From Client-side Encryption to Secure Web Applications

by

Emily Stark

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

April 24, 2013

Certified by

Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by

Leslie Kolodziejski
Chairman, Department Committee on Graduate Students
From Client-side Encryption to Secure Web Applications

by
Emily Stark

Submitted to the Department of Electrical Engineering and Computer Science
on April 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis presents an approach for designing secure web applications that use client-side encryption to keep user data private in the face of arbitrary web server compromises, as well as a set of tools, called CryptFrame, that makes it easier to build such applications. CryptFrame allows developers to encrypt and decrypt confidential data in the user’s browser. To ensure an adversary cannot gain access to the decryption keys or plaintext data, CryptFrame provides a browser extension that stores the keys and allows only sensitive regions in the web page to access them. CryptFrame performs templated verification of sensitive regions to grant small amounts of trusted client-side code access to plaintext data in the browser. Finally, CryptFrame provides a principal graph to help users safely change permissions on shared data in the presence of active adversaries. We use CryptFrame to modify several existing Django-based applications, requiring few source code modifications and incurring moderate performance overhead.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor
Acknowledgments

I would like to thank my advisor, Nickolai Zeldovich, and my collaborator Raluca Ada Popa for their ideas, guidance, and assistance on this project. I am also grateful to the members of the PDOS group for listening to presentations, reading paper drafts, and participating in discussions that helped me improve this work greatly.
## Contents

1 Introduction 11

2 Related work 15  
2.1 Web server security 15  
2.2 Isolation in a web browser 15  
2.3 Cryptographic protection 16  
2.4 Computation on encrypted data 17  
2.5 User interface security 17

3 Overview 19  
3.1 Threat model 19  
3.2 Building secure web applications 20  
3.3 Trusted components 20

4 Sensitive regions 23  
4.1 Display integrity 23  
4.2 Visual indicator 24  
4.3 Django integration 24  
4.4 Backwards compatibility 25

5 Templatized verification 27

6 Principal graph 29  
6.1 Principal graph structure 29  
6.2 Principal graph API 31  
6.3 Mapping the principal graph to applications 32

7 Implementation 35  
7.1 CryptFrame admin 35

8 Case studies 37  
8.1 gradapply 38  
8.1.1 Developer effort 38  
8.1.2 Security analysis 40  
8.2 django-forum 41  
8.2.1 Developer effort 41
8.2.2 Admin integration ........................................ 42
8.2.3 Searching on encrypted data ............................... 42
8.2.4 Security analysis ........................................... 43
8.3 django-messages ............................................. 43
8.4 Password manager ........................................... 44

9 Unambiguous user interfaces ................................. 45
  9.1 UI manipulation ............................................. 45
  9.2 Certificate chain manipulation ........................... 46

10 Performance evaluation ...................................... 47
  10.1 Throughput .................................................. 47
  10.2 Latency ..................................................... 48

11 Conclusion .................................................... 51
List of Figures

3-1 An overview of an application using CryptFrame. Shaded components are untrusted. Dashed lines indicate components that CryptFrame adds to the original application. .......................................................... 19

4-1 CryptFrame displays a green shield icon to inform users when they are entering data into a sensitive region. .......................................................... 24

5-1 An example illustrating the Django template language. .......................... 27

6-1 An example of a principal graph for a forum application. Principal nodes are represented as ellipses and shaded principal nodes represent users with public keys provided by the identity provider. Principals are labeled with their public keys and users with their names, for clarity. Rectangles represent data nodes. A dashed edge from \( P \) to \( Q \) indicates that \( P \) has created a certificate for \( Q \) with the labeled attribute. A solid edge from \( P \) to \( Q \) indicates an access relationship: \( P \) has access to all data that \( Q \) has access to. 31

8-1 CryptFrame template code from gradapply for a form with encrypted inputs. The argument to the \('\text{% sensitive %}'\) tag is the ID of the sensitive region. The Javascript code inside the SR tells CryptFrame to encrypt sensitive fields of the form with the principal returned by \text{find_principal}. The “encrypted” class on the textarea element indicates that the value will be encrypted before the form is submitted. ......................................................... 39

8-2 Example Django template code from gradapply that uses CryptFrame to decrypt a ciphertext with a given principal. ................................. 39

8-3 Two possible SRs for adding a reviewer in gradapply. ......................... 40

10-1 Throughput for individual types of requests in django-forum. ............. 48

10-2 Latency for different operations in django-forum. ............................. 49

10-3 Sources of latency imposed by a sensitive region. The baseline is a page that loads the posts in a thread, using a single sensitive region. Each bar shows the decrease in client-side latency that results from removing a single step that CryptFrame performs when loading the sensitive region. ............ 49
List of Tables

6.1 The API exposed to sensitive regions for working with the principal graph. Three calls (decrypt, certify, and grant_access) require a principal’s secret key, which is obtained by following key chains starting from the user’s key. ................................................................. 31

8.1 Changes to four existing applications needed to encrypt sensitive data using CryptFrame. ................................................................. 37
Chapter 1

Introduction

A web application’s security is at the mercy of the web server, which can often be compromised despite the best intentions of the web application developer, leading to disclosure of confidential data managed by the application.

A promising approach to protect confidential user data in web applications is to use Javascript to encrypt the data before it leaves the browser \([2, 4, 6, 22, 24]\). The hope is that the server sees only encrypted data and does not learn the decryption key. One example of such a setting is a competent, honest web developer who wants to host a web application on a shared hosting provider, but does not trust the hosting provider to always install operating system security updates or maintain the server’s physical security. Another example is a developer who runs his own web server, but wants to use an inexpensive and potentially untrustworthy CDN for parts of his application.

In the web application setting, client-side encryption is not sufficient if the web server is compromised, because a malicious server could tamper with the client-side encryption code, steal secret keys from the browser, or read sensitive data off the page. Moreover, recent proposals for computing on encrypted data at the server (e.g., CryptDB for databases \([16]\) and searching on encrypted text \([23]\)) promise to allow servers to deliver useful functionality without being able to decrypt confidential data. When these systems are placed in the web application setting, they no longer provide any security guarantees for the previously mentioned reasons: a malicious server could provide compromised client-side code that steals secret data or keys from the browser. Therefore, in the current web security model, client-side encryption provides only a small increase in security (if any) over encrypting data on the server \([14]\). As we discuss in Chapter 2, existing solutions have one or both of the following shortcomings: either they do not defend against arbitrary server-side compromises, or they are specific to certain applications.

How can we preserve the benefits of client-side encryption in the web application setting? In this thesis, we show how to build secure web applications where client-side encryption indeed protects confidential data – that is, where user data confidentiality is protected against arbitrary server compromises. Our contributions are three-fold: (1) we identify three key challenges that developers must address when building secure web applications using client-side encryption; (2) we provide a set of tools that help developers address these challenges; and (3) we demonstrate how to apply these ideas to several existing web applications.

The three challenges, along with our approach for addressing each one, are as follows:
Challenge 1: Verifying code. The server can send arbitrary code to run on the client, so we must prevent malicious code from accessing sensitive data or the decryption keys, while allowing legitimate code to access sensitive data for formatting it for display or performing client-side computation on it.

To address this challenge, we identify an application-agnostic primitive called a *sensitive region* that can be added to browsers to facilitate secure web applications. A sensitive region is an iframe with a separate origin that runs only signed code, so that cryptographic keys and sensitive data are accessible only to code signed by the developer.

Challenge 2: Verifying data. Even if the user's browser is running the correct web application code, this code might use dynamic data provided by the untrusted server, e.g. embedded in the HTML document or received as a response to an Ajax request. This data could be used in security-sensitive ways, and a malicious server could manipulate the data to influence what operation the client-side code performs.

We identify access control changes as a common situation in which this challenge arises. We provide a *principal graph* abstraction that uses key chaining to cryptographically enforce access control. The principal graph uses *certificate chaining* to verify all principal graph information received from the untrusted server, so that the server cannot perform unauthorized access changes by manipulating information sent to the client. For example, to grant access to data, a sensitive region receives a public key from the server to use for encrypting the data, and the sensitive region verifies a certificate chain to ensure that the received key is correct.

Challenge 3: Unambiguous user interfaces. An attacker may manipulate the user interface to mislead the user into thinking he is performing one operation when he is in fact performing another. This could resemble well-known attacks like clickjacking or phishing, or be as simple as changing labels on a form asking for sensitive data.

