Noname manuscript No.
(will be inserted by the editor)

ShadowFPE: New Encrypted Web Application Solution based on Shadow DOM

Xiaojie Guo[1,2] · Yanyu Huang[1,2] · Jinhui Ye[1,2] · Sijie Yin[1,2] · Min Li[1,2] · Zhaohui Li[1,2,*] · Xiaochun Cheng[3,*]

Received: date / Accepted: date

Abstract Most users hesitate to use third-party web applications because of security and privacy concerns. An ideal solution would be to allow apps to work with encrypted data, so that users might be more willing to provide just the encrypted version of their sensitive data. ShadowCrypt, proposed in CCS 2014, is the first and so far only solution that can achieve this by leveraging the encapsulation provided by Shadow DOM V0, without the need for the users to trust neither server nor client codes of web applications. Unfortunately, researchers shown ShadowCrypt to be vulnerable to several attacks. Furthermore, since 2015 ShadowCrypt is no longer compliat to the updated W3C standard, and some attacks have been proposed for ShadowCrypt. Hence, currently there is no effective and secure solution to serve the purpose.

In this paper, we present ShadowFPE, a novel format-preserving encryption that makes use of a robust property in Shadow DOM to obtain a feasible solution. Compared to ShadowCrypt, our solution does not destroy the data format and make the data usable in most cloud web applications. We confirmed the effectiveness and the security of ShadowFPE through case studies on web applications. Our results show that ShadowFPE is practical as it has low overhead and requires minimal modification in existing applications compared to the existing applications.

Keywords Format-preserving encryption · ShadowCrypt · data privacy · Shadow DOM · encrypted web applications.

Xiaojie Guo
E-mail: xiaojieg.veda@gmail.com

Yanyu Huang
E-mail: onlyerir@163.com

Jinhui Ye
E-mail: 18119946720@163.com

Sijie Yin
E-mail: yin_sijie@qq.com

Min Li
E-mail: limintj@nankai.edu.cn

Zhaohui Li
E-mail: lizhaohui@nankai.edu.cn

Xiaochun Cheng
E-mail: X.Cheng@mdx.ac.uk

[1] TianjinKey Laboratory of Network and Data Security Technology & College of Cyber Science, Nankai University, 300350, Tianjin, China
[2] College of Computer Science, Nankai University, 300350, Tianjin, China
[3] Middlesex University, The Burroughs, Hendon, London NW4 4BT, UK

1 Introduction

With the increasing popularity of cloud computing, more and more web applications are deployed in clouds to provide services. These web applications are called Software as a Service (SaaS). While SaaS reduces the cost of software delivery and maintenance, most users still hesitate to use these web applications due to the security and privacy concerns, because cloud servers and client-side codes are both untrusted. Weak authentication, application vulnerabilities, employees' misbehavior, and other attacks on web, all might lead to data leakage and further reduce the confidence of customers.

In this section, we will introduce the encrypted web applications in details. Moreover, we will describe merits and limitations of ShadowCrypt. Finally, our contributions show that the proposed ShadowFPE can solve the ciphertext storage and identification problem.

1.1 Encrypted web applications

To encrypt web application data, most proposals adopt the approach of encrypting the data before storage while requiring as little modification as possible for applications. The encryption can occur in three locations which can also be called "chokepoints", as shown in Figure 1.

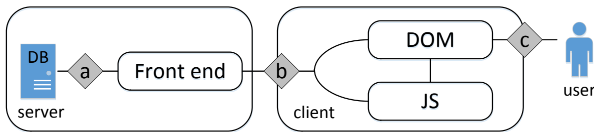


Fig. 1: Chokepoints (a), (b), (c) for data encryption in web application.

Chokepoint (a) is between the front end and the database, where data is encrypted before reaching the database server. A typical scheme is CryptDB [5]. This design protects user privacy against an untrusted/compromised database server, however, it suffers attacks such as SQL injection or XSS attacks from web browsers. And the user has no control on the decryption keys.

Chokepoint (b) is between the client and the network, where the application's client-side JavaScript code encrypts the data before sending it to the server-side. This design is the most commonly used solution today. Popular applications (e.g., LastPass [15], Mega, CryptoCat [16]) and some researches (e.g., [13, 14]) all apply this design. Although private data can be accessed only by application's client side (i.e., JavaScript/HTML), bugs in the client-side codes (e.g., XSS) could also compromise the security by leaking data in plaintext. For example, LastPass and CryptoCat both have suffered from client-side vulnerabilities. Client-side codes may not be trustable too [15, 16].

The ideal location for encryption is in Chokepoint (c), where data is encrypted before the application codes (including the client-side codes) can access it. This design makes the application only able to access encrypted data. "ShadowCrypt" is based on this idea and was proposed in 2014, which runs as a browser extension and ensures that the secure encryption/decryption operations are controlled by the user. ShadowCrypt can achieve a higher security level against attacks in application server-side, the network, and even the client-side application code (i.e., the DOM and JavaScript).

1.2 Merits and limitations of ShadowCrypt

Merits: Encryption in Chokepoint (c) was unexplored until ShadowCrypt was proposed in 2014, which brings a new design idea for encrypted web applications. As far as we know, ShadowCrypt is the only solution for it.

The key innovation of ShadowCrypt is to build a secure input/output environment by leveraging the encapsulation power of Shadow DOM in the web browser. The novelty stems from a functionality of Shadow DOM ("multiple shadow roots", see Section 2 for more details) that allows the web browser to render DOM elements without putting them into the main document DOM tree, and thus it can defend the malicious JavaScript/HTML codes fetching the clear texts from the input/output elements.

Another contribution of ShadowCrypt is to provide a generic solution for text-based web applications (e.g., Gmail, Facebook, Twitter) such that without any modification in the applications, users are able to encrypt data while still able to use most of the functionalities. There are two challenges in order to achieve this: ciphertext identification from HTML codes and keyword search [1, 2] (the only frequent operation over encrypted text). For the former, ShadowCrypt adds both the prefix and postfix to the encrypted text. For the latter, it computes the hash of each word and appends them at the end of the encrypted text.

Limitations: Unfortunately, Shadow DOM is an evolving standard. With the upgrade of W3C standard, "multiple shadow roots" is no longer supported. The solution provided by ShadowCrypt is not feasible in the new W3C standard. Hence, to have a proper solution for this question, we should find an alternative design that can provide a similar solution for encryption at Chokepoint (c). In addition to this issue, researchers also raised concerns about the actual security of some attacks for Shadow DOM, which makes ShadowCrypt insecure [21]. The question is whether Shadow DOM can still be used to build a secure input/output environment for web applications and if yes, how.

