Building web applications on top of encrypted data using Mylar

Raluca Ada Popa, Emily Stark,† Jonas Helfer, Steven Valdez,
Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan
MIT CSAIL and †Meteor Development Group

Crypto ePrint Archive, updated version, 2016-08-29

ABSTRACT

Web applications rely on servers to store and process confidential information. However, anyone who gains access to the server (e.g., an attacker, a curious administrator, or a government) can obtain all of the data stored there. This paper presents Mylar, a platform that provides end-to-end encryption to web applications. Mylar protects the confidentiality of sensitive data fields against attackers that gained access to servers. Mylar stores sensitive data encrypted on the server, and decrypts that data only in users' browsers. Mylar addresses three challenges in making this approach work. First, Mylar allows the server to perform keyword search over encrypted documents, even if the documents are encrypted with *different* keys. Second, Mylar allows users to share keys securely in the presence of an active adversary. Finally, Mylar ensures that client-side application code is authentic, even if the server is malicious. Results with a prototype of Mylar built on top of the Meteor framework are promising: porting 6 applications required changing just 36 lines of code on average, and the performance overheads are modest, amounting to a 17% throughput loss and a 50 ms latency increase for sending a message in a chat application.

1 INTRODUCTION

Using a web application for confidential data requires the user to trust the server to protect the data from unauthorized disclosures. This trust is often misplaced, however, because there are many ways in which confidential data could leak from a server. For example, attackers could exploit a vulnerability in the server software to break in [48], a curious administrator could peek at the data on the server [9, 11], or the server operator may be compelled to disclose data by law [22]. Is it possible to build web applications that protect data confidentiality against attackers with *full access* to data on the servers?

A promising approach is to give each user their own encryption key, encrypt a user's data with that user's key in the web browser, and store only encrypted data on the server. This model ensures that an adversary would not be able to read any confidential information on the server, because they would lack the necessary decryption keys. In fact, this model has been already adopted by some privacy-conscious web applications [32, 46].

Unfortunately, this approach suffers from three significant security, functionality, and efficiency shortcomings. First, a compromised server could provide malicious client-side code to the browser and extract the user's key and data. Ensuring that the server did not tamper with the application code is difficult because a web application consists of many files, such as HTML pages, Javascript code, and CSS style sheets, and the HTML pages are often dynamically generated.

Second, this approach does not provide data sharing between users, a crucial function of web applications. To address this problem, one might consider encrypting shared documents with separate keys, and distributing each key to all users sharing a document via the server. However, distributing keys via the server is challenging because a compromised server can supply arbitrary keys to users, and thus trick a user into using incorrect keys.

Third, this approach requires that all of the application logic runs in a user's web browser because it can decrypt the user's encrypted data. But this is often impractical: for instance, doing a keyword search would require downloading all the documents to the browser.

This paper presents Mylar, a new platform for building web applications that stores sensitive data encrypted on the server. Mylar makes it practical for many classes of applications to protect confidential data in this way. It leverages the recent shift in web application frameworks towards implementing logic in client-side Javascript code, and sending data, rather than HTML, over the network [33]; such a framework provides a clean foundation for security. Mylar addresses the challenges mentioned above with a combination of systems techniques and novel cryptographic primitives, as follows.

Data sharing. To enable sharing, each sensitive data item is encrypted with a key available to users who share the item. To prevent the server from cheating during key distribution, Mylar provides a mechanism for establishing the correctness of keys obtained from the server: Mylar forms certificate paths to attest to public keys, and allows the application to specify what certificate paths can be trusted in each use context. In combination with

a user interface that displays the appropriate certificate components to the user, this technique ensures that even a compromised server cannot trick the application into using the wrong key.

Computing over encrypted data. Keyword search is a common operation in web applications, but it is often impractical to run on the client because it would require downloading large amounts of data to the user's machine. While there exist practical cryptographic schemes for keyword search, they require that data be encrypted with a single key. This restriction makes it difficult to apply these schemes to web applications that have many users and hence have data encrypted with many different keys.

Mylar provides the first cryptographic scheme that can perform keyword search efficiently over data encrypted with *different* keys. The client provides an encrypted word to the server and the server can return all documents that contain this word, while keeping the word and the contents of the documents encrypted.

Verifying application code. With Mylar, code running in a web browser has access to the user's decrypted data and keys, but the code itself comes from the untrusted server. To ensure that this code has not been tampered with, Mylar checks that the code is properly signed by the web site owner. This checking is possible because application code and data are separate in Mylar, so the code is static. Mylar uses two origins to simplify code verification for a web application. The primary origin hosts only the top-level HTML page of the application, whose signature is verified using a public key found in the server's X.509 certificate. All other files come from a secondary origin, so that if they are loaded as a top-level page, they do not have access to the primary origin. Mylar verifies the hash of these files against an expected hash contained in the top-level page.

To evaluate Mylar's design, we built a prototype on top of the Meteor web application framework [33]. We ported 6 applications to protect various confidential data fields using Mylar: a medical application for endometriosis patients, a web site for managing homework and grades, a chat application called kChat, a forum, a calendar, and a photo sharing application. The endometriosis application is used to collect data from patients with that medical condition, and was designed under the aegis of the MIT Center for Gynepathology Research by surgeons at the Newton-Wellesley hospital (affiliated with the Harvard Medical School) in collaboration with biological engineers at MIT.

Our results show that Mylar requires little developer effort: we had to modify an average of just 36 lines of code per application. We also evaluated the performance of Mylar on three of the applications above. For example, for kChat, our results show that Mylar incurs modest overheads: a 17% throughput reduction and a 50 msec latency increase for the most common operation (sending a message). These results suggest that Mylar is a good fit for multi-user web applications with data sharing.

2 RELATED WORK

Mylar is the first system to provide significant confidentiality guarantees for a wide range of web applications against attackers that gained access to server data. In the rest of this section, we relate Mylar to prior work on securing web applications, building systems using untrusted servers, and computing over encrypted data.

2.1 Web application security

Much of the work on web application security focuses on preventing security vulnerabilities caused by bugs in the application's source code, either by statically checking that the code follows a security policy [12, 50], or by catching policy violations at runtime [19, 28, 52]. In contrast, Mylar assumes that the attacker already succeeded in stealing the server data, either as a result of software vulnerabilities or because the server operator is untrustworthy, and protects data confidentiality in this setting.

On the browser side, prior work has explored techniques to mitigate vulnerabilities in Javascript code that allow an adversary to leak data or otherwise compromise the application [1, 17, 51]. Mylar assumes that the developer does not inadvertently leak data from client-side code, but in principle could be extended to use these techniques for dealing with buggy client-side code.

There has been some work on using encryption to protect confidential data in web applications, as we describe next. Unlike Mylar, none of them can support a wide range of complex web applications, nor compute over encrypted data at the server, nor address the problem of securely managing access to shared data.

A position paper by Christodorescu [13] proposes encrypting and decrypting data in a web browser before sending it to an untrusted server, but lacks any details of how to build a practical system.

Several data sharing sites encrypt data in the browser before uploading it to the server, and decrypt it in the browser when a user wants to download the data [15, 32, 40]. The key is either stored in the URL's hash fragment [32, 40], or typed in by the user [15], and both the key and data are accessible to any Javascript code from the page. As a result, an active adversary could serve Javascript code to a client that leaks the key. In contrast, Mylar's browser extension verifies that the client-side code has not been tampered with.

Several systems transparently encrypt and decrypt data sent to a server [7, 14, 38, 39]. These suffer from the same problems as above: they cannot handle any active attacks,

and cannot compute over encrypted data at the server without revealing a significant amount of information.

Cryptocat [46], an encrypted chat application, distributes the application code as a browser extension rather than a web application, in order to deal with active attacks [45]. Mylar's browser extension is general-purpose: it allows verifying the code of web applications without requiring users to install a *separate* extension for each application. Cryptocat could also benefit from Mylar's search scheme to perform keyword search over encrypted data at the server.

2.2 Untrusted servers

SUNDR [29] protects file system integrity, providing fork consistency in the face of a malicious server. SPORC [16] and Depot [31] extend SUNDR's design to build applications on top of an encrypted server. For example, SPORC provides conflict resolution using operational transforms, and consistently handles access control changes. These systems do not allow an application to perform server-side computation, such as Mylar's server-side keyword search. Furthermore, with SPORC, the application logic is hard-coded into the client, whereas with Mylar, the application logic is determined at runtime, based on the URL that the user visits.