Addressing this challenge requires building user interfaces that unambiguously identify the operation being performed. To help users unambiguously identify when they are entering sensitive data, we provide a visual indicator in our browser extension. We also explore the challenges in building unambiguous user interfaces that perform access control changes; for example, if a user is choosing to move a forum post into one of two threads with the same name, the UI must allow him to disambiguate the two, or else the post might be moved into a forum other than the one the user intended. We use several case study applications to show how sensitive regions and the principal graph can be combined with thoughtful UI design to build secure web applications with dynamic access control changes.

To help developers follow this approach, we provide a browser extension and a server-side framework, which together we call CryptFrame. The browser extension implements sensitive regions, and the server-side framework provides libraries for storing and manipulating an application's principal graph, along with deployment scripts.

We used CryptFrame to modify four existing Django applications to protect sensitive data in the event of a server compromise. Our evaluation shows that CryptFrame works well for a range of real applications, including a graduate admissions web site, a forum
application, a messaging application, and a password manager. Using CryptFrame, we adapted these applications to protect sensitive data by changing under 150 lines of code. Our design requires no additional network round-trips in many cases, and our early prototype reduces throughput by 31% and adds 100-215 msec latency on the client.

CryptFrame and existing approaches for computing on encrypted data [9, 12, 16, 23] are complementary and can be used together. Without CryptFrame, proposals for computing on encrypted data lose their security guarantees in the web setting (e.g., a compromised server can inject Javascript code in the client application and steal the decryption key). CryptFrame helps these approaches to retain their properties in the web application setting, and we show an example in Chapter 8.2.3 to demonstrate this point.

The rest of this thesis is organized as follows. Chapter 2 surveys related work. Chapter 3 provides an overview of our approach, assumption, and goals. Chapters 4, 5, and 6 describe the techniques that we use to build secure web applications. Chapter 7 describes our prototype implementation of CryptFrame. Chapter 8 describes how we used CryptFrame to modify several web applications to protect user data from a malicious server, and Chapter 9 explains challenges we encountered in designing UIs for use with CryptFrame. Chapter 10 evaluates CryptFrame’s performance, and Chapter 11 concludes.
Chapter 2

Related work

CryptFrame helps developers build web applications that protect data confidentiality despite arbitrary server compromises, and this is the first work that fully tackles this threat model. The rest of this chapter describes the relation between our contributions and prior work in more detail.

2.1 Web server security

Many techniques have been proposed to deal with security vulnerabilities in server-side web application code: statically checking the application code [3, 26], checking security rules at runtime [15, 28], or structuring the web application to minimize the impact of vulnerabilities [10, 13]. While these techniques make it difficult for an adversary to exploit a vulnerability in the web application, they do not address arbitrary server-side compromises and they assume the adversary does not get access to certain components; once the adversary compromises these components, he could gain access to confidential data. We present applications built with CryptFrame that prevent an adversary with full access to the server from getting access to the plaintext data.

Separating authentication from the rest of an application is often done using single sign-on systems such as OpenID. As we describe later, CryptFrame’s identity provider adopts a similar approach for user login, and for mapping human-readable identities (such as an email address) to a public key.

2.2 Isolation in a web browser

Several techniques aim to address vulnerabilities in a web application’s client-side code (typically Javascript). BFlow [27] and Clilet [8] prevent untrusted client-side widgets (e.g., written in Javascript) from leaking confidential data, and privilege separation can be used to reduce the amount of trusted code in HTML5 applications [1]. Our design uses privilege separation in client-side code – between trusted code, which is signed and runs in the sensitive region with access to the plaintext data, and the rest of the code, which can be altered by the server and does not have access to plaintext – to protect against arbitrary server-side compromises.
Code signing is used to designate trusted code in Java, ActiveX, and others. An important advantage of code signing is that the signing key need not be available during normal operation, thus preventing an adversary that compromises the server from modifying signed code. As we describe later, CryptFrame's templatized verification can be thought of as code signing for the HTML and Javascript that runs in a sensitive region.

2.3 Cryptographic protection

A position paper by Christodorescu [4] proposes encrypting and decrypting data in a web browser before sending it to an untrusted web server, but does not describe how to privilege separate client-side code or distribute keys among users. Several “pastebin” sites encrypt text in the browser before uploading it to the server, and decrypt it in the browser when the user comes back to the same page [6, 22]. In these sites, the key is either stored in the URL's hash fragment [22], or typed in by the user [6], and can be accessed by any Javascript code from the page. As a result, these techniques work only in the face of passive snooping attacks. In contrast, we use CryptFrame to develop applications that protect confidential data in a web application despite active attacks (which can inject arbitrary Javascript code into the web browser).

Cryptocat [24], an ongoing project, originated as a web application using Javascript cryptography to allow encrypted chat, but was critiqued for not considering the issues associated with client-side cryptography [25], which are the very challenges that CryptFrame addresses. As a result, subsequent versions distributed Cryptocat as a browser extension rather than a web application. However, this does not provide a general solution because users would have to install a separate extension for each application that uses client-side encryption. With CryptFrame, Cryptocat could still be implemented as a web application and achieve similar security properties, while installing only one extension for all applications.

Scramble [2] is a Firefox extension that uses OpenPGP to encrypt and decrypt data in social networking sites so that only the intended users can read data and not the social networking site itself. Scramble suffers from two main limitations. First, Scramble does not prevent a malicious site from reading data off the page as the user enters it or after the browser decrypts and displays it, so it does not offer any real security guarantees against a malicious server. CryptFrame does prevent this attack, by using sensitive regions. Second, Scramble offers only limited functionality for access control changes, which can be tedious: when sharing a piece of data with other users, one selects the users from a Scramble “address book” UI, which allows setting permissions on only one piece of data at a time. To give a user access to a forum with Scramble, one has to go through every post in every thread of the forum and give the user access to that post. In contrast, our principal graph abstraction allows developers to use their existing access control UIs. Therefore, using CryptFrame, one can directly give a user access to an entire forum.
2.4 Computation on encrypted data

As mentioned earlier, CryptFrame can be integrated with techniques for computation over encrypted data [5, 12, 16, 18, 19]. We give an example of this integration in Chapter 8.2.3.

CryptDB [16] processes SQL queries over encrypted data and provides some guarantee of security in the event that the application server is compromised: that only data of logged-in users is leaked. CryptFrame provides a stronger guarantee (that no data is leaked in the event of a compromise, even for logged-in users), does not require a trusted proxy server, and addresses a setting in which the client is running code provided by the untrusted server, which CryptDB does not address.

Data preservers [12] use trusted hardware to perform user-approved computations over confidential data on an untrusted server. Data preservers focus on a web service setting, where the untrusted server stores and computes on user data but does not serve a web application that runs on the client. In contrast, we address the web application setting, where an untrusted server can send malicious Javascript code to the browser to read sensitive data or change access permissions. In the web application setting, secure data preservers could be combined with CryptFrame to give the server an interface to compute on sensitive user data, while protecting against active attackers who manipulate the client-side web application to gain access to the data (CryptFrame’s contribution).

2.5 User interface security

In Chapter 9, we discuss how to use CryptFrame to build secure permission-granting user interfaces in the presence of active adversaries. Our approach builds on earlier work [11, 20, 21], but applies these ideas to more dynamic permission granting interfaces, when the trusted user interface may be talking to a compromised server over the network.
Chapter 3

Overview

This chapter provides an overview of our approach for building secure web applications using client-side encryption, by describing how a developer can build such an application with the help of CryptFrame. Figure 3-1 shows the major components of a CryptFrame-based application.

3.1 Threat model

We define a secure web application to be a web application that protects the confidentiality of sensitive user data without trusting any part of the web server. We assume that the web application developer is honest and can audit his own client-side application code, but does not want to trust the entire server stack with users’ sensitive data.

CryptFrame helps developers defend against an active attacker who is in complete control of the web server except for the developer’s code signing key. For example, CryptFrame is designed to help defend against an attacker with physical access to the web server and database, as long as the signing key is not stored in either of those locations.

CryptFrame’s goal is to help developers build applications that keep sensitive user data confidential from a complete server compromise, even if users view this data in their browser or change access control permissions.

