Towards Enabling Secure Web-Based Cloud Services using Client-Side Encryption

Martin Johns
TU Braunschweig
m.johns@tu-bs.de

Alexandra Dirksen
TU Braunschweig
a.dirksen@tu-bs.de

ABSTRACT

The recent years have brought an influx of privacy conscious applications, that enable strong security guarantees for end-users via end-to-end or client-side encryption. Unfortunately, this application paradigm is not easily transferable to web-based cloud applications. The reason for this lies within adversary's enhanced control over client-side computing through application provided JavaScript. In this paper, we propose CryptoMembranes – a set of native client-side components that allow the development of web applications which provide a robust isolation layer between the client-side encrypted user data and the potentially untrusted JavaScript, while maintaining full interoperability with current client-side development practices. In addition, to enable a realistic transition phase, we show how CryptoMembranes can be realized for currently existing web browsers via a standard browser extension.

1 INTRODUCTION

1.1 Motivation

At the latest since the disclosures of Edward Snowden, messenger applications offering end-to-end encryption such as Signal [5] or Threema [6] are gaining in popularity. Consequently, provider of other Cloud Software or Social Networks are coming under increasing pressure to encrypt the data of their user as well. In response to this trend more and more large companies provide end-to-end encrypted services to their users [7, 8, 11, 25, 39]. In case of cloud-based applications, the adoption of the client-side encryption paradigm is attractive to strengthen the cloud's privacy profile [28].

Nowadays many cloud-based applications utilize web technologies, i.e., HTML and JavaScript, to implement their user interfaces so that they can be accessed directly from the browser. Unfortunately, the underlying security requirements of applications that leverage client-side encryption cannot be adapted to web-based applications in general and SaaS scenarios in particular [18]. The reason for this lies in the core of the web application paradigm: Web frontends allow the cloud provider, and thus the potential adversary, to execute JavaScript code on the client-side within the user's web browser, where the confidential data resides unencrypted. This in turn enables an active attacker to access and leak this data without the user's knowledge and consent, rendering the application inherently untrustworthy.

First approaches to answer the growing demand for client-side privacy enforcement materialized in the form of browser extensions. Those add a client-side encryption layer to existing web applications, often without the consent or cooperation of the operators of the targeted applications [1, 2, 4].

However, as shown in [17] hamfisted solutions to extend existing web applications with encryption mechanisms often lead to a cat-and-mouse game between the extensions and the application developer, when it comes to e.g. automatically identifying the input fields that are intended for sensible user data.

In contrast, the motivation of this work is to provide website developers native tools, i.e., standardized DOM elements, which integrate out-of-the box encryption into the browser. On one hand developer can create web applications which allow their users to maintain his/her privacy from scratch while preserving the usability due to seamless client-side compatibility. On the other hand, the user does not have to be confronted with the decision whether to put the use of an application above her privacy.

1.2 Overview

More specifically, we propose a new methodology to build web application with client-side encryption from the ground up using enhanced web browser capabilities. To substantiate the underlying idea, we first instantiate this approach via direct integration into the core of the web browser. Furthermore, we show how browser extensions could be leveraged to realize the concept using today's web browsers:

Secure Building Blocks for Client-side Encrypted Cloud Applications. In this paper, we propose CryptoMembranes, a set of client-side components for SaaS cloud applications. CryptoMembranes leverage client-side encryption for user confidentiality goals and are secure against active JavaScript attackers. The core of our work is a new type of DOM/HTML element, the CMembraneElements, which provide seamless functionality in the web browser with the standard DOM API but provide strong protection of confidential data. A CMembraneElement maintains two representations of the confidential data simultaneously: An encrypted instance that is available for the (untrusted) client-side code to enable rich application functionality and an unencrypted instance for the user to consume and interact with. The two value instances are linked – if one instance is changed through active code or user interaction, the other is updated accordingly through instantaneous and transparent en/decryption operations. CMembraneElements closely mirror the form and functionality of the corresponding DOM elements, which enable rapid and easy adaption of existing SaaS cloud application code towards the secure paradigm.

Providing Isolated Crypto-elements for Legacy Browsers. The current generation of web browsers do not offer native elements to enable secure SaaS cloud applications that utilize client-side encryption. In this paper, we introduce the idea for web browser extension, that only uses technologies readily available today, to provide strong isolation between the confidential user data and

the untrusted client-side code of the cloud application, without impacting the client-side functionality of the application. Using a sophisticated combination of on-the-fly rewriting of DOM content, strong compartmentalization that leverages the Web's Same-Origin Policy, and modern cross-origin communication primitives, a seamless integration of the newly introduced DOM containers into the cloud application code is provided.

2 PROBLEM STATEMENT

2.1 Principals and Security Objectives

In the context of this paper we consider three main principals: The *user*, the *database operator*, and the *cloud application operator*. These principals have the following relations:

- The *user* uses the cloud services of the *cloud application operator*. For this purpose he utilizes standard web browser technology to access the application.
- The *cloud application operator* provides, operates and maintains the SaaS (Software as a Service) cloud application, offered to the *user* via web-based technologies. He uses the *database operator* to store the *user*'s data.
- The *database operator* offers data storage services to the *cloud provider* and *user* in a IaaS (Infrastructure as a Service) fashion.

In general, the three principals can be regarded as mutually distrusting. However in the context of this document, we mainly consider the *user*'s distrust in relation to the other two actors. Thus, we define the primary security objective of the *user* as follows:

Security Objective: The *user* want to keep his personal data private from the other parties, while still using their SaaS and IaaS services.

