter. To eliminate this possibility, and allow base objects to be used inside our code with peace-of-mind, we *freeze* `Object.prototype`. Prototype lookups on objects could be avoided by other means, albeit tediously. Literal object definitions, for instance, do not cause the setters to be invoked in the prototype chain. In the same example, `var data = {key: "X"};` would *not* invoke the setter.

Using `freeze` is simpler than breaking prototype chains [11] and eased development. Freezing `Object.prototype` is convenient, but we have found it to break some websites (Gmail) and some libraries (d3.js) in subtle ways, so we also had to allow certain redefinitions, such as `toString`, to restore expected behavior.

Future language features could allow discovering hidden content (e.g., a new exception mechanism that allows inspecting stack variables, a new unhandled event type, or additional object introspection) and could affect the integrity of the platform. We cannot claim our defenses are future-proof. On the other hand, the platform approach allows code to be exposed and vetted by enthusiast and professionals. The software platforms underlying the Web evolve constantly – everything built on top of them must necessarily co-evolve, or die. We do not believe future uncertainty should be a deterrant to the creation of novel privacy-protecting technologies.

We note that the number of global wrappers introduced by the Runtime is tractable: only for event listener registration, selected event getters, and `shadowRoot` interactions (all discussed in Section 3.1). We have been diligent in consulting specifications and available documentation to cover possible aliases and equivalent functions. Also, because it is difficult to determine, without inspecting the browser's source code, whether a built-in (e.g., `appendChild`) internally accesses object properties that can be modified by the application, we perform DOM manipulations in the content script rather than in the page. Hopefully, further review of the published code would clear out any oversight on our part.

3.3 Privacy Indicator

Ideally, we would like to have visible at all times the full list of participants involved in a particular stream. Unfortunately, Chrome allocates space for no more than a single icon per-extension in its decorations. The icon can display a badge of text, but even that is limited to 4 characters. The compromise we found was to use the icon as a notification and use its popup activation menu to display additional details.

The indicator toggles between two modes: "protected" and "unprotected". As long as there is no user activity, the indicator rests in its default "unprotected" state. However, when the Agent receives a keyboard or click event for a private area of the page, it transitions the indicator to "protected mode" for *that* area. The indicator changes its appearance to notify the user, and a marker is added to describe the nature of the events that have caused the transition (See Figure 2).

This transition activates a locking mechanism released by a timer. While the lock is held on the area, events dispatched for areas of the DOM unrelated to the stream will be aborted. Inactivity will expire the lock and switch back to unprotected mode. On the other hand, continued activity in a private area of the stream will renew the timer. In the future, we would reserve a CTRL-* key sequence to prolong the lock indefinitely and unlock it on-demand.

Activating the icon brings up information about the one specific locked area and its associated stream. Recall that there may be multiple private areas shown on the page for a given stream. The information shown for the stream would apply to all of the areas.

The monitor can simultaneously adorn four markers. A mouse click or keyboard event in a private area of the page will lock the monitor (with "M" or "K" markers, respectively). In addition to text areas, Beeswax provides built-in support for private images. A mouse click in a private image area allows users to input new

Fig. 2. The Privacy Indicator "unprotected" (left), "protected" due to keyboard events ("K") in a private area (middle), and showing a security warning (right).

images. When the click is intercepted, Beeswax opens a file chooser dialog locked to the area. To differentiate the file chooser opened by the platform from one opened by the application, we add an "F" marker on the indicator. Lastly, a "*" marker attracts the user's attention to answer a prompt, e.g., to confirm sending or accepting an invitation or friendship.

3.4 Cryptography and Key Management

All keys are stored in local storage in the background page. Because Chrome's model is limited to a single store per extension, we avoid collisions of keys from different applications (but equal streamids) with a canonical naming scheme. To save/load a key to/from storage, we combine the key id (e.g., a streamid passed in the API) with the domain origin of the application, as well as the extension user's account name. The browser reports the tab URL associated with a content script's message port to the background page. This metadata and the naming scheme allow authorizing API requests access to only the keys associated with their application.

