ShadowCrypt: Encrypted Web Applications for Everyone

Warren He
UC Berkeley
-w@berkeley.edu

Devdatta Akhawe
UC Berkeley
devdatta@cs.berkeley.edu

Sumeet Jain
UC Berkeley
sumeetjain@berkeley.edu

Elaine Shi
University of Maryland
elaine@cs.umd.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

ABSTRACT

A number of recent research and industry proposals discussed using encrypted data in web applications. We first present a systematization of the design space of web applications and highlight the advantages and limitations of current proposals. Next, we present ShadowCrypt, a previously unexplored design point that enables encrypted input/output without trusting any part of the web applications. ShadowCrypt allows users to transparently switch to encrypted input/output for text-based web applications. ShadowCrypt runs as a browser extension, replacing input elements in a page with secure, isolated shadow inputs and encrypted text with secure, isolated cleartext. ShadowCrypt's key innovation is the use of Shadow DOM, an upcoming primitive that allows low-overhead isolation of DOM trees. Evaluation results indicate that ShadowCrypt has low overhead and of practical use today. Finally, based on our experience with ShadowCrypt, we present a study of 17 popular web applications, across different domains, and the functionality impact and security advantages of encrypting the data they handle.

ACM Classification: D.4.6 Security and Protection

1. INTRODUCTION

Users today face the privacy diffusion problem: a number of web applications handle their data but users lack control of and visibility into who can access their data, who can modify it, and who can summarize or embed their data without permission. Violations of the user's expectation of data usage abound, from rogue employees [8] and government agencies [49], to unexpected changes in policies by the web application itself [20, 28].

A promising solution to this problem is providing only encrypted data to web applications. The user can control the decryption keys and only provide them to trusted principals. A large body of prior work in the cryptography community [7, 16, 44] as well as in the systems community [37, 38] discusses techniques to encrypt and compute on data handled by web (or cloud-based) applications.

Few, if any, of these proposals have achieved broad adoption. A possible reason could be that all the proposals require significant application rewrites. The resulting deployment and usability difficulty is an insurmountable mountain for typical users and developers. Requiring application rewrites also means that users cannot make the switch to encrypted data; instead, they have to wait for developers.

Clearly, there is a need for a secure, usable mechanism for encrypting data in web applications that puts the users back in control. For this reason, companies such as Virtru [48] have emerged. Virtru offers a browser plugin that performs email encryption, such that web-mail providers like Gmail cannot see users' data in the clear. However, Virtru provides only a point solution for a handful of Webmail providers, and does not generalize to other web applications.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660326.

1.1 Our Contributions: ShadowCrypt

We present ShadowCrypt, a general solution for encrypting textual data for existing web applications. With ShadowCrypt, security conscious users are back in control of their data: they have the *choice* of sending encrypted data to web apps (e.g., Gmail, Facebook, Twitter, Reddit, etc.), while still being able to use much of the functionality of existing web apps.[1]

ShadowCrypt sits between the web application and the user, where it captures user input and provides encrypted data to the application. When the application displays encrypted data to the user, ShadowCrypt again transparently captures encrypted text in the page and renders decrypted text instead. Our experiments indicate that ShadowCrypt causes minimal overhead on web pages, which is unnoticeable to the user.

ShadowCrypt is designed to be secure against potentially malicious or compromised web applications. Therefore, a key challenge in developing ShadowCrypt is successfully isolating private data from the web application's JavaScript and HTML code. While the browser provides primitives to isolate JavaScript code, the user only interacts with the DOM (i.e., the UI tree). Secure input/output requires securely isolating the application's DOM from the DOM containing the private data in the clear. To isolate the DOM, ShadowCrypt relies on Shadow DOM, an upcoming W3C standard already supported in modern web browsers like Google Chrome and Firefox. In this sense, ShadowCrypt's design minimizes the trusted computing base to the browser and ShadowCrypt itself. As we discuss in Section 2.1, in all previous proposals of encrypting data to web applications, the web application's JavaScript/HTML code can access the user's data in the clear, while in some proposals even the server-side PHP or Java code can access the user data in the clear.

ShadowCrypt defaults to random encryption, a good fit for textual data common in web applications. ShadowCrypt also supports deterministic encryption. This allows search to continue working without application modification, a trade-off also made in previous work [37]. To study to what extent ShadowCrypt would impact

[1]ShadowCrypt is available for download in the Google Chrome Store as well as open-source online [43].

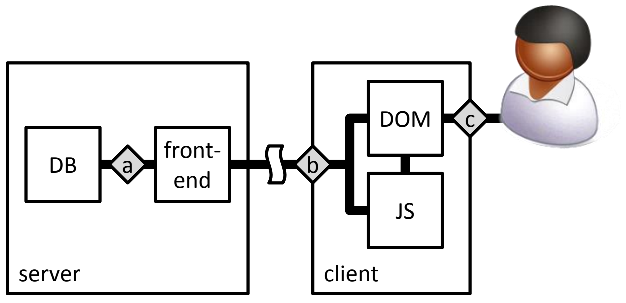


Figure 1: Architecture of typical web applications and chokepoints for data encryption.

the functionality of existing web applications, we tested ShadowCrypt with 17 popular web applications covering a broad range of functionality such as email, social networks, task management, and medical records. We found that the combination of deterministic and random encryption facilitates transparent data encryption while still maintaining prominent functionality in 14 out of the 17 applications we studied, including popular applications like Gmail.

In our current prototype, users manually share the encryption keys with collaborators through a separate, trusted channel. We designed ShadowCrypt to handle key distribution independently of its isolation system. We hope that ShadowCrypt provides a new, easily deployed platform for the broader community to develop and evaluate novel key management and sharing systems for security and usability.

Our experience with popular web applications is (to our knowledge) the first empirical study of the trade-off between functionality and data-encryption in wide-ranging applications such as publishing, task management, and medical records. Section 7 provides the details of our case studies. The issues we present help identify future work opportunities for encrypted data in the cloud. Some of the issues we identify are fundamental: for example, it is not clear how to support current advertising practices with encrypted user data. Supporting encryption requires advertisers to adopt new targeting and ad-serving practices.

2. PROBLEM AND APPROACH OVERVIEW

A number of recent proposals investigated encrypting data handled by web applications. In this section, we first systematize the design space and organize previous work throughout this space (Section 2.1). Then, we present ShadowCrypt and position it in the design space (Section 2.2). We also discuss our threat model and the security goals ShadowCrypt aims to ensure (Section 2.3).

2.1 Understanding the Design Space

Consider a simplified architecture of typical web applications (including our running example), outlined in Figure 1. The server consists of a database and a publicly accessible “frontend.” The database stores user data, and the frontend (written in, for example, PHP, NodeJS, or Rails) generates HTML from this data and serves it to clients over the network. The frontend also receives data from the client, processes it, and stores it in the database.

The client is a web browser, which transforms the HTML it receives into the “DOM,” a tree-structured representation of the application’s user interface. The user interacts with the application through the DOM.

The application’s HTML can also include JavaScript code, which the browser runs. JavaScript interacts with the user indirectly, through the DOM’s APIs. Both JavaScript (via, e.g., XMLHttpRequest) and the DOM (via, e.g., form submissions) can send data to the server. We refer to the combination of HTML, JavaScript, and CSS code as client-side code.

All proposals to encrypt web application data attempt to reduce the amount of code that can access data in the clear. Typically, these proposals operate at a “chokepoint” to ensure complete mediation. Figure 1 shows three chokepoints (a, b, and c).

An encryption system running at a chokepoint enforces the invariant that all code to the left (in Figure 1) of the chokepoint can access only encrypted data. Code to the right of the encryption system’s chokepoint is part of the application’s trusted computing base (TCB); it has access to data in the clear, and users must trust it to maintain the data’s confidentiality.

A system using a chokepoint farther to the right has the benefit of a smaller TCB, but it is less flexible. By contrast, a typical web application without any data encryption has a large TCB that encompasses all of its code, including the database.

(a) Between front end and database

Systems that rely on chokepoint (a) (Figure 1) give both the server-side and client-side application code (i.e., the DOM and JavaScript code) access to data in the clear. Data is encrypted before reaching the database server. This design protects against a curious database administrator or an untrusted/compromised database server.