Figure 3-1: An overview of an application using CryptFrame. Shaded components are untrusted. Dashed lines indicate components that CryptFrame adds to the original application.
3.2 Building secure web applications

A secure web application that allows users to enter and view sensitive data must isolate this data on the page and encrypt it before sending it to the untrusted server, and the server should not be able to trick the user's web browser into running arbitrary code in the isolated context. To accomplish this with CryptFrame, the application developer decides on sensitive data that should be encrypted, and marks sensitive regions in the application's web pages that collect or display sensitive data. The developer then signs the sensitive region code with his private signing key. Users download and install the CryptFrame browser extension, and log in at a trusted third-party identity provider, which associates meaningful user identifiers, such as email addresses, with public keys. Many applications can use the same identity provider, and the application needs to contact the identity provider only when new users register or when users switch browsers.

When a user visits a web application that uses CryptFrame, the CryptFrame browser extension obtains the developer's public key (using one of the methods described below), verifies sensitive region code, and isolates sensitive regions from the rest of the page. The extension gives verified sensitive regions access to the user's key, which can be used to encrypt and decrypt sensitive data.

Sensitive regions enforce that the correct code runs in users' browsers, but this code can receive data from the untrusted server, which could be used to influence which security-sensitive operations are performed. In particular, CryptFrame provides a principal graph abstraction that helps developers cryptographically enforce dynamic access control. However, the server could provide the wrong key to the principal graph library in the sensitive region when encrypting sensitive data. For this reason, the principal graph uses certificate chains to ensure that the keys received from the untrusted server are correct. Sensitive region code can use a CryptFrame client-side library to securely retrieve keys corresponding to application-specific principals and use these keys to set access permissions. The principal graph is stored on the untrusted server, but the key chains and certificate chains ensure that an attacker cannot make unauthorized changes to the principal graph.

3.3 Trusted components

A secure web application that uses CryptFrame relies on the following assumptions.

- **Uncompromised end user machines.** End users must be running uncompromised browsers with the CryptFrame extension obtained from a trusted source. The extension is application-independent, so users have to download and install it just once to use any CryptFrame application.

- **Developer signing key.** The developer must have a private key for signing code. This key can be kept offline and used only when deploying new code to production. The user's browser must have some trustworthy means of obtaining the developer's public key for verifying code; the key could be obtained as an extension to an SSL certificate, or via an SSL connection to a trusted directory service.
• **Identity provider.** CryptFrame requires a trusted third-party identity provider that associates human-meaningful names, such as email addresses, with public keys. The identity provider can also serve as storage for private keys, so that users can switch browsers without needing to export keys. We believe it is reasonable to assume a trusted identity provider, since it is similar to existing OpenID providers, and its functionality is limited, application-agnostic, and not on the critical path of most operations.

• **Sensitive region code.** When using CryptFrame, developers isolate portions of their code inside a sensitive region. The code inside sensitive regions is trusted: a vulnerability inside the sensitive region can be used to steal user data. Developers should audit their sensitive region code for vulnerabilities (e.g., cross-site scripting).

• **Visual indicators.** To maintain full security against arbitrary compromises, the user must determine when he is on the correct website and when he is entering data into a sensitive region. Otherwise, an attacker who controls the server could perform a phishing-like attack by redirecting the user to a different site entirely, or by serving a page that looks the same as the correct one except with all sensitive regions removed, so that the user enters sensitive data into an unisolated form field. The TLS visual indicators implemented by all major browsers help with the first of these requirements; users should always check for the TLS lock icon and verify that the domain name is correct before entering sensitive data. To help with the second requirement, the CryptFrame browser extension includes a visual indicator that tells the user when he is interacting with a form field inside a sensitive region. Security-conscious users should check for this visual indicator before entering sensitive data in a form field. Visual indicators such as the TLS lock icon have been shown to be ineffective for most real users [7]. For that reason, we expect that in practice the CryptFrame visual indicator will be of use only to expert users. However, applications built with CryptFrame can still provide some guarantee even if users ignore the visual indicators: in this case, a secure web application will guarantee that, in the event of a server compromise, all sensitive user data that existed in the application before the attack remains confidential. Data that is entered during the attack might be leaked because a user might be entering data into a part of the page that is not contained in a sensitive region.
Chapter 4

Sensitive regions

A sensitive region (SR) is a section of a web application’s HTML code that is isolated from the rest of the page (“outer page”). We provide this primitive for the purposes of displaying or collecting sensitive data, or for granting access to sensitive data. Code running on the outer page, which is provided by the malicious server, is isolated from the sensitive data inside the SR. For example, an input widget for entering confidential text, or a form that moves a post to a different forum, should be contained in a sensitive region. Since these regions control access to sensitive data, they must be isolated from the rest of the page, which is untrusted.

To implement sensitive regions, the CryptFrame browser extension runs SR code in an iframe, which runs in a different origin from the outer page. The Same-Origin Policy ensures that the outer page cannot read the contents of the sensitive region, or run any code inside it. Because the CryptFrame extension gives decryption keys only to code running on its own iframes, the outer page cannot obtain decryption keys.

4.1 Display integrity

The isolation provided by the Same-Origin Policy stops the untrusted page from reading sensitive data that appears inside the sensitive region, but this isolation cannot stop an attacker who controls the outer page from manipulating the UI to trick the user into granting access to sensitive data. Such attacks are known as clickjacking or UI redressing attacks. For example, suppose a sensitive region contains a button that, when clicked, makes a piece of data public. An attacker could cover up the sensitive region, place another button on top of the “Make Public” button (say “Keep Private”), and entice the user into clicking on the attacker’s button. The attack code can remove the button at the last second, letting the click fall through to the “Make Public” button, which the user did not intend to click.

To prevent such attacks, the CryptFrame browser extension enforces display integrity for sensitive regions. For example, the sensitive region should be fully visible for a human-perceivable amount of time before mouse clicks or keystrokes are passed to it. We do not discuss this further, since a variety of techniques for display integrity have been explored in prior work [11, 21].
4.2 Visual indicator

As discussed in Chapter 3.3, for a web application to protect all sensitive data from a malicious server, the CryptFrame browser extension must help users determine when they are entering data into an encrypted text input or file upload widget in a sensitive region. The CryptFrame extension displays a green shield icon, shown in Figure 4-1, near the address bar when form fields inside sensitive regions have focus. For example, when a user clicks on a text box inside a sensitive region, the green shield icon will appear, alerting the user that the data being entered cannot be read by the web server, and the icon will remain there for as long as the text box has focus. This mechanism allows security-conscious users to avoid phishing attacks where the attacker removes a sensitive region from the page without changing its appearance.

The Same-Origin Policy allows any HTML element in a page to obtain focus, even if it steals focus from a different origin. As a result, an attacker can change focus to a page element in the outer page while the user is typing, thereby obtaining at least a few characters from the user (until the user notices the shield icon is gone). The user can verify that none of their sensitive keystrokes were pilfered in this way by ensuring the green shield icon remains after they finish typing; an adversary cannot change focus back to an input element in a sensitive region after stealing focus.

If a user misinterprets CryptFrame’s shield indicator, the application will achieve a weaker form of security: an attacker who compromises the server will not be able to learn anything about data that existed in the application before the compromise occurred. To steal data as it is being entered into the application, the attacker could perform a phishing-style attack by replacing sensitive regions with HTML that looks the same but is not isolated or verified. However, the attacker cannot decrypt ciphertexts already stored on the server. To obtain private keys for decrypting these ciphertexts, the attacker would have to manipulate a user into granting access to them, but the principal graph abstraction discussed in Chapter 6 ensures that access can only be granted within a verified sensitive region.

4.3 Django integration

As a concrete example of how sensitive regions can be used in real applications, the CryptFrame server-side framework allows developers to denote sensitive regions using a {% sensitive %} template tag in their Django templates. CryptFrame also provides several template tags that can be used within sensitive regions to perform common tasks. For example, the {% ciphertext %} tag is translated into Javascript code that decrypts a ciphertext.