2.2 Partial Solution: Client-side Data Encryption

In [27] Popa et. al employ encrypted databases to partially address the outlined security objective: In this application model, the user encrypts all his data locally before sending it to the application provider/database. The application and database operate purely on encrypted values leveraging technologies such as partially/fully homomorphic encryption [19], order-preserving encryption [13] or searchable encryption [33].

In this model, the encryption key never leaves the user's control and the data remains fully encrypted while being processed by the application provider and database operator. Thus, the privacy of the user's data is robustly enforced for all computation locations out of the user's own infrastructure.

2.3 Problem: Client-side Leakage of Private Data

While using client-side encryption works well for certain applications, it is not suitable for the outlined cloud scenario. In web-based cloud applications utilizing encrypted databases, the user uses his **web browser** to interact with the application. Within the browser, the user's data is entered and displayed in plaintext.

The encryption/decryption in such scenarios is in general conducted by a dedicated client-side component, such as a web proxy or browser extension, that decrypts incoming data, before it is passed to the browser's rendering engine and encrypts it, before the HTTP traffic leaves the user's infrastructure [18].

The security culprit here is, that the UI code (HTML/JavaScript) is provided on runtime by the *cloud application operator*. The user's data exists in unencrypted form within the browser, as otherwise the user would be unable to enter/read the data. Thus, all active JavaScript code is also capable of reading the data. Hence, a malicious or compromised cloud application operator could modify the JavaScript code that is sent to the client such that it can read the data *after* it has been decrypted or right after the user entered it *before* it is encrypted. Unencrypted data subsequently can be leaked to the malicious principal in clear text via various HTTP means, such as XMLHttpRequests [36] or WebSockets [23].

What makes matters worse is the fact that malicious JavaScript is not restricted to accessing unencrypted data that was actively queried by the user during the application usage session. As the JavaScript code runs within the authorized web session of the user, it can covertly create arbitrary, additional data queries to the application's backend, well hidden in invisible iframes. The decryption proxy cannot differentiate between these rogue HTTP requests and legitimate requests, that were initiated by the user and thus will readily decrypt all incoming data. Therefore, the attacker is able to obtain all data from the server, that was stored under the user's account and was presumed to be protected by the client-side encryption process.

3 RELATED WORK

With the initial publication of the CryptDB paper [27], building practical applications over encrypted databases have become an area of constant attention. Several variants of this application scenario have been subsequently been published. Furthermore, papers have proposed to use proxies for client-side encryption of web data, in many cases without active cooperation of the server-side infrastructure [16, 18, 31]. However, none of these works considers the unique client-side leakage attacks in SaaS, which are the focal point of this paper.

Several attacks of client-side information leakage on web applications have been discussed in the past. For one, it has been shown that client-side timing of HTTP responses can disclose cross-origin, sensitive information [21, 35]. This requires that the communication between the trusted and untrusted realms happen within the browser and do not rely on HTTP or other network traffic. Furthermore, it has been shown, that the information-flow restricting directives of the communication service provider can be subverted [15, 34]. These attacks require the attacker to execute JavaScript in the same DOM as the sensitive values reside, a circumstance robustly prevented if the Same-origin Policy is respected, which is one of the key elements of our work.

The conceptional closest work that addresses client-side encryption is ShadowCrypt, proposed by He et. al [22]. This work takes advantage of Shadow DOM, a browser primitive that was standardized by W3C [20] and is part of Web Components [38]. These elements can be leveraged to create an encapsulated DOM subtree,

which is then attached to an existing DOM tree. This helps developers to avoid website errors due to conflicting CSS or JavaScript selectors.

Even if projects like ShadowCrypt try to leverage Shadow DOM in order to isolate user data, it was never meant to be used as a security feature. This procedure is even strongly disadvised by Shadow DOM's inventor Hayato Ito [24]. One of the reasons is that the Shadow DOM, which contains the user's sensitive data, is appended to a regular untrusted DOM element, named Shadow Host. Consequentially, the isolated Shadow DOM is hosted in the same origin as the untrusted DOM above and so is the user's data. As long as the Shadow Host is untrusted, the data remains vulnerable for e.g. XSS attacks or SQL injection. This limitation was even pointed out by the authors of ShadowCrypt in [22, Sec. 2,1]. In [17] Freyberger et. al investigates the limitations of securing input in internet browsers, with a focus on implementations using ShadowCrypt. Even after the introduction of the `closed` tag for Shadow DOM in 2017 [14] the authors still warn about relying on Shadow DOM for security isolation, since there is no way to stop an attacker from hijacking the attachment of a Shadow DOM.

A more promising approach to tackle this problem is the use of the DOM element iframe [9], which strictly enforces the Same-origin Policy if the `sandboxed` tag is used. Those elements can be considered as a separate HTML page specifically designed to isolate content from access outside of the same origin, be it an untrusted DOM element or JavaScript.

One project that utilized iframes for the isolation of sensible user data was Priv.ly [3]. With the help of a browser extension this project identified input fields on websites which are intended to contain sensible user data. Instead of pasting the user's plaintext into the untrusted DOM, the extension replaces it with URLs. They point to a server where the user's encrypted content is located. This data is decrypted and placed in iframes which in turn are inserted into the DOM. This project was shut down due to multiple reasons, which the founders list on their website [3]. One of the reasons coincides with the main incentive of our work – subsequent encryption of data on existing applications inevitably leads to a cat and mouse game between the data protectors and the company behind the target application. As already shown in [17] it is nearly impractical to automatically find each input field on a website worth protecting. Either because any change to the layout may result in the fields no longer being recognized. Or because the website is not even using predefined input elements for the user's input, as it is the case with e.g. Google Docs.