Previous Work. CryptDB [37] adopts this design. CryptDB modifies the database schema to store encrypted values. CryptDB also includes a proxy that interposes on application queries and translates normal queries into queries on the encrypted database. This allows an application’s unmodified queries to work transparently, easing the system’s adoption.

Limitations. The user needs to trust *both* the server-side and client-side code for ensuring data confidentiality. While cryptographic and remote attestation techniques can help prove code identity [36], authentic applications could still be vulnerable to (for example) SQL Injection or XSS attacks leading to data theft.

CryptDB does support a user-aware mode to mitigate the large TCB, but using it requires providing the CryptDB proxy with the whole application’s access control logic, which impacts backwards compatibility. Further, if an administrator with access to all users’ data logs in, an adversary could still get all the data in the clear.

(b) Between the client and the network

A system at this chokepoint (Figure 1 (b)) allows only the application’s client side (i.e., JavaScript/HTML) to access private data in the clear. The application’s client-side JavaScript code encrypts data before sending any data to the server-side

Previous Work. This design is the most commonly used solution today. Popular applications encrypting at this chokepoint include password managers (e.g., LastPass), file hosting providers (e.g., Mega), messaging applications (e.g., CryptoCat), and secure note providers (e.g., LastPass).

Christodorescu proposed the idea of separating the client’s UI and networking code and inserting a crypto layer in between [9], but their proposal requires a browser and application rewrite. Mylar is an extension of the Meteor JavaScript framework for building applications that encrypt all their data sent to the server [38]. Developers need to write their applications in Meteor (affecting backwards compatibility) and tell Mylar what data needs encryption.

Limitations. In this design, everything to the right of the chokepoint (Figure 1 (b)) is part of the TCB. Thus, security of user data requires authenticating the client-side (i.e., JavaScript/HTML) code.

Bugs in the client-side code (e.g., XSS) could also compromise security by leaking data in the clear.

Applications such as LastPass and Mega trust their servers (via an HTTPS channel) to provide correct code and thus only protect against the passive adversary at the server side. CryptoCat is a browser extension and does not load code from remote servers (except, of course, during installation and updates). Mylar takes a middle route: it requires the developer sign all client-side code and a browser extension verifies this signature before allowing the load.

These authentication measures only prove the identity of the code, not that it is bug free. It is not trivial to implement encryption at chokepoint (b) correctly. The browser does not provide any API to interpose on all network channels. In addition to cross-site scripting attacks, the application needs to protect against HTML injection attacks, which can leak sensitive data without any JavaScript code execution.

For example, if an attacker can insert the HTML string `<img src='http://evil.com/log.cgi?` (unclosed single quote) before a secret value in HTML, the user's secret data would end up on the attacker's servers *without* any JavaScript involved. Zalewski [52] and Heiderich [24] identify a number of attacks to steal sensitive data even in the absence of code injection attacks.

Given the large size and complexity of modern HTML5 applications, ensuring correctness of client-side code and that it does not leak sensitive data is difficult. LastPass, Mega, and CryptoCat all have suffered from client-side vulnerabilities [10, 12, 29]

2.2 ShadowCrypt

ShadowCrypt works at the chokepoint (c) in Figure 1. This chokepoint encrypts data before the application code (including the client-side code) can access it. The application can only view an encrypted version of the data. This requires isolating the input and output fields while still providing the application access to the encrypted data.

Choosing this chokepoint means that no application code is in the TCB. This leads to a system secure against attackers at the client-side as well as the server-side. It also gives the user complete control over the data. In contrast, previous proposals required trusting application developers to handle data in a privacy-preserving manner.

The key challenge is providing a secure user interface and an isolated environment in which to store keys and to perform the encryption. ShadowCrypt relies on the browser extension infrastructure to provide a secure isolated environment for executing code and storing keys. Unfortunately, this is not sufficient since the user only interacts with the DOM (Figure 1), which is shared between the application and browser extensions. ShadowCrypt relies on a new upcoming primitive, the Shadow DOM [19], to securely isolate cleartext from the application code (Section 3).

Key Storage. ShadowCrypt stores encryption keys on the user's computer. Only the ShadowCrypt code has access to them. A ShadowCrypt user can share keys with anyone she wants via ShadowCrypt's key import/export interface. ShadowCrypt's user interface supports multiple keys at run time, and the user can choose which key to use. This design puts the user in control of her data and helps mitigate the privacy diffusion problem.

2.3 Threat Model

Our key threat model is the web attacker. We do not trust the application's server and client-side code (including the DOM and JavaScript code).

We declare two threats that ShadowCrypt does not defend against. These two issues are not addressed by the same-origin policy either.

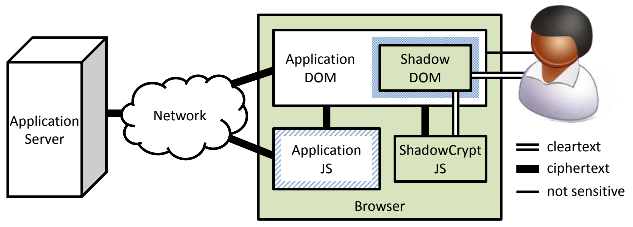


Figure 2: Overview of the ShadowCrypt architecture. ShadowCrypt runs in the browser, isolated from web pages but modifying the DOM via secure APIs. It creates an isolated Shadow DOM that interacts with the user.

Web security research is working towards addressing these two issues on the web in general.

First, ShadowCrypt does not defend against side channel data leakage. Because ShadowCrypt causes the page to render with the cleartext, the dimensions of the cleartext are available to the untrusted page. In comparison, many features of HTML, such as images, allow access to the dimensions resources loaded from other origins.

Second, one mode of text input supported by ShadowCrypt does not defend against clickjacking. ShadowCrypt allows users to enter text into a text field directly on the page (another supported method is to open a dedicated window). It may be possible to create an arrangement of transparent elements that intercepts keystrokes while showing a proper encrypted input underneath. In comparison, this kind of attack also affects sensitive cross-origin content that is included through an `iframe`.

Our system's TCB includes the operating system and the browser—we do not protect against a compromised browser or OS. Currently, we also do not provide protection against side-channel attacks such as length of text on the encrypted text.

2.4 Goals

Security Goal. ShadowCrypt aims to ensure that any data entered into a secure input widget that encrypts the data with key k is only visible to principals with knowledge of the key k. We rely on the user to authenticate the secure input widget via a passphrase (Section 5) and not to input sensitive data in a widget that lacks this passphrase.

Usability Goals. ShadowCrypt focuses on supporting current, popular web applications *transparently*. The user should be able to interact with the application in the same way whether ShadowCrypt is enabled or disabled. ShadowCrypt aims to preserve, as much as possible, the application's look and feel. A user should be able to use ShadowCrypt on any site without having to modify the site. This goal aims to ensure broad usability and backwards compatibility. Previous experience has shown that secure systems with serious usability constraints fail to achieve adoption beyond a niche.

Non-Goals. We do not aim to protect against denial-of-service attacks by the application. A trivial denial-of-service attack on our system would be for the web application to simply delete all encrypted data. We also do not guarantee the "freshness" of the application's data. Applications are free to display a previously entered value, whether by intended functionality or by malicious interference.

3. SHADOWCRYPT DESIGN

Figure 2 summarizes the ShadowCrypt architecture. ShadowCrypt runs as a browser extension and relies on the security of the user's browser and operating system.

The user interacts with the page as normal. ShadowCrypt seamlessly replaces encrypted data in the page with the cleartext stored in an isolated Shadow DOM. ShadowCrypt also replaces input elements in the page with new inputs, isolated from the page. The user provides her sensitive data to these isolated input elements and ShadowCrypt provides only encrypted text to the rest of the application's DOM and JavaScript code.

ShadowCrypt needs to isolate its own JavaScript code, which stores keys and performs encryption. We rely on the browser extension infrastructure to provide this isolated environment, which we describe in Section 3.1.

Isolating ShadowCrypt's JavaScript code is not sufficient for protecting the data that it handles. ShadowCrypt needs to show the decrypted data to the user as well as accept sensitive data input. This needs to occur in the application's DOM, since the user does not interact with JavaScript directly. In Section 3.2.1, we show how ShadowCrypt relies on an upcoming primitive, Shadow DOM, to securely isolate content in the DOM.

3.1 Isolating JavaScript

In order to meet our security goal, ShadowCrypt must isolate cleartext data, decryption keys, and its own logic from the application. ShadowCrypt is a browser extension, written in HTML and JavaScript just like a web page. Extensions can interact with and modify web pages via the page's DOM, just like scripts on the web page.