After denoting sensitive regions with template tags, the developer runs a CryptFrame deployment script to prepare the templates for the client’s browser by annotating sensitive regions with a special HTML attribute. (This deployment script also signs the sensitive
region code, as described in Chapter 5.) When a page is rendered in the browser, the CryptFrame browser extension scans the page for regions that are marked as sensitive with the special HTML attribute. It replaces these regions with iframes that serve a page on the extension’s origin. The extension uses the postMessage API to send the contents of the sensitive region to the iframe page to evaluate. Since secret keys are only available to the extension origin, the sensitive region code cannot read confidential data or make access control changes until it is run inside the iframe. In the next chapter, we describe how CryptFrame prevents the attacker from supplying arbitrary code to run inside the sensitive region.

4.4 Backwards compatibility

Some users might be unwilling or unable to install a browser extension, and we would like to be able to provide some isolation between the untrusted server and the sensitive data in this case. Sensitive regions can be implemented without a browser extension, albeit without any protection against UI redressing and phishing attacks. A trusted server can serve a static HTML and Javascript page that the developer includes in an iframe. This trusted page can verify and run sensitive region code. As long as the trusted server is on a different origin than the rest of the application, the Same-Origin Policy stops the untrusted part of the application from reading sensitive data that appears inside the iframe. This affords some security to users who do not have the CryptFrame browser extension installed. However, without an extension, an attacker can perform a UI redressing attack on an access-granting sensitive region or a phishing attack to trick the user into entering sensitive data outside a sensitive region, so the application is not guaranteed to be secure without the extension installed.
Chapter 5

Templatized verification

Since sensitive region code has access to private keys and sensitive data in plaintext, a SR should not run arbitrary code provided by the outer page. Instead, the developer should use a secret signing key to sign the code that runs in sensitive regions, and only SRs running verified code should be given access to the user's private keys.

What makes this challenging is that many modern web applications are built on frameworks such as Django, in which developers write templates that are rendered into HTML documents at runtime. For compatibility with this style of development, we propose a technique called templatized verification, in which a developer signs a template of the code that can run inside a SR, and the browser verifies that the running code conforms to this signed template.

In our prototype, the CryptFrame server-side framework includes a deployment script for generating templatized signatures. CryptFrame generates a tree representing the sensitive region contents written in the Django template language, hashes the contents of each node of the tree, and signs the tree of hashes, including flags for subtrees that can be repeated or omitted (corresponding to loops and conditionals). This tree of hashes is sent to the browser along with the sensitive region contents, and the CryptFrame browser extension verifies that the sensitive region code conforms to the signed template representation. The extension also sanitizes the contents of template variables, since the sanitization performed by the Django

```plaintext
{% sensitive "messages" %}
<ul>
  {% for msg in msgs %}
  <li>
    {{ msg.title }}
    {% if msg.author == user %}(sent by you){% endif %}
  </li>
  {% endfor %}
</ul>
{% endsensitive %}
```

Figure 5-1: An example illustrating the Django template language.
runtime cannot be trusted. Only when the SR code has been verified and sanitized does the CryptFrame browser extension give it access to private keys.

More specifically, the CryptFrame implementation of templatized verification allows developers to specify sensitive regions in their templates by surrounding the regions with the \{ sensitive \} tag. The developer then runs a CryptFrame deployment script that encodes SR checksums in JSON. Django parses the template into a list of nodes, each of which contains a combination of strings, template variables, and other nodes. For each node in this list, the CryptFrame deployment script appends to the checksum structure a hash if the node just contains a string, or an object containing the type of the node (e.g. loop, conditional, template variable) and a recursively computed checksum of the contents of the node.

The deployment script appends a sensitive region’s checksum structure inside a <checksum> tag with a signature, and also includes HTML comments to assist the browser extension in matching the rendered template to the templatized checksum. For example, CryptFrame surrounds template variables with <!-- dyn --> comments to indicate to the browser extension that the contained string is part of a template variable and that the next template node begins after the closing <!-- /dyn --> comment. Before running a SR’s code, the CryptFrame browser extension uses these HTML comments to parse the SR’s contents into a list of nodes with checksums, and verifies that this list matches the structure inside the <checksum> tag and signed by the developer. In our prototype implementation, the CryptFrame browser extension retrieves the developer’s key by asking the IDP for the key corresponding to the page’s origin, which means there must be one global IDP for all applications using CryptFrame. As discussed in Chapter 3.3, a variety of other key distribution mechanisms could be used instead, such as including the key in an extension to the SSL certificate.

For example, consider the template shown in Figure 5-1. The signature on this template allows the sensitive region to contain an HTML document made up of a <ul> tag, followed by any number of list items, followed by a </ul> tag. The contents of each list item can be made up of any string, optionally followed by the string "(sent by you)".

In some cases, templates take input from the untrusted server, giving an attacker some power to interfere with the contents of the sensitive region. For example, an attacker can substitute any string for the contents of a template variable, since the developer does not know at signing time what the contents of the variable should be. For this reason, security-sensitive text such as form labels should be static strings instead of template variables. In Chapter 9, we discuss these concerns further and give examples of how developers can use templatized verification safely and prevent such attacks from compromising security.
Chapter 6

Principal graph

Sensitive regions and templatized verification greatly reduce the adversary’s ability to extract sensitive data or cryptographic keys from the user’s browser. However, sensitive region code still receives data from the untrusted server that might be used in security-sensitive ways. For example, consider a web forum application, which allows a user to move a secret thread to a new forum. A typical UI for this action allows the user to choose the new forum from a dropdown menu. This dropdown and the associated form should be contained in a sensitive region to stop malicious code from moving a thread on the user’s behalf. However, when the user submits the form, the untrusted server must provide the public key of the new forum, so that Javascript code in the user’s browser can encrypt the contents of the secret thread using the new forum’s key. If an attacker supplies his own public key, instead of the new forum’s public key, the attacker will be able to decrypt the contents of the secret thread using his own secret key.

To allow developers to safely obtain and manipulate keys in UIs such as this one, we introduce a principal graph abstraction for managing access control permissions that are stored and served by an untrusted server. Since the principal graph uses cryptography to enforce access permissions, modifications cannot be made without a user’s private key; therefore, the principal graph can be modified only by code running in a sensitive region in a user’s browser, not by the server. Thus, we must also ensure that code running in a sensitive region cannot be tricked by the untrusted server to make unintended changes to the principal graph.

6.1 Principal graph structure

The principal graph stores two kinds of nodes. Principal nodes represent any entity in the application’s access control policy, such as users, groups, messages, and discussion threads. Principals are identified by public keys, and each principal has a corresponding private key. Data nodes, on the other hand, represent confidential data that is accessible only to a specific set of principals.

The principal graph also keeps track of two kinds of relationships (edges) between nodes. An access relationship from principal $P$ to principal $Q$ indicates that $Q$ can access all of $P$’s
data. This relationship is implemented by encrypting $P$’s private key with $Q$’s public key. These wrapped keys form key chains, which enforce access permissions cryptographically.

Certificate relationships help secure web applications determine the correct key to use for encrypting confidential data. For example, if a user wants to move a forum post to a particular discussion thread, the sensitive region must obtain the public key for the principal corresponding to this thread. But how does the sensitive region verify that a given public key is indeed the key for the thread that the user selected? Attribute certificates help applications perform this verification. Any principal $P$ can create a certificate for another principal $Q$, certifying that $Q$ has a particular attribute, which is a \{name: “value”\} pair.

Like other certificate-based systems, the principal graph does not have a cryptographic mechanism for ensuring that revoked certificates are actually deleted; a principal that is certified at one time can replay that certificate later even if the signer asks for the certificate to be deleted. A revocation mechanism, such as a certificate revocation list stored on the identity provider, could be combined with the principal graph to prevent this problem.

The principal graph uses the trusted identity provider as a root of trust, but it is undesirable to have the identity provider involved with all application-specific access control changes. The principal graph allows us to limit the involvement of the identity provider to just providing keys for individual users.

As an example of how to use the principal graph, we return to the example from the beginning of this chapter, where we described a sensitive region that allows a user to move a thread into a forum. Suppose a user Alice wants to move a thread into the “Secrets” forum created by Bob. Let $B$ be the principal representing Bob, and let $F$ be the principal representing Bob’s “Secrets” forum. When Bob creates the forum, $B$ produces a certificate for $F$ specifying that $F$ has the attribute \{forum-name: “Secrets”\}. Later, when Alice (whose principal is $A$) wants to move a thread into the “Secrets” forum, the sensitive region running on behalf of $A$ can verify that it is encrypting the thread to the intended forum principal: the sensitive region can use $B$’s public key (obtained from the identity provider) to verify that the principal received from the server has been certified by Bob as having the \{forum_name: “Secrets”\} attribute. Any principal can create a certificate for any other principal, so certificates can be chained to express more complex chains of trust, and using attributes on these certificate chains allows sensitive regions to express application-specific trust relationships.