CryptDB [36] protects confidential data in a SQL database server by running SQL queries over encrypted data. However, in a typical database-backed web application, the application server gets access to unencrypted data, and receives each user's key when the user logs in. Consequently, while CryptDB protects against attacks on the database server, it provides no guarantees for users logged in during an attack on the application server. For example, if an administrator with access to all data is logged in when the application server is compromised, then the attacker can compromise all data. Finally, CryptDB cannot compute over data encrypted with different keys as in Mylar's multi-key keyword search. On the other hand, CryptDB allows computing more functions over encrypted data than Mylar.

2.3 Computation over encrypted data

Theoretical results on fully homomorphic encryption and functional encryption have shown that it is possible for an untrusted server to compute arbitrary functions over encrypted data [18, 21, 30]. However, such general-purpose schemes are too slow to be practical.

Many schemes for performing keyword search over encrypted data have been proposed [24, 41]. All of these schemes for keyword search have assumed that the data is encrypted with a single key; Mylar provides the first practical scheme for performing keyword search over data encrypted with different keys. Proxy re-encryption [3] allows switching the key under which some data is encrypted in the context of public-key encryption, but this does not provide an efficient search scheme.

Searchable encryption schemes, including Mylar's, inherently leak some information. Cash et al categorize the various leakage profiles of seachable encryption schemes [10]. We discuss the guarantees that Mylar provides in more detail in §3.4 and in a longer technical report [35]. Grubbs et al [23] analyze the security of Mylar and consider three attack scenarios; two of these attacks fall outside of Mylar's threat model (as Grubbs et al acknowledge) and the third was already considered by Mylar's design and prevented using the **allow_search** API call, as we describe in §5.4.

2.4 Trusted hardware

An alternative approach for computing over encrypted data is to rely on trusted hardware [2, 4, 25]. Such approaches are complementary to Mylar, and could be used to extend the kinds of computations that Mylar can perform over encrypted data at the server, as long as the application developer and the users believe that trusted hardware is trustworthy.

3 MYLAR ARCHITECTURE

There are three different parties in Mylar: the users, the web site owner, and the server operator. Mylar's goal is to help the site owner protect the confidential data of users in the face of a malicious or compromised server operator.

3.1 System overview

Mylar embraces the trend towards client-side web applications; Mylar's design is suitable for platforms that:

1. Enable client-side computation on data received from the server.
2. Allow the client to intercept data going to the server and data coming from the server.
3. Separate application code from data, so that the HTML pages supplied by the server are static.

AJAX web applications with a unified interface for sending data over the network, such as Meteor [33], fit this model. Such frameworks provide a clean foundation for security, because they send data separately from the HTML page that presents the data. In contrast, traditional server-side frameworks incorporate dynamic data into the application's HTML page in arbitrary ways, making it difficult to encrypt and decrypt the dynamic data on each page while checking that the fixed parts of the page have not been tampered with [42].

3.1.1 Mylar's components

The architecture of Mylar is shown in Figure 1. Mylar consists of the four following components:

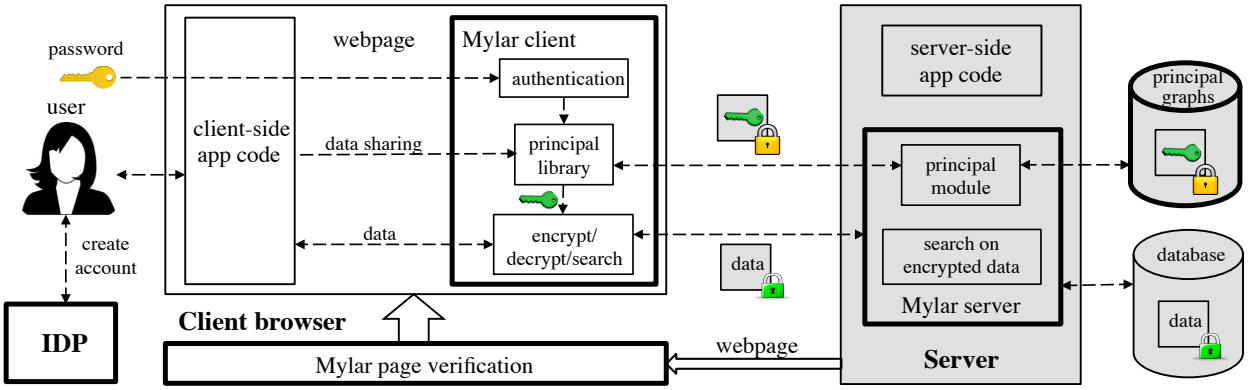

Figure 1: System overview. Shaded components have access only to encrypted data. Thick borders indicate components introduced by Mylar.

Browser extension. It is responsible for verifying that the client-side code of a web application that is loaded from the server has not been tampered with.

Client-side library. It intercepts data sent to and from the server, and encrypts or decrypts that data. Each user has a private-public key pair. The client-side library stores the private key of the user at the server, encrypted with the user's password.[1] When the user logs in, the client-side library fetches and decrypts the user's private key. For shared data, Mylar's client creates separate keys that are also stored at the server in encrypted form.

Server-side library. It performs keyword search over encrypted data at the server.

Identity provider (IDP). For some applications, Mylar needs a trusted identity provider service (IDP) to verify that a given public key belongs to a particular username. An application needs the IDP if the application has no trusted way of verifying the users who create accounts, and the application allows users to choose whom to share data with. For example, if Alice wants to share a sensitive document with Bob, Mylar's client needs the public key of Bob to encrypt the document. A compromised server could provide the public key of an attacker, so Mylar needs a way to verify the public key. The IDP helps Mylar perform this verification by signing the user's public key and username. An application does not need the IDP if the site owner wants to protect against only passive attacks (§3.4), or if the application has a limited sharing pattern for which it can use a static root of trust (see §4.2).

An IDP can be shared by many applications, similar to an OpenID provider [34]. The IDP does not store per-application state, and Mylar contacts the IDP only when a user first creates an account in an application; afterwards, the application server stores the certificate from the IDP.

[1]The private key can also be stored at a trusted third-party server, to better protect it from offline password guessing attacks and to recover from forgotten passwords without re-generating keys.

3.2 Mylar for developers

The developer starts with a regular (non-encrypted) web application implemented in Mylar's underlying web platform (Meteor in our prototype). To secure this application with Mylar, a developer uses Mylar's API (Figure 2), as we explain in the rest of this paper. First, the developer uses Mylar's authentication library for user login and account creation. If the application allows a user to choose what other users to share data with, the developer should also specify the URL and public key of a trusted IDP.

Second, the developer specifies which data in the application should be encrypted, and who should have access to it. Mylar uses principals for access control; a principal corresponds to a public/private key pair, and represents an application-level access control entity, such as a user, a group, or a shared document. In our prototype, all data is stored in MongoDB collections, and the developer annotates each collection with the set of fields that contain confidential data and the name of the principal that should have access to that data (i.e., whose key should be used).

Third, the developer specifies which principals in the application have access to which other principals or to search queries. For example, if Alice wants to invite Bob to a confidential chat, the application must invoke the Mylar client to grant Bob's principal access to the chat room principal.

Fourth, the developer changes their server-side code to invoke the Mylar server-side library when performing keyword search. Our prototype's client-side library provides functions for common operations such as keyword search over a specific field in a MongoDB collection.