4 CRYPTOMEMBRANES: SECURE CLIENT-SIDE ENCRYPTED CLOUD APPLICATIONS

As mentioned in the introduction of this paper, we aim to overcome the current problem of web applications using client-side encryption, i.e., their weakness to active JavaScript data leakage attacks. To this purpose, we introduce CryptoMembranes – a secure solution to prevent the leakage of decrypted values by malicious JavaScript on the client-side.

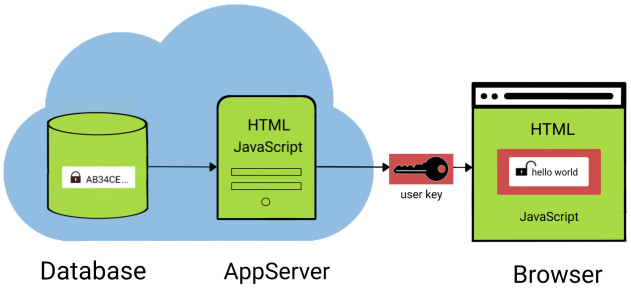

Figure 1: Overview of the system architecture

4.1 Core Concept

The core concept of CryptoMembranes is based on *in-place usage of* CMembraneElements *instead of regular DOM elements*. For each regular DOM element that can carry data, such as numeric values or text, a corresponding CMembraneElement exists, e.g., `DIV` elements are matched with `CryptoDIV` elements. A given CMembraneElement exposes exactly the same interface and behavior as the corresponding regular DOM element.

4.2 Architecture

The coarse architecture of the presented systems mirrors the general set-up of a cloud application utilizing an encrypted database (see Sec. 2.2 and 2.3):

(1) Encrypted database backend: Stores the user's data. The data resides in encrypted form, due to client-side encryption.
(2) Cloud application: The cloud application queries the database to obtain the user's data and compiles HTML/JavaScript/JSON content containing the encrypted user data.
(3) Web browser: The web browser renders the cloud application's HTML/JavaScript to provide the user with the application's user interface
(4) Client-side de/encryption mechanism and key store: A dedicated unit resides in the user's browser, which transparently decrypts the incoming data and encrypts values in outgoing HTTP traffic. It provides secure storage and management of the user's encryption keys.
(5) To protect the decrypted values from malicious JavaScript, the data is kept in CMembraneElements within the browser.

Please refer to Figure 1 for a graphical overview of the system.

4.3 CMembraneElements

The core contribution of CMembraneElements is the fact that a CMembraneElement provides both decrypted (for the user) and encrypted (for the application's JavaScript) access to the contained values:

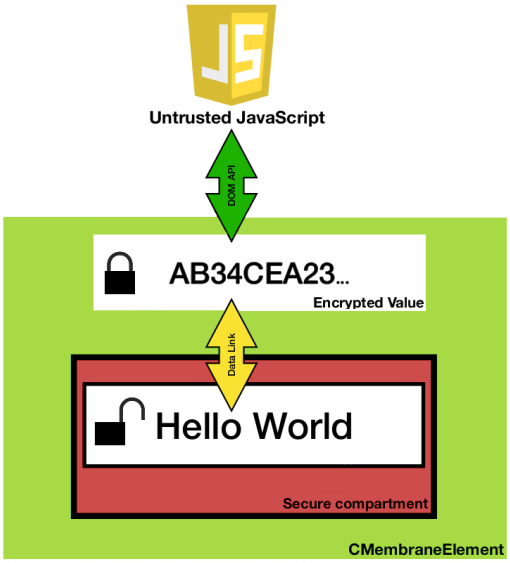

Figure 2: Schematic view of a CMembraneElement

Definition: A CMembraneElement is a dedicated DOM element with the following characteristics:

- It mirrors in form, API and functionality a corresponding, regular DOM element (e.g., a `DIV` element).
- It carries a least one value, such as a numeral, a string or a text node.
- It maintains the contained value in two forms: Unencrypted and encrypted.
- The unencrypted value is presented to the user via the browser UI.
- The encrypted value is available to the enclosing website's JavaScript.
- If one of the two representations is changed via running JavaScript code or through user interaction, the corresponding representation changes accordingly. This means, the client-side de/encryption mechanism transparently transforms the changed values, i.e., updates the encrypted or unencrypted version via the corresponding cryptographic operation.
- The enclosing web page's JavaScript has **no** access to the unencrypted value.

Please refer to Figure 2 for a graphical representation of the CMembraneElement concept.

4.4 CMembraneElement Syntax:

A CMembraneElement adheres to the following syntactic conventions:

- *Name:* The name of the corresponding HTML element is of the form `Crypto` followed by the name of the mirrored DOM element, e.g., `CryptoDIV`
- *KeyID attribute:* A CMembraneElement carries an optional HTML attribute named `CMkeyID`, which specifies the locally stored cryptographic key that should be used for the cryptographic operations.
- *Algorithm attribute:* A CMembraneElement carries an second optional HTML attribute named `CMALgID`, which specifies the cryptographic algorithm that should be used for the cryptographic operations, e.g., "OrderPreserving" for a suitable order preserving algorithm.

If one or both of the attributes are missing, the client-side mechanism chooses the *default* key and/or algorithm to process the data. For simple applications, in general a single key per algorithm-class is sufficient.

4.5 CMembraneElement Types

For functional reasons two distinct types of CMembraneElements exist: *Display* and *Input* CMembraneElements.

4.5.1 Elements for data output. Display CMembraneElementshave the purpose to present confidential data to the user. While the actual values are kept in encrypted form in the database, and thus are out of reach for potential attackers, they are shown to the user in clear text, when displayed by the browser.