Browsers protect extension logic by running the extension code in a separate JavaScript environment—even code that interacts directly with the web page. The JavaScript isolation also includes isolated DOM APIs. Barth et al. present details about this isolation in Google Chrome [22]. This mechanism protects ShadowCrypt's logic from the untrusted page. Mozilla Firefox implements a similar isolation mechanism called NativeWrappers [35].

The extension's JavaScript code is more privileged than the page's JavaScript code. It can execute code in the page's JavaScript environment by inserting `<script>` elements into the page. But the page's JavaScript code cannot inject code into the extension's JavaScript environment.

3.2 Isolating DOM

The DOM isolation mechanism should isolate both input and output widgets, and it should maintain the look and feel of the application. Isolating parts of the DOM while preserving fine-grained styling has not been explored in previous work. In this section, we first discuss two unsatisfactory approaches. Then, we introduce the upcoming Shadow DOM standard and discuss how we used it in ShadowCrypt.

Strawman 1. `iframe`s leverage a browser's built-in frame and origin isolation properties. This provides strong, browser-vetted secrecy, but `iframe`s create discord, because they disregard the surrounding page's text styling.[2]

Furthermore, cross-origin `iframe`s are resource intensive, particularly given Chrome's plans for out-of-process `iframe`s [47]. ShadowCrypt creates a large number of isolated widgets for applications with many pieces of user data per page, like Facebook and Reddit.

Strawman 2. We could modify the DOM API such that it hides the secret content from JavaScript code, for example, by overriding the getters/setters of an input element to return encrypted values.

DOM has an intricate, complex API that is hard to reason about, so implementing this technique securely is difficult. We are not aware of any work on formalizing the DOM API and its semantics. Even if such a formal model existed, the DOM API is continuously evolving, and ShadowCrypt would always run the risk of vulnerability by falling out of sync with the browser's DOM implementation.

Second, there exist pure HTML mechanisms for sending data to the server, such as forms. Since form submission only involves the native browser behavior, it is not subject to our modified API, and it acts on the secret data.

```
<p>Enter your name: <input id="nameField"></p>
<p style="color:red" id="nameDisplay"></p>
<script>
nameField.addEventListener('change', function (event) {
  nameDisplay.textContent = 'Hello, ' + nameField.value + '!';
});
</script>
```

Figure 3: Simple Hello World Application.

3.2.1 Shadow DOM

ShadowCrypt relies on an upcoming standard, the Shadow DOM, for isolating cleartext input/output in the DOM. Shadow DOM specifies a way for an application to define a separate "shadow" tree for a particular node in the DOM. The browser then renders a composition of the main document and shadow trees. We explain the nature of this composition in more detail with an example in Section 3.2.2.

ShadowCrypt identifies input and output nodes in the main application document and defines a new shadow tree for each. The shadow tree contains cleartext, while the original node only sees the ciphertext. The browser composes the main document and shadow trees, and the user sees the cleartext from the shadow trees.

An explicit goal of the Shadow DOM standard is encapsulation. It specifies only a few explicit ways for the application to access the content of a shadow tree. The list of JavaScript objects that can cross the boundary between DOM and Shadow DOM is a whitelist[3] and ShadowCrypt redefines these objects to `null`s.

Keystroke events still traverse the encapsulation boundary. We rely on the privileged ShadowCrypt extension code to ensure that keystroke events for secure input do not trigger the application's listeners. ShadowCrypt checks the `target` property of any keystroke events and stops the event propagation if the target is a secure input widget.

Together, the above two ensure isolation of the shadow tree with clear text from the main document. The W3C is currently considering proposals to extend the Shadow DOM standard to include browser-vetted isolation [18].

3.2.2 Example

Consider a simple Hello World application (shown in Figure 3). The application waits for a user, Alice, to type in her name and updates the page with a simple greeting.

Input Shadowing. When Alice loads this page with ShadowCrypt enabled, the ShadowCrypt JavaScript code notices the presence of a text input element. ShadowCrypt creates a shadow tree for this input element containing a "shadow" input. The shadow input handles the cleartext data in place of the original, application-provided input element (Figure 4, left). The shadow input is in a separate shadow tree, but the browser composes it with the main document and renders the shadow input element next to the "Enter your name:" message.

[2]The `seamless` attribute could help with the styling issue, but neither Chrome, Firefox, nor Internet Explorer support it.

[3]Appendix A discusses this list in more detail.

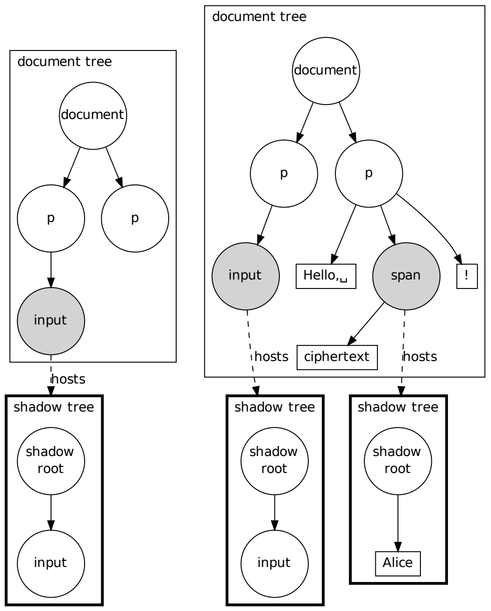


Figure 4: Left: The example application in its initial state. A text input hosts a shadow input. Right: The example application presenting a personalized greeting. A `span` element hosts cleartext.

```
<p style="color:red" id="nameDisplay">
    Hello, <span>Alice</span>!
</p>
```

Listing 1: The browser renders a composed tree equivalent to this HTML snippet

When Alice clicks on the *rendered* input element and starts typing, she is interacting with the shadow input element created by ShadowCrypt. As she types, the browser notifies ShadowCrypt of each keystroke. Each time Alice types a letter of her name, ShadowCrypt reads the cleartext in the shadow element, encrypts it, and updates the original input node's value with the ciphertext. The encrypted text also includes the fingerprint of the key used for encryption as well as a sentinel string (`=?shadowcrypt`) to help easily identify ciphertext in the page.

Application Processing. When Alice finishes typing, the application receives a `change` event. Line 4, Figure 3 reads the `value` of the original input node, which is the ciphertext set by ShadowCrypt. The application concatenates it with "Hello" before setting the `textContent` property of the `p` element (Line 5).

Output Substitution. When the application sets the `textContent` property, it adds a text node to the document. ShadowCrypt's JavaScript code notices the sentinel value in the text node and finds the ciphertext inserted. ShadowCrypt then uses the key fingerprint to identify the decryption key and attempts to decrypt the ciphertext.

If decryption succeeds, ShadowCrypt's JavaScript code has the sensitive data in the clear. It now needs to show this data to the user while still isolating it from the application. To do this, it surrounds the ciphertext string with a `span` element, creates a shadow tree for the `span` element, and writes the cleartext into the shadow tree. Figure 4 (right) shows the resulting DOM trees.

The composed tree of the output `p` element would look like the HTML in Listing 1. Alice sees a personalized greeting as the original application would show.

3.3 Defending against Active Client-side Attacker

Providing a thorough defense against an active attacker at the client side is difficult. We found a number of challenges in defending against such a powerful adversary, especially while building a user interface that fits in with the untrusted page's visual style. We discuss two concrete attacks below.

In order to match the page's look and feel, ShadowCrypt inherits CSS styles when displaying data to the user. Combined with the ancillary information such as the dimensions of the output element, an active attacker can learn information about the cleartext displayed to the user. For example, Heiderich et al. present an efficient way to determine the characters in a string from its size by using custom fonts [24]. Similarly, Stone demonstrated a number of cross-origin timing attacks leaking text across cross-origin `iframes` [46].

For seamless integration, ShadowCrypt allows untrusted content and secure input/output widgets to run in a single page. This design is vulnerable to mashup integrity attacks like clickjacking. For example, an attacker could try to layer an unencrypted input over a passphrase-bearing encrypted input.

Defending against such UI attacks is a broad problem. Extensions such as NoScript already provide protections against such attacks, while Huang et al. present the InContext defense, which is also on standards track at the W3C [25, 32].