Finally, as part of installing the web application, the site owner generates a public/private key pair, and signs the application's files with the private key using Mylar's bundling tool. The web application must be hosted using `https`, and the site owner's public key must be stored in the web server's X.509 certificate. This ensures that even if the server is compromised, Mylar's browser extension

Function	Semantics
idp_config(*url*, *pubkey*)	Declares the *url* and *pubkey* of the IDP and returns the principal corresponding to the IDP.
create_user(*uname*, *password*, *auth_princ*)	Creates an account for user *uname* which is certified by principal *auth_princ*.
login(*uname*, *password*)	Logs in user *uname*.
logout()	Logs out the currently logged-in user.
collection.**encrypted**({*field*: *princ_field*}, ...)	Specify that *field* in *collection* should be encrypted for the principal in *princ_field*.
collection.**auth_set**([*princ_field*, *fields*], ...)	Authenticate the set of *fields* with principal in *princ_field*.
collection.**searchable**(*field*)	Mark *field* in *collection* as searchable.
collection.**search**(*word*, *field*, *princ*, *filter*, *proj*)	Search for *word* in *field* of *collection*, filter results by *filter* and project only the fields in *proj* from the results. Use *princ*'s key to generate the search token.
princ_create(*name*, *creator_princ*)	Create principal named *name*, sign the principal with *creator_princ*, and give *creator_princ* access to it.
princ_create_static(*name*, *password*)	Create a static principal called *name*, hardcode it in the application, and wrap its secret keys with *password*.
princ_static(*name*, *password*)	Return the static principal *name*; if a correct password is specified, also load the secret keys for this principal.
princ_current()	Return the principal of currently logged in user.
princ_lookup($name_1$, ..., $name_k$, *root*)	Look up principal named $name_1$ as certified by a chain of principals named $name_i$ rooted in *root* (e.g., the IDP).
granter.**add_access**(*grantee*)	Give the *grantee* principal access to the *granter* principal.
grantee.**allow_search**(*granter*)	Allow matching keywords from *grantee* on *granter*'s data.

Figure 2: Mylar API for application developers split in three sections: authentication, encryption/integrity annotations, and access control. All of the functions except `princ_create_static` and `searchable` run in the client browser. This API assumes a MongoDB storage model where data is organized as collections of documents, and each document consists of fieldname-and-value pairs. Mylar also preserves the generic functionality for unencrypted data of the underlying web framework.

will know the site owner's public key, and will refuse to load client-side code if it has been tampered with.

3.3 Mylar for users

To obtain the full security guarantees of Mylar, a user must install the Mylar browser extension, which detects tampered code. However, if a site owner wants to protect against only passive attacks (§3.4), users don't have to install the extension and their browsing experience is entirely unchanged.

3.4 Security guarantees

Threats. The server and some user machines could be compromised. Some users may collude with the server, either because the adversary is a user of the application, or because the adversary broke into a user's machine.

Mylar considers two types of attackers, passive and active, and provides different security guarantees for each.

- A *passive* (or honest-but-curious) attacker eavesdrops on all the data on the server, but does not actively insert or modify any data or query results. In particular, the server responds to all client requests as if it were not compromised.

- An *active* (or malicious) attacker occurs when the application and the database servers can be *fully* controlled by an adversary: the adversary may obtain all data from the server, cause the server to send arbitrary responses to web browsers, and so on. The attacker could also compromise some users. This model subsumes a wide range of real-world security problems, from bugs in server software to insider attacks, as well as the passive attacker.

Assumptions. Mylar assumes that the web application as written by the developer will use the Mylar API correctly, will not send user data or keys to untrustworthy recipients, and cannot be tricked into doing so by exploiting bugs (e.g., cross-site scripting). Our prototype of Mylar is built on top of Meteor, a framework that helps programmers avoid many common classes of bugs in practice. Mylar also assumes that Mylar's code verification module has the correct public key for the developer of the web application.

Mylar also assumes that the IDP correctly verifies each user's identity (e.g., email address) when signing certificates. To simplify the job of building a trustworthy IDP, Mylar does not store any application state at the IDP, contacts the IDP only when a user first registers, and allows the IDP to be shared across applications.

Finally, Mylar assumes that the user checks the web browser's security indicator (e.g., the `https` shield icon) and the URL of the web application they are using, before entering any sensitive data. This assumption is identical to what users must already do to safely interact with a *trusted* server. If the user falls for a phishing attack, neither Mylar nor a trusted server can prevent the user from entering confidential data into the adversary's web application.

Security guarantees. Mylar provides different guarantees against the two attackers above.

Passive attacker guarantees. For data fields that the developer marks **encrypted**, Mylar protects the contents of those fields using standard encryption. In other words, for these fields, Mylar provides the guarantees of end-to-end encryption. To achieve this, Mylar assumes that none of the users with legitimate access to these data fields used a compromised machine or purposefully leaked the data or keys.

Mylar does not protect any other data. This includes data not marked for encryption by the developer and metadata, such as which data can be accessed by a given principal. Mylar does not hide data access patterns, or communication and timing patterns in the application. ORAM-based techniques [20, 44] could be used to hide access patterns, at an additional performance cost.

For fields marked as **searchable**, the security guarantees are weaker. Mylar encrypts both the search queries and the words in the searchable fields. The words are encrypted with IND-CPA security (randomized encryption), which leaks only the length of the word. The search queries are encrypted with deterministic encryption. As is typical with seachable encryption, there is "side" leakage caused by the number of words per data field, by the search queries, and by the positions in which the queries match. Cash et al. [10] categorize typical leakage profiles in searchable encryption schemes; our scheme fits between their L2 and L3 profiles, reaching either depending on the situation. We formalize the security of Mylar's search in [35], by giving a precise security definition and proving that Mylar's scheme achieves it.

Mylar's search scheme assumes that the application (possibly with the user's help) allows search to happen only over trustworthy principals; that is, users with access to those principals (and their machines) are not compromised. This falls under Mylar's assumption that the application developer will correctly use the Mylar API.

Active attacker guarantees. Against the active attacker, Mylar provides two guarantees. First, the client software cannot be tampered with by an active attacker at the server. This ensures that, even in the presence of an arbitrary corruption of the server, the Mylar client will run untampered code. Second, when a client looks up a key for a principal, either for granting access to some data or for allowing search, Mylar ensures that the server provides the correct key.

Mylar does not aim to prevent other active attacks. In particular, Mylar does not prevent integrity attacks (such as tampering) with respect to any data, queries, or any part of search. The marked fields remain encrypted, but in some cases, such attacks may indirectly lead to some information leakage. A follow-up system to Mylar, Verena [26], was designed to protect against many of these attacks and was built with Mylar in mind.

3.5 Security overview

At a high level, Mylar achieves its goal as follows. First, it verifies the application code running in the browser (§6), so that it is safe to give client-side code access to keys and plaintext data. Then, the client code encrypts the data marked sensitive before sending it to the server. Since users need to share data, Mylar provides a mechanism to securely share and look up keys among users (§4). Finally, to perform server-side processing, Mylar introduces a new cryptographic scheme that can perform keyword search over documents encrypted with many different keys, without revealing the content of the encrypted documents or the word being searched for (§5).

4 Sharing data between users

Many web applications share data between users according to some policy. A simple example is a chat application, where messages are shared between the sender and the recipients. In Mylar's threat model, an application cannot trust the server to enforce the sharing policy, because the server is assumed to be compromised. As a result, the application must encrypt shared data using a key that will be accessible to just the right set of users.

Mylar allows an application to specify its security policy in terms of application-defined principals. In particular, each principal has an application-chosen *name*, a *public key* used to encrypt data for that principal, and a *private key* used to decrypt that principal's data.

In addition to allowing the application to create principals, and to use the principals' keys to encrypt and decrypt data, Mylar provides two critical operations to the application for managing principals:

- Find a principal so that the application can use the corresponding private key to decrypt data. The goal is to ensure that only authorized users can get access to the appropriate private key.

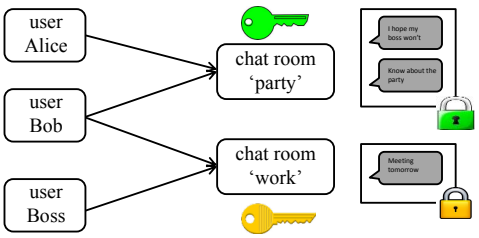

Figure 3: Example access graph for a chat application. Rounded rectangles represent principals, and arrows represent access relationships. Alice and Bob share the chat room "party" so they both have access to the principal for this room. Messages in each chat room are encrypted with the key of the room's principal.

- Find a principal so that the application can use the corresponding public key to encrypt or share data with other users. The goal is to ensure that a malicious server cannot trick Mylar into returning the wrong public key, which could lead the application to share confidential data with the adversary.

Mylar cryptographically enforces the above goals by forming two graphs on top of principals: an *access graph*, which uses key chains to distribute the private keys of shared principals to users, and a *certification graph*, which uses certificate chains to attest to the mapping between a principal name and its public key.