Key Agreement. As mentioned above, our platform supports an abstraction called a friendship channel. A friendship channel is a virtual private channel between the Beeswax extensions of two users. Beeswax has an API for one party to launch an authenticated key agreement protocol with another party. Assuming the parties have the appropriate public keys and Twitter userid bindings, and that the protocol completes, the two parties will possess two symmetric keys, one for encryption and one for authentication. Those keys will be used subsequently to encrypt-then-mac all communication over the friendship channel. Namely, two friends use the channel to send each other stream invitations. Note that channels are between two users only, but streams may involve more than two users (see Section 2.1).

For the authenticated key agreement, we implement the well-known `AKEP1` protocol of Bellare and Rogaway [4] in the public key model. The JSON-like wire format of our protocol is in Table 2. We leave implementing a Diffie-Hellman style perfect forward secrecy key agreement protocol to future work.

Key Distribution. To publish a new set of keys, the extension posts three tweets. The body of the first contains `#encryptkey`, a time stamp, and the user's public encryption key. The body of the second contains `#signkey`, the same time stamp, and the user's public signature key. The body of the third contains `#keysig`, the same time stamp, the expiration date of the keys, a signature over the user's Twitter username and ID, the two public keys formerly identified, the same time stamp, and an expiration date (of about one week). The extension requires that the user be signed in to Twitter, so the extension can publish on the user's behalf.

The extension polls the user's Twitter feed to find tweets with the above markers and timestamps. For a given triple, the extension checks four conditions: the public keys in the body of the tweets are the same as the user's locally bound public keys, the time of the tweets is within `delta` (a configurable parameter in the order of minutes) of the time stamp in the body of the tweets, the time of the tweets is before the expiration time in the body of the `#keysig` tweet, and the signature in the `#keysig` tweet is valid. If any of these checks fail the extension raises an alarm to the user. If all pass, the extension binds the two public keys to the given Twitter ID, regardless of the state of an existing key binding.

Key Revocation. A Beeswax extension subscribes to key announcements of other Twitter accounts. Revocation is achieved in Beeswax by publishing a new key over an old one, which causes other users' extensions to notice the change. Cases where certificates are absent, have expired, or have changed since the last validation are handled similarly. Whenever a certificate is determined invalid, associated keys (including stream keys obtained over invalidated channels) are kept, but marked invalid. The case of a certificate having changed differs only slightly in that there is opportunity to re-establish a valid friendship channel.

In our current model, the participants in a stream must trust the stream owner's extension to verify the validity of other participant's certificates, and re-key appropriately (e.g., new friendship requests and new invites). From the moment a stream participant's extension detects the stream owner's certificate being invalid, the stream cannot be used to encrypt new content. The extension provides the application with an API error code in this event. It is the application's responsibility to launch a new stream. A page is allowed to load with content from invalid streams (e.g., content encrypted

Table 2. Alice contacts Bob for the first time and negotiates a shared secret with him using our Key Agreement Protocol (KAP) (via `get_friend`). Messages flow between users' extensions in the direction of the arrows. Alice's keys are denoted by the letter A, and Bob's with B. Key subscript $\cdot_s$ denotes signing, and $\cdot_e$ encryption. Public keys are suffixed with $+$, private ones with $-$. Functions `add_sign` and `add_HMAC` compute signature and HMAC (respectively) and return their augmented input.

Alice		Bob
Input: (A_s+, A_s-, A_e+, A_e-), (B_s+, B_e+)		*Input:* (B_s+, B_s-, B_e+, B_e-)
• Generates A_{FID}: contribution to friendship ID (challenge nonce). • Sends `KAP_MSG1` Message.	→ `type: KAP_MSG1, hdr: { To: "B", From: "A", AFID: A_FID, BFID: "" }, payload: null`	• Receives `KAP_MSG1` • Finds A_s+ and A_e+ if unknown (See Section 2.2). • Validates `To` . ↓ • Generates B_{FID}: contribution to friendship ID (challenge nonce). • Generates MK: the master key of the friendship. • Replies with `KAP_MSG2` message.
• Receives `KAP_MSG2`. • Validates `hdr`. • Validates signature with B_s+. • Decrypts `payload` with A_e- • Saves plaintext as MK. ↓ • Sends `KAP_MSG3` message. • Derives & stores friendship keys: F_{mac}= HMAC(MK, "mac") F_{enc}= HMAC(MK, "enc")	← `add_sign(k=B_s-, { type: KAP_MSG2, hdr: { To: "A", From: "B", AFID: A_FID, BFID: B_FID }, payload: enc(k=A_e+, m=MK) })` → `add_sign(k=A_s-, { type: KAP_MSG3, hdr: { To: "B", From: "A", AFID: A_FID, BFID: B_FID }, payload: null })`	• Receives `KAP_MSG3`. • Validates `hdr`. • Verifies signature with A_s+. • Transitions to A-has-friendship-keys state ↓ • Derives & stores friendship keys: F_{mac}= HMAC(MK, "mac") F_{enc}= HMAC(MK, "enc") • Sends `KAP_MSG4` Message.
• Receives `KAP_MSG4`. • Validates hdr. • Validates HMAC. • Transitions to B-has-friendship-keys state	← `add_HMAC(k=F_mac, { type: KAP_MSG4, hdr: { To: "A", From: "B", AFID: A_FID, BFID: B_FID }, payload: null })`	