For instance, Listing 1 shows the source code for a `CryptoDIV` CMembraneElement. The tag was included in the web page's source code by the cloud application. The actual encrypted data between the opening and closing tag was retrieved from the database. Upon rendering the web page, the browser's rendering engine passed the encountered value to the decryption mechanism on client-side, with the information that the corresponding crypto key has the id "123" and that the "Order Preserving Encryption" algorithm should be applied. Both these information was obtained from the element's attributes.

The decryption mechanism decrypts the value on the client-side and causes the rendering engine to create a graphical representation of the decrypted value, that is inserted in the displayed DOM at the position where the `CryptoDIV` would appear. Any JavaScript running in the page would only see the encrypted value, as the decrypted version is strongly contained within the CMembraneElement's container.

The usage of HTML within a Display CMembraneElements is permitted. However, to avoid accidental or targeted data leakage, JavaScript execution and referencing of external HTTP content is prevented.

```
1 <CryptoDIV CMKeyID="123" CMAlgID="OrderPreserving">
2 AB34CEA23...
3 </CryptoDIV>
```

Listing 1: Example for a `CryptoDIV` element, carrying a value that was encrypted with the key 123 and the algorithm "OrderPreserving".

If JavaScript code in the page would alter the content of the element, e.g., through direct manipulation, this operation first only alters the encrypted representation of the value. Subsequently, the client-side crypto mechanism checks, if the new value adheres to the expected syntactic conventions (e.g., charset or size) of the element's

crypto algorithm. If this is the case, it attempts to decrypt the new value and display the result in the isolated rendering segment. This functionality allows the update of the CryptoDIV with new values, which have been obtained on run-time or the transfer of encrypted content within the DOM, without any loss of confidentiality.

4.5.2 Elements for date entry. Input CMembraneElements are DOM elements that occur in HTML forms to interactively query data from the user. Any HTML element, such as INPUT or TEXTAREA, has a matching CMembraneElement.

Listing 2 shows a CryptoINPUT that allows the user to enter text in string form. After the user has finished his input, the client-side crypto mechanism uses the referenced key (in the depicted case the key with the id "345") to encrypt the value. The resulting encrypted value is available via the element's DOM API, i.e., through the element's value property. Furthermore, when the enclosing HTML form is submitted, the corresponding HTTP POST request also carries the encrypted value. The cleartext value never leaves the client-side CMembraneElementcontainer.

```
1 <CryptoINPUT Type="text" Name="confinput" CMKeyID="345"
  ↪ CMAlgID="Deterministic">
```

Listing 2: Example for a CryptoINPUT element.

4.6 Secure data entry for Input CMembraneElements

To combat social engineering attacks, in which the user might be tricked into entering confidential information into non-CMembraneElement inputs, the browser clearly indicates that the current element, which receives the input is indeed a secure CMembraneElement. How this is realized, depends on the system platform on which the browser is executed:

For standard web browsers running on desktop operating systems the browser creates an input field, that cannot be imitated with standard HTML/CSS. The most straight-forward method to do so, is to implement the actual input field outside of the web page and within the general browser UI, e.g., within an expanding entry field that originates out of the browser's address bar. This procedure may be similar to the address bar's lock-icon which indicates that the current website serves a valid certificate.

For mobile web browsers running on mobile platforms, such as Android or iOS, other non-spoofable UI strategies are preferable, as the browser UI is comparatively small in these cases. Such strategies could include companion apps, that take the task of data entry, or specific clearly marked secure keyboards, that cannot be invoked by untrusted, non-CMembraneElement code.

4.7 Client-side programming using CMembraneElements

As the CMembraneElements external API and behavior is designed to be indistinguishable form the API and behavior of the mirrored elements, the corresponding JavaScript coding does not need to be altered. This is especially relevant in cases in which existing applications are adapted to work with client-side encryption.

Display Elements		
HTML	**CMembrane**	**Remark**
<H1>	<CryptoH1>	For <H2> - <H6> likewise
<DIV>	<CryptoDIV>	
<LI>	<CryptoLI>	Requires closing <\CryptoLI>
<P>	<CryptoP>	Requires closing <\CryptoP>
<SPAN>	<CryptoSPAN>	
<A>	<CryptoA>	Applies crypto to the link text, not the URL
<TH>	<CryptoTH>	
<TD>	<CryptoTD>	
Input Elements		
HTML	**CMembrane**	**Remark**
<INPUT>	<CryptoINPUT>	
<OUTPUT>	<CryptoOUTPUT>	
<TEXTAREA>	<CryptoTEXTAREA>	

Table 1: Overview of CMembraneElements.

As long as the cryptographic keys and algorithms match, data can be freely moved in between CMembraneElements. Take for example Listing 3: Both CMembraneElements share the same key and algorithm. As soon as the user has entered text into the CryptoINPUT CM2, the onchange eventhandler (line 4) calls the JavaScript function moveData (line 7). This function, running in the untrusted part of the DOM has no access to the cleartext that has been entered by the user. Instead the reading operation in line 10 cause the encryption mechanism to encrypt the entered value with the CMembraneElement's key and algorithm and pass the result of this operation to the cValue variable. The operation in line 11 conducts the opposite process – the encrypted value is passed to the CryptoDIV element, which causes the crypto mechanism to decrypt the received value and update the unencrypted representation within the element's secure boundaries. The unencrypted value is subsequently shown to the user within the displayed DOM.