4.1 Access graph

To ensure that only authorized users can access the private key of a principal, Mylar requires the application to express its access control policy in terms of *access* relationships between principals. Namely, if principal A can access principal B's private key, then we say A *has access to* B. The *has access to* relation is transitive: if B in turn has access to C, then A can access C's private key as well. To express the application's policy in the access graph, the application must create appropriate *has access to* relationships between principals. The application can also create intermediate principals to represent, say, groups of users that all should have access to the same private keys.

As an example, consider a chat application where messages in each chat room should be available only to that room's participants. Figure 3 shows the access graph for this scenario. Both Alice and Bob have access to the key encrypting the "party" room, but the boss does not.

Key chaining. To enforce the access graph cryptographically, Mylar uses key chaining, as in CryptDB [36]. When an application asks to add a new *has access to* edge from principal A to principal B, Mylar creates a *wrapped key*: an encryption of B's private keys under the public key of principal A. This ensures that a user with access to A's private key can decrypt the wrapped key and obtain B's private key. For example, in Figure 3, the private key of the "party" chat room is encrypted under the public key of Alice, and separately under the public key of Bob as well. The server stores these wrapped keys, which is safe since the keys are encrypted.

In practice, *has access to* relationships are rooted in user principals, so that a user can gain access to all of their data when they initially log in and have just the private key of their own user principal. When Mylar needs to decrypt a particular data item, it first looks up that data item's principal, as specified by the **encrypted** annotation (Figure 2). Mylar then searches for a chain of wrapped keys, starting from the principal of the currently logged in user, and leading to the data item's principal.

4.2 Certification graph

Mylar applications must look up public keys of principals when sharing data, for two broad purposes: either to encrypt data with that key, or to give some principal access to that key. In both cases, if a compromised server tricks the client application into using the public key of the adversary, the adversary will gain access to confidential data. For example, in the chat example, suppose Bob wants to send a confidential message to the "work" chat room. If the server supplies the adversary's public key for the chat room principal and the application client uses it, the adversary will be able to decrypt the message. Preventing such attacks is difficult because all of the wrapped keys are stored at the server, and the server may be malicious.

To prevent such attacks, Mylar relies on a certification graph, which allows one principal to vouch for the name and the public key of another principal. The nodes of this graph are principals from the access graph together with some *authority* principals, which are principals providing the root of trust (described in §4.3). Applications create certificate chains for principals, rooted in an authority principal. For instance, in the chat example, the application can sign the "chatroom:work" principal with the key of the "user:boss" principal that created the chat room. Using the certification graph, applications can look up the public key of a principal by specifying the name of the principal they are looking for, along with a chain of certifications they expect to find.

Since the server is not trusted, there is no single authority to decide on the public key for a given principal name: in our chat example, both the real boss and a malicious server may have created chat rooms named "work." To prevent such naming ambiguity, one approach is to display the names in a certification chain to the user, similar to how web browsers display the hostname from an X.509 certificate for `https` web sites. As we describe later in §8, if the chat application displays the email address of the chat room creator (who signed the chat room principal), in addition to the name of the chat room, the user could distinguish a correct "work" chat room, created by the boss, from an impostor created by an attacker.

This requires Mylar applications to unambiguously map human-meaningful names, such as the "work" chat room and the identity of the Boss user, onto principal names, such as "chatroom:work" and "user:boss."

Mylar's certificate chains are similar to X.509; the difference is that X.509 typically has fixed roots of trust and fixed rules for what certificate chains are allowed, whereas Mylar allows the application to specify different roots of trust and acceptable chains for each lookup.

4.3 Principals providing the root of trust

The authority principals can be either the IDP or *static principals*. Static principals are access control entities fixed in the application's logic. For example, the endometriosis medical application has a group called "surgeons" representing the surgeons that have access to all patient data. Similarly, the homework submission application has a group called "staff" representing staff members with access to all student homework submissions and grades. In these applications, static principals can altogether remove the need for an IDP.

A developer can create a static principal by running **princ_create_static**(*name*, *password*) with the help of a command-line tool. This generates fresh keys for a principal, and encrypts the secret keys with *password*, so they can be retrieved only by providing *password* to **princ_static**. The resulting public key and encrypted secret key are hardcoded into the application's source code. This allows the application to refer to the static principal by name without relying on the IDP.

Static principals can also certify other principals. For example, in the endometriosis application, all user accounts are manually created by surgeons. This allows all user principals to be certified by the static "surgeons" principal, avoiding the need for an IDP to do the same.

4.4 User principals

To create an account for a new user, the application must invoke **create_user**, as shown in Figure 2. This causes the Mylar client to generate a new principal for the user, encrypt the secret key with the user's password, and store the principal with the encrypted secret key on the server.

To enable the application to later look up this user's public key, in the presence of active adversaries, the principal must be certified. To do this, the application supplies the *auth_princ* argument to **create_user**. This is typically either a static principal or the IDP. For static principals, the certificate is generated directly in the browser that calls **create_user**; the creator must have access to the private key of *auth_princ*. For example, the endometriosis application, where all users are manually created by a surgeon, follows this model. If *auth_princ* is the IDP, the Mylar client interprets *uname* as the user's email address, and contacts the IDP, which verifies the user's email address and signs a certificate containing the user's public key and email address.

Even though multiple applications can share the IDP, a buggy or malicious application will not affect other applications that use the same IDP (unless users share passwords across applications). This property is ensured by never sending passwords or secret keys to the IDP, and explicitly including the application's origin in the certificate generated by the IDP.

4.5 Data integrity

To prevent an attacker from tampering with the data, Mylar provides two ways to authenticate data, as follows.

First, all encrypted data is authenticated with a MAC (message authentication code),[2] which means that clients will detect any tampering with the ciphertext. However, an adversary can still replace the ciphertext of one field in a document with any other ciphertext that was encrypted using the same key.

To protect against such attacks, developers can specify an *authentication set* of fields whose values must be consistent with one other, using the **auth_set** annotation. This annotation guarantees that if a client receives some document, then all fields in each authentication set were consistent at some point, according to the corresponding principal. Mylar enforces authentication sets by computing a MAC over the values of all fields in each set.

For example, in a chat room application, each message has several fields, including the message body and the (client-generated) timestamp. By putting these two fields into an authentication set, the developer ensures that an adversary cannot splice together the body of one message with the timestamp from another message.

Mylar does not guarantee data freshness, or correctness of query results. An adversary can roll back the entire authentication set to an earlier version without detection, but cannot roll back a *subset* of an authentication set.

5 COMPUTING ON ENCRYPTED DATA

The challenge facing Mylar in computing over encrypted data is that web applications often have many users, resulting in data encrypted with many different keys. Existing efficient encryption schemes for computation over encrypted data, such as keyword search, assume that all data is encrypted with a single key [24, 41]. Using such a scheme in Mylar would require computation over one key at a time, which is inefficient.

For example, consider a user with access to N documents, where each document is encrypted with a different key (since it can be shared with a different set of

[2] For efficiency, Mylar uses authenticated encryption, which conceptually computes both the ciphertext and the MAC tag in one pass.

users). Searching for a keyword in all of these documents would require the user to generate N distinct cryptographic search tokens, and to send all of them to the server. Even for modest values of N, such as 1000, this can result in noticeable computation and network costs for the user's machine. Moreover, if the N keys are not readily available in the client browser, fetching these keys may bring further overhead.

To address this limitation, Mylar introduces a *multi-key search* scheme, as described in the rest of this section.

5.1 Multi-key search

Mylar's multi-key search scheme provides a simple abstraction. If a user wants to search for a word in a set of documents on a server, each encrypted with a different key, the user's machine needs to provide only a single search token for that word to the server. The server, in turn, returns each encrypted document that contains the user's keyword, as long as *the user has access* to that document's key.

The intuition for our scheme is as follows. Say that the documents that a user has access to are encrypted under keys $k_1, \ldots, k_n$ and the user's own key is uk. The user's machine computes a search token for a word w using key uk, denoted tk^w_{uk}. If the server had $tk^w_{k_1}, \ldots, tk^w_{k_n}$ instead of tk^w_{uk}, the server could match the search token against the encrypted documents using an existing searchable encryption scheme.