prior to the invalidation), but the privacy indicator will display a lasting warning graphic (right-hand side of Figure 2).

Crypto Library. For cryptographic routines, we use the Stanford JavaScript Crypto Library [31] (sjcl) with AES and ECC support. The version of the library we use relies on the browser platform's `crypto.getRandomValues` to seed its pseudo random number generator (which is described in [16]). We have adopted sjcl because of its more mature ECC support vs WebCrypto [2] at the time the project started, but we could easily move to an alternate, native and faster, cryptographic library supported directly in the browser when it officially supports the primitives we need (ElGamal). Keys are stored in JSON format, using base64 encoding when appropriate.

We use elliptic curve cryptography (ECC) for all asymmetric material, in particular ECDSA for signing messages and ElGamal for encrypting small messages. Our symmetric encryption keys are 256bit long and we use the AES cipher in CCM mode (Counter with CBC-MAC, with 128bit IV and 64bit tag size). For HMAC, we use SHA-256 hashing, also with 256bit keys. All elliptic curve operations take place on NIST's recommended P-192 curve, which is available in sjcl. Moving to a potentially more secure curve, such as Curve25519 [5], or implementing ciphersuite negotiation is future work.

4 Evaluation

Beeswax aims to provide mechanisms and APIs that are fit for the development of modern web applications. To test this claim, we have used the platform primitives presented earlier on two applications. First, we transformed an existing web communication application, KiwiIRC (v0.9.0), into one that also offers encrypted communications between groups of users. Second, we developed a new photo gallery application, PicSure, from scratch to demonstrate the abilities of the platform to handle richer media types. Below we discuss our experience in developing these applications and our platform's performance overheads.

4.1 Functionality and Experience

KiwiIRC is a web IRC client software package that contains both Web2.0 single-page client-side code, and server code to proxy messages between the browser and the IRC network. The IRC client already routed different types of messages between users and channels. It was therefore relatively easy to add another class of messages to carry Beeswax payloads within the IRC network. We changed KiwiIRC to route platform messages using information present in their headers. One difficulty we surmounted was that encrypted messages had to be fragmented to respect the length limitations of the underlying relay protocol and pass integrity checks.

The client was augmented with new commands:
`/joinenc [streamid]`: Takes a stream id (or creates one if unspecified), deterministically forms an IRC channel name from it, joins that channel, and allows typing new messages using the stream key.
`/inviteenc streamid [user]+`: Invites each user in the list to partake in the stream identified by `streamid`. Once a target user accepts the invitation, he/she automatically joins the conversation using `/joinenc`.

We have added graphical menu options to befriend other users in a chat room, as well as buttons to toggle encryption of input messages in the chat room panel.

When typing in regular messages, the application uses the arrow keys to recall elements from a history buffer and uses the "enter" key to signal the readiness for submission. The application cannot read keycodes from private areas, thus additional buttons are added outside the private area to recover this functionality.

For a seamless look, private areas hosting encrypted messages are styled like plain messages. That is, they are added with the other messages, along with information coming from the network (nickname of author, timestamp, etc.). The platform can apply highlighting to designated words in incoming messages, such as nicknames, but we have yet to explore fancier formatting, as there is often a tradeoff between functionality and side-channel leakage.

The server-side code did not require any modification. The approximate number of lines of code added (not counting whitespace and comments) was around 400, constituting a 7% increase in code size. This gives an estimate of how little work is needed to add key-agreement and end-to-end encryption to an existing application using Beeswax.