For the JavaScript and the user, the cryptographic operations are completely transparent and invisible. Exchanging the CMembraneElements in the example with regular DIV and Input HTML elements would result in an application that would look and behave fully identical.

```
1  <CryptoDIV ID="CM1" CMKeyID="911" CMAlgID="Deterministic">
2  </CryptoDIV>
3
4  <CryptoINPUT ID="CM2" Type="text" name="conf" CMKeyID="911"
   ↪ CMAlgID="Deterministic" onchange="moveData()">
5
6  <script>
7  function moveData(){
8      var cm1 = document.getElementById("CM1");
9      var cm2 = document.getElementById("CM2");
10     var cValue = cm2.value   // cValue is encrypted
11     cm1.innerText = cValue;
12 }
13 </script>
```

Listing 3: Client-side programming with CMembraneElements.

4.8 CMembraneElement Element Overview

As specified above, the CMembraneElement-types span two distinct classes, *display* and *input* CMembraneElements. The class of *display* CMembraneElements contains all DOM elements, that can carry `text` child-nodes, most notably CMembraneElementsfor the `div`, `span` and `td` DOM elements. The contained text of these elements is subject to the transparent crypto operations.

The class of *input* CMembraneElements encompasses all DOM elements, that can be used to query free-form input from the user in the context of HTML forms, most notably CMembraneElements for the `input` and `textarea` DOM elements. We leave the design of suitable CMembraneElements for modal input elements, such as `option` for future work.

Please refer to Table 1 for an overview of all covered CMembraneElements.

5 ExtensionMembranes: PROVIDING ISOLATED CRYPTO-ELEMENTS FOR LEGACY BROWSERS

We envision CryptoMembranes to be a fully supported approach for future browser generations. A native built-in implementation of CMembraneElements in web browsers is straight forward, as the browser engine runs in a higher security context as a website's JavaScript. Thus, the browser engine can conduct the code compartmentalization and content isolation of the sensitive data.

In this section, we show how CMembraneElements can be realized for the existing browser generation purely by using existing web standards and a web browser extension.

5.1 ExtensionMembranes: Overview

We introduce ExtensionMembranes, a full realization of the CryptoMembranes-concept using only already existing, standardized HTML elements and JavaScript APIs, leveraging standard browser extension technology.

ExtensionMembranes browser extension: The system is realized in the form of a standard web browser extension, a technology that is supported by all major web browsers. Extension code runs on the client-side with higher privileges and robustly isolated from untrusted, web-retrieved JavaScript code. Thus it can securely enforce the system's security objectives. The extension provides the following functionality:

Secure key storage The extension securely persists the user's cryptographic material, such as encryption keys, and offers tools for key management.

Page rewriting The extension processes all incoming HTTP responses and inserts secure element compartments into the DOM for all sensitive values (more on this process below).

Secure connection between untrusted JavaScript and sensitive data: The extension introduces technical measures and communication primitives that realize the CryptoMembranes-like linking of values between the trusted and the untrusted realms.

For the storage and processing of the sensitive data, the extension utilizes ExtensionMembraneElements:

ExtensionMembraneElements: To encapsulate the confidential data and provide the transparent interaction with untrusted JavaScript, we introduce ExtensionMembraneElements. In essence, a ExtensionMembraneElement consists of the following components:

- *Container element:* A DOM element that encapsulates the ExtensionMembraneElement content and logic. In most cases realized with `DIV` or `SPAN` elements.
- *Isolated compartment:* An isolated subdocument hosted on a unique, non-web origin, that contains the sensitive data and the decryption logic. In the proposed implementation, this compartment is realized using `iframe` elements.
- *Connecting JavaScript logic:* Essential JavaScript APIs and properties of the container element are redefined to trigger the secure handling of the sensitive value and the connection between the untrusted environment to the trusted isolated origin within the `iframe`.

The details of these components will be explained in dedicated sections of this document later.

Page loading process: On the server-side web content that is designed to leverage ExtensionMembraneElements uses specifically marked HTML elements that are replaced by the browser extension with ExtensionMembraneElements during page load. While the specific method of marking in these custom elements is flexible and arbitrary (as long as the marking is unambiguous), for the remainder of this document, we assume that the syntactic conventions of CMembraneElements, as introduced in Section 4.4, are used.

On a coarse level, for each incoming HTML document, the browser extension conducts the following steps:

(1) Replace CMembraneElements with container DOM elements
(2) Insert isolated `iframes` into the containers
(3) Decrypt the initial values and pass the resulting data to the iframe DOM
(4) Alter the APIs and properties of the container element, such that they transparently update the `iframe` content.

Please refer to Figure 3 for a system flow diagram. The precise implementation of these steps is provided in the following sections.

5.2 Initial rewriting of custom ExtensionMembraneElements

As the proposed custom CMembraneElements are unknown to the browser, any CMembraneElement element contained in a web page's HTML has to be replaced with existing DOM elements. The preferable method for this is *on-the-fly* rewriting the incoming HTML source code before it is passed to the rendering engine. Some browsers, such as Firefox, allow direct manipulation of the HTTP traffic conducted by the browser extension. For other browsers, such as Google Chrome, instead the extension has to leverage ServiceWorkers [30] for this task. In either case, the extension establishes an HTTP interception mechanism that modifies the incoming HTML on the network layer, similar to an intercepting man-in-the-middle proxy.