ShadowCrypt cannot be applied to web applications that handle data with a pre-defined length and format such as Customer Relationship Management (CRM) system and Office Automation (OA) system. The first problem to solve is how to store the encrypted data in the original database, because ShadowCrypt will change the data length and format; the second problem is how to identify ciphertext from HTML codes with limited storage length. We refer these two problems as the "ciphertext storage" and "identification" problem. Our target is to provide a secure solution for

the encryption in Chokepoint (c) while solving the storage and identification problem such that encrypted data can still be stored in the original database of the web application.

Contributions: In this paper, we present ShadowFPE, a new solution for user-controlled encrypted web applications deployed in untrusted clouds. Similar to ShadowCrypt, ShadowFPE provides a practical solution for encryption in Chokepoint (c) which is the ideal location for encrypting sensitive data before the application codes (including the client-side codes) can access it. There are two difficulties that ShadowFPE has overcome. First, without the multiple shadow roots (deprecated in the latest W3C standard), it becomes impossible to make use of Shadow DOM to provide an encapsulated secure input/output environment. To tackle this problem, we finally identified a more robust property in DOM (roughly speaking, creating new branch in DOM tree, thus we refer our method as the "new-branch" method) to leverage Shadow DOM to obtain a more long-term and robust solution. Our solution would still be valid as long as DOM is supported and Shadow DOM is not deprecated as it does not rely on the low level details of Shadow DOM (unlike ShadowCrypt). The security and practicality of ShadowFPE are verified by case studies on real web applications. The results show that ShadowFPE is effective as it has low overhead and requires minimal modification to the existing applications.

Second, ShadowFPE also solves the "ciphertext storage and identification problem". Following the idea in privacy-preserving data publishing (PPDP) [19], ShadowFPE requires the cloud service providers (CSPs) to mark the sensitive data explicitly to provide more confidence to users to purchase their services. ShadowFPE defines a unified specification named "tag-rule" based on HTML tags and attributes to mark the sensitive data and their formats; provides the "tag-based identification method" for the user to identify and encrypt/decrypt them; and applies format-preserving encryption (FPE) to solve the ciphertext storage problem. The security and practicality of ShadowFPE are verified by case studies on real web applications, e.g., CRM, OA, Meeting Management System (MMS). The results show that ShadowFPE has a high usability and is efficiency with minimal modification overhead (lightweight).

Organization: In this paper, we introduce the background encrypted web applications and characteristic of ShadowCrypt in Section 1. Then, we describe the details of Shadow DOM and ShadowCrypt in Section 2. In Section 3, we analyze the limitations of state of art solution. In Section 4, we present a new-branch method and apply this method to ShadowFPE which presented in Section 5. We also present more case studies in Section 6 and evaluates ShadowFPE in Section 7. We draw some conclusion in Section 8.

2 Preliminaries

In this section, we will introduce the background about Shadow DOM and ShadowCrypt in Section 2.1 and Section 2.2. And we will describe the Format-preserving encryption simply in Section 2.3.

2.1 Shadow DOM

Shadow DOM provides an encapsulation for JavaScript, CSS, and templating in a web component. It can block JavaScript code from accessing the shadow tree. It has been updated from version V0 to V1, and the lastest W3C Working Draft was published in September 2017. We summarize the main differences between Shadow DOM V0 and V1:

1. *Shadow root mode*. The new mode "closed" is added in V1. The design goal of the *closed* mode is to forbidden access to Shadow DOM from outside.
2. *Multiple shadow roots*. It is deprecated in V1 (see the attack in Section 3.1 for multiple shadow roots).
3. *Can be a shadow host*. In V0, every element can be a shadow host; while in V1, only a limited number of elements are allowed to be shadow hosts.
4. *Selectors*. In V0, the usage of */deep/* or *::shadow* selector can visit Shadow DOM. In V1, these selectors have been deprecated.
5. *Function of creating shadow root*. In V1, the new function of *Element.attachShadow* is used to create a shadow root with different modes.

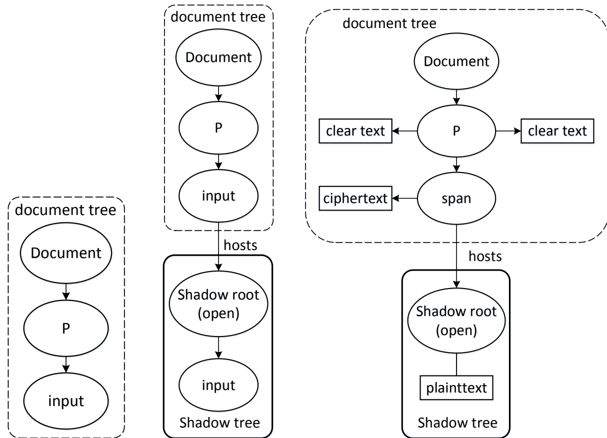
2.2 ShadowCrypt

ShadowCrypt [6] builds the secure input/output environment which is isolated from the application DOM based on shadow DOM V0. It runs as follows.

Initial state. In Figure 2(a) we show an example with only one input element. From it, we can see the initial state of DOM before using ShadowCrypt.

Input isolation. The input element in the page will be replaced with a **new input** which is isolated from the page, as shown in Figure 2(b). Then, it captures user's data from shadow input, encrypts it using the AES-CCM algorithm, and updates the original input node's value with the ciphertext. The final encrypted text will be added with a prefix such as "`=?shadowcrypt`" to identify it as a ciphertext.

Output isolation. When the encrypted text needs to be decrypted and rendered, ShadowCyrpt will locate the text with the prefix of "`=?shadowcrypt`", decrypts it, and creates a *span* element hosting the cleartext which is shown in Figure 2(c).



(a) Initial state of DOM without ShadowCrypt. (b) Initial state of DOM with ShadowCrypt. (c) ShadowCrypt presents ciphertext. A *span* element hosts plaintext.

Fig. 2: The basic procedure of ShadowCrypt.

2.3 Format preserving encryption

Format-preserving encryption (FPE) [7–9, 11, 22] has been proposed recently to solve the ciphertext storage problem. The goal of applying FPE is to generate ciphertext, which falls in the same domain as that of the plaintext, thus FPE can ensure that the ciphertext has the same format as the plaintext while encrypting sensitive information.

Some practical FPE schemes have been proposed for common domains like integer (e.g., FFSEM [8]) and character data (e.g., FFX [11]). With the format of the ciphertext preserved, FPE enables the upgrade of database or application security in a transparent way, without changing the database structure and data types. It has been widely used in some integrated solutions and business products.

3 Limitations of state of art solution

In this section, we point out the reasons why ShadowCrypt cannot work after the upgrade of W3C standard, and discuss the potential attacks and other issues about it.

Problem 1: *ShadowCrypt is not compliant with the new W3C standard, so there is no solution which can build a secure input/output environment using Shadow DOM at Chokepoint (c).*

The reason why ShadowCrypt cannot work is due to the following three facts in the new W3C standard:

Fact 1 Some HTML elements (e.g., input element, video element) have already included a "user-agent Shadow root". Take input element as an example, it can be shown in Figure 3(b).