Our idea is to enable the server *to compute these tokens by itself*; that is, to adjust the initial tk^w_{uk} to $tk^w_{k_i}$ for each i. To allow the server to perform the adjustment, the user's machine must initially compute *deltas*, which are cryptographic values that enable a server to adjust a token from one key to another key. We use $\Delta_{uk \to k_i}$ to denote the delta that allows a server to adjust tk^w_{uk} to $tk^w_{k_i}$. These deltas represent the user's access to the documents, and crucially, these deltas can be reused for every search, so the user's machine needs to generate the deltas only once. For example, if Alice has access to Bob's data, she needs to provide one delta to the server, and the server will be able to adjust all future tokens from Alice to Bob's key.

In terms of security, our scheme guarantees that the server does not learn the word being searched for, and does not learn the content of the documents. All that the server learns is whether the word in the search token matched some word in a document, and in the case of repeated searches, whether two searches were for the same word. Knowing which documents contain the word being searched for is desirable in practice, to avoid the overhead of returning unnecessary documents.

This paper presents the multi-key search scheme at a high level, with emphasis on its interface and security properties as needed in our system. We provide a rigorous description and a cryptographic treatment of the scheme (including formal security definitions and proofs) in a technical report [35]. Readers that are not interested in cryptographic details can skip to §5.3.

Client-side operations:

procedure KEYGEN() ▷ Generate a fresh key
  $key \leftarrow$ random value from $\mathbb{Z}_p$
  return key

procedure ENC(key, $word$)
  $r \leftarrow$ random value from $\mathbb{G}_T$
  $c \leftarrow \langle r, H_2(r, e(H(word), g)^{key}) \rangle$
  return c

procedure TOKEN(key, $word$)
  ▷ Generate search token for matching $word$
  $tk \leftarrow H(word)^{key}$ in $\mathbb{G}_1$
  return tk

procedure DELTA(key_1, key_2)
  ▷ Allow adjusting search token from key_1 to key_2
  $\Delta_{key_1 \to key_2} \leftarrow g^{key_2/key_1}$ in $\mathbb{G}_2$
  return $\Delta_{key_1 \to key_2}$

Server-side operations:

procedure ADJUST(tk, $\Delta_{k_1 \to k_2}$)
  ▷ Adjust search token tk from k_1 to k_2
  $atk \leftarrow e(tk, \Delta_{k_1 \to k_2})$ in $\mathbb{G}_T$
  return atk

procedure MATCH(atk, $c = \langle r, h \rangle$)
  ▷ Return whether c and atk refer to same word
  $h' \leftarrow H_2(r, atk)$
  return $h' \stackrel{?}{=} h$

Figure 4: Pseudo-code for Mylar's multi-key search scheme.

5.2 Cryptographic construction

We construct the multi-key search scheme using bilinear maps on elliptic curves, which, at a high level, are functions $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are special groups of prime order p on elliptic curves. Let g be a generator of $\mathbb{G}_2$. Let H and H_2 be certain hash functions on the elliptic curves. e has the property that $e(H(w)^a, g^b) = e(H(w), g)^{ab}$. Figure 4 shows pseudo-code for our multi-key search scheme.

When encrypting a document with this scheme, Mylar randomizes the order of the keywords.

5.3 Indexing search

One efficiency issue with this algorithm is that the server has to scan through every word of every document to identify a match. This can be slow if the documents are large, but is unavoidable if the encryption of each word is randomized with a different r, as in Figure 4.

To enable the construction of an efficient index over the words in a searchable document, Mylar supports an indexable version of this multi-key search scheme. The idea

is to remove randomness without compromising security. Intuitively, randomness is needed to hide whether two words encrypted under the same key are equal. But for words within one document, Mylar can remove the duplicates at the time the document is encrypted, so per-word randomness is not needed within a document.

Therefore, to encrypt a document consisting of words $w_1, \ldots, w_n$, the client removes duplicates, chooses one random value r, and then uses the same r when encrypting each of the words using ENC().

When searching for word w in a document, the server performs the adjustment as before and obtains atk. It then computes $v \leftarrow \text{COMBINE}(r, atk) = \langle r, H_2(r, atk) \rangle$ using the document's randomness r. If one of the words in the document is w, its encryption will be equal to v, because they use the same randomness r. Therefore, the server can perform direct equality checks on encrypted words. This means that it can build an index over the encrypted words in the document (e.g., a hash table), and then use that index and v to figure out in constant time if there is a match without scanning the document.

A limitation is that the server has to use an index per unique key rather than one holistic index.

5.4 Integrating search with the principal graph

Mylar integrates the multi-key search scheme with the principal graph as follows. When a principal P is created, Mylar generates a key k_P using KEYGEN (Figure 4). Whenever P receives access to some new principal A, Mylar includes k_A in the wrapped key for P. The first time a user with access to P comes online, the Mylar client in that user's browser retrieves k_A from the wrapped key, computes $\Delta_{k_P \to k_A} \leftarrow \text{DELTA}(k_P, k_A)$, and stores it at the server. This delta computation happens just once for a pair of principals.

To encrypt a document for some principal A, the user's browser encrypts each word w in the document separately using $\text{ENC}(k_A, w)$. Cash et al. [10] point out that security can be strengthened by randomly shuffling the keywords before uploading the document, and padding the list of keywords to a fixed size for each document. Since the multi-key search scheme does not support decryption, Mylar encrypts all searchable documents twice: once with the multi-key search scheme, for searching, and once with a traditional encryption scheme like AES, for decryption.

To search for a word w with principal P, the user's client uses $\text{TOKEN}(k_P, w)$ to compute a token tk, and sends it to the server. To search over data encrypted for principal A, the server obtains $\Delta_{k_P \to k_A}$, and uses $\text{ADJUST}(tk, \Delta_{k_P \to k_A})$ to adjust the token from k_P to k_A, obtaining the adjusted token atk_A. Then, for each document encrypted under k_A with randomness r, the server computes $v \leftarrow \text{COMBINE}(r, atk_A)$ and checks if v exists in the document using an index. The server repeats the same process for all other principals that P has access to.

Integrating the access graph with keyword search brings up two challenges. The first comes from the fact that our multi-key search scheme allows adjusting tokens just once. In the common case of an access graph where all paths from a user to the data's encryption key consist of one edge (such as the graph in Figure 3), Mylar associates the search delta with the edge, and stores it along with the wrapped key. In our chat example, this allows a user's browser to search over all chat rooms that the user has access to, by sending just one search token.

Some applications can have a more complex access graph. For example, in the endometriosis application, all doctors have access to the *staff* principal, which in turn has access to all patient principals. Here, the optimal approach is to use the ADJUST() function on the server between principals with the largest number of edges, so as to maximize the benefit of multi-key search. For instance, if a doctor wanted to search over patient records, the doctor's browser should fetch the *staff* principal it has access to, and produce a search token using the *staff* principal's private key. The server would then use ADJUST() to look for matches in documents encrypted with each patient's key. Because most of our applications have simple access graphs, our prototype does not automate this step, and a developer must choose the principal with which to search.

The second challenge comes from the fact that searching over data supplied by an adversary can leak the word being searched for. For example, suppose an adversary creates a document containing all the words in a dictionary, and gives the user access to that document. If the user searches for a word w in all of the documents he has access to, including the one from the adversary, the server will see which of the words in the adversary's document matches the user's token, and hence will know which dictionary word the user searched for. To prevent this, users must explicitly *accept* access to a shared document, and developers must invoke the **allow_search** function, provided by Mylar for this purpose, as appropriate.

More concretely, before user P can search on principal A, the application's client code must call P.**allow_search**(A), shown in Figure 2. It is the responsibility of the application developer to ensure that, prior to calling **allow_search**, the application is certain that principal A is trustworthy. (Otherwise, any user with access to principal A will be able to see all search queries that P ever issues, by the attack discussed above.) The application may trust A due to application-specific logic (e.g., because A is a globally trusted principal in the application), or the application may ask the user for input to decide if A should be trusted with seeing all the search queries of P. Mylar's certificate chain can help the user

```
procedure ProcessResponse(url, cert, response)
                                  ▷ url is the requested URL
                          ▷ cert is server's X.509 certificate
    if cert contains attribute mylar_pubkey then
        pk ← cert.mylar_pubkey
        sig ← response.header["Mylar-Signature"]
        if not VerifySig(pk, response, sig) then
            return Abort
    if url contains parameter "mylar_hash=h" then
        if hash(response) ≠ h then return Abort
    return Pass
```

Figure 5: Pseudo-code for Mylar's code verification extension.

and the developer establish the correct identity of the creator of A even in the presence of an adversary.