Richer Sharing. We have also built a web photo gallery called PicSure to show the capabilities of the platform to handle richer media types and allow application-defined sharing rules between users. It consists of approximately 2000 lines of script code, server and client combined. Put simply, it allows users to create albums of photos with simple descriptions and share individual albums with other users. All the data inside an album is part of the same Beeswax stream. The action of sharing albums is done through forms (drop-downs and lists) in the application. Unlike with overlay approaches, the user does not need to leave the application to define keys or assign them to the various objects. Rather, the application gives sensible sharing controls to the user: the owner of an album may invite any user currently online to collaborate on an album.

Editing descriptions is done with toggles and confirmation icons, like many edit-in-place applications, and uploading images is done using the Beeswax image file chooser. There are a few usability problems with the current platform, but they have solutions. First, having no capability to reduce images to thumbnails, PicSure renders album photos as full resolution images. Second, the sizes of the images are not known by the application, so it imposes a fixed size area to render them in the markup. A new appearance modifier could generate a thumbnail or resize the host element to match the image's aspect ratio. One other difference from a normal photo gallery is that the encrypted image data is retrieved and fed using data-URIs, as our implementation does not yet pass `src="{url}"` files through the decryption. This translates into encoding/decoding delays. Despite these usability issues, the application remains quite practical to share images quickly and securely.

4.2 Performance

We first report on the performance overhead of Beeswax, and conclude qualitatively with our own experience. The performance overhead of the platform as a whole can be attributed to several aspects:

Runtime initialization Page load delays incurred by the initialization of the Page Runtime.

Events Interception of every DOM event to prevent a possible data exfiltration (e.g., keycodes).

Message passing Costs associated with passing messages between the page, content script, and background page.

Encryption Cost associated with the encryption and decryption of user data.

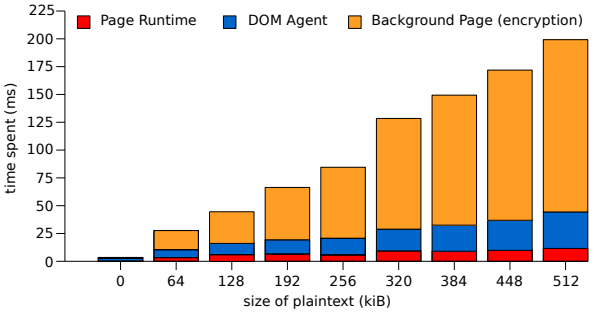

Fig. 3. Median time (N=51) spent in each component when encrypting DOM elements, by plaintext size. Blocks (from bottom: Page Runtime, DOM Agent, and Background) do not overlap.

The cost of initializing the runtime and intercepting the events is always present, even when the page in the tab does not use Beeswax. This is the base cost of loading the content script, injecting the runtime in the page, initializing the event hooks, and trapping on the first few load events. We compare load times of a bare bones HTML page with and without the extension enabled. This constitutes a base cost for loading any website. For $N = 400$, the minimal page's mean load time is under 65.5ms (σ=10.1 median=65 min=42 max=100) with the extension, versus 13.7ms (σ=4.7 median=14 min=5 max=25) without (cost average of 51.8ms).

Invocations of API methods do have an associated cost when generating events. This is because the Page Runtime needs to perform sanitization of these events. To calculate the overhead in event dispatch, we compare the time taken to generate and dispatch 1000 keyboard events. With the extension loaded, over N=100 rounds, 1000 events take on average 130.2ms to dispatch (σ=5.2 median=128 min=125 max=153) versus 51.8ms without ($\sigma = 5.2$ median=51 min=46 max=64). Event dispatch takes 252% the time of the baseline, which is a considerable *relative* overhead, but in practice this is not noticeable.

The cost of message passing and encryption is predictable and is linear with the size of the plaintext submitted. Figure 3 shows where time is spent, by component, during calls to encrypt, i.e., with `get_cipher`.

The time spent in the Page Runtime includes the serialization of a message containing a node identifier (the one to encrypt) and 2 event dispatches with JSON serializations (on the call and on the return). The time spent in the DOM Agent includes reading the private DOM element's contents and passing the plaintext to the Background Page (and ciphertext back) over a message port (these ports support "structured clone" copies to avoid JSON serialization). Lastly, the majority of the time is spent in the Background Page performing the AES encryption routine and forming the stringified ciphertext (returned ciphertext objects contain a few parameters, namely an IV, which needs to be kept along with the text). Figure 3 shows median information over 51 runs. The top of a bar estimates the roundtrip time for encrypting private content of that size.