Fact 2 The HTML elements are forced to host up to one shadow root, i.e., multiple shadow roots are no longer supported that makes ShadowCrypt not feasible.

Fact 3 Many HTML elements, such as input, textarea, etc., cannot be the shadow hosts because these elements have already hosted a default shadow tree whose root is called "user-agent shadow root" (Fact 1).

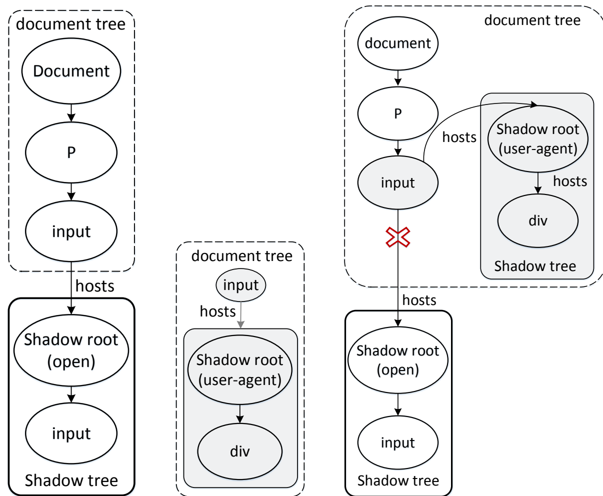
As described in Section 2.2, a shadow root will be added to the input element in ShadowCrypt which is shown in Figure 3(a). However, the input element has already included the "user-agent Shadow root" (Fact 1); so, ShadowCrypt would lead to the result which is illustrated in Figure 3(c). We can see that the input element in ShadowCrypt would host "*multiple shadow roots*". This is a violation of Fact 2. Moreover, making input element as a shadow host also violates Fact 3.

`Remark`. The `iframes` have been mentioned to build the secure input/output environment in [21]. However, He et al. [6] have pointed out that the `iframes` have usability limitations, especially when rendering the decrypted text.

Problem 2: *ShadowCrypt may suffer from some potential attacks in the old W3C standard.*

There are three attacks for ShadowCrypt, in the followings, we retrieve these attacks described in [21]:

- **Attack 1**: **"new-ShadowRoot attack".** In V0, because Shadow DOM allows an element to have multiple shadow roots, the newest shadow root will have the ability to access the old shadow root objects through their *olderShadowRoot* property.
- **Attack 2**: **"CSS selectors attack".** In V0, two CSS selectors of */deep/* and *::shadow* can "pierce" the Shadow DOM. They can access the Shadow elements from the outside world. Figure 5 describes an example of how to launch this attack.
- **Attack 3**: **Hijack attack.** In V0, the *Element.prototype.createShadowRoot* method can be replaced by the web application with a version that saves the references to the created ShadowRoots, allowing the web



(a) Shadowcrypt's input shadowing. A text input hosts a shadow input.

(b) Input element hosts an user-agent shadow root.

(c) Reason for InvalidStateError.In latest W3C standard, Shadow root cannot be created on a host which already hosts an user-agent shadow tree.

Fig. 3: In latest W3C standard, Shadow root cannot be created on a host which already hosts an user-agent shadow tree.

application to access the contents of these shadow roots.

Problem 3: *There is no practical solution to support both the user-controlled encryption and to preserve application functionalities for cloud web applications.*

The encryption algorithm used in ShadowCrypt will change the original data format. When the data is encrypted by ShadowCrypt, these data cannot be stored in the original database in its original form. So, ShadowCrypt can only be applied to text-based web applications, like webmail and social networking sites. In details, the encryption process of ShadowCrypt has the following two problems:

- It encrypts the plaintext using the AES-CCM algorithm, a traditional block cipher algorithm with block length of 128 bits, which introduces the problem of ciphertext storage. The length of the ciphertext is more than a block size characters, which exceeds the maximum length of the original data field.
- It attaches a format signature (`=?shadowcrypt-`) to identify ciphertext strings in the process of decryption and an EOF sequence (`?=`) to denote the end of ciphertext string. These special patterns for ciphertext identification also cause changes to the original format and make it difficult to store the ciphertext in the original database.

Another problem is that the encryption may cause some functions of the application not usable. For example, the web application may check the validity of the format of the input, in this case, since ShadowCrypt cannot preserve the input format after encryption, the web application may not be used. This is a common problem of encrypted web application.

4 The new-branch method

In this section, we present the basic idea of a new solution named "new-branch method" in Section 4.1 and describe the details of this solution for the isolation of input data in Section 4.2. Unlike ShadowCrypt, our method is based on a more robust property in DOM, rather than the low-level properties in Shadow DOM. We also analyze the security of this method in Section 4.3.

4.1 Basic idea

To isolate the DOM, we still use Shadow DOM to realize secure input/output environment. Instead of adding Shadow DOM for the existing text input element, we create a new HTML element which can be the shadow host and does not cause the multiple shadow root problem. Besides, in order to disallow any access to a node in a shadow root from the outside world, we set the mode of shadow root *closed*. More specifically, we create a new branch under the same parent node of the protected element, i.e., we insert a precedent node for the protected element. So, we name this method as "new-branch method".

It avoids involving Shadow DOM internal implementation details, therefore it can work properly in all versions of browsers supporting Shadow DOM.

4.2 The details of our method

As illustrated in Figure 4, there are four steps in this method as described in the following:

Step 1: *Hide the replaced element.* We set the style of the original input element as "*display:none*". Since users will interact with the isolated input environment, the existence of the original input element will cause a collision in display.

Step 2: *Create shadow host.* We create an HTML tag, *<span>*, to host the shadow tree. Because in the latest W3C standard, the shadow tree cannot be hosted by the original input element, which has already hosted a user-agent shadow root.

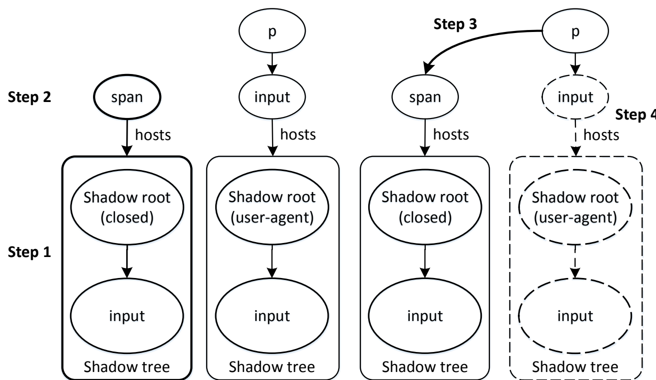


Fig. 4: The steps of "new-branch method". Step 1: hide the replaced element; Step 2: create a shadow host; Step 3: insert into DOM tree; Step 4: create shadow tree.