6 Verifying client-side code

Although Mylar uses encryption to protect confidential data stored on the untrusted server, the cryptographic keys and the plaintext data are both available to code executing in the user's web browser. The same-origin policy [53] ensures that applications from *other* origins running in the browser do not access the data in the Mylar application. However, Mylar must also ensure that code running in the application's origin has not been tampered with.

Since the code in a web page is static in Mylar, a strawman solution is to sign this code and verify the signature in the browser. The strawman does not suffice because of a combination of two factors. On the one hand, most web applications (including those using Mylar) consist of multiple files served by the web server. On the other hand, the only practical way to control what is loaded in a browser is to interpose on individual HTTP requests.

The problem arises because at the level of individual HTTP requests, it is difficult to reason about what code the browser will execute. For example, if an image is loaded in the context of an `<IMG SRC=...>` tag, it will not execute Javascript code. But if the same image is loaded as a top-level page, the browser's content-sniffing algorithm may decide the file is actually HTML, and potentially execute Javascript code embedded in the image [6]. Thus, a well-meaning developer must be exceedingly careful when including any content, such as images, in their web application. If the developer inadvertently includes a malicious image file in the application, an adversary can cause the browser to load that file as a top-level page [5] and trigger this attack. Similar problems can arise with other content types, including CSS style sheets, PDF files, etc.

Two-origin signing. To address this problem, Mylar uses two origins to host an application. The *primary* origin hosts exactly one file: the application's top-level HTML page. Consequently, this is the only page that can gain access to the application's encryption keys and plaintext data in the browser. All other files, such as images, CSS style sheets, and Javascript code, are loaded from the *secondary* origin. Mylar verifies the authenticity of these files to prevent tampering, but if an adversary tries to load one of these files as a top-level page, it will run with the privileges of the secondary origin, and would not be able to access the application's keys and data.

To verify that the application code has not been tampered with, Mylar requires the site owner to create a public/private key pair, and to sign the application's top-level HTML page (along with the corresponding HTTP headers) with the private key. Any references to other content must refer to the secondary origin, and must be augmented to include a mylar_hash=h parameter in the query string, specifying the expected hash of the response. The hash prevents an adversary from tampering with that content or rolling it back to an earlier version. Rollback attacks are possible on the top-level HTML page (because signatures do not guarantee freshness), but in that case, the entire application is rolled back: hashes prevent the adversary from rolling back some but not all of the files, which could confuse the application.

This signing mechanism can verify only the parts of an application that are static and supplied by the web site owner ahead of time. It is up to the application code to safely handle any content dynamically generated by the server at runtime (§3.4). This model is a good fit for AJAX web applications, in which the dynamic content is only data, rather than HTML or code.

Browser extension. Each user of Mylar applications should install the Mylar browser extension in their web browser, which verifies that Mylar applications are properly signed before running them. Figure 5 shows the pseudo-code for the Mylar browser extension. The site owner's public key is embedded in the X.509 certificate of the web server hosting the web application. Mylar assumes that certificate authorities will sign certificates for the web application's hostname only on behalf of the proper owner of the web application's domain (i.e., the site owner). Thus, as long as the site owner includes the public key in all such certificates, then users visiting the correct web site via `https` will obtain the owner's public key, and will verify that the page was signed by the owner.

7 Implementation

We implemented a prototype of Mylar by building on top of the Meteor web application framework [33]. Meteor allows client-side code to read and update data via MongoDB operations, and also to issue RPCs to the server. Mylar intercepts and encrypts/decrypts data accessed via the MongoDB interface, but requires developers to explicitly handle data passed via RPCs. We have not found this to be necessary in our experience.

We use the SJCL library [43] to perform much of our cryptography in Javascript, and use elliptic curves for most public-key operations, owing to shorter ciphertexts and higher performance. As in previous systems, Mylar uses faster symmetric-key encryption when possible [36]. For bilinear pairings, we use the PBC C++ library to improve performance, which runs either as a Native Client module (for Chrome), as a plugin (for Firefox), or as an NDK-based application (for Android phones). To verify code in the user's browser, we developed a Firefox extension. Mylar comprises ∼9,000 lines of code in total.

When looking up paths in the principal graphs, Mylar performs breadth-first search. We have not found this to be a bottleneck in our experience so far, but more efficient algorithms, such as meet-in-the-middle, are possible.

8 Building a Mylar application

To demonstrate how a developer can build a Mylar application, we show the changes that we made to the kChat application to encrypt messages. In kChat, users can create chat rooms, and existing members of a chat room can invite new users to join. Only invited users have access to the messages from the room. A user can search over data from the rooms he has access to. Figure 6 shows the changes we made to kChat, using Mylar's API (Figure 2).

The call to *Messages*.**encrypted** specifies that data in the "message" field of that collection should be encrypted. This data will be encrypted with the public key of the principal specified in the "roomprinc" field. All future accesses to the *Messages* collection will be transparently encrypted and decrypted by Mylar from this point. The call to *Messages*.**searchable** specifies that clients will need to search over the "message" field; consequently, Mylar will store a searchable encryption of each message in addition to a standard ciphertext.

When a user creates a new room (create_room), the application in turn creates a new principal, named after the room title and signed by the creator's principal. To invite a user to a room, the application needs to give the new user access to the room principal, which it does by invoking **add_access** in invite_user.

When joining a room (join_room), the application must look up the room's public key, so that it can encrypt messages sent to that room. The application specifies both the expected room title as well as the room creator as arguments to **princ_lookup**, to distinguish between rooms with the same title. By displaying both the room title and the creator email address, as in Figure 7, the application helps the user distinguish the correct room from an identically named room that an adversary created.

To send a message to a chat room, kChat needs to specify a principal in the **roomprinc** field of the newly inserted document. In this case, the application keeps the current room's principal in the *room_princ* global variable. Similarly, when searching for messages containing a word, the application supplies the principal whose key should be used to generate the search token. In this case, kChat uses the current user principal, **princ_current**().

```
// On both the client and the server:
idp = idp_config(url, pubkey);
Messages.encrypted({"message": "roomprinc"});
Messages.auth_set(["roomprinc", ["id", "message",
                                 "room", "date"]]);
Messages.searchable("message");

// On the client:
function create_user(uname, password):
    create_user(uname, password, idp);
function create_room(roomtitle):
    princ_create(roomtitle, princ_current());
function invite_user(username):
    global room_princ;
    room_princ.add_access(princ_lookup(username, idp));
function join_room(room):
    global cur_room, room_princ;
    cur_room = room;
    room_princ = princ_lookup(room.name,
                              room.creator, idp);
function send_message(msg):
    global cur_room, room_princ;
    Messages.insert({message: msg, room: cur_room.id,
                     date: new Date().toString(),
                     roomprinc: room_princ});
function search(word):
    return Messages.search(word, "message",
                           princ_current(), all, all);
```

Figure 6: Pseudo-code for changes to the kChat application to encrypt messages. Not shown is unchanged code for managing rooms, receiving and displaying messages, and login/logout (Mylar provides wrappers for Meteor's user accounts API).

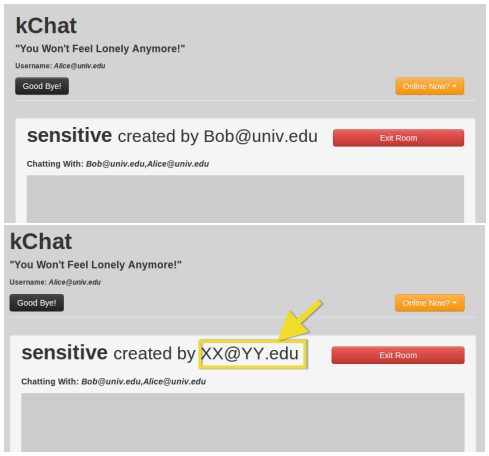

Figure 7: Two screenshots from kChat. On the top, Alice is chatting with Bob as intended. On the bottom, the server provided a fake "sensitive" chat room created by the adversary; Alice can detect this by checking the creator's email address.