Overall, we did not perceive any performance impact in using the IRC client with and without privacy enabled. Similarly, we have run the extension on our desktops over the course of months without perceiving negative effects on non-Beeswax sites. A full user study is outside the scope of this paper.

5 Limitations

The API maintains data confidentiality, but this loss of visibility comes at a cost in interactivity for the application. Beeswax is not suitable for all applications. For instance, applications heavily relying on data-mining might find this cost prohibitive. However, from our experience building applications with the platform, we believe that many of the typical social-networking features such as comments, change notifications, "likes", replies, view counts, and untargeted ads (i.e., without access to content) can still be implemented. Also, where greater control is required over one data element (rich text format, image manipulation, etc.), appearance modifiers can fill in some of the gap. Lastly, applications which combine data with different confidentiality levels (public and private) might be drawn to Beeswax. Below, we detail more subtle restrictions and suggest improvements.

Private Area Interactions. Our event sanitization conceals most interactions with private areas, but can be modified to let out more events. For instance, to allow the application to detect when an input image changes, Beeswax currently synthesizes a "change" event on the host element after the platform's file chooser is used. An improvement would be to interpret "Enter" strokes in private input boxes as an intent to submit, and synthesize a corresponding event interceptable by the application. Although arbitrary input validation is impossible without access to content, simple validators such as length enforcement could be communicated declaratively via DOM attributes. Leaving these decisions up to the application is an aspect where the platform approach shines over overlays.

The ability to drag-and-drop files onto private areas would be useful, especially for applications using pho-

tos, but difficult to implement securely. The intention of moving a file into a private area would need to be known ahead of time so that drag events could be suppressed along the way (to hide the file object from the non-private areas) until the file reached its destination.

Running other extensions. The load order of multiple concurrently installed Chrome extensions is, to our knowledge, unspecified. A foreign extension could modify the browser's JavaScript runtime in unsafe ways (e.g., by exposing the original implementations of some global functions) and break our assumptions. Whitelisting safe extensions to run concurrently with Beeswax is possible, but requires careful inspection. In Chrome's model, a permission exists to selectively enable or disable extensions. To ensure our extension loads first, a workaround is to disable all other extensions whenever a Beeswax-application is in use. If our API were part of the browser core, this limitation would vanish.

Visual flexibility. Only CSS and "appearance modifiers" can be used to change the look and feel of private areas. We support basic word highlighting, which is admittedly limiting, but the platform approach shows potential. It could be extended to support markdown, BBCode, or other wiki-syntax display. There could be platform support to crop and rotate images, a platform-provided rich-text editor, etc. Like all web platforms and standards, we anticipate features would mature over time, guided by popular demand.

6 Related Work

We have looked at alternatives before settling on ShadowDOM to implement isolation in the UI. Namely, priv.ly [30] uses iframes, but their approach has two drawbacks. First, iframes are isolated by origin and behave like black boxes: it becomes difficult for the embedding application to manipulate and receive events for them. Second, the added dependence on another service or domain (in their case the content stores) to store and retrieve user data, as well as an ever-up-to-date look and feel (that has to match all websites), was unappealing.

The goals and limitations of overlays were discussed earlier. While they provide private UI channels, they do not defend against UI spoofing and require key definition and sharing to happen out-of-band from the application. Our architecture, with its integrated privacy monitor and in-band key distribution, allows a user to safely verify the sharing intentions of the application. In our case, this makes it possible to implement application sharing semantics which are in-line with those of the elements protected.

The data isolation mechanisms in ShadowCrypt [8] are similar, but not identical, to ours. We have found several design flaws in their latest implementation [15] which we detail below. Each of them allows an attacking website to exfiltrate the private data that ShadowCrypt is trying to hide. To their credit, their code is open-source and receives suggestions for improvement. We note that since the project's inception, changes in Chrome's DOM core moved some object properties to getters/setters on the prototype chains [25]. This introduced breakage which has not yet been fixed, but the attacks listed below are valid regardless of the location of these properties (i.e., before or after said change).