Step 3: *Insert into DOM tree*. We insert the newly-created shadow host into the DOM tree as the precedent node of the original text input element.

Step 4: *Create shadow tree*. We create a shadow tree which consists of a new input element replacing the original one. Secure input requires the isolation of the application's DOM from the DOM containing the sensitive information.

Claim 1 *The new-branch method is independent of internal implementation of Shadow DOM, so that it can work properly with the upgrade of Shadow DOM.*

Proof: ShadowCrypt cannot work with the upgrade of Shadow DOM, because its design relies on the internal implementation details of Shadow DOM. However, the proposed "new-branch method" does not rely on the low-level details of Shadow DOM. It makes use of a more robust property in DOM rather than Shadow DOM. In our method, Shadow DOM is only used for encapsulation, its internal implementation is transparent to our method. So, "new-branch method" will run correctly as long as DOM is supported and Shadow DOM is not yet deprecated.

4.3 Security proof

Fact 1 *The goal of the shadow DOM is to provide functional encapsulation, which is primarily concerned with establishing functional boundaries in a document tree.*

Proof 1 *The DOM tree is made up of numerous functional subtrees-one or more DOM nodes which implement a certain functionality. The goal of the shadow DOM is to provide functional encapsulation for these subtrees. This is achieved by keeping functional subtrees separate from the document tree (and each other). This separation of shadow DOM subtrees is known as the shadow boundary. Javascripts, CSS rules and DOM queries do not cross the shadow boundary, and thus provide encapsulation and can be used to build secure input/output environment.*

Theorem 1 *Secure input/output environment can be built by the new-branch method if Shadow DOM achieves the ideal encapsulation.*

Proof 2 *As an evolving standard, Shadow DOM has not yet fully achieved the goal of functional encapsulation in the version of V0. It has suffered some attacks. With the upgrade of Shadow DOM, the encapsulation has been greatly enhanced. In the version of V1,* we can prove that these attacks have been invalid by experiments.

Against new-ShadowRoot attack*. Because "multiple shadow roots" has been deprecated in Shadow DOM V1. It is obvious that our "new-branch method" will not suffer this attack. To verify that, we test it in Chrome55 supporting the new W3C standard. The experiment shows that it will raise a DOMException.*

Against CSS selectors attack*. The* /deep/ *and* ::shadow *selectors have already been deprecated in Shadow DOM V1. So our proposed method will not suffer this attack, too.*

To verify that, we test on a span *element which will host a shadow tree. We first test the attack on Shadow DOM V0, as shown in Figure 5, the input value in the Shadow DOM is stolen successfully. Then, we test the attack on Shadow DOM V1, as shown in Figure 6, we couldn't get anything of the inside of Shadow DOM.*

```
> var host = document.getElementsByTagName("span")[0];
  var root = host.createShadowRoot();
  var input = document.createElement('input');
  input.setAttribute("value", "abc");
  root.appendChild(input);
  var x = document.querySelectorAll('*::shadow input');
  console.log(x[0].value);
  x = document.querySelectorAll('HTML /deep/ input');
  console.log(x[0].value);
  abc                                                VM41:7
  abc                                                VM41:9
```

Fig. 5: In V0, "CSS selectors attack"can steal content by */deep/* or *::shadow* selector.

Against Hijack attack*. We believe this attack is impossible. Because browsers protect extension logic by running the extension code in a separate JavaScript environment. The application and the browser extension are not sharing the same JavaScript global prototypes. So the Hijack attack won't happen.*

To verify that, we write a simple browser extension to simulate the secure input environment. We create a

```
> var host = document.getElementsByTagName("span")[0];
  var root = host.attachShadow({mode:'closed'});
  var input = document.createElement('input');
  input.setAttribute("value", "abc");
  root.appendChild(input);
  var x = document.querySelectorAll('*::shadow input');
  console.log(x[0].value);
  x = document.querySelectorAll('HTML /deep/ input');
  console.log(x[0].value);
✖ ▶Uncaught DOMException: Failed to execute                VM43:2
   'attachShadow' on 'Element': Shadow root cannot be
   created on a host which already hosts a shadow tree.
       at <anonymous>:2:17
```

Fig. 6: In V1, "CSS selectors attack" cannot cross the boundary of shadow DOM.

closed shadow tree for the span *element. Then, to simulate a malicious application code, we rewrite the* Element.prototype.attachShadow *function in it. The code as shown below.*

```
//application:
<span>Hello, world!</span>
<script>
console.log("applicaiton JS start");
Element.prototype.attachShadow = function (para
    ) {
  console.log("malicious function executing");
  }
console.log("applicaiton JS end")
</script>
//browser extension:
var host = document.getElementsByTagName("span"
    )[0];
var root = host.attachShadow({mode:'closed'});
var input = document.createElement('input');
root.appendChild(input);
```

When the Element.prototype.attachShadow *function is redefined to replace the function in extension, the Hijack attack will happen. To make sure that, we set the extension running at* document_end *which is a parameter in manifest file, i.e., the extension code will not be run until the application code is executed completely. We also log the execution sequence in the console to verify that.*

If the attack succeeds, the string "malicious function executing" in the Element.prototype.attachShadow *will be output. But in our experiment, we don't see it from the console. This shows that the extension and application use different JavaScript global prototypes. As a result, the Hijack attack won't happen.*

5 Our Solution: ShadowFPE

In this section, we present our ShadowFPE solution. In particular, we present the basic idea in Section 5.1, system design in Section 5.2 and tag method in Section 5.3. We also implement our scheme and analyze the security of ShadowFPE in Section 5.4 and Section 5.5.

5.1 Basic idea

Although ShadowFPE can apply the "new-branch method" to build the secure input/output environment, we still have to solve the "ciphertext storage and identification problem". As described in Section 2.3, FPE can solve the ciphertext storage problem. It looks perfect, but in fact, for a web application, we must solve the problems including: 1) how to know which inputs should be encrypted, 2) how to identify the ciphertext from HTML codes, 3) how to maximize the support of application functions, and so on.

For text-based web applications, we can work as ShadowCrypt. But for other web applications, we only consider encrypting the primary sensitive data (e.g., *name*, *telephone*), like that in anonymization method [19] of privacy-preserving data publishing. For marking the sensitive data, we require the developers/CSPs (cloud service providers) to add some identification information (e.g., HTML tags and attributes) to the web application codes. Meanwhile, we design the identification method to automatically identify them.