Both the attacks above result from our design constraints of seamless integration and usability. It is a trivial extension to use separate tabs/windows instead of Shadow DOM; such a design would be secure from the clickjacking attack mentioned above. We implement this kind of input for discretionary use: pressing a keyboard command (Ctrl+Backtick) with a secure input selected causes ShadowCrypt to open a dedicated window with the user's passphrase and a single text input.

Mitigating the first attack is also easy, as long as we disregard the page's look and feel: the Shadow DOM specification allows the construction of shadow trees that do not inherit styles.

Our experience shows that enacting these protective measures tremendously limits the usability of our system, and thus we default to in-page input and output despite the vulnerabilities above. In the future, we will investigate mechanisms that compromise between information leakage and styling flexibility.

3.4 Functionality Impact

Enabling ShadowCrypt means that the main application code never gets access to the user data in the clear. As a result, the unmodified application cannot process the data (save for operations that work equally well on encrypted data). Depending on the application, this can minimally or severely impact the functionality of the application. We discuss our experience with various classes of applications and the functionality impact in Section 7.

One limitation that we observed broadly is that the application code can no longer sanitize the encrypted text. Rich text inputs (such as comments or blog posts) normally work by providing HTML code to the application. The application can then sanitize it before rendering it again. However, with ShadowCrypt enabled, the application only sees an encrypted blob and cannot sanitize the untrusted input values. If ShadowCrypt rendered this potentially untrusted input, it would create an XSS vulnerability.

Instead, in our current design, we always render the decrypted value as plaintext. We can also add support for simple formatting tags, but the fundamental limitation remains: ShadowCrypt does not have information on which HTML constructs to allow or disallow.

=?shadowcrypt-`ce4d75a0b5022f0b31cf62e150474d3`
`0adced1007de8b46c82a546fa9eff97a1`**?**`/FnXuSdlLi9`
`UEV554/CF9RYeMQ/le2ZkErLeyDU=`**??=**

Figure 5: An encrypted messsage. We have set delimiters of the major parts in bold.

4. SHADOWCRYPT IMPLEMENTATION

We have implemented ShadowCrypt as a Google Chrome browser extension. The extension is available on the Chrome Store for anyone to try out [43]. Since we do not use any non-standard features, we believe ShadowCrypt should work just as well on any other modern, standards-compliant browser with extension support. We are currently working on an implementation for Firefox.

4.1 Shadowing Application UI

ShadowCrypt monitors the application's user interface to identify encrypted text and input widgets, inserting appropriate shadow inputs and outputs as needed. Content can make its way into the application's web page in different ways, such as when the page first loads as an HTML file, when JavaScript code inserts markup through `document.write()`, or when JavaScript code uses the DOM API to modify the document tree.

To monitor the page as it changes, ShadowCrypt relies on the standard `MutationObserver` interface to register a callback that the browser will invoke after every modification to the document. Recall that while the application code and ShadowCrypt share the DOM, the browser provides trusted, isolated (or native) DOM APIs to ShadowCrypt. Relying on `MutationObservers` has the added advantage that these observers also run as the page is progressively rendered.

ShadowCrypt searches all insertions for HTML text input widgets and inserts shadows as needed. ShadowCrypt also supports the `textarea` element and elements with the `contentEditable` attribute set to true. The core idea behind supporting these elements is the same as presented in Section 3.2.2 for `input type=text` elements. We discuss the element specific details in Appendix B. The essential concept behind ShadowCrypt applies equally to input fields of type "file." We are currently working to expand our coverage to files, particularly images.

One limitation of our current implementation is that creating a shadow tree does not result in a mutation event: shadow trees are separate from the main document. As a result, an application using Shadow DOM for its own input would not work with ShadowCrypt today. We did not find any application currently relying on Shadow DOM.

To work around this limitation, we can modify the application's DOM APIs to detect new shadow trees and insert ShadowCrypt mutation observers into the shadow trees as well. We have not done this in our implementation, as we did not find any application relying on Shadow DOM right now. We stress that the above issue does not affect the security of ShadowCrypt. We already require the user to authenticate ShadowCrypt inputs via a passphrase.

4.2 Encrypting Text

ShadowCrypt encrypts text using the AES-CCM algorithm. In the near future, the Web Cryptography API will make native and hardware-based cryptography available to the web platform. For now, we used the Stanford JavaScript Crypto Library's (SJCL) implementation [45]. Figure 5 presents an example of an encrypted message. This encoded payload includes a format signature, the fingerprint of the key used to encrypt, and the encoded ciphertext, including initialization vector and authentication data.

Format Signature (`=?shadowcrypt-`) This eases implementation of output decryption. ShadowCrypt's mutation observer examines content as it enters the document tree. The format signature explicitly marks all ciphertext strings, making them easy to find via a regular expression search.

Key Fingerprint (`ce4d`...`97a1`) ShadowCrypt uses this fingerprint to look up the right key to decrypt the ciphertext, since there may be data encrypted with different keys on the same page. A question mark follows this as a separator.

Payload (`/FnX`...`yDU=`) This base64-encoded string internally consists of the random initialization vector, the raw ciphertext, and the message authentication code. Another question mark follows this as a separator.

EOF Finally, the sequence "`?=`" denotes the end of the ciphertext string.

4.2.1 Other Encryption Schemes

ShadowCrypt, by default, uses AES-CCM with a random IV for maximum security, but it can easily support arbitrary encryption schemes for more functionality. We currently also support a searchable encryption scheme that works transparently with existing web services such as Gmail and Facebook.

The key difference between the searchable encryption scheme and the default scheme is the addition of encrypted keywords at the end of the encrypted message. When encrypting text, ShadowCrypt's code computes a deterministic hash (keyed with the encryption key) of each unique word in the input. ShadowCrypt then appends these hashes at the end of the encrypted text sent to the application.

To search, ShadowCrypt prepares an encrypted query by performing just the keyword extraction procedure on the cleartext query, resulting in the hashes of each cleartext keyword in the query. The application can use its original keyword searching functionality to find encrypted documents from this query.

Our searchable encryption scheme creates a new side-channel vulnerability, but we chose this scheme because it works transparently with the search functionality of current web applications. It is not difficult to support other (more-secure) searchable encryption schemes if we can modify application code. Further, users interested in stronger security can always use the more secure encryption schemes (but without search functionality).

We also built a variant of ShadowCrypt that uses asymmetric encryption. This variant uses the OpenPGP.js library [40] to handle the cryptographic operations. Thus far, we have not incorporated PGP into our release due to performance reasons. The asymmetric cryptography takes a long time to process each message. The slowest part has been decrypting a message, when we used RSA keys. The keys tend to have small public exponents, so encrypting was much faster than decrypting. This meant that, while one could enter text relatively smoothly, the page would freeze up whenever the application tried to display encrypted content. See Section 6 for our timing measurements.

4.2.2 Manifest Files

By default, ShadowCrypt applies AES-CCM encryption to all input elements on a page. Often, encryption of all inputs would break critical functionality. For example, while encrypting the email body in Gmail is OK, encrypting the contents of the "To" field would render the message undeliverable. ShadowCrypt supports a keyboard command (Ctrl+Space) to disable encryption on a particular input widget.

To ease usability, we have also implemented support for manifest files in ShadowCrypt. These files specify which input fields

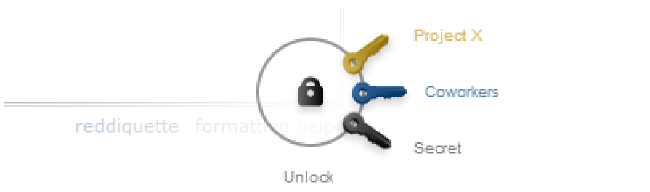


Figure 6: User interface for managing keys at the input widget. Clicking on "Unlock" turns off encryption, and clicking on a particular key encrypts with that key.

ShadowCrypt should not encrypt. Manifest files can also list which input fields should apply the searchable encryption scheme or the encrypted query generation scheme. We envision a centralized, trusted market for these manifest files. We currently have manifest files for Gmail, Twitter, Facebook, and Asana.

Manifest files are a usability improvement and do not affect the security of ShadowCrypt. Users can still verify the presence of secure input based on the passphrase.