Principals, key chains, and certificate chains form a principal graph, an example of which is shown in Figure 6-1. This graph represents the access control permissions for a forum application. Users create groups, forums, threads, and posts. Only users who have a key chain to the principal for a post can decrypt the post’s key in order to read the post. When granting a group access to a forum, the sensitive region can verify a certificate chain to ensure that the group key received from the server corresponds to a particular human-meaningful group name specified by the group’s creator.

The principal graph is stored on the untrusted web server, but an attacker can create certificates or read principals’ data only when he knows the corresponding private keys. For example, an attacker controlling the web server cannot read the data in a particular thread unless he knows the private key corresponding to that thread, or the private key of a principal who has access to that thread, because no private keys are stored in plaintext in the principal graph.
Figure 6-1: An example of a principal graph for a forum application. Principal nodes are represented as ellipses and shaded principal nodes represent users with public keys provided by the identity provider. Principals are labeled with their public keys and users with their names, for clarity. Rectangles represent data nodes. A dashed edge from $P$ to $Q$ indicates that $P$ has created a certificate for $Q$ with the labeled attribute. A solid edge from $P$ to $Q$ indicates an access relationship: $P$ has access to all data that $Q$ has access to.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_principal(optional name) → Principal</td>
<td>Creates a new principal by generating a key pair and storing its public key at the server. Optionally creates and stores a certificate for the name of the principal, signed by the currently logged-in user.</td>
</tr>
<tr>
<td>lookup_principal(attr1, ..., attrN, authority) → Principal</td>
<td>Looks up a principal with attr1, as certified by a principal with attr2, ..., as certified by a principal with attrN, as certified by the user named authority. Returns a principal only if each of the certificates is valid.</td>
</tr>
<tr>
<td>Principal.encrypt(plaintext) → ciphertext</td>
<td>Encrypts plaintext data with the principal’s public key.</td>
</tr>
<tr>
<td>Principal.decrypt(ciphertext) → plaintext</td>
<td>Decrypts the ciphertext with the principal’s secret key.</td>
</tr>
<tr>
<td>Principal.certify(subject_princ, attr) → Certificate</td>
<td>Signs (subject_princ, attr) and stores the certificate at the server.</td>
</tr>
<tr>
<td>Principal.grant_access(grantee)</td>
<td>Gives grantee access to all data that Principal has access to, by encrypting Principal’s secret key with grantee’s public key and storing the wrapped key at the server.</td>
</tr>
</tbody>
</table>

Table 6.1: The API exposed to sensitive regions for working with the principal graph. Three calls (decrypt, certify, and grant_access) require a principal’s secret key, which is obtained by following key chains starting from the user’s key.

6.2 Principal graph API

The CryptFrame framework provides a server-side library for storing and serving the principal graph, and a client-side library that runs inside sensitive regions. The client-side
library allows sensitive regions to look up principals, make changes to the principal graph, and encrypt/decrypt data. Table 6.1 shows the API that the client-side library exposes to sensitive regions. The principal graph API is designed so that the principal graph client library can verify that the results of calls to the API are correct for the arguments that were passed in: e.g., if the application requests a key with a certain attribute, the client library can check that the resulting key has a certificate with that attribute.

The principal graph API is designed to be used by Javascript code, but the job of the Javascript code is to faithfully perform operations requested by the user. Attributes help the Javascript code translate user intent into principal graph operations by allowing the code to directly map user inputs (e.g. dropdown items selected by the user) into attributes on certificates. This means that attributes are often human-meaningful strings. For example, suppose that Alice is sharing her data with “Bob’s group.” A secure sensitive region for performing this action would allow Alice to select from a dropdown menu that lists group names along with their creators. To look up the principal for the selected group, the sensitive region code should retrieve the name and creator that are displayed in the selected dropdown item and use those as attributes in a call to lookup_principal(). Using human-meaningful names such as “Bob’s group” (instead of, for example, randomly generated identifiers) allows user interactions to be translated directly into principal graph operations, without giving the attacker a chance to substitute different attributes and thereby affect the principal being returned. In Chapter 9, we give further examples to explain why human-meaningful attributes are necessary for building safe sensitive regions that link user input directly to principal graph manipulations.

6.3 Mapping the principal graph to applications

Typically, a developer uses the principal graph by associating principals with some or all of the models in the application, and adding principal fields to the models to make this association explicit. For example, in a forum application, a developer might modify the thread model to include a principal field. When an object $O$ is created, a new principal $P$ should be generated and $O.principal$ should be set to $P$.

Next, the developer decides which fields represent sensitive data, such as the body field of a post model in the forum example. An object’s sensitive fields should be encrypted and decrypted inside sensitive regions. In many cases, if an object $O$ has a sensitive field $x$, then $O.principal$ will be used to encrypt and decrypt the contents of $O.x$.

Other fields of $O$ may represent objects that have access to $O$’s sensitive data. For example, if $O$ is an object representing a forum post, then $O.thread$ has access to $O$. When $O.principal$ is created or changed, $O.thread.principal$ should be given access to $O.principal$ using the principal graph grant_access call.

When principals for objects are retrieved from the untrusted server, the sensitive region must verify attribute certificate chains for those principals. A model with an associated principal should have a human-meaningful name field. When users perform actions in the application, they make security-sensitive decisions based on these human-meaningful names (such as the title of the forum in which they are placing a thread). Finally, a model with a human-meaningful name should have one or more certifier fields, indicating principals that
create the model's certificates. For example, if \( F \) is an object representing a forum, \( F.title \) is the human-meaningful name of the forum, and \( F.creator \) is a certifier for the forum: when \( F \) is created, \( F.creator.principal \) (the principal associated with the user who created the forum \( F \)) creates a certificate saying that \( F.principal \) represents the forum with title \( F.title \).

These relationships define the behavior of sensitive regions in terms of the application's models. When a new object \( O \) is created, each certifier field should create a certificate for \( O \)'s human-meaningful name. If \( O.x \) is a certifier field, then when data is encrypted with \( O.principal \), the principal must have a certificate signed by \( O.x \), who should have a certificate signed by a certifier field of \( O.x \), and so on until the chain ends in a user certified by the IDP. Similarly, any fields who have access to \( O \) should be given access via a `grant_access` call, using certificate chains formed from the certifier fields to fetch the principals.

In Chapter 7.1, we describe how a developer can provide simple annotations denoting these relationships, which allow CryptFrame to automatically generate secure interfaces in an admin application for adding and editing data.
Chapter 7

Implementation

We implemented a prototype of CryptFrame for Django applications, although our design can be generalized to other MVC web application frameworks. The server component, including code to manage the principal graph and sign application templates, consists of 791 lines of Python and 105 lines of C++ (for cryptography). A prototype identity provider consists of 283 lines of Python and HTML.

The client-side code consists of 1285 lines of Javascript and 481 lines of C++ (for cryptography), and provides the CryptFrame browser extension (using the Chrome extension API), the client-side library for manipulating the principal graph, and helper code for sensitive regions (e.g., for communicating the encrypted contents of input values to the outer page, encrypting files before uploading them, and redirecting link clicks inside the sensitive region to the outer page). The browser extension does cryptography by sending requests to a local C++ server that wraps the functionality of the NaCl cryptography library. For ease of use by end users, this local server could be deployed as a NPAPI browser plugin.

7.1 CryptFrame admin

We bundle CryptFrame with a custom version of the Django admin application, which is a built-in management application used by many other Django applications. The Django admin allows administrative users to create, edit, and view data stored in the application by providing a generic web-based interface to editing objects. We modified the Django admin to work with CryptFrame; after the developer provides a few simple annotations on their data models, users in the CryptFrame admin can view and edit sensitive data without leaking it.