The rewriting is done as follows: each encountered *display* CMembraneElement, e.g., `CryptoDIV`, is replaced with the

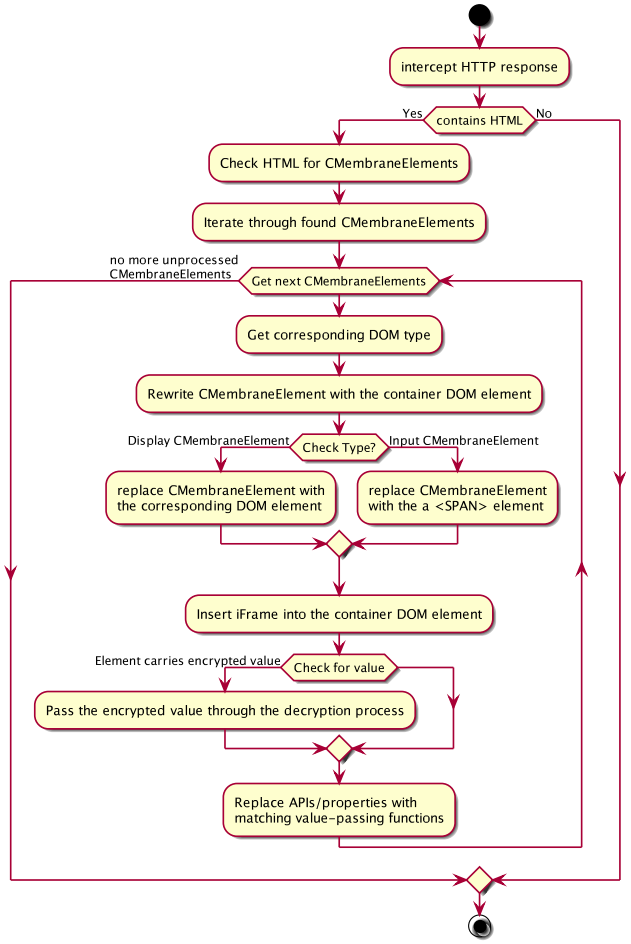

Figure 3: Processing of incoming HTTP responses

corresponding DOM element, e.g., `DIV`, with an additional attribute that signifies to the subsequent processes, which CMembraneElement the element represents (see Listing 4). *Input* CMembraneElements are initially replaced with `SPAN` elements, which are set via CSS to have exactly the same dimensions as the replaced element. Please refer to Listing 4 for an example.

Alternatively, for browsers that do not support either on-the-fly rewriting method, the custom HTML elements remain in the page until the rendering process has terminated. As the custom HTML elements are unknown to the browser, they are ignored during the rendering process, but can still be located via JavaScript. After the initial rendering has terminated, the custom elements are replaced in in place using JavaScript, as described above.

5.3 Subsequent rewriting of custom ExtensionMembraneElements on runtime

Modern web applications frequently rely on client-side creation of HTML markup, using JavaScript libraries, such as JQuery [12] or AngularJS [10]. Hence, not all ExtensionMembraneElements can

```
1  The following CMembraneElement:
2
3  <CryptoDIV ID="CM1" CMKeyID="911" CMAlgID="Deterministic">
4  AB34CEA23...
5  </CryptoDIV>
6
7  will be rewritten into:
8
9  <DIV rel-crypto="cryptodiv" ID="CM1" CMKeyID="911"
   ↪  CMAlgID="Deterministic">
10 AB34CEA23...
11 </DIV>
```

Listing 4: Rewriting of CMembraneElements into ExtensionMembraneElements.

be processed on-the-fly. To accommodate this development methodology, the extension augments all DOM-APIs that are capable of adding further HTML content to the document, i.e, `document.write` and all variants of `HTMLElement.innerHTML`, similar to the methodology presented by Musch et al. [26]. The augmented APIs intercept the HTML-strings before they are passed to the browser's HTML parser. This way, all additional DOM content can be processed following the method discussed in Sec. 5.2.

5.4 Insertion of isolated compartments

The inserted DOM elements (in most cases `DIV`, `SPAN` or `TD` elements) serve as *containers* for the secure value compartments. To realize the isolated compartment CryptoMembranes leverages `iframe` elements. For each of the container elements, the browser extension inserts an `iframe` as a direct child node. The origin of the web content displayed in the `iframe` is unique for the browser session, unreachable for the untrusted web content and owned by the browser extension. Please refer to Listing 6, lines 5 - 17, for an example in which a DIV ExtensionMembraneElementis created.

Thus, thanks to the browser's same-origin policy [29], the untrusted JavaScript code cannot reach into the `iframe` to read its content. The browser extension sets the CSS properties within the iframe to match the properties that would be applied to the unaltered, replaced element, causing the content displayed in the iframe to seamlessly match its surroundings.

5.5 Initial decryption of existing data

Next, for each processed CMembraneElement, the browser extension checks, if the element contained encrypted values. They must be clearly identifiable as encrypted and appear either as childnodes (in case of display elements) or value attributes (in case of input elements). If such content is found, it is removed from the visible DOM and placed in a custom property of the container DOM element. Furthermore, the browser extension retrieves the corresponding crypto-key and decrypts the value within the secure realm of the privileged extension code. The resulting unencrypted value is set as the content of the securely isolated iframe, such that the browser can display it to the user.

5.6 Rendering of the sensitive content

The browser extension retrieves the applicable CSS styles for the replaced DOM element and propagates them into the confinements of the iframe, resulting in a seamless rendering of the element. Display ExtensionMembraneElements are permitted to display HTML markup to structure the data. However, to avoid active data leakage attacks, such as stored XSS, the following measures are taken:

- No HTTP content from outside the origin of the browser extension can be loaded into the iframe and no HTTP/network requests are allowed to be created from within the iframe.
- No JavaScript execution is allowed in the iframe, besides the privileged value-passing JavaScript owned by the browser extension.