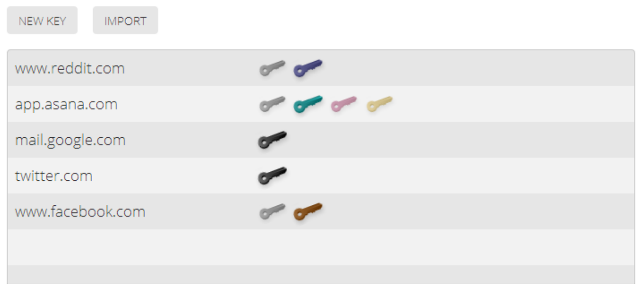
5. USER INTERFACE

Our goal with ShadowCrypt is transparent usability as well as security. A good user interface that helps the user make the right decision by default is a key component of ShadowCrypt. Critical to security is ensuring that the user only enters sensitive data in the ShadowCrypt elements and a good key management interface.

Authenticating to the User. ShadowCrypt ensures that the user only enters clear text input into the secure shadow inputs by authenticating shadow inputs with a secret passphrase. While setting up ShadowCrypt, the user and ShadowCrypt identify a secret passphrase. ShadowCrypt includes this passphrase in all the shadow inputs it creates.

We rely on the user to input sensitive data only after checking for the presence of this passphrase. Figure 6 shows an example widget with the secret passphrase. Such reverse passwords are common in web applications [2, 3].

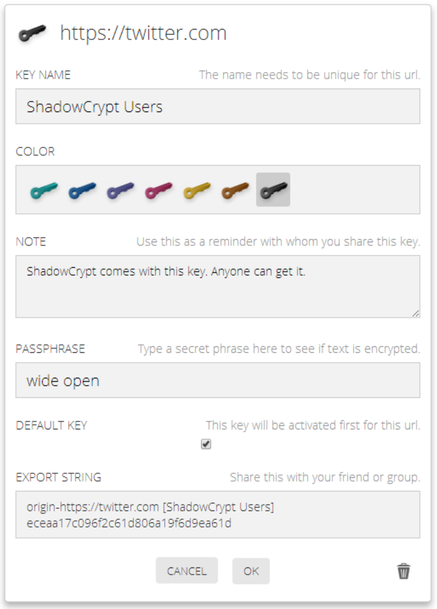
Key Management UI. ShadowCrypt maintains a key database in the private storage area that the browser provides to each extension. Google Chrome automatically synchronizes this database across all of a user's browsers. The database supports an arbitrary number of keys tied to each application (identified by the origin). Our released version of ShadowCrypt only supports symmetric encryption, and keys are 128 bits. Figure 7 shows the key management page for ShadowCrypt.



www.reddit.com
app.asana.com
mail.google.com
twitter.com
www.facebook.com

Figure 7: Main ShadowCrypt key management user interface.

When the user clicks on a particular key icon, the individual key's management page pops up (Figure 8). Each key has a friendly mnemonic name and a unique color. Users can also set a default key for the application. Users can import and export keys by copying/pasting a text string. ShadowCrypt also allows users to annotate keys with text reminders to help remember their history and provenance. ShadowCrypt thus relieves the privacy diffusion problem by putting the users back in charge of sharing data (by controlling key distribution).



https://twitter.com

KEY NAME The name needs to be unique for this url.

ShadowCrypt Users

COLOR

NOTE Use this as a reminder with whom you share this key.

ShadowCrypt comes with this key. Anyone can get it.

PASSPHRASE Type a secret phrase here to see if text is encrypted.

wide open

DEFAULT KEY This key will be activated first for this url.

✓

EXPORT STRING Share this with your friend or group.

origin-https://twitter.com [ShadowCrypt Users]
eceaa17c096f2c61d806a19f6d9ea61d

Figure 8: User interface for managing individual keys, reached by clicking on a key icon in Figure 7.

Key management is a well-known, hard, but important problem that is orthogonal to our work. We designed our system to handle key distribution independently of the isolation system that we have developed. ShadowCrypt also provides a new platform for the broader community to develop and evaluate novel key management systems. Many of today's well-known encryption products also rely on the user to manually export and share keys, including Google's recently released End-to-End extension. PGP uses a collection of public key servers to distribute keys and associate them with identities. We are currently working on integrating several key management mechanisms such as keybase.io [27] and cloud-based key management services proposed in usable security literature [14].

We are currently extending the current key management mechanism in three directions. First, we are adding support for per user (or user group) keys instead of the current per-site keys. A user can designate a particular key (across applications) for (say) her family or for her co-workers. We believe that such a design will be more intuitive for the typical user. Second, we are investigating designs for managing public keys and specifying multiple recipients for our PGP-based variant. Finally, we are investigating an easier interface for sharing and distributing keys (e.g., via a centralized service, such as Keybase [27]).

The shadow input widgets that ShadowCrypt creates have a couple of features to help with key management. First, the shadow input widget has a color border around it that matches the color of the key currently in use. Second, ShadowCrypt adds a lock icon at the bottom right corner of the widget. Clicking the lock icon, users can choose alternate keys or disable encryption altogether (Figure 6).

6. PERFORMANCE EVALUATION

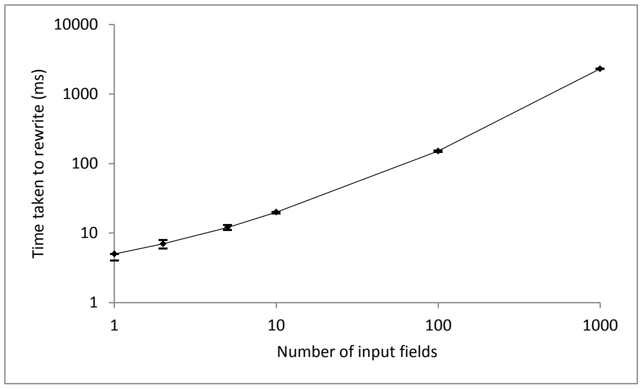


Figure 9: Time taken for ShadowCrypt to create shadow inputs. Median and quartile measurements from 100 trials on pages as the number of inputs varies.

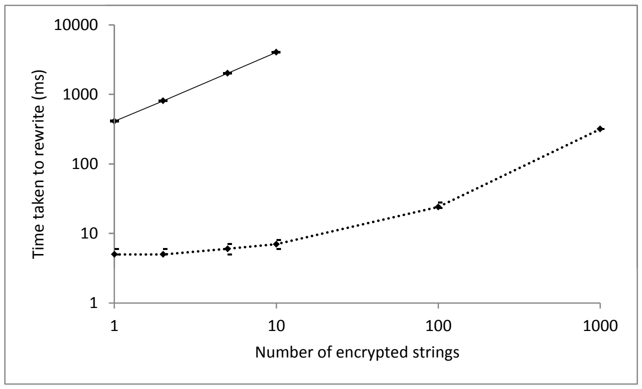


Figure 10: Time taken for ShadowCrypt to rewrite encrypted messages on a page. Median and quartile measurements from 100 trials on pages as the number of encrypted messages varies. Solid line shows PGP variant. Dotted line shows AES-CCM variant.

Next, we discuss the performance overhead of ShadowCrypt on the user's browsing.

We first measured the overhead introduced by ShadowCrypt's `MutationObserver` to the user's day-to-day browsing. We conducted these tests on an Intel 2.5GHz x 4 with 8GB of RAM. We ran the Dromaeo DOM benchmark [13]. This test only measures the overhead introduced by ShadowCrypt to a page's normal operation since the page does not contain any encrypted input/outputs. We found that the Dromaeo benchmark scores dropped from 1,617 runs/second to 1,153 runs/second. We find that this loss is not noticeable by a user. In contrast, popular extensions like LastPass (with over a million users) drop the score to 1494 runs/second.

Second, we measured the performance overhead of ShadowCrypt as the number of encrypted input/output elements changed. We conducted these tests on an Intel Core i7 3.40GHz with 16GB of RAM.

We created pages with between 1 and 1000 input elements and between 1 and 1000 encrypted messages (each 5 characters long). We measured the extra time ShadowCrypt takes to replace these elements with their isolated shadows containing the cleartext.

Figure 9 and Figure 10 plot the median time overhead for 100 trials for each case. We find that ShadowCrypt has an overhead of 151 ms for 100 inputs and 24 ms for 100 encrypted messages. The PGP variant was much slower at decrypting messages, taking 4 seconds on just 10 encrypted messages.