5.2 System Design

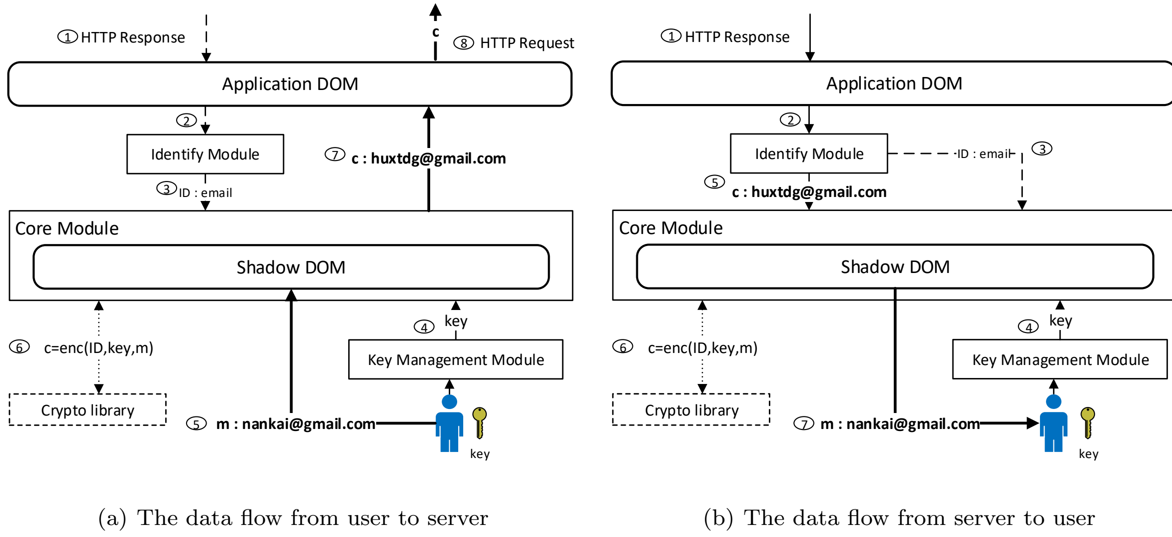
ShadowFPE consists of three modules:

- Identifying module. It implements the identification of sensitive data and its format.
- Core module. It constructs a secure environment by isolating application DOM and accomplishes encryption/decryption on sensitive data.
- Key management module. It enables user to select the key and control his/her privacy information. It can be implemented as that in the ShadowCrypt.

Notations. Denote $Enc\ (ID, key, m)$ as the encryption algorithm, in which key is the key for encryption, ID is the format controlling string for input data, m denotes the plaintext from the user input. Similarly, denote $Dec\ (ID, key, c)$ as the decryption process, where c is the ciphertext and key is the decryption key.

Data flow. We briefly describe the data flow of ShadowFPE (from user to server and vice versa).

User to Server. Figure 7(a) describes the data flow from user to server. When the client receives the HTTP messages (*Step 1*), ShadowFPE will traverse application's DOMs, identify the tagged elements and their formats within these DOMs (*Step 2*), and then transmit these elements (*Step 3*) and the key (*Step 4*) to ShadowFPE's core module. The core module will generate an isolated input environment, monitor user inputs and encrypt them.



(a) The data flow from user to server

(b) The data flow from server to user

Fig. 7: Data flow of ShadowFPE.

User input will only interact with Shadow DOM and the web application code can only access the encrypted data. For example, when a user inputs plaintext, "*nankai@gmail.com*" (*Step 5*), Shadow DOM in ShadowCrypt's core module will capture it and use the identifying module to identify its format (*email*). With the key received from the key management module, Shadow DOM encrypts it using a chosen encryption scheme in crypto library (*Step 6*). Then Shadow DOM updates the data in application DOM to the encrypted data, "*huxtdg@gmial.com*" (*Step 7*). The application will then send an HTTP request message (*Step 8*). We emphasize that encryption will not change the data format.

Server to user. Figure 7(b) describes the data flow from server to user. After client received the HTTP message sent by server (*Step 1*), the browser will parse it. The identifying module in the ShadowFPE will traverse the application DOMs and identify the tagged ciphertext and its format (*Step 2*), then transmit the format (*Step 3*), the key (*Step 4*) and the ciphertext data (*Step 5*) to the core module of ShadowFPE. Finally, ShadowFPE will decrypt the ciphertext (*Step 6*) and display it (*Step 7*) to the user through an isolated output environment.

5.3 Tag Method

We define "tag-rule" to mark the sensitive data, and design "tag-based identification method" to identify the input data to be encrypted and the output data to be decrypted.

5.3.1 Tag-rule

The tag-rule is based on the mechanism of HTML tags and custom attributes. All the current browsers support them, because the custom attributes only require appending a prefix, "*data-*", when defining such attributes.

Key elements. There are two key elements:

- The attribute named "*data-crypt*": In the user-to-server data flow, we add attribute *data-crypt* into input elements to mark the sensitive data and its format. Its value could be one item in {"AES", "FPE", "INT", "STRING", "EMAIL", "DATETIME"}.
- The *span* tag. In the server-to-user data flow, we add the tag *span* and attribute *data-crypt* to mark the ciphertext and data type of the sensitive information.

Example. The input element, whose HTML code is *<input type="text" name="themail"/>*, will be changed to *<input type="text" name="themail" data-crypt= "EMAIL"/>*, where the "*data-crypt*" marks this element as a sensitive data and the required encryption algorithm is FPE for email. The ciphertext, whose value is <%=*themail*%>, will be changed to <*span* data-crypt="EMAIL"><%=*themail*%></*span*>.

5.3.2 Tag-based identification method

This method is used in the browser side for the identification of the input data to be encrypted and the output data to be decrypted.

Input data identification. How to identify sensitive data in the data flow from user to server is described here in two aspects, elements identification and formats identification.

Elements identification. ShadowFPE will traverse each node in application DOMs and each attribute of input elements (such as input, textarea). It will identify each input element with *data-crypt* attribute as an element to be encrypted.

Formats identification. This identification relies on the value of *data-crypt* attribute of that element: (1) if it is "AES", the field is not format-sensitive, that is, this field does not have a specific format to preserve. ShadowFPE will encrypt it using the AES algorithm; (2) if it is one of the items in the set {"INT", "STRING", "DATETIME", "EMAIL"}, this field is identified as format-sensitive and will be encrypted by the related FPE algorithm in the set {FFSEM [8], FFX [11], FPE-DateTime [10], FFX [8]}; (3) if it is "FPE", the field is format-sensitive but the web application did not provide the concrete format of this field. In this situation, ShadowFPE will execute an automatic identification for this field.

To describe the above process, we use an array *encElements* to store the elements of the field to be encrypted, and use an array *encFormat* to store the corresponding format for each field that needs to be encrypted. The function *IdentifyAutoFormat()* realizes the automatic identification. The pseudo-code is shown in Algorithm 1.