To use the CryptFrame admin, the developer must first decide which models have associated principals and add principal fields to these models. Next, the developer adds four types of annotations to a data model: human_name, sensitive, has_access, and certifier, whose meanings are described in Chapter 6.3. In Chapter 8, we give more examples of how these annotations are used in case study applications. After the developer runs a deployment script, administrative users can securely add and edit data in the CryptFrame admin, where data is encrypted and the principal graph manipulated using appropriate certificate chains.
The main challenge in modifying the Django admin for use with CryptFrame is that it uses one template to render a form for editing an object of any model. This template uses dynamically generated labels on input fields, which are unsafe to use with CryptFrame because an attacker can manipulate the labels without failing the templatized verification check. To specialize these “polymorphic” templates to a particular application, CryptFrame provides a deployment script that instantiates this template using the labels corresponding to each model in the application. CryptFrame’s admin puts forms for editing annotated models inside sensitive regions, and adds code inside these sensitive regions to encrypt and decrypt data, and grant access to the correct principals when objects are created or edited.

Adapting Django admin to use CryptFrame required a 74-line Python deployment script and 492 lines of Javascript and HTML, as well as 133 lines of Python added or changed from the original application.
Chapter 8
Case studies

We used CryptFrame to modify four real applications to protect sensitive data in the case of a server compromise: a graduate school admissions website, a forum application, a messages

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of code changed or added</th>
</tr>
</thead>
<tbody>
<tr>
<td>gradapply</td>
<td>1 create user principal 1 change type of rec. field</td>
</tr>
<tr>
<td></td>
<td>2 admin annotations</td>
</tr>
<tr>
<td></td>
<td>5 create statement and letter principals</td>
</tr>
<tr>
<td></td>
<td>6 app settings changes</td>
</tr>
<tr>
<td></td>
<td>9 add principal to model</td>
</tr>
<tr>
<td></td>
<td>13 enter/view reviews</td>
</tr>
<tr>
<td></td>
<td>13 add new reviewers</td>
</tr>
<tr>
<td></td>
<td>28 enter/view statements</td>
</tr>
<tr>
<td></td>
<td>37 enter/view rec. letters</td>
</tr>
<tr>
<td></td>
<td><strong>115 total (original application: 17,390)</strong></td>
</tr>
<tr>
<td>django-forum</td>
<td>2 create user principal</td>
</tr>
<tr>
<td></td>
<td>4 key chain caching</td>
</tr>
<tr>
<td></td>
<td>7 app settings changes</td>
</tr>
<tr>
<td></td>
<td>7 view thread</td>
</tr>
<tr>
<td></td>
<td>9 add principals to models</td>
</tr>
<tr>
<td></td>
<td>10 admin annotations</td>
</tr>
<tr>
<td></td>
<td>21 create new post</td>
</tr>
<tr>
<td></td>
<td>24 search thread</td>
</tr>
<tr>
<td></td>
<td>33 create new thread</td>
</tr>
<tr>
<td></td>
<td><strong>117 total (original application: 855)</strong></td>
</tr>
<tr>
<td>django-messages</td>
<td>62 total (original application: 881)</td>
</tr>
<tr>
<td>password manager</td>
<td>12 total (original application: 499)</td>
</tr>
</tbody>
</table>

Table 8.1: Changes to four existing applications needed to encrypt sensitive data using CryptFrame.
application, and a browser-based password manager. In this chapter, we describe these case studies to demonstrate the following:

First, our approach for developing secure web applications captures the security policy of real applications and allows the applications to provide meaningful security guarantees. The applications' access control policies map naturally into principal graphs, leading to intuitive calls to CryptFrame's API.

Second, using CryptFrame requires minimal developer effort. Moreover, applications that use the Django admin for access control can instead use our CryptFrame admin application with just a few developer annotations.

8.1 gradapply

gradapply manages the graduate school admissions process for a department. There are four types of users in gradapply: applicants, recommenders, reviewers, and a single department chair. The chair is specified at install time, while reviewers, recommenders, and applicants are all created dynamically.

In gradapply, we chose to encrypt three types of sensitive data: the statement of objectives essays written by applicants, the letters of recommendation uploaded by recommenders, and the reviews written by reviewers. The application gives all reviewers access to all applicants' data and to all letters uploaded by recommenders. We mapped this access control policy to the principal graph as follows.

- At install time, the department chair creates a principal $R$ with a certificate for attribute \{group: "reviewers"\}.
- When an application creates a statement of objectives, a new principal is created with the attribute \{name: "statement"\} and $R$ is given access to this principal. The statement of objectives is encrypted to the \{name: "statement"\} principal.
- When a recommender enters a new letter of recommendation, a new principal is created with the attribute \{name: "letter"\} and $R$ is given access to this principal. The letter of recommendation is encrypted to the \{name: "letter"\} principal.
- When a new reviewer is added, the reviewer's public key is looked up at the IDP, and the reviewer is given access to $R$.

8.1.1 Developer effort

Modifying gradapply to encrypt/decrypt data and using the principal graph to implement the above access control policy took 115 lines of code, as shown in Table 8.1. For each form that collects sensitive data and each location that displays it, we surrounded the region with the sensitive template tag provided by CryptFrame. Ciphertexts can be decrypted and displayed by passing the ciphertext and the principal to use for decryption to the ciphertext template tag. If a SR contains a form that takes sensitive data as input, then the developer uses the cryptframe.register_encrypted_form function exposed by
[% sensitive "statement" %]
<script>
cryptframe.register_encrypted_form(
    document.getElementById("statement_form"), // form to encrypt
{
    "find_principal": function (next) {
        // Find the principal to use for
        // encrypting data in this form.
        cryptframe.lookup(
            [new cryptframe.CertAttr("name", "statement")],
            cryptframe.user_email(),
            next
        );
    }
});
</script>

<form id="statement_form" action="" method="POST">
    <label>Enter your statement of objectives here:</label>
    <textarea name="statement_objectives" class="encrypted">
    </textarea>
    <input type="submit" />
</form>

{end_sensitive %}

Figure 8-1: CryptFrame template code from gradapply for a form with encrypted inputs. The argument to the (% sensitive %) tag is the ID of the sensitive region. The Javascript code inside the SR tells CryptFrame to encrypt sensitive fields of the form with the principal returned by find_principal. The “encrypted” class on the textarea element indicates that the value will be encrypted before the form is submitted.

{% sensitive "view-statement" %}
{% ciphertext folder.statement_objectives folder.principal %}
{end_sensitive %}

Figure 8-2: Example Django template code from gradapply that uses CryptFrame to decrypt a ciphertext with a given principal.
the CryptFrame client-side library to specify which principal to use for encrypting and decrypting input values. In the example in Figure 8-1, the input is encrypted with a principal with the attribute \{name: "statement"\}, as certified by the currently logged-in user. The encrypted class on the input element specifies that the value of the input field should be encrypted before sending it outside the SR.

One SR in the application performs dynamic access control: any reviewer can add another user as a reviewer by providing the new reviewer’s email address. On form submission, the SR looks up the new reviewer at the identity provider and then looks up the principal with the attribute \{group: "reviewers"\} as signed by the hard-coded department chair’s email address. The SR then calls grant_access to give the new reviewer access to the principal for the reviewers group.

8.1.2 Security analysis

With sensitive regions added into the application for viewing and entering sensitive data, as long as users enter sensitive data only in form fields that show the CryptFrame visual indicator, an attacker cannot read sensitive data unless he knows the secret key of the department chair, a reviewer, or a principal that has been added as a reviewer.

The “add reviewer” form is the most likely pitfall when using CryptFrame, because the attacker has the ability to manipulate the page outside the sensitive region or the template within the sensitive region in order to trick the user about the function of the form. For example, suppose that the sensitive region on this page included only the text input field where the user is intended to enter the email address of the reviewer to add, but no context about what the form does. Then the attacker could change the page outside the sensitive region to trick the user into believing that the text input field is for searching for or banning a particular user.
Similarly, templatized verification allows the attacker to make limited manipulations within the sensitive region without failing the signature check. For example, the sensitive region shown in Figure 8-3a is not secure because the attacker could substitute any text in the email_label variable, including “Enter the email address of a user to ban.” Therefore, forms that change access permissions should be entirely contained in sensitive regions with static context indicating their function, as shown in Figure 8-3b.

8.2 django-forum

django-forum [17] is a standard forum application written in Django. It organizes posts into threads, and each thread belongs to a forum. Users are organized into groups, and each forum has a list of groups associated with it. If forum \( F \) is associated with group \( G \), then users in \( G \) are allowed to access the posts assigned to threads in \( F \).