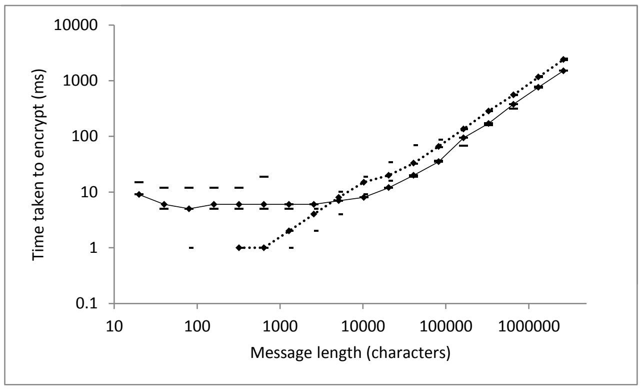


Figure 11: Time taken for ShadowCrypt to encrypt a message. Median and quartile measurements from batches of 100 messages for each length. Solid line shows PGP variant. Dotted line shows AES-CCM variant.

Such a number of inputs/outputs are not representative of typical web application behavior. For example, Gmail normally loads 50 email threads on the inbox page and only has 3 input elements. ShadowCrypt's impact on performance in those cases is negligible.

To measure the overhead of cryptographic operations as the length of the message varies, we measure the timing of ShadowCrypt's encryption functions as we varied the length of cleartext from 20 to 2,621,440 characters. Figure 11 shows the results of 100 trials of each. We find that encryption takes under 1 ms for messages up to 640 characters long. Thus, ShadowCrypt's repeated encryption for user inputs does not add any noticeable overhead on a page.

The Web Cryptography API is already available in Chrome's Canary and Firefox's Nightly release channel. We benchmarked the Web Cryptography API in Google Chrome Canary and found that it performs AES operations 4.7 times as fast as SJCL (encrypting a 1 KiB message with a 256-bit key 1,000 times) and RSA operations 12 times faster than OpenPGP.js (encrypting/decrypting a 256-bit message with a 2048-bit key 1,000 times). It performs better than the JavaScript-based cryptography implementation we used in our overall performance tests, so we have not tried to optimize the cryptography code.

To estimate ShadowCrypt overhead for real-world applications, we loaded a typical page from three of our case studies (discussed next) and estimated the overhead if all inputs and user-generated strings switched to using ShadowCrypt. Table 1 lists the number of user-generated strings and input fields for each page and the estimated load time increase in milliseconds. To estimate the percentage overhead, we measured the page load time on a cold and hot cache.[4] Table 1 also lists these times and the percentage overhead introduced by ShadowCrypt in each case.

Overall, we find that ShadowCrypt does not impose significant overheads on the page, with a high of 8% overhead on hot cache for a long Reddit comments page. Our estimates of load times are conservative (particularly in the hot cache case). Google Analytics' Site Speed Data, which measures real-world page-load times (even with caching), reports a median (mean) page load time of just under 3000 ms (6000 ms) around the world [26].

7. CASE STUDIES

[4]We used the `webpagetest.org` infrastructure (standard amongst the Web Performance community) with the connection type set to "Cable" [34]. The load times on our machines are comparable.

Application	**Str**	**In**	**OH (ms)**	**Load (ms)**	**Load2 (ms)**
Reddit comments page	196	6	61	2,507 (2%)	731 (8%)
Twitter user page	33	13	49	3,017 (2%)	1,492 (3%)
Facebook profile page	25	1	23	2,984 (1%)	2,177 (1%)

Table 1: Load time of a typical page in popular web applications. "Str" is the number of user-generated strings present on the page. "In" is the number of text inputs on the page. "OH" is the estimated overhead from using ShadowCrypt. "Load" is page load time (without ShadowCrypt) and the overhead as a percentage of this time. "Load2" is the load time with the cache populated and the overhead as a percentage of this time.

Application	**Fields encrypted**
OpenEMR	freeform patient information
WordPress	posts and comments
Blogger	posts (plaintext only) and comments
Tumblr	text posts
Reddit	text submissions and comments
Pinterest	submission descriptions and comments
Asana	task titles, descriptions, and comments
Trello	task titles, descriptions, and comments
Wunderlist	tasks titles, descriptions, and comments
Etherpad classic	document and chat
Gmail	subject and body
Twitter	tweets
Facebook	status updates and comments

Table 2: List of case studies that retained prominent application functionality and the fields encrypted.

We tested the ShadowCrypt extension on a wide variety of popular applications that handle textual data, which is the focus of ShadowCrypt. While encrypting data always impacts some application functionality, we find that for a broad range of applications, encrypting textual data still retains prominent functionality.

ShadowCrypt's contribution is providing the *option* of encrypting text and putting the user back in control of her data. ShadowCrypt is a general mechanism for secure input/output in web applications. Our experience (discussed below) highlights the challenges and limitations of moving to encrypted data for web applications. Future work can investigate how we can enable lost functionality by modifying application code.

7.1 Applications Retaining Prominent Functionality

Table 2 lists 14 applications that retained prominent functionality when we used them with ShadowCrypt. Table 2 also lists the data encrypted with ShadowCrypt. After switching to encrypted data, typical functionality affected includes targeted advertising and rich text output. Encryption also disables application-mediated data sharing; instead the user needs to explicitly share keys with users she wants to share data with.

OpenEMR manages a database of patient data. Typical input to the OpenEMR application is free-form text. ShadowCrypt was able to encrypt all the free-form text fields including patient name, symptoms, physician notes, and so on. We used the deterministic encryption scheme to allow search, but this meant that non-keyword searches stopped working. For example, searching for "Jo" would previously return patients named "John," but would not under our deterministic encryption scheme. Nonetheless, this did demonstrate that ShadowCrypt is scalable to applications with a large number of user input fields with no additional effort.

We also tested publishing applications, namely **WordPress**, **Reddit**, **Blogger**, **Tumblr**, and **Pinterest**. For all these applications, ShadowCrypt was able to support encrypted post-titles, posts, and comments. Per Section 3.4, ShadowCrypt does disable rich-text input/output including turning URLs in plaintext to links.

For all the applications above, the decrypted post title correctly showed up in the page body. However, ShadowCrypt could not replace the value in the `title` tag (shown in the tab/window bar), since HTML does not allow markup (i.e., the shadow tree) in the `title` element.

We also tested ShadowCrypt with popular task management applications such as **Asana**, **Trello**, and **Wunderlist** (each of which has millions of users [1, 11, 15]). We used ShadowCrypt to encrypt any textual task data. These three services all supported task descriptions and comments, which we were able to encrypt with ShadowCrypt. Additionally, all three of these services support realtime sync across multiple clients, which continued to work.

ShadowCrypt did break Asana's "diff" feature. Asana uses this to highlight modifications to a particular task but in the case of encrypted tasks, the server-side diff mangled the encoded ciphertext.

A possible solution for this issue is that ShadowCrypt offer a platform for side-effect free computations; the server can provide JavaScript code (e.g., diff) that ShadowCrypt could execute on the decrypted data in an isolated environment. This solution could also mitigate the formatting/sanitization issue discussed above. Of course, this would require modifications to the application.

We also used **Gmail** with ShadowCrypt. We were able to use ShadowCrypt to encrypt the subject and body of email messages. Gmail was not able to serve relevant ads next to email threads; it mistakenly showed ads related to short letter sequences present in the ciphertext. One feature that broke due to ShadowCrypt was the message preview in message lists (e.g., at the inbox)—the snippet preview in the thread list view would truncate the body's ciphertext and ShadowCrypt refuses to decrypt the invalid ciphertext.

We also used ShadowCrypt to tweet on **Twitter** and post status updates on **Facebook**. In both cases, ShadowCrypt successfully encrypted and decrypted the messages. Since the servers do not see the contents of the messages, they cannot detect "mentions" of another user and send notifications. While it is trivial to modify ShadowCrypt to not encrypt any word following an @-sign, it is not clear whether users would understand the security implications of this change.

Twitter's length limitation combined with our lengthy ciphertext encoding meant that using ShadowCrypt limited users to 45 character tweets. A specialized encoding scheme will fare better,[5] but we did not investigate this further.

Encryption and Key Sharing. ShadowCrypt allows a user to switch to encrypted data. The user can now choose the strength of encryption and collaborators to share keys with. For example, deterministic encryption schemes allow search functionality but without the security offered by random encryption.