Algorithm 1 input element identification

1: $encElements \leftarrow [\,],\ encFormats \leftarrow [\,],\ i \leftarrow 0$
2: **for** $eachElement \in document.getElementByTagName(\text{'}input\text{'})$ **do**
3:   **if** $eachElement.getAttribute() \neq null$ **then**
4:     **if** $eachElement.getAttribute() = \text{"}FPE''$ **then**
5:       $encElements[i] \leftarrow eachElement$
6:       $encFormats[i] \leftarrow IdentifyAutoFormat(eachElement)$
7:     **else**
8:       $encElements[i] \leftarrow eachElements$
9:       $encFormats[i] \leftarrow eachElement.getAttribute(\text{"}data - crypt'')$
10:     **end if**
11:     $i{+}{+}$
12:   **end if**
13: **end for**

Output data identification. How to identify the sensitive data in the data flow from server to user consists of two aspects, elements identification and formats identification.

Elements identification. ShadowFPE will traverse each node and find out elements with the tag *span*, in which the text will be decrypted and later displayed.

Formats identification. Like that in "input data identification", it identifies the data format through the value of *data-crypt* attribute.

Algorithm 2 shows the process code, where *decElements* is adopted to store the elements of the fields to be decrypted.

Algorithm 2 output element identification

1: $decElements \leftarrow [\,],\ decFormats \leftarrow [\,],\ i \leftarrow 0$
2: **for** $eachElement \in document.getElementByTagName(\text{'}span\text{'})$ **do**
3:   **if** $eachElement.getAttribute(\text{"}data - crypt'') \neq null$ **then**
4:     **if** $eachElement.getAttribute(\text{"}data - crypt'') = \text{"}FPE''$ **then**
5:       $decElements[i] \leftarrow eachElement$
6:       $decFormats[i] \leftarrow IdentifyAutoFormat(eachElement)$
7:     **else**
8:       $decElements[i] \leftarrow eachElement$
9:       $decFormats[i] \leftarrow eachElement.getAttribute(\text{"}data - crypt'')$
10:     **end if**
11:     $i{+}{+}$
12:   **end if**
13: **end for**

5.4 Implementation

Input isolation. To isolate the plaintext, ShadowFPE firstly obtains input elements through the "data-crypt" attribute. Then, it utilizes the "new-branch method" to isolate them. Concretely, it creates a new shadow host with tag <*span*> instead of the original input element, inserts the new one, and sets the original one display none. Finally, it captures the plaintext from the new input element, encrypts it according to the marked data format, and updates the ciphertext to the original one. This process is described as follows.

```
for (var i=0;i<encElements.length;i++){
    var oldInput=encElements[i];
    oldInput.setAttribute("style","display:none")
        ;
    var mySpan=document.createElement("span");
    oldInput.parentNode.insertBefore(mySpan,
        oldInput);
    var host=mySpan;
    var root=host.attachShadow({mode:'closed'});
    var input=document.createElement("input");
    input.setAttribute("data-format",encFormats[i
        ]);
    root.appendChild(input);
    input.addEventListener("keyup",function(event
        ) {
        var ret=Enc(this.getAttribute("data-
            format"),key,this.value);
        this.parentNode.host.nextSibling.
            setAttribute("value",ret); });
}
```

Output isolation. For "output isolation", ShadowFPE selects a proper algorithm according to the identified ciphertext and its format, decrypts the data,

and displays it to the user through the isolated Shadow DOM.

To display the plaintext after decryption, the core module generates a shadow tree on the host of *span* tag and puts the sensitive data in the newly-created shadow tree. The following code shows this whole process.

```
for(var i=0;i<decElements.length;i++){
 var host=decElements[i];
 var ret= DEC(decFormats[i],key,host.innerText);
 var root=host.attachShadow({mode:'closed'});
 var retNode=document.createTextNode(ret);
 root.appendChild(retNode);
}
```

5.5 Security Analysis

For a text-based web application, threat model and security of ShadowFPE are both as same as ShadowCrypt. If the input/output environment is secure, ShadowFPE would be secure against data leakages on clouds, network, and clients. We can prove it by the following reasons: 1) the "new-branch method" has been proved that it can defend the proposed attacks in Shadow DOM V1 in Section 4.3; 2) the FPE algorithms are adopted the famous ones that have provable security; 3) the encryption/decryption operations are executed in the client side, and the key is managed by users.

For cloud web applications, we must consider other attacks caused by the new feature, i.e., the application codes are allowed to be modified by clouds. In this case, attacker may *add wrong tags* or *dishonestly mark the sensitive data*, to launch attacks. Because ShadowFPE doesn't reply for wrong tags, and thus the latter becomes the biggest threat of ShadowFPE, in which the malicious cloud can obtain the plaintext by dishonestly marking the sensitive data. In fact, the ideal threat model for CSPs should be "honest but curious", because dishonest will make them lose customers. Even if not, there are some cryptographic methods that can help us to solve this attack. Typically, ShadowFPE can require the signature of the modified pages and their elements which need to be marked by the Certification Authority (CA).

6 Case Studies

ShadowFPE has strong applicability and it can be widely used in a variety of applications. In this section, we wil l show how to apply ShadowFPE in various kinds of web applications, not limited to text-based web applications. Since ShadowFPE applies FPE and allows customers to design their own identification rules, ShadowFPE is found to be applicable in many typical open-source web applications, such as CRM, OA, CMS and email systems, listed in Table 1.

Table 1: List for case studies. We modified some applications to protect some sensitive fields using ShadowFPE.

Product	System type	Encryption fields
X2CRM	CRM	Most sensitive fields
Dolibarr	ERP	Sensitive fields related to cooperations
Fengoffice	OA	Sensitive fields about OA tasks
OpenConf	Meeting System	Author information in a submission

Among them, we demonstrate how to modify a popular and open-source CRM system, X2CRM, using ShadowFPE and later conduct an analytical study of the modification. At the time of our experiment, 6.0.1 is the latest version of X2CRM. To protect user privacy, we mark sensitive fields according to the tag-rule of ShadowFPE, including *title*, *phone*, *website*, *email*, *address*, *employees*, and so on. To preserve the "fuzzy query" function in the original web application, we did not mark the "*name*" field as sensitive.

The screenshot of "create account" page without ShadowFPE extension is shown in Figure 8(a). And the screenshot of "create account" page after using ShadowFPE extension is shown in Figure 8(b). To distinguish between these two pages, we use dashed border lines for input field and make it a little shorter than the original input field (we can maintain the same style in practical application). Users' privacy information are all collected by secure input elements with dashed border lines, and will be later encrypted by ShadowFPE.

To verify the effectiveness of sensitive data protection, we opened the "account detail" page of the user we had just created in the firefox browser without our ShadowFPE extension. All the sensitive data which is shown in Figure 8(a) is displayed as ciphertext with their original format preserved: data in employees, phone, and postal cede field are integers of the same length, data in website field is still in the format of website and so on. Other fields that we did not mark as sensitive information are still shown in plaintexts. Figure 8(b) shows how the Google Chrome browser with ShadowFPE extension displays the information of Alice. As we can see, Alice's information is displayed in plaintext. Therefore, we prove that ShadowFPE prevents user privacy from adversaries but works transparently from the perspective of users.