In order to demonstrate that CryptFrame integrates with existing techniques for computing on encrypted data, we modified django-forum to include a feature for searching within a thread, as we discuss in Chapter 8.2.3. This feature added 43 lines of code to the original application.

In this case study, we describe how we modified django-forum to encrypt forum posts. In Chapter 10, we compare the performance of this version of the application to the original, and to a version in which we encrypt forum and thread titles as well as posts.

8.2.1 Developer effort

The django-forum application allows users to create new threads and posts. It uses the Django admin to allow users to create forums, move threads, move posts, create groups, and assign groups to forums. We assigned a principal to each forum, thread, group, and user. Forum principals have access to principals for their threads, and posts are encrypted with the principals for their threads. When a new thread is created, the parent forum's principal creates a certificate for the name of the thread. The only subtlety lies in certifying the new thread's title using the parent forum's principal instead of the user who creates it. This is because the groups assigned to the forum determine who can see the data. If a user creates a new post, knowing the thread's forum as well as the thread title is important for deciding which thread to put the post in: there might be two threads with the same name, created by the same user, in both the “Public” and the “Secret” forums, so a certificate signed by the forum principal (not by the user) is required to differentiate these threads.

After deciding on the principal graph, we modified the django-forum application to encrypt/decrypt data and to add access to the correct principals when new threads or posts are created.

django-forum requires several key chains per page load (e.g., one per post in a thread). Computing key chains by querying the principal graph database is expensive, so the CryptFrame server-side API allows the application to cache all key chains starting at a particular principal, and to extend key chains that have already been cached. We used these in django-forum to preload a user’s key chains on login, and to extend the key chains of all currently logged-in users when a new post or thread is created.
8.2.2 Admin integration

django-forum relies heavily on the admin application, where users can create and edit forums, threads, and posts. We modified the application to use the CryptFrame admin, using the following annotations:

- On the forum model, we marked title as the human-meaningful name of the model, and creator as the certifier field. These annotations allow the admin to translate user actions directly into principal graph operations. For example, when a user creates a forum, the admin generates a certificate for the forum principal’s title, signed by the creator. We marked groups as a has_access field, indicating that any users in one of the forum’s groups can decrypt any data that the forum has access to.

- On the thread model, we marked title as the human-meaningful name. We marked forum as a has_access field and as a certifier field. The certifier annotation indicates that a thread’s forum is responsible for creating a certificate for the thread’s name.

- On the post model, we marked body as sensitive, which tells the admin to encrypt its contents. We marked thread as has_access. Since a post does not have a principal of its own, the admin encrypts a post’s sensitive fields with the principals of each of its has_access fields (in this case, the thread principal).

8.2.3 Searching on encrypted data

To demonstrate that CryptFrame can be integrated with techniques for computing on encrypted data, we modified django-forum to include a feature that allows users to search for keywords in posts within a thread. We used the technique described by Song et al. [23] as implemented in CryptDB [16]. This algorithm uses a key to encrypt a plaintext $P$, and can also use the key to compute a token for a keyword $W$. Anyone with a ciphertext for $P$ and the token for $W$ can determine whether the plaintext $P$ contains the keyword $W$, but cannot learn any other information about $P$ or $W$.

The client uses this algorithm to encrypt posts, using a thread principal’s key for all the posts in that thread. After the user enters a search keyword inside a sensitive region, the CryptFrame browser extension uses the thread key to compute a token. The token is sent to the server, where the application passes it to the CryptFrame server-side library to search ciphertexts for the keyword. The server learns which ciphertexts match the user’s query, but cannot learn which keyword the user is searching for. The ciphertexts are passed back to the client and displayed in a sensitive region.

CryptFrame can therefore be easily extended to support computation over encrypted data using existing techniques. One might also want to implement a keyword search over all the posts in the application. Using the Song et al. scheme, the client would have to compute tokens for every single thread key, which might be a prohibitive amount of client-side computation. If a future cryptosystem allows search over data encrypted with different keys, then such a system could be integrated as above.
8.2.4 Security analysis

The admin provides security for django-forum by encrypting sensitive data with keys obtained from appropriate certificate chains, extracted from the above annotations. As an example, we explain why the admin sensitive region for editing a thread cannot be used by an attacker to gain access to sensitive data. The SR includes context explaining what the form is for (“Change Thread”), as well as all the form input fields with static labels such as “Title.” The SR also prominently displays the email address of the thread owner, and the button to submit the form. Because the static context tells the user the function of the form and of each input field, an attacker cannot mislead the user about what an interaction with the form will do.

The form displays the thread’s title and creator, along with a dropdown menu for selecting a forum. When the sensitive region to edit an existing thread loads, the forum dropdown is prepopulated with the list of forum titles from the server, each listed with the email addresses of their owners. When a user moves a thread into a new forum and submits the form, the SR retrieves two principals with lookup_principal: one for the thread and one for the forum selected in the dropdown. To retrieve the forum principal, the SR takes the name and email address directly from the selected dropdown item, and it requests a principal with the selected title, as certified by the selected creator. To retrieve the thread principal, the SR takes the thread name from the input field and the owner from the place where it is displayed in the SR, again forming the certificate chain from these prominently displayed values. Since the user will submit the form only if the thread and forum are the ones he intended, and since the certificate chain attributes are populated directly from the fields that the user sees, the SR performs principal graph changes corresponding exactly to the user’s intended action. In other words, if the user intends to move thread X to forum Y, then the attacker cannot manipulate the outer page or the template within the sensitive region in such a way as to cause the user to make any changes to the principal graph other than giving the Y principal access to the X principal.

Since all django-forum access control changes happen in the CryptFrame admin, achieving security in the actual application is fairly simple. Sensitive regions with forms for creating a new thread or post include static context describing the form’s functionality, and they display the forum and thread name. Certificate chains are formed from these displayed names.

8.3 django-messages

django-messages\(^1\) allows users to send messages to each other, and we modified the application to encrypt the contents of these messages. Since the application has a fairly simple access control policy, most of the changes involved simply encrypting and decrypting data, using the CryptFrame ciphertext tag and the register_encrypted_form call to the client-side library. Each message has its own principal, and both the sender and recipient principals have access to the message principal.

\(^1\)https://github.com/arneb/django-messages
django-messages uses the Django admin application to allow administrators to view and edit messages, as well as assign a message to be readable by a group of users. We annotated the message model to be compatible with the CryptFrame admin; a message's sender, recipient, and group fields are all annotated as has_access fields, and its body is marked as sensitive.

### 8.4 Password manager

Our final case study application is a simple browser-based password manager\(^2\). This application consists of a model to represent passwords, and it uses the Django admin for all password viewing, editing, and sharing. Such applications are very easy to adapt to CryptFrame using our custom admin, requiring just a few annotations on the application's models. For the password model, we marked the name field as the model's human-meaningful name, the user field as the certifier, the group and user fields as has_access, and the password as sensitive.

\(^2\)https://github.com/ghickman/django-password
Chapter 9

Unambiguous user interfaces

In previous chapters, we discussed how to isolate sensitive data from a malicious server and allow dynamic access control changes. The remaining challenge in building secure web applications is for the developer to design sensitive region UIs that unambiguously extract the user’s intention from his interactions when granting access to sensitive data (such as discussed in Chapter 8.2.4). This task is application-specific, so we do not provide one solution that works for all applications. Instead, we discuss guidelines that we followed to design secure UIs for our case study applications; we believe that these principles should apply to other applications as well.

There are two main classes of attacks that developers should defend against: one in which the attacker can manipulate the UI to mislead the user about the function of the sensitive region, and one in which the attacker can manipulate the attributes of a certificate chain without the user noticing. Below we describe how to defend against each of these types of attacks, using examples drawn from our case study applications.

9.1 UI manipulation

The sensitive region could allow the attacker to mislead the user about the function of the SR, by manipulating either the outer page context or the dynamically generated parts of the sensitive region. The attack discussed in Chapter 8.1.2 is an example of this problem, where an attacker can cause the user to think that he is banning a user when he is in fact adding a reviewer who can access all data.

To avoid this mistake in our case study applications, for each form that grants access to sensitive data, we included a description of the form’s function and a label on each form input field inside the sensitive region. These labels are static in the sense that they are hardcoded strings (not template variables) and they appear in every possible rendering of the template (i.e. not inside conditionals).