Application	LoC before	LoC added for Mylar	Number and types of fields secured	Existed before?	Keyword search on
kChat [27]	793	45	1 field: chat messages	Yes	messages
endometriosis	3659	28	tens of medical fields: mood, pain, surgery, ...	Yes	N/A
submit	8410	40	3 fields: grades, homework, feedback	Yes	homework
photo sharing	610	32	5 fields: photos, thumbnails, captions, ...	Yes	N/A
forum	912	39	9 fields: posts body, title, creator, user info, ...	No	posts
calendar	798	30	8 fields: event body, title, date, user info, ...	No	events
WebAthena [8]	4800	0	N/A: used for code authentication only	Yes	N/A

Figure 8: Applications ported to Mylar. "LoC before" reports the number of lines of code in the unmodified application, not including images or Meteor packages. "Existed before" indicates whether the application was originally built independent of Mylar.

9 EVALUATION

This section answers two main questions: first, how much developer effort is required to use Mylar, and second, what are the performance overheads of Mylar?

9.1 Developer effort

To measure the amount of developer effort needed to use Mylar, we ported 6 applications to Mylar. Two of these applications plan to start using Mylar in production in the near future: a medical application in which endometriosis patients record their symptoms, and a web site for managing homework and grades for a class at MIT. We also ported an existing chat application called kChat, in which users share chat rooms by invitation and exchange private messages, and a photo sharing application. We also built a Meteor-based forum and calendar, which we then ported to Mylar. Finally, to demonstrate the generality of Mylar's code verification, we used it to verify the code for WebAthena [8], an in-browser Javascript Kerberos client.

Figure 8 summarizes the fields we secured with Mylar in the above applications, along with how much code the developer had to change. In the case of the endometriosis application, fields were stored in the database as field name and field value pairs, so encrypting the generic "value" field secured tens of different kinds of data. In the other apps, a field corresponded to one kind of sensitive data. The results show that Mylar requires little developer effort to protect a wide range of confidential data, averaging 36 lines of code per application.

9.2 Performance

Mylar's performance goal is to avoid significantly affecting the user experience with the web application. To evaluate whether Mylar meets this goal, we answer the following questions:

- How much latency does Mylar add to the web application's overall user interface?
- How much throughput overhead does Mylar impose on a server?

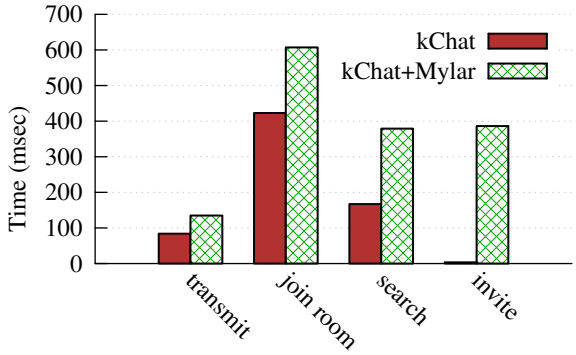

Figure 9: End-to-end latency of four operations in kChat. Transmit includes the time from when one user sends a message to when another user receives it.

- Is Mylar's multi-key search important to achieve good performance?
- How much storage overhead does Mylar impose?

To answer these questions, we measured the performance of kChat, the homework submission application ("submit"), and the endometriosis application. Although kChat has only one encrypted field, every message sent exercises this field. We used two machines running recent versions of Debian Linux to perform our experiments. The server had an Intel Xeon 2.8 GHz processor and 4 GB of RAM; the client had eight 10-core Intel Xeon E7-8870 2.4 GHz processors with 256 GB of RAM. The client machine is significantly more powerful to allow us to run enough browsers to saturate the server. For browser latency experiments, we simulate a 5 Mbit/s client-server network with 20 msec round-trip latency. All experiments were done over `https`, using nginx as an `https` reverse proxy on the server. We used Selenium to drive a web browser for all experiments. We also evaluated Mylar on Android phones and found that performance remained acceptable, but we omit these results for brevity.

End-to-end latency. Figure 9 shows the end-to-end latency Mylar introduces for four main operations in kChat: transmitting a message, joining a room, searching for a

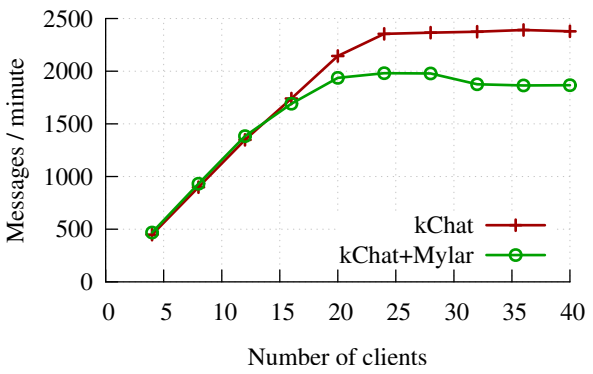

Figure 10: Server throughput for kChat.

word in all rooms, and inviting a user to a room. For message transmission, we measured the time from the sender clicking "send" until the message renders in the recipient's browser. This is the most frequent operation in kChat, and Mylar adds only 50 msec of latency to it. This difference is mostly due to searchable encryption, which takes 43 msec. The highest overhead is for inviting a user, due to principal operations: looking up and verifying a user principal (218 msec) and wrapping the key (167 msec). Overall, we believe the resulting latency is acceptable for many applications, and subjectively the application still feels responsive.

We also measured the latency of initially loading a page. The original kChat application loads in 291 msec. The Mylar version of kChat, without the code verification extension, loads in 356 msec, owing to Mylar's additional code. Enabling the code verification extension increases the load time to 1109 msec, owing to slow signature verification in the Javascript-based extension. Using native code for signature verification, as we did for bilinear pairings, would reduce this overhead. Note that users experience the page load latency only when first navigating to the application; subsequent clicks are handled by the application without reloading the page.

We also measured the end-to-end latency of the most common operations in the endometriosis application (completing a medical survey and reading such a survey), and the submit application (a student uploading an assignment, and a staff member reading such a submission); the results are shown in Figure 11. For the submit application, we used real data from 122 students who used this application during the fall of 2013 in MIT's 6.858 class. Submit's latency is higher than that of other applications because the amount of data (student submissions) is larger, so encryption with search takes longer. For comparison, we also show the latency of submit when search is turned off. The search encryption can happen asynchronously so the user does not have to wait for it.

Throughput. To measure Mylar's impact on server throughput, we used kChat, and we set up many pairs of browsers—a sender and a receiver—where the sender continuously sends new messages. Receivers count the total number of messages received during a fixed interval. Figure 10 shows the results, as a function of the total number of clients (each pair of browsers counts as 2 clients). Mylar decreases the maximum server throughput by 17%. Since the server does not perform any cryptographic operations, Mylar's overhead is due to the increase in message size caused by encryption, and the encrypted search index that is added to every message to make it searchable.

Figure 11 also shows the server throughput of the endometriosis and class submit application when clients perform representative operations.

Search. To evaluate the importance of Mylar's multi-key search, we compare it to two alternative approaches for secure search. The first alternative is single-key server-side search, in which the client generates a token for every key by directly computing the adjusted token from our multi-key search. This alternative is similar to prior work on encrypted keyword search. In this case, the client looks up the principal for every room, computes a token for each, and the server uses one token per room. The second alternative is to perform the search entirely at the client, by downloading all messages. In this case, the client still needs to look up the principal for each room so that it can decrypt the data.

Figure 12 shows the time taken to search for a word in kChat for a fixed number of total messages spread over a varying number of rooms, using multi-key search and the two alternatives described above. We can see that multi-key search is much faster than either of the two alternatives, even with a small number of rooms. The performance of the two alternatives is dominated by the cost of looking up the principal for each room and obtaining its private key. Multi-key search does not need to do this, because the server directly uses the deltas, and it achieves good performance because both ADJUST and MATCH are fast, as shown in Figure 13.

