Practical Security for Multi-User Web Application Databases

by

Catherine M.S. Redfield

B.S., Computer Science and Engineering (2011)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
In partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
June, 2012

Certified by . . . .
Nickolai Zeldovich
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Practical Security for Multi-User Web Application Databases

by

Catherine M.S. Redfield

Submitted to the Department of Electrical Engineering and Computer Science
June, 2012
In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Online web applications are continuously vulnerable to attacks on their users' data. Outside adversaries can gain unauthorized access by exploiting unknown vulnerabilities; curious or malicious database administrators can examine or alter the data *in situ*.

Multiple Principal CryptDB protects against attacks on web application servers. By chaining encryption keys to user passwords, an attacker gaining access to decrypted data through issuing arbitrary queries to the database through CryptDB cannot access data belonging to offline users. A logging system and distributed key storage for CryptDB constrain the pool of possibly compromised data after an attack.

Multiple Principal CryptDB can be used to secure the data of six web applications examined, with 2-8 lines of altered source code and 15-111 annotations added to the schema. On the phpBB web forum application, Multiple Principal CryptDB reduces throughput by only 14.5%, with 24 sensitive fields encrypted, and adds less than 26ms of latency to each individual query.

Thesis Supervisor: Nickolai Zeldovich
Title: Assistant Professor of Electrical Engineering and Computer Science

for my father

who always wants to hear about my research

Acknowledgments

This research was performed under the supervision of Professor Nickolai Zeldovich, and it is a joint work with Raluca Ada Popa and Professor Hari Balakrishnan. The work on Multiple Principal CryptDB was included in the CryptDB published at the 23rd Symposium on Operating Systems in the CryptDB paper.

I would like to thank Nickolai for being brilliant and impossibly kind to me. For the past year and a half, I have been continually inspired by his enthusiasm and his confidence in my abilities.

CryptDB is the brainchild of Raluca, so naturally without her, this thesis would not have existed. But beyond letting me work on her awesome ideas, I want to thank Raluca for being such a generous mentor and friend. Without her help and advice and support, there is no way this thesis would ever have been finished.

I would also like to thank my family: my father for encouraging me to always want to learn more and do better; my mother for reminding me that learning is not the only important part of living; and my sisters for making me laugh.

I am also grateful to all the members of the PDOS group, SITB, and Hari Balakrishnan for their help and support on this project.

This work was supported in part by the National Science Foundation under grant 6923594.

Overview

Contents

List of Figures

Chapter 1

Introduction

Online web applications are continuously vulnerable to attacks on their users' data: as they receive information from the user, as they process it, and even after they have stored it in a database. Databases and application servers are also vulnerable: they can be attacked by adversaries exploiting unknown vulnerabilities and gaining unauthorized access. Databases are also vulnerable to attack by curious or malicious database administrators [2], or to a physical attack [8]. Storing private user data on the database in an encrypted form, and never allowing the database management system access to the encryption key would reduce the damage that would result from such an attack.

A typical database-backed application has three main components: the user (client) machines, the application server, and the database management server (database server). Figure 1-1 shows these components in white. Our idea for of ensuring user privacy is to have the database management system maintain the data with the same efficiency in the unencrypted case, but without having access to the data (encrypting the data). There have been several proposals for systems that create an encrypted database. The simplest suggestion would be to encrypt and decrypt the data as it passes through the application server. However, altering an existing application to handle the additional work of encrypting and decrypting all the data that passes through it would be a huge task. This straw-man would also transfer all the computation and filtering, which database systems are specifically optimized to handle, to the application server, which is not designed to deal with it. Overall, this design would require extensive time from the programmer, as well as an