To robustly enforce these security properties, the compartment iframes are outfitted with a strict Content Security Policy (CSP) [37]. The applied CSP disallows all network communication through an empty whitelist for **all** CSP directives. Furthermore, **all** inline scripts are forbidden, along with **all** scripts, that do not originate directly from the browser extension. Furthermore, dynamic creation of script code is prevented. Please refer to Listing 5 for the precise CSP.

```
Content-Security-Policy: default-src 'self'
```

Listing 5: The strict CSP applied to all ExtensionMembraneElements. Only 'self', i.e., the unique origin of the browser extension is whitelisted.

5.7 Instantiation of JavaScript handlers for value-passing

Finally, to enable JavaScript-based interaction with the elements (see Listing 3), the extension rewrites the public properties and APIs of the container element, to cause the browser extension to conduct the corresponding de/encryption operations on read, write and interactive update operations, that target the container element. This is done by replacing all APIs and properties that target values within the element:

Read operations All APIs/properties that grant read access to the contained value are altered to return the encrypted value which is held in a specific custom property of the element.

Write operations All APIs/properties that grant write access to the contained value are altered as follows: First, the passed new value is written into the specific custom property of the element. Subsequently, the new value is passed, using the `postMessage` API [32] into the `iframe` for cryptographic processing and presentation to the user.

Update operations Accordingly to the *write operations*, a message handler is instantiated, that receives `postMessages` from within the `iframe`. Such messages are created by the `iframe` whenever user interaction changes the cleartext version of the value in the `iframe`, e.g., in case the user enters information into a `INPUT` field. In this process, the extension retrieves the entered/changed cleartext value, applies the corresponding encryption and sends the ciphertext via `postMessage` to the encapsulating ExtensionMembraneElement. The ExtensionMembraneElement's message handler retrieves the new encrypted value from the `postMessage` and updates the specific custom property of the element.

For instance, in the example of Listing 3, the `innerText` property of the element `cm1` (line 11), would have been rewritten to do the following: First, the internal store of the elements would be updated to the value passed via the property call. Next, a `postMessage` call is created to pass the encrypted value securely into the `iframe`. Please refer to Listing 6, lines 19-26, for the corresponding property instrumentation code. The code for the update operation is left out for brevity sake.

Within the `iframe`, a receiving script processes the incoming value, passes it to the browser extension, which conducts the decryption. Finally, the decrypted value is inserted into the isolated DOM fragment within the `iframe`, and thus, presented to the user.

```
1  // Get all DIV-CMembrane placeholders from the document
2  var CMDivs =
   ↪  document.querySelectorAll('div','[rel-crypto]');
3
4  // Process the first CMembraneDIV
5  var odiv = CMDivs[0];
6
7  // Create ExtensionMembrane container
8  var exMemDiv = document.createElement("div");
9
10 // Store encrypted value
11 exMemDiv.cryptText = odiv.innerText;
12
13 // Create isolated iframe compartment
14 var EMframe = document.createElement("iframe");
15 EMframe.src="[origin of the extension]";
16 exMemDiv.appendChild(EMframe);
17 exMemDiv.compFrame = EMframe;
18
19 // Replace read/write element properties
20 Object.defineProperty(exMemDiv, "innerText", {
21     get: function(){return this.cryptText},
22     set: function (val) {
23         this.cryptText = val;
24         // Pass the encrypted value into the compartment
25         this.compFrame.contentWindow.postMessage(val,
           ↪  [origin of the extension]);
26 }});
27
28 // replace placeholder with ExtensionMembrane
29 odiv.parentNode.replaceChild(exMemDiv, odiv);
```

Listing 6: Creating a ExtensionMembraneElement and replacing native data-access properties with value-passing methods – Exemplified with the `innerText` property

5.8 Interaction with the server-side

No specific actions are required by the extension to enable secure interaction with the server-side, e.g., via `FORM` submission or sending data via `XMLHttpRequests`. The user's sensitive data always resides within the untrusted DOM in encrypted form. The ExtensionMembraneElements are designed to maintain their full native functionality, thus, e.g., on `FORM` submission, the encrypted values are reliably send to the server. The connection value-passing JavaScript handlers ensure that whenever the user interacts with the page and enters new values, the corresponding ciphertext is updated accordingly in realtime.

5.9 Security assessment

Finally, in this section, we briefly revisit the crucial security properties of the application scenario and show that they are satisfied by ExtensionMembranes.

5.9.1 Isolation properties. The isolation of the sensitive values from the untrusted JavaScript is robustly enforced through the browser's native Same-origin Policy [29], as the content within the iframe compartment is hosted within the unique extension origin.

5.9.2 Protection against code injection attacks. As specified in Section 5.6, the compartment iframes carry a very strict and prohibitive CSP, disabling all script execution. Thus, code injection attacks are robustly thwarted.

5.9.3 Protection against ClickJacking attacks. Finally, the adversary could attempt to overlay Input ExtensionMembraneElements with untrusted input elements. As keystrokes are never propagated over the boundaries of web origins, any intercepted input never reaches the attacked ExtensionMembraneElement. Thus, any such attack would immediate lead to noticeable malfunctioning of the application.

6 CONCLUSION

In this paper we proposed CryptoMembranes, an extension to the web browser's HTML and DOM-API which offers native support to securely realize cloud-applications that leverage client-side encryption to protect user data against sever compromise. CryptoMembranes are designed to fit seamlessly into the currently existing browser paradigm, due to their close mirroring of related DOM-elements and their respective JavaScript interfaces. Thus, provided the underlying cryptographic algorithms permit, they are out of the box compatible with already existing client-side processing functionality written in JavaScript, such as JavaScript UI frameworks or libraries.

We firmly believe that adding native and robust primitives for client-side encryption to the web platform would be a key enabler for new and exiting application scenarios, which are currently infeasible due to the inherent shortcomings of the ecosystem.