The user also needs to share keys with other users to share data. For example, social applications such as Twitter, Facebook, Gmail or any blogging application, the user will need to manually share the encryption keys. While an additional step, it is a necessary one to put the user back in control of her data.

[5]Recall that Twitter's length limitation is really 140 *Unicode codepoints*, not ASCII characters.

Application	Comments
Google Drive Spreadsheets	cannot evaluate encrypted formulas; static data only
Google Drive Docs	custom keystroke-based input; would require application redesign to encrypt
Office 365	
Etherpad lite	

Table 3: List of applications that did not work or had severe loss of functionality with ShadowCrypt.

In the case of applications like Wunderlist (for task management), the need for sharing keys and the encryption strength depends on the user. Often, users rely on task-management applications for personal use without sharing their data with others. We also found these list-based applications usable without search functionality. Again, this is a user-specific decision.

7.2 Applications with degraded functionality

In other cases, turning on ShadowCrypt severely degraded application functionality or ShadowCrypt was unable to achieve encryption for textual data. Table 3 lists these applications and the reasons for their lost functionality.

Spreadsheets. On using ShadowCrypt with the Google Drive **Spreadsheet** application, we immediately hit an error: our cipher-text encoding scheme surrounds the cipher-text with a sentinel value that starts with an =-sign. The spreadsheet program interprets this as an (invalid) formula and throws an error.

We were able to work around this by using a different sentinel string. This experience does point to a fundamental issue: whatever sentinel string we choose, we always run the risk of interfering with some application.

The ShadowCrypt+Google Spreadsheets application works best with textual data and Google Forms, which in turn fill in data into the spreadsheet. Using ShadowCrypt does break functionality like sorting and arithmetic on numeric values.

Word Processing Applications. We tested **Google Drive**, **Office 365**, **Etherpad classic**, and **Etherpad lite**. Of these, ShadowCrypt only succeeded in encrypting the document in Etherpad classic.

On investigating further, we found that Google Drive Docs, Office 365 Word, and Etherpad lite, did not rely on standard HTML input widgets. Instead, these applications relied on keystroke events to build their own text editing functionality. In view of such a design, it is unlikely that a ShadowCrypt like system can ever work with these applications without changes.

Further, while ShadowCrypt does work with Etherpad classic, it does break the author attribution feature. Etherpad classic tries to maintain the author information of each character in the document. Due to the nature of random encryption, each changed character appears as if the user rewrote the entire document. One direction for future work is how to support such a design while still maintaining the privacy of document content.

8. RELATED WORK

We provided a detailed comparison to closely related work in Section 2.1. Here, we discuss literature we did not cover earlier.

Privly [39] is a browser extension that allows users to share encrypted text on existing web applications, somewhat like ShadowCrypt. Instead of storing encrypted text with the web application, Privly stores the encrypted text on a third-party dedicated storage server. Privly creates a hyperlink to the message with the decryption key in the hyperlink's fragment identifier [33]. Privly sends this hyperlink to the web application, in place of the text. Thus, the decryption key is never sent to the storage server, but it is visible to the web application's client-side code. In contrast, ShadowCrypt does not trust the web application with the decryption key. Additionally, Privly requires a dedicated storage server, which increases the cost and reduces performance. Finally, Privly relies on replacing hyperlinks with `iframe`s that renders the decrypted text, which has performance and usability limitations (Section 2.1).[6]

Virtru [48] is another browser extension, which focuses on email. Virtru supports Outlook, Gmail, and Yahoo! Mail. Users of the Virtru extension interact with Virtru's own server to exchange encryption keys. This allows the service to revoke access to messages after they have been sent. ShadowCrypt, by contrast, is more general, and targets any web application. ShadowCrypt users share keys through any channel other than the untrusted site, so there is no way to revoke access to a message that has already been sent.

A number of researchers proposed cryptographic constructs that allow an untrusted server to "blindly" compute on encrypted user data. Constructions for general functions include Fully Homomorphic Encryption [16], Functional Encryption [6, 42], Oblivious RAM [21], and secure computation in either the circuit [51] or the RAM model [23, 30]. Researchers have also proposed more efficient schemes for specific functionalities, such as searchable encryption [7, 44].

While these schemes offer strong security guarantees, they have a high performance cost and often require a rewrite of application code handling encrypted data. As a result, we do not currently rely on them but these schemes are not at odds with ShadowCrypt. Applications that want to rely on such schemes only need to modify their code and implement the scheme in ShadowCrypt as another type of encryption algorithm.

Instead of relying on encrypted data in the cloud, another option is to rely on remote attestation and trusted computing techniques to ensure that the cloud handles the data per the user's expectations. Maniatis et al. provide an overview of all the related techniques in this direction [31]. We are not aware of work in this direction that specifically targets web applications.

Data breaches and privacy violating information flows in web applications can occur due to bugs (such as missing access checks). A large body of research aims to improve the developer's ability to reason about data flow in web applications: Giffin et al. provide a comprehensive survey of research in the system's community [17]. Yang et al. [50] present Jeeves, a new language mechanism for enforcing access checks as well as survey research in the programming languages community.

These techniques typically require application rewrites and only protect the user from unintentional developer mistakes. In contrast, ShadowCrypt does not require application rewrite and aims to put the user back in control—intentional or unintentional violations of user expectation by the developer notwithstanding.

ShadowCrypt aims to provide a secure, isolated UI widget inline to an application. Roesner et al. propose secure widgets as a permission granting mechanism [41]. Huang et al. discuss a secure defense against clickjacking attacks [25].

9. CONCLUSION

We presented ShadowCrypt, a system for transparently switching to encrypted text for web applications. In contrast to previous approaches, ShadowCrypt does not trust any part of a web application with the user data. Instead, ShadowCrypt puts the user back

[6]We were unable to conduct a security review of Privly's isolation mechanism because it is currently invitation-only. We did not find a publication detailing its technical architecture.

in control by sharing keys with the principals she wants. We also highlighted the functionality and usability challenges of switching to encrypted data in modern applications. Future work can focus on extending the ShadowCrypt design to protect against active attackers.

ShadowCrypt's secure infrastructure and usable interface design provide a basis for implementing wide variety of encryption schemes. We are currently working on supporting additional schemes that can work transparently such as Format Preserving Encryption (FPE) [4] and Attribute-based Encryption [5,42] (ABE). The former allows us to textual and non-textual fields that place constraints on its format (e.g., images) and the latter enables easier key management. In the longer run, we aim to support encryption schemes that rely on modifications to existing web applications to work, such as Searchable Encryption [7,44], or Fully Homomorphic Encryption [16].

ShadowCrypt's contribution lies in providing the user the *choice* of encrypting arbitrary fields—the user can enable/disable (random or deterministic) encryption as she desires. ShadowCrypt also supports manifest files that identify fields to encrypt/not-encrypt and the algorithm to use for a particular application. We envision seeding ShadowCrypt with sane default manifests for popular applications.

10. ACKNOWLEDGEMENTS

We would like to thank our shepherd Graham Steel, Joel Weinberger, Prateek Saxena, Matthew Finifter, Ashley Thomas, and Jassiem Ifill for their useful feedback. This material is based upon work supported by the National Science Foundation under Grants No. 0831501 and 1239166. This material is based upon work supported by the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539. This work is partially supported by the Office of Naval Research under MURI Grant No. N000140911081. This research was supported by Intel through the ISTC for Secure Computing. This research is partially funded by the National Science Foundation under grant CNS-1314857, a Sloan Research Fellowship, as well as a Google Faculty Research Award.

11. REFERENCES

[1] 6WUNDERKINDER. Let's talk comments for wunderlist and our 5+ million users. `http://goo.gl/PcjOR6`.

[2] AGARWAL, N., RENFRO, S., AND BEJAR, A. Phishing forbidden. *Queue 5*, 5 (2007), 28–32.

[3] AGARWAL, N., RENFRO, S., AND BEJAR, A. Yahoo!'s sign-in seal and current anti-phishing solutions. In *Proceedings of Web 2.0 Security & Privacy Workshop* (2007).

[4] BELLARE, M., RISTENPART, T., ROGAWAY, P., AND STEGERS, T. Format-preserving encryption. In *Selected Areas in Cryptography* (2009), pp. 295–312.

[5] BETHENCOURT, J., SAHAI, A., AND WATERS, B. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), SP '07, pp. 321–334.