Because we chose not to encrypt the data in name and revenue field, the primary functionality of the application was preserved: "fuzzy query" on the name field and statistical analysis on the revenue field are still sup-

(a) The X2CRM "account details" page in the browser without ShadowFPE extension. It shows the ciphertext of sensitive fields.

(b) The X2CRM "account details" page in the browser with ShadowFPE extension. It displays the plaintext instead of ciphertext in sensitive fields.

Fig. 8: The screenshot of "create accoun" page without and with ShadowFPE extension

ported, which are the basis of X2CRM's distinguishing functionalities.

7 Evaluation

In this section, we evaluate the performance of ShadowFPE and the overhead of this modification in Section 7.1. The results show that ShadowFPE is practical as it has lower overhead compare to the original application. Additionally, we evaluate the modification cost for the applicability in existing web applications in Section 7.2.

For the consideration of efficiency and practicality in ShadowFPE, our evaluation focuses on two aspects: (1) the response time of the browser which consists of the time taken to create the shadow input, the time taken to perform the encryption and decryption algorithm, the performance loss in practice; (2) application modification overhead. We evaluate the time required for the modification for an existing web application in cloud and verify that ShadowFPE is of practical use.

7.1 Performance evaluation

We conducted the test on an Intel Core i7-6700 3.41GHz x 8 with 16GB of RAM.

Cost of creating secure input environment. We first test the additional cost of using ShadowFPE to create a secure input environment. In this test, we established web pages with 1-1000 input elements, and the result is shown in Figure 9. Exactly as it illustrates, the time cost for generating shadow input is in linear relationship with the number of input fields. The more the number of input elements, the more the time cost is. Because the increase of the number of elements will cause an increase in the time of operations. The figure shows that it takes less than 10ms for creating 100 shadow input elements, and the cost of creating 1000 shadow input is approximately 60ms, which is imperceptible to users. For a normal web page, the number of input elements is far fewer than 1000, even fewer than 100. Thus, the time cost in this step can be ignored for user experience.

Time cost of encryption. ShadowFPE listens on the keystroke events and encrypts the input data. To estimate the effect of different encryption schemes, we choose three kinds of encryption schemes: (1) for integers of length from 1 to 30, we encrypted them with FFSEM, a kind of FPE algorithm; (2) for strings of length from 1 to 50, we choose another kind of FPE algorithm, FFX; (3) for strings of length from 1 to 1000, we adopt AES algorithm. That is because the length of the format sensitive fields in reality will not be too long, and integer and string are the two general types of formats that most applications will distinguish. The long data usually preserves certain format, thus we use AES to accelerate the process of encryption.

The result of the test is illustrated in Figure 11 and 12. We can see that: (1) when we use FFSEM to encrypt integers of length within 30, the time cost is between 1ms and 2ms; (2) the time cost for encryption strings of length within 50 by using FFX is between 1ms and 2ms. As the length of sting increases, the time cost also increases with a linear blow up; (3) the time cost of using AES is the least among the time cost of using three encryption schemes. The time cost approaches 1ms when it encrypts strings of 500 characters. The results are consistent with our expectation that AES is the fastest while FPE schemes, although slower, can preserve data formats for shorter data values.

Overall speaking, the performance is reasonably fast. The time cost for the encryption of format sensitive fields is only within 3ms. The time cost for encrypting long text is similar to the time cost in ShadowCrypt that also adopts AES. We can conclude that ShadowFPE is practical.

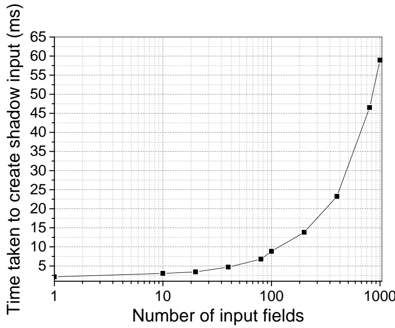


Fig. 9: Time cost for ShadowFPE to create shadow input.

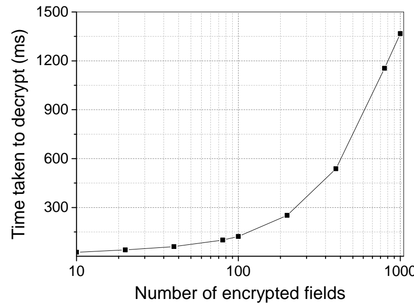


Fig. 10: Time cost for ShadowFPE to decrypt messages on a page.

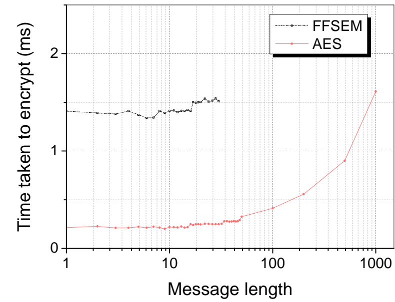


Fig. 11: Time cost for ShadowFPE to encrypt message in different formats by FFSEM and AES.

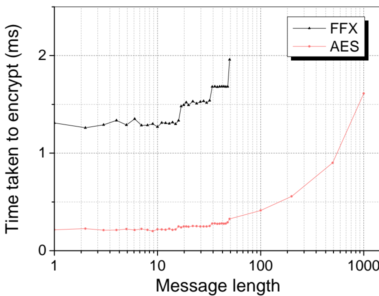


Fig. 12: Time cost for ShadowFPE to encrypt message in different formats by FFX and AES.

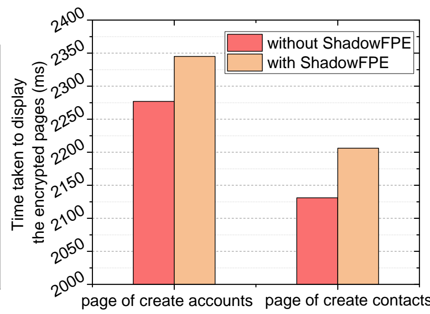


Fig. 13: Time cost to load web pages when using ShadowFPE or not.

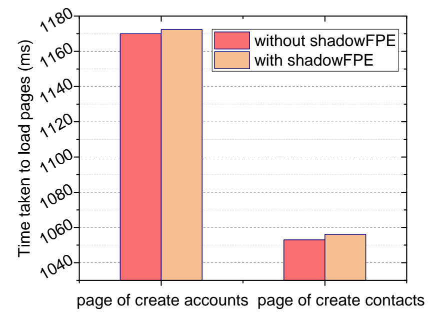


Fig. 14: Time cost to display the encrypted web pages when using ShadowFPE or not.