Attack 1) Improper attribute deletion. Events targeting input boxes for encrypted content are canceled by ShadowCrypt. The decision to cancel an event is based on the target of the event, i.e., `evt.target`. During an event's dispatch, the target will change when crossing a shadow tree boundary. By introducing an additional shadow tree between the root of the document and the shadow tree for the "secure" input box, we can make an event appear as if it were targeting another element than the input box's host. The method `Element.createShadowRoot` is not deleted properly from the globals, allowing an application to mount this attack and let secure keypress events flow to the application. This attack can be mounted in two lines of code.

Attack 2) Improper attribute deletion. The property `shadowRoot` of elements gives access to Shadow DOM contents. ShadowCrypt deletes this property from elements at the moment they are added to the DOM, but this is too late. It is possible for a malicious application to create an element, redefine this property as being non-configurable, and add the element to the DOM after. The attempt to delete the property by ShadowCrypt will fail silently, leaving the "secure" contents readable by the application.

Attack 3) Unprotected globals. So-called "deep" selectors can be used to retrieve elements across shadow tree boundaries. ShadowCrypt overrides `querySelector` to prevent selectors containing the substring `'/deep/'`, which would allow access to the hidden input element. However, it uses the global prototype method `window.RegExp.test` to perform the check. Because the method is accessed through the global `window`, a malicious application can redefine it to lie when a regular

expression matches for `/deep/`, and later obtain a reference to the secure input element.

While attack 1) is simple, attacks 2) and 3) indicate that a rigorous construction of the functions modifying the runtime are necessary. Properly implementing these defenses without bugs is hard. We believe the systematic way in which our Page Runtime is constructed helps with this, namely by avoiding the use of modified globals, and making assertions about the presence and absence of properties (e.g., a property should be there before deletion and absent after). Pushing our platform into the browser's core would make some of these defenses irrelevant, but for the moment, they are crucial.

There exists a body of work concerned with controlled JavaScript execution, which could allow Beeswax to make private areas less opaque to applications, but with possible privacy implications. TreeHouse [9] uses Web Workers to isolate scripts and provide them with a reduced view of the DOM, which is very much in line with our data protection strategy in Beeswax. AdSafe [18] and Caja [13] are language-based sandboxes that constrain language features and DOM access to allow for controlling the scope of effects of script execution.

Regarding hardening our Page Runtime against possible intrusions by the application code, there is also a wealth of research covering secure ways of isolating concurrently running JavaScript programs from one another. Many techniques offer ways of modifying JavaScript in safe ways for a containing page. Those include enforcing policies on scripts [12, 23] or at the language level [10] (for the older ECMAScript 3). They do fairly well in restricting the parts of the language used for safe operation, but the model of operation is reversed in our case. Our page runtime hosts an application whose programs cannot be modified (instead of an untrusted program that must conform to certain rules). As such these are not directly applicable to us.

The work on writing protected wrappers [10, 11], even though written for an older version of ECMAScript, still provides valuable formal descriptions of actual attacks. The event models (i.e., sanitization) in the browser are, however, not described in this previous work. Some the problems, e.g., protecting the globals against tampering, are addressed previously in work on JavaScript compilation [7], but for it to be useful would require rewriting our Page Runtime in ML.

Our work also relates to work on information flow control. In COWL [17], the authors make use of the existing isolation in browsers (contexts) to control flow between domain origins. A finer-grained labeling scheme than per-origin would be required to apply to our needs.

7 Discussion and Future Work

Our simple invite API could be improved to support other kinds of groups, such as closed groups or groups where all existing parties need to agree before a member can invite a new person. Assuming a user trusts another friend (and their extension) to forward invites, then this could allow knowing the full extent of the users with whom the keys have been shared. Lastly, our streams can be improved to defeat replay attacks from the application. At the moment, a malicious application could reorder and resend a stream's messages. We plan to fold in sequence numbers as authenticated metadata and display them in the privacy indicator as a simple defense.

Ideally, the platform would support applications for both desktop and mobile. Popular mobile platforms may be trustworthy, but are not as open and extensible as desktop environments. There currently is not good support for third-party extensions on mobile web browsers. We do not eliminate the possibility of adapting our techniques to an OS kernel and/or rendering APIs, but we suspect it would require a substantial rewrite. By reusing an existing mobile browser engine customized with our platform tools, we may support web-based applications designed for mobile.