The Django admin is an example of an application in which it is difficult to include this static context on sensitive region forms. The Django admin uses one template for editing objects of many different models, so one request might render the template with completely different form fields than another request. For example, editing a forum post in the Django admin uses different form fields than editing a thread, but the same template is used when
editing a post or a thread. This motivates the preprocessing step in the CryptFrame admin, in which each template is preprocessed into several templates, one for each type of data in the application: for django-forum, the CryptFrame admin preprocesses the template for editing a generic model into one template for editing a forum, one template for editing a thread, and one template for editing a post. Each of these outputted templates has the static form context that is necessary to ensure the template’s UI is unambiguous.

9.2 Certificate chain manipulation

When granting access to sensitive data, the sensitive region can use certificate chains to verify the authenticity of keys retrieved from the untrusted server. But if the sensitive region populates this certificate chain with values provided by the attacker and not approved by the user, then the certificate chain provides no security.

To avoid this mistake in our case study applications, we always constructed certificate chains either from hardcoded values that are known at install time (like the department chair’s email address in gradapply) or from values displayed to the user during their interaction with the SR. Consider the example given in Chapter 8.2.4 of moving a thread to a different forum. Suppose that the user wants to move thread X to forum Y. When the user submits the form, the sensitive region reads the displayed thread title off the page and looks up a principal with a certificate for that title. If the title were not displayed on the page, then the attacker controlling the server could replace it with any thread, and the user would have no way of knowing that the thread being moved to forum Y is not thread X but in fact a different thread. This example shows that any dynamic attributes in a certificate chain must be displayed to the user in such a way that the user would not perform the action if the attributes were changed by an attacker.
Chapter 10

Performance evaluation

Although the primary focus of CryptFrame is security, this chapter quantifies CryptFrame’s performance overheads. We measure the throughput and latency of CryptFrame for django-forum, because it had the most complex access control policy of our case study applications. We ran django-forum on an Intel Xeon server (using only one core for consistency) running Linux 2.6.32, Django 1.3, Apache 2.2.16, and MySQL 5.1.49. We preloaded the forum with 50 users organized into five groups, and 250 posts organized into 25 threads split among five forums. To measure latency, we ran Chrome 26.0.1410.43 an Intel Q9400 server with 4GB RAM with the CryptFrame browser extension installed, and we measured the time from the beginning of the request until the page was fully rendered, averaged over 50 runs. For throughput, we ran django-forum and the CryptFrame principal graph application on the same machine, and we ran 10 parallel clients on a 16-core AMD Opteron 8350 system (so clients were not a bottleneck) for two minutes, simulating users browsing the forum and creating threads and posts.

For both latency and throughput, we compared the original django-forum against two versions of the application modified to use CryptFrame: one version that encrypts just the contents of forum posts, and another that encrypts forum posts, thread titles, and forum titles.

10.1 Throughput

Using CryptFrame to encrypt forum posts, thread titles, and forum titles in django-forum results in a 31% decrease in overall throughput on our workload, compared to running unmodified django-forum. When CryptFrame is used to encrypt just the forum posts, the decrease in overall throughput is 25%. To understand the sources of this decrease, we measured throughput for each of the individual requests, shown in Figure 10-1.

The clusters labelled “List forums,” “List threads,” and “Read thread” show throughput for requests where the clients only read data but do not write anything. The throughput loss in this type of request is primarily due to extra Django template variables, which take about 1 msec to resolve a template variable to a value. CryptFrame requires two additional template variable resolutions for each ciphertext template variable: the principal used for decryption, and a key chain from the currently logged-in user principal to the
decryption principal. “Read thread” requests incur more overhead due to more ciphertexts. Similarly, when posts and titles are encrypted instead of just posts, each page load contains more ciphertext template variables, leading to a higher loss in throughput compared to the application that just encrypts posts.

The clusters labelled “Create thread” and “Create post” show throughput for requests that insert new data into the database. In the original application, these requests are at most two database inserts, but with CryptFrame, the server has to process up to five extra requests to the CryptFrame principal graph application: looking up the principal for the parent thread or forum, looking up a key chain from the currently logged-in user to that principal, storing the principal created for the new object, storing a certificate for the principal, and storing the wrapped key after calling `grant_access`. The latter three requests do not occur when a new post is created, since the post is encrypted with the thread’s principal and a new principal is not created. Similarly, the “Search” cluster shows a large decrease in throughput because the server has to look up the principal for the thread, look up a key chain for the currently logged-in user to that principal, and make a foreign function call to the C++ implementation of the encrypted search described in Chapter 8.2.3.

10.2 Latency

Figure 10-2 shows client-side latency imposed by CryptFrame for different operations in django-forum. The leftmost three clusters show the latency for loading three pages in django-forum: the home page (“List forums”), a page listing the threads in a forum (“List threads”), and a page showing the posts in a thread (“Read thread”). The rightmost three clusters show the time taken from submitting a form (either to create a thread, create a post, or search a thread’s posts) to receiving a response from the server. CryptFrame adds about 200 msec latency in the worst case. For the leftmost three clusters, CryptFrame adds no extra round-trips to the server. The rightmost three bars do add round trips to fetch and store principals and key chains, but we believe that the CryptFrame API could be modified to allow developers to avoid some of these round trips. For example, the “Create thread” form could include the parent forum’s principal and key chain in the template at render-time, instead of the client fetching this information when the form is submitted.
Figure 10-2: Latency for different operations in django-forum.

Figure 10-3: Sources of latency imposed by a sensitive region. The baseline is a page that loads the posts in a thread, using a single sensitive region. Each bar shows the decrease in client-side latency that results from removing a single step that CryptFrame performs when loading the sensitive region.

To understand where CryptFrame adds latency, we used a page that loads the posts in a thread in a single sensitive region, and we measured how latency decreases when individual operations are removed from the process of loading the sensitive region, as shown in Figure 10-3. The sources of latency shown in the graph are, from left to right,

- Verifying the developer’s signature on the sensitive region’s contents (“No sig. verification on template”).
- Checking that the sensitive region’s contents conform to the developer’s signed template (“No templ. verification”).
- Sanitizing dynamic data in the sensitive region to ensure that an attacker cannot, for example, insert a malicious script as the contents of a template variable (“No sanitization”).
- Decrypting the contents of the forum posts (“No decryption”).
- Unwrapping the key chain of the thread principal, to obtain a secret key with which to decrypt the forum posts ("No keychain unwrap").

- Resolving template variables that look up keychains from the currently logged-in user to the thread principal ("Static keychains").

- Resolving template variables that look up the thread principal ("Static principals").

All but the last two latency sources occur on the client. At least half the overhead from each cryptographic operation comes from the XMLHttpRequest to the local server doing cryptography in C++, so a native-speed Javascript cryptography API would decrease the client-side latency added by CryptFrame.
Chapter 11

Conclusion

This thesis showed how to protect confidential data in web applications in the face of arbitrary server compromises, by leveraging client-side encryption. We identified three key challenges to making client-side encryption secure: verifying code, verifying dynamic data, and designing unambiguous user interfaces. We further presented CryptFrame, a set of tools that helps developers build such applications by providing three mechanisms: sensitive regions, templatized verification, and the principal graph. We used CryptFrame to modify four existing Django applications to protect user data from the server, showing that with minimal effort, developers can use CryptFrame to achieve strong security guarantees.

CryptFrame works best with applications that store and serve data for users without computing on it at the server, though CryptFrame can be integrated with cryptosystems that allow computation on encrypted data to achieve these benefits of these systems in the web application setting. One direction for future work is to integrate CryptFrame with other cryptosystems to allow more types of computation on encrypted data at the server. Further, developing cryptosystems that allow computation over data encrypted with multiple keys would be useful for CryptFrame, since a user’s data is often encrypted with different keys to support sharing with other users.

Another direction for future work is applying the ideas presented in this thesis to desktop or mobile applications. The code delivery model in these settings simplifies the problem and removes the need for sensitive regions and templatized verification, but ideas from CryptFrame’s principal graph can still be useful when an untrusted server delivers keys to the client.

We hope that CryptFrame enables web developers to use client-side encryption effectively, by building web applications that offer both functionality and confidentiality against a compromised web server.
Bibliography