Time cost of decryption. To measure the time cost for decryption, we established web pages including 1-1000 ciphertext fields. Each page contains various types of sensitive data, including integers, mobile numbers, email addresses, ID numbers, normal strings. And encryption schemes include the corresponding F-PE algorithms and AES. The length of each field is set randomly, aiming at simulating the pages in reality for the evaluation of the average time cost.

As illustrated in Figure 10, the time cost increases with the increase in the number of fields to be decrypted. Though when there are approximately 800 fields to be decrypted in one page, the time cost reaches 1 second, this situation will hardly happen in reality. Even for the CRM system that has many sensitive data, the number of sensitive fields within a web page is normally fewer than 50, and the number is much smaller in other web applications. Thus, users usually cannot realize this change in real applications.

Performance comparison. We also measure the performance loss between ShadowFPE and the original application. We choose two pages of X2CRM to do the test. One of them is a page for accounts creation (*http://website/X2CRM-master/x2engine/index.php/accounts/create*), the other is for contacts creation (*../index.php/contacts/create*). The former has 19 input elements of which 9 is sensitive, the latter has 18 input elements of which 12 is sensitive. These sensitive fields contain mobile numbers, email address, website address, etc.

As shown in Figure 13 and 14, we can see that: (1) when loading web pages, ShadowFPE only needs a few milliseconds to create the shadow input for the sensitive fields; (2) when displaying the encrypted pages, ShadowFPE may take tens of milliseconds to decrypt the sensitive information. So, ShadowFPE fulfills the requirements of practical uses.

7.2 Modification cost evaluation

ShadowFPE requires minimal code rewriting in existing web applications and we evaluate the overhead of this modification. We present primary data in our modification of some applications mentioned in Section 6 (Table 2), including code quantity, the degree of knowledge (the range is from 0 to 1, and 1 identifies the developer is extremely familiar with the application code), number of developers, the time spent on code reading, the

Table 2: Modification overhead of some applications.

	X2CRM	Dolibarr	fengoffice	OpenConf
Code quantity (MB)	84.3	31.1	44.3	2.55
Degree of knowledge	0.3	0.3	0.4	0.3
Number of modification staffs	1	2	2	1
Time spent on code reading (h)	5.5	2	1.5	1.5
Time spent on modifying (h)	1	0.5	0.5	0.5
Quantity of code changes (KB)	2.48	0.89	1.13	1.59

time spent on modifying, and the overhaul of the source code.

8 Conclusions

We present ShadowFPE, a solution for user-controlled encrypted web applications. In contrast to previous approaches, ShadowFPE not only preserves the web applications' functionality but also does not trust any part of web applications based on clouds. ShadowFPE only requires applications to do tiny modification to provide secure input/output operations through a browser extension. ShadowFPE applies format preserving encryption (FPE) to encrypt sensitive data, thus will not violates the format of database fields. ShadowFPE can be widely applied in most kinds of web applications based on clouds. We do experiments on some applications and verify the high usability of ShadowFPE.

Like ShadowCrypt, ShadowFPE does not address how to resist XSS attack. But XSS vulnerability can be avoided through strengthening ShadowFPE. Through censoring user input and filtering the harmful ones, ShadowFPE can defend against XSS attack. This is what we will focus on in our future work.

References

1. S. Kamara, C. Papamanthou and T. Roeder, *Dynamic searchable symmetric encryption*, Proceedings of the 2012 ACM conference on Computer and communications security (CCS), ACM, pp. 965-976, 2012.
2. R. Cheng, J. Yan, C. Guan, F. Zhang and K. Ren, *Verifiable searchable symmetric encryption from indistinguishability obfuscation*, Proceedings of the 2015 ACM conference on Computer and communications security (CCS), ACM, pp. 621-626, 2015.
3. K. Florian, *Frequency-Hiding Order-Preserving Encryption*, Proceedings of the 2015 ACM conference on Computer and communications security (CCS), ACM, pp. 656-667, 2015.
4. J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan and S.W. Duncan, *L-EncDB: a lightweight framework for privacy-preserving data queries in cloud computing*, Knowledge-Based Systems, Elsevier, VOL. 79, pp. 18-26, 2015.
5. R. A. Popa, C. Redfield, N. Zeldovich and H. Balakrishnan, *CryptDB: protecting confidentiality with encrypted query processing*, In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM, pp. 85-100, 2011.
6. W. He, D. Akhawe, S. Akhawe, E. Shi and D. Song, *Shadowcrypt: Encrypted web applications for everyone*, In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, pp. 1028-1039, 2014.
7. B. John and R. Phillip, *Ciphers with Arbitrary Finite Domains*, Topics in Cryptology–CT-RSA, Springer, VOL. 2271, pp. 114-130, 2002.
8. T. Spies, *Feistel finite set encryption mode*, NIST Proposed Encryption Mode, 2008.
9. B. Morris, P. Rogaway and T. Stegers, *How to encipher messages on a small domain*, In Advances in Cryptology-CRYPTO 2009, Springer, pp. 286-302, 2009.
10. Z. Liu, C. Jia and J. Li, *Format-Preserving encryption for datetime*, 2010 IEEE International Conference on Intelligent Computing and Intelligent Systems, IEEE, pp. 201-205, 2010.
11. M. Bellare, P. Rogaway and T. Spies, *The FFX Mode of Operation for Format-Preserving Encryption*, NIST submission, 2010.
12. *Ciphercloud*, https://www.ciphercloud.com/.
13. M. Christodorescu, *Private use of untrusted web servers via opportunistic encryption*, W2SP 2008: Web 2.0 Security and Privacy, 2008.
14. R.A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich and H. Balakrishnan, *Securing web applications by blindfolding the server.* In Proceedings of the USENIX Symposium of Networked Systems Design and Implementation (NDSI), 2014.
15. Lastpass blog: *Cross site scripting vulnerability reported, fixed*, Feb. 2011. http://goo.gl/4MDNjU.
16. Cryptocat blog: *Xss vulnerability discovered and fixed*, Aug. 2012. http://goo.gl/Nq7tVk.
17. D. Pouliot and C. V. Wright, *The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption.* In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, pp. 1341-1352, 2016.
18. P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart and V. Shmatikov, *Breaking web applications built on top of encrypted data.* In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, pp. 1353-1364, 2016.
19. B. Fung, K. Wang, R. Chen and P. S. Yu, *Privacy-preserving data publishing: A survey of recent developments.* ACM Computing Surveys (CSUR), 2010, 42(4): 14.
20. T. Daleniusl, *Finding a needle in a haystack.* Journal of official statistics, 2(3), 329-336, 1986.
21. S. Ruoti, D. Zappala and K. Seamons, *MessageGuard: Retrofitting the Web with User-to-user Encryption.* arXiv preprint arXiv:1510.08943, 2015.
22. B. Mihir and T.H. Viet, *Identity-Based Format-Preserving Encryption.* CCS, 2017.