Storage overhead. For kChat, the server storage overhead after inserting 1,000 messages with Mylar was 4× that of unmodified kChat. This is due to three factors: principal graphs (storing certificates and wrapped keys), symmetric key encryption, and searchable encryption. Our prototype stores ciphertexts in base-64 encoding; using a binary encoding would reduce storage overheads.

10 DISCUSSION

Mylar focuses on protecting confidential data in web applications. However, Mylar's techniques for searching over encrypted data and for verifying keys are equally applicable to desktop and mobile phone applications; the

Application	Operation for latency	Latency w/o Mylar	Latency with Mylar	Throughput w/o Mylar	Throughput with Mylar	Throughput units
submit	send and read a submission	65 msec	606 msec	723	394	submissions/min
submit w/o search			70 msec		595	
endometriosis	fill in/read survey	1516 msec	1582 msec	6993	6130	field updates/min

Figure 11: Latency and throughput of different applications with and without Mylar. The latency is the end-to-end time to perform the most common operation in that application. For submit, the latency is the time from one client submitting an assignment until another client obtains that submission. For endometriosis, the latency is the time from one client filling out a survey until another client obtains the survey.

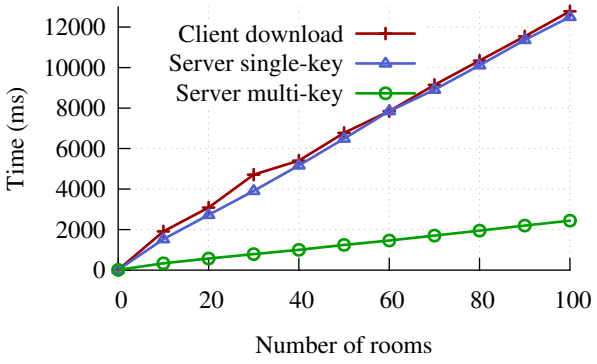

Figure 12: End-to-end latency of keyword search in kChat, searching over 100 6-word messages, spread over a varying number of rooms.

Encrypt	Delta	Token	Adjust	Match
6.5 ms	7.1 ms	0.9 ms	5.6 ms	0.007 ms

Figure 13: Time taken to run each multi-key search operation.

primary difference is that code verification becomes simpler, since applications are explicitly installed by the user, instead of being downloaded at application start time.

Mylar relies on X.509 certificates to supply the web site owner's public key for code verification. Alternative schemes could avoid the need for fully trusted certificate authorities [47, 49], and the Mylar extension could allow users to manually specify site owner public keys for especially sensitive web sites.

Revoking access to shared data is difficult, because Mylar cannot trust the server to forget a wrapped key. Complete revocation requires re-encrypting shared data under a new key, and giving legitimate users access to the new key. In less sensitive situations, it may suffice to try deleting the key from the server, which would work if the server is not compromised at the time of the deletion.

11 Conclusion

Mylar is a novel web application framework that enables developers to protect confidential data against attackers who gain access to the data stored on the server. Mylar leverages the recent shift to exchanging data, rather than HTML, between the browser and server, to encrypt all data stored on the server, and decrypt it only in users' browsers. Mylar provides a principal abstraction to securely share data between users, and uses a browser extension to verify code downloaded from the server that runs in the browser. For keyword search, which is not practical to run in the browser, Mylar introduces a cryptographic scheme to perform keyword search at the server over data encrypted with different keys. Experimental results show that using Mylar requires few changes to an application, and that the performance overheads of Mylar are modest.

Mylar and the applications discussed in this paper are available at `http://css.csail.mit.edu/mylar/`.

Update log

2016-08-29 This version expands on the original Mylar paper [37] as follows. Expanded the discussion of Mylar's security guarantees in §3.4. Clarified claims of security for passive vs active attackers throughout the paper. Discussed follow-up work by Karapanos et al [26], Cash et al [10], and Grubbs et al [23].

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mike Freedman, for their feedback. We also thank Linda Griffith, John Guttag, Nicolaas Kaashoek, and Michelle Park for providing a real medical use case for Mylar with their endometriosis application. This research was supported by NSF award IIS-1065219, by DARPA CRASH under contracts #N66001-10-2-4088 and #N66001-10-2-4089, by Quanta, and by Google.

REFERENCES

[1] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *Proceedings of the 21st Usenix Security Symposium*, Bellevue, WA, Aug. 2012.

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, Jan. 2013.

[3] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2006.

[4] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 205–216, Athens, Greece, June 2011.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th Usenix Security Symposium*, San Jose, CA, July–Aug. 2008.

[6] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[7] F. Beato, M. Kohlweiss, and K. Wouters. Scramble! your social network data. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium*, Waterloo, Canada, July 2011.

[8] D. Benjamin. Adapting Kerberos for a browser-based environment. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Sept. 2013.

[9] D. Borelli. The name Edward Snowden should be sending shivers up CEO spines. *Forbes*, Sept. 2013. `http://www.forbes.com/sites/realspin/2013/09/03/the-name-edward-snowden-should-be-sending-shivers-up-ceo-spines/`.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[11] A. Chen. GCreep: Google engineer stalked teens, spied on chats. *Gawker*, Sept. 2010. `http://gawker.com/5637234/`.

[12] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[13] M. Christodorescu. Private use of untrusted web servers via opportunistic encryption. In *Proceedings of the Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2008.

[14] CipherCloud. Cloud data protection solution. `http://www.ciphercloud.com`.

[15] Defuse Security. Encrypted pastebin. `https://defuse.ca/pastebin.htm`, Sept. 2013.

[16] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[17] R. Fischer, M. Seltzer, and M. Fischer. Privacy from untrusted web servers. Technical Report YALEU/DCS/TR-1290, Yale University, Department of Computer Science, May 2004.

[18] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178, Bethesda, MD, May–June 2009.

[19] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 1996.

[21] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 555–564, Palo Alto, CA, June 2013.

[22] Google, Inc. User data requests – Google transparency report, Sept. 2013. `http://www.google.com/transparencyreport/`

userdatarequests/.

[23] P. Grubbs, R. McPherson, M. Naveed, T. Risenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. Oct. 2016.

[24] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, Oct. 2012.

[25] J. Kannan, P. Maniatis, and B.-G. Chun. Secure data preservers for web services. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, Portland, OR, June 2011.

[26] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *37th IEEE Symposium on Security and Privacy*, 2016.

[27] KiqueDev. kChat. https://github.com/KiqueDev/kChat/.

[28] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, Dec. 2004.

[30] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, New York, NY, May 2012.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[32] Mega. The privacy company. https://mega.co.nz/#privacycompany, Sept. 2013.

[33] Meteor, Inc. Meteor: A better way to build apps. http://www.meteor.com, Sept. 2013.

[34] OpenID Foundation. OpenID. http://openid.net, Sept. 2013.

[35] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508, Aug. 2013. http://eprint.iacr.org/.

[36] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[37] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2014.

[38] K. Puttaswamy, C. Kruegel, and B. Zhao. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Cascais, Portugal, Oct. 2011.

[39] F. Y. Rashid. Salesforce.com acquires SaaS encryption provider Navajo Systems. *eWeek.com*, August 2011.

[40] S. Sauvage. ZeroBin - because ignorance is bliss. http://sebsauvage.net/wiki/doku.php?id=php:zerobin, Feb. 2013.

[41] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, May 2000.

[42] E. Stark. From client-side encryption to secure web applications. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2013.

[43] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in Javascript. In *Proceedings of the Annual Computer Security Applications Conference*, Honolulu, HI, Dec. 2009.

[44] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, Oct. 2013.

[45] The Cryptocat Project. Moving to a browser app model. https://blog.crypto.cat/2012/08/moving-to-a-browser-app-model/, Aug. 2012.

[46] The Cryptocat Project. Cryptocat. http://www.cryptocat.com, Sept. 2013.

[47] Thoughtcrime Labs. Convergence. http://convergence.io/, 2011.

[48] J. Tudor. Web application vulnerability statistics, June 2013. http://www.contextis.com/files/Web_Application_Vulnerability_Statistics_-_June_2013.pdf.

[49] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008.

[50] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Usenix Security Symposium*, Vancouver, Canada, July 2006.

[51] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar. 2009.

[52] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[53] M. Zalewski. *The Tangled Web*. No Starch Press, 2012.