8 Conclusion

The Beeswax platform allows developing interactive, multi-user web-based applications. It balances the desire of developers to maintain control over the look and feel and functionality of their applications with the users' desire to know and control who has access to their private data. The Beeswax API forms secure communication channels between users and prevents data exfiltration by the applications. Beeswax allows the community to focus its scrutiny on just the platform, instead of all applications using it. Like all platforms, Beeswax's functionality is not set in stone. We anticipate that Beeswax will evolve to accommodate developer and user needs, and either move into the core of one or more browsers or be maintained by an open-source community.

Acknowledgements We would like to thank Arthur Loris for contributions to the implementation, as well as our shepherd, David Fifield, and reviewers for their invaluable feedback.

References

[1] "Mozilla Security Blog: Plugging the CSS History Leak," https://blog.mozilla.org/security/2010/03/31/plugging-the-css-history-leak/, 2010.

[2] "W3C Web Cryptography API, Candidate Recommendation," http://www.w3.org/TR/WebCryptoAPI/, December 2014.

[3] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *NDSS'2010*.

[4] M. Bellare and P. Rogaway, "Entity authentication and key distribution," in *CRYPTO '93*.

[5] D. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *Public Key Cryptography - PKC 2006*, vol. 3958, pp. 207–228.

[6] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *USENIX Security 2012*.

[7] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully Abstract Compilation to JavaScript," in *POPL '2013*.

[8] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "ShadowCrypt: Encrypted Web Applications for Everyone," in *ACM CCS 2014*.

[9] L. Ingram and M. Walfish, "Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves." in *USENIX ATC 2012*.

[10] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating JavaScript with Filters, Rewriting, and Wrappers," in *ESORICS'2009*.

[11] J. Magazinius, P. H. Phung, and D. Sands, "Safe Wrappers and Sane Policies for Self Protecting Javascript," in *NordSec'10*.

[12] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser," in *IEEE S&P 2010*.

[13] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized JavaScript," Google, Inc, Tech. Rep., June 2008.

[14] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting javascript," in *ASIACCS'09*.

[15] "ShadowCrypt github page," https://github.com/sunblaze-ucb/shadowcrypt, used commit fa2de4e61dd7c0f18d177ba4c810dc935c68f32d.

[16] E. Stark, M. Hamburg, and D. Boneh, "Symmetric Cryptography in Javascript," in *ACSAC '2009*.

[17] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting Users by Confining JavaScript with COWL," in *OSDI'2014*.

[18] "ADSafe homepage," http://www.adsafe.org.

[19] "Beeswax Platform: Home Page." https://web-priv.github.io/beeswax/, (code for platform and applications).

[20] "ChatStep: Secure / Private / Beautiful Online Group Chat. Home Page," https://chatstep.com/.

[21] "Chrome Developer: Content Scripts," https://developer.chrome.com/extensions/content_scripts.

[22] "Cryptocat: Project Homepage," https://crypto.cat/, (Site closed temporarily by developers in Feb 2016.).

[23] "Content Security Policy Level 2 W3C Recommendation," http://www.w3.org/TR/2015/CR-CSP2-20150219/, February 2015.

[24] "Shadow DOM W3C Working Draft," http://www.w3.org/TR/shadow-dom/, June 2014.

[25] "Chrome Developers: DOM Attributes now on the prototype chain," https://developers.google.com/web/updates/2015/04/DOM-attributes-now-on-the-prototype-chain, December 2010.

[26] "DOM Level 3 Event Flow (UI Events). Working Draft," https://www.w3.org/TR/DOM-Level-3-Events/#event-flow, December 2015.

[27] "Chatting off the record. Google Chat Help." https://support.google.com/chat/answer/29291.

[28] "Keybase.io homepage," https://keybase.io.

[29] "Kiwi IRC homepage," https://kiwiirc.com, used version 0.9.0.

[30] "Priv.ly project homepage," https://priv.ly.

[31] "Stanford Javscript Crypto Library (SJCL) homepage," http://bitwiseshiftleft.github.io/sjcl/, used version 1.0.0 compiled with AES and ECC.

[32] "Slack Privacy Policy," https://slack.com/privacy-policy.

[33] "Subrosa.io github page," https://github.com/subrosa-io/.

[34] "Public Search API for tweets (developer docs)," https://dev.twitter.com/rest/reference/get/search/tweets.