[6] BONEH, D., SAHAI, A., AND WATERS, B. Functional encryption: Definitions and challenges. In *Proceedings of the 8th Conference on Theory of Cryptography* (2011), TCC'11, pp. 253–273.

[7] CASH, D., JARECKI, S., JUTLA, C. S., KRAWCZYK, H., ROSU, M.-C., AND STEINER, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO* (2013), pp. 353–373.

[8] CHEN, A. Gcreep: Google engineer stalked teens, spied on chats. `http://gawker.com/5637234/`.

[9] CHRISTODORESCU, M. Private use of untrusted web servers via opportunistic encryption. *W2SP 2008: Web 2.0 Security and Privacy 2008* (2008).

[10] CONSTANTIN, L. Mega: Bug bounty programme resulted in seven vulnerabilities fixed so far, Feb. 2013. `http://www.computerworld.co.nz/article/488585/`.

[11] CONSTINE, J. Twitter and linkedin manage tasks with asana, new api means robots can too. `http://goo.gl/M8monn`.

[12] Cryptocat blog: Xss vulnerability discovered and fixed, Aug. 2012. `http://goo.gl/Nq7tVk`.

[13] Dromaeo: Javascript performance testing. `http://dromaeo.com/`.

[14] FAHL, S., HARBACH, M., MUDERS, T., AND SMITH, M. Confidentiality as a service–usable security for the cloud. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on* (2012), IEEE, pp. 153–162.

[15] GALLAGHER, J. Thanks a million! `http://blog.trello.com/thanks-a-million/`.

[16] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC* (2009), pp. 169–178.

[17] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIERES, D., MITCHELL, J., AND RUSSO, A. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)* (2012), pp. 47–60.

[18] GLAZKOV, D. [shadow]: Consider isolation. `https://www.w3.org/Bugs/Public/show_bug.cgi?id=16509`.

[19] GLAZKOV, D. Shadow dom. `http://goo.gl/G4j3L4`.

[20] GOEL, V., AND WYATT, E. Facebook privacy change is subject of f.t.c. inquiry. `http://nyti.ms/19IWMV8`.

[21] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

[22] GOOGLE. Content scripts. `http://goo.gl/G2r47g`.

[23] GORDON, S. D., KATZ, J., KOLESNIKOV, V., KRELL, F., MALKIN, T., RAYKOVA, M., AND VAHLIS, Y. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[24] HEIDERICH, M., NIEMIETZ, M., SCHUSTER, F., HOLZ, T., AND SCHWENK, J. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 760–771.

[25] HUANG, L.-S., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 22–22.

[26] JAIN, A., AND TIKIR, M. Is the web getting faster? `http://goo.gl/kFXL7r`.

[27] KEYBASE. Keybase. `https://keybase.io/`.

[28] KURT OPSAHL. Facebook's eroding privacy policy: A timeline. `http://goo.gl/BkRknm`.

[29] Lastpass blog: Cross site scripting vulnerability reported, fixed, Feb. 2011. `http://goo.gl/4MDNjU`.

[30] LU, S., AND OSTROVSKY, R. How to garble ram programs. In *EUROCRYPT* (2013).

[31] MANIATIS, P., AKHAWE, D., FALL, K., SHI, E., MCCAMANT, S., AND SONG, D. Do you know where your data are?: Secure data capsules for deployable data protection.

In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 22–22.

[32] MAONE, G., HUANG, D. L.-S., GONDROM, T., AND HILL, B. User interface security directives for content security policy. http://www.w3.org/TR/UISecurity/.

[33] MCGREGOR, S. Zerobin. http://goo.gl/blY1zx.

[34] MEENAN, P. Webpagetest - website performance and optimization test. http://www.webpagetest.org/.

[35] MOZILLA DEVELOPER NETWORK, AND INDIVIDUAL CONTRIBUTORS. Xpconnect wrappers. http://goo.gl/8eZzQ8.

[36] PARNO, B., MCCUNE, J., AND PERRIG, A. Bootstrapping trust in commodity computers. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), pp. 414–429.

[37] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 85–100.

[38] POPA, R. A., STARK, E., VALDEZ, S., HELFER, J., ZELDOVICH, N., AND BALAKRISHNAN, H. Securing web applications by blindfolding the server. *NDSI* (2014).

[39] PRIVLY. Privly. http://priv.ly/.

[40] RECURITY LABS GMBH. Openpgp.js. http://openpgpjs.org/.

[41] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 224–238.

[42] SAHAI, A., AND WATERS, B. Fuzzy identity-based encryption. In *EUROCRYPT* (2005), pp. 457–473.

[43] Shadowcrypt code release. http://shadowcrypt-release.weebly.com/.

[44] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2000), IEEE Computer Society.

[45] STARK, E., HAMBURG, M., AND BONEH, D. Symmetric cryptography in javascript. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual* (2009), IEEE, pp. 373–381.

[46] STONE, P. Pixel perfect timing attacks with html5.

[47] THE CHROMIUM AUTHORS. Design plans for out-of-process iframes. http://goo.gl/VqR4sv.

[48] Virtru. http://www.virtru.com.

[49] WIKIPEDIA. Global surveillance disclosures (2013–present). http://goo.gl/3YWjY9.

[50] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices 47*, 1 (2012), 85–96.

[51] YAO, A. C.-C. How to generate and exchange secrets. In *IEEE symposium on Foundations of Computer Science (FOCS)* (1986).

[52] ZALEWSKI, M. Postcards from the post-xss world. http://lcamtuf.coredump.cx/postxss/.

APPENDIX

A. THE SHADOW DOM BOUNDARY

`host.shadowRoot`. The host has a dedicated property that provides access to its shadow tree through the shadow tree's root.

`shadowRoot.olderShadowRoot`. Shadow DOM allows one element to host multiple shadow trees. The `shadowRoot` property returns the "youngest" shadow tree (the one registered last), and the `olderShadowRoot` property on each shadow tree provides access to the next "older" shadow tree.

`insertionPoint.getDistributedNodes()`. Shadow DOM defines a mechanism for combining an older shadow tree's content with younger shadow tree's content. When this happens, the `getDistributedNodes()` method provides access to the content taken from the older shadow tree.

We harden the encapsulation by removing the host element's `shadowRoot` property and preventing the creation of any more shadow trees on the element.

B. SUPPORTING HTML TEXT INPUTS

In Section 3.2.2, we focused mainly on the HTML `input type=text` widget. HTML also defines two other ways to declare inputs: the `textarea` element and the `contenteditable` attribute on any element (for rich input). In this section, we detail how our Chrome implementation handles these elements.

Recall that the browser notifies ShadowCrypt's mutation observer about everything that the application puts in the document tree. To find all text input widgets, ShadowCrypt uses `querySelectorAll()` (with the CSS selector `input,textarea,[contenteditable]`) to find candidate text inputs in the added subtree. It then examines the candidate elements for more detailed criteria: `input` elements must be of `text` type, and elements with the `contenteditable` attribute must have it set to `true`.

Next, it proceeds with the transformation described in Section 3.2.1, creating a shadow input. ShadowCrypt privately annotates the original element to prevent further attempts to rewrite it, in case the application removes the element and adds it back later.

ShadowCrypt needs to support bidirectional data flow between the application (which has the ciphertext) and the shadow tree (which has the cleartext). First, when the user modifies the cleartext in the shadow tree, ShadowCrypt needs to update the corresponding ciphertext in the document. When the user types into ShadowCrypt's created input element, the browser dispatches `input` events. ShadowCrypt registers an event listener that encrypts the new data and updates the original element with the ciphertext.

In the other direction, the application's JavaScript code can set the value of an input widget, and ShadowCrypt needs to update the corresponding cleartext. For `input` and `textarea` elements, ShadowCrypt defines a custom setter on the `value` property on the web page's DOM object. This custom setter exists in the application's JavaScript environment. When the application assigns a new value to this property, the custom setter notifies ShadowCrypt's JavaScript environment by dispatching a custom event carrying the new value. An event listener in ShadowCrypt's environment decrypts the ciphertext string provided by the application and updates the shadow input with the clear text.

For elements with the `contenteditable` attribute, the application sets the value by modifying the element's descendants. This triggers ShadowCrypt's document mutation observer, and ShadowCrypt updates the shadow input with the decrypted data. Recall that ShadowCrypt privately annotates input widgets having a shadow tree. This allows ShadowCrypt to differentiate changes to `contenteditable` inputs from other document mutations.