7 ACKNOWLEDGMENT

This research was supported by the Lower Saxonian Ministry for Science and Culture as part of the research program MOBILISE (Mobility in Engineering and Science).

REFERENCES

[1] German-based, end-to-end encryption solution that integrates seamlessly with dropbox. https://www.dropbox.com/app-integrations/boxcryptor. visited 2020-07-28.
[2] Communicating securely with mailvelope. https://www.mailvelope.com/de. visited 2020-07-28.
[3] Priv.ly - share priv(ate).ly. https://priv.ly/.
[4] Sendsafely encryption for chrome and gmail. https://chrome.google.com/webstore/detail/sendsafely-encryption-for/glpichgelkekjnccdflklcclhnoioblm. visited 2020-07-30.
[5] Signal messenger. hhttps://signal.org.
[6] Threema messenger. https://threema.ch/.
[7] End-to-end encrypted file sync & sharing. https://www.salesforce.com/content/dam/web/en_us/www/documents/reports/wp-platform-encryption-architecture.pdf. visited 2020-04-29.
[8] Whatsapp encryption overview. *White paper*, 2016.
[9] Html5 - the iframe element. https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html, 2017.
[10] AngularJS - Superheroic JavaScript MVW Framework. [online], https://www.google.com/search?client=safari&rls=en&q=angular+js&ie=UTF-8&oe=UTF-8, 2020.
[11] Introducing google cloud confidential computing with confidential vms. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms, 2020.
[12] jQuery: The Write Less, Do More, JavaScript Library. [software], https://jquery.com, 2020. visited 2020-04-29.
[13] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007632. URL http://doi.acm.org/10.1145/1007568.1007632.
[14] Eric Bidelman. Shadow dom v1: Self-contained web components. https://developers.google.com/web/fundamentals/web-components/shadowdom, 2017.
[15] Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP*, 2012. URL http://www.w2spconf.com/2012/papers/w2sp12-final11.pdf.
[16] M. H. Diallo, B. Hore, E. C. Chang, S. Mehrotra, and N. Venkatasubramanian. Cloudprotect: Managing data privacy in cloud applications. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 303–310, June 2012. doi: 10.1109/CLOUD.2012.122.
[17] Michael Freyberger, Warren He, Devdatta Akhawe, Michelle L Mazurek, and Prateek Mittal. Cracking shadowcrypt: Exploring the limitations of secure i/o systems in internet browsers. *Proceedings on Privacy Enhancing Technologies*, 2018(2):47–63, 2018.
[18] Benny Fuhry, Walter Tighzert, and Florian Kerschbaum. Encrypting analytical web applications. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop*, CCSW '16, pages 35–46, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4572-9. doi: 10.1145/2996429.2996438. URL http://doi.acm.org/10.1145/2996429.2996438.
[19] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: 10.1145/1536414.1536440. URL http://doi.acm.org/10.1145/1536414.1536440.
[20] Dimitri Glazkov and Hayato Ito. Shadow dom. working draft, w3c, june 2014, 2015.
[21] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and Conquer: Exposing Cross-Origin Resource Size. In *25th USENIX Security Symposium (USENIX Security)*, 2016.
[22] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1028–1039, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660326. URL http://doi.acm.org/10.1145/2660267.2660326.
[23] Ian Hickson. The Web Sockets API. W3C Working Draft WD-websockets-20091222, http://www.w3.org/TR/2009/WD-websockets-20091222/, December 2009. URL http://www.w3.org/TR/2009/WD-websockets-20091222/.
[24] Hayato Ito. Add closed flag to createshadowroot (bugzilla: 20144). https://github.com/w3c/webcomponents/issues/100, 2015.
[25] Joseph Menn. Facebook to expand encryption drive despite warnings over crime. https://www.reuters.com/article/us-facebook-privacy-encryption/facebook-will-widen-access-to-encryption-feature-test-safety-measures-idUSKBN1XF2MJ, 2019.
[26] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2019.
[27] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the*

23rd ACM Symposium on Operating Systems Principles (SOPS), 2011.

[28] Krishna P. N. Puttaswamy, Christopher Kruegel, and Ben Y. Zhao. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, 2011.

[29] Jesse Ruderman. The Same Origin Policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001. URL http://www.mozilla.org/projects/security/components/same-origin.html.

[30] Alex Russell, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. Service Workers. W3C Working Draft, 2 November 2017, https://www.w3.org/TR/service-workers-1/, 2017.

[31] E. Saleh and C. Meinel. Hpisecure: Towards data confidentiality in cloud applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 605–609, May 2013. doi: 10.1109/CCGrid.2013.109.

[32] Eric Shepherd. window.postmessage. [online], https://developer.mozilla.org/en/DOM/window.postMessage, last accessed 02/12/12, October 2011. URL https://developer.mozilla.org/en/DOM/window.postMessage.

[33] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 44–55, 2000. doi: 10.1109/SECPRI.2000.848445.

[34] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 853–864, 2016.

[35] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[36] Anne van Kesteren. The XMLHttpRequest Object. W3C Working Draft, http://www.w3.org/TR/XMLHttpRequest, April 2008. URL http://www.w3.org/TR/XMLHttpRequest/.

[37] W3C. Content Content Security Policy Level 3. W3C Editor's Draft, 10 May 2017, https://w3c.github.io/webappsec-csp/, May 2017. URL https://w3c.github.io/webappsec-csp/.

[38] W3C. Web components. https://github.com/w3c/webcomponents/issues/100, 2019. visited 2020-07-23.

[39] Eric S. Yuan. End to end encryption update. https://blog.zoom.us/end-to-end-encryption-update/, 2020.