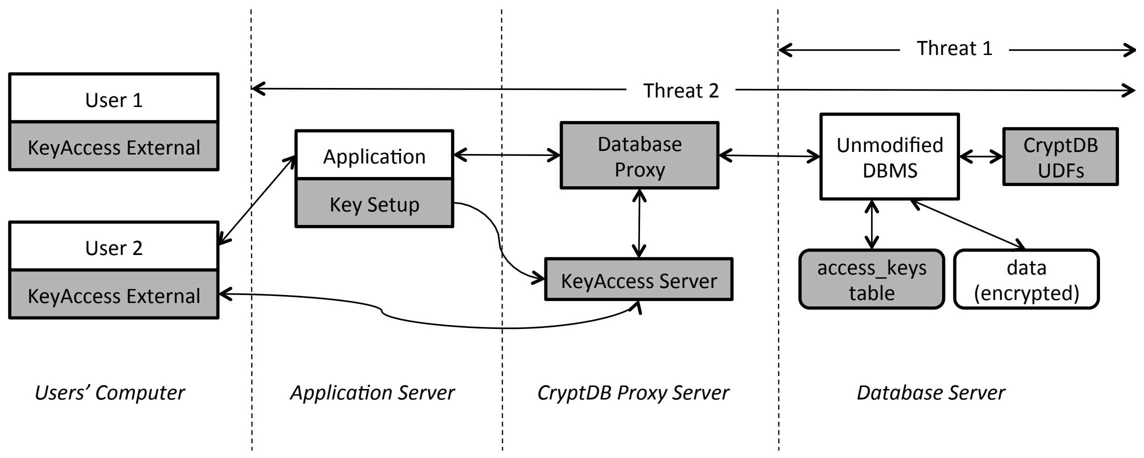


Figure 1-1: The CryptDB architecture. Shaded blocks show sections of CryptDB; the unshaded sections are unmodified.

increase in computation on the application server, which would slow down database accesses. Recent developments in fully homomorphic encryption – where queries are performed on the encrypted data without decryption – make that a theoretically possible solution, but the current implementations would create an overhead impractical for real world systems [6]. In both proposals, an attacker gaining control of the application server would still be able access sensitive data by querying the database through the application.

Popa et al. [12] propose CryptDB: a practical system which, through the observation that there are a finite and small number of possible queries to a standard database, can perform queries on an encrypted database. Figure 1-1 illustrates the design of CryptDB and the threats it is intended to protect against. CryptDB is designed to act as a proxy for the application to the database management system (DBMS), and for the DBMS to the application. Both the application and the DBMS are unchanged and interact with the proxy as they are designed to interact with each other. As queries pass through the main proxy of CryptDB, it encrypts the sensitive data, and issues queries to the DBMS with all sensitive fields encrypted.

The first threat CryptDB is designed to address (Threat 1 in Figure 1-1) is that of a curious or malicious database administrator, or of a physical attack on the database. By encrypting the data in the CryptDB proxy, and never giving the DBMS access to any of the keys used for encryption, an adversary, while still able to corrupt or erase un-targeted data, can no longer read any of the sensitive information. A major challenge in addressing this threat is the tension between securing the encryption and promoting the efficient execution of queries. Current approaches for computing over encrypted data, as mentioned above, tend to be slow or impractical in other ways. CryptDB addresses this problem with the idea of *SQL-aware encryption*, in which different encryption schemes directly related to the possible SQL operators (equality, inequality, search, aggregates such as sum, and joins) are used in combination to allow the DBMS to execute queries as it would usually do.

This thesis describes Multiple Principal CryptDB, which is intended to provide protection against Threat 2 (in Figure 1-1), an attack to the application server. In this case, an attacker could cause the application to issue arbitrary queries to the database through CryptDB, and thus could gain access to the decrypted information. To protect against this threat, we use the idea of *chaining encryption keys to user passwords* (as introduced in [12]), where the key for encrypting a data item is only accessible through a chain of encryption based at a user password. If we consider data to be owned by one or more users, then an item can be decrypted only if a user who has ownership over a given item is online.

While these design choices prohibit some privacy violations, attacks are still possible, and any data owned by a user who was logged on during the period of the attack may have been compromised. For this eventuality, we also propose a logging and auditing system for CryptDB, wherein the system logs which keys are used, providing a record of all accesses. Thus in the case of an attack, the log constrains the pool of possibly compromised data. To ensure that this log is inaccessible to an attacker, we also propose a variation of Multiple Principal CryptDB, where users run each their own copy of our key chaining program, shown as External KeyAccess in Figure 1-1. Each External KeyAccess stores the access keys and logs CryptDB's accesses to them. Using this distributed key access system, it would be possible to greatly reduce the set of possibly compromised data during an attack.

Multiple Principal CryptDB does not encrypt the entire database – only the fields which are designated sensitive by the application maintainer. Using an annotation scheme we developed for the common web forum phpBB, Multiple Principal CryptDB has a low overhead, reducing throughput by 14.5% and increasing database size by 1.2x, without auditing functionality. For normal application usage, Multiple Principal CryptDB increases latency by

5-26ms without auditing, and 3-78ms with auditing. Although there are certain types of queries Multiple Principal CryptDB cannot process, an examination several applications shows that these types of queries are rare. An examination of six different web applications showed that the design paradigm of Multiple Principal CryptDB is viable for their access control implementations, and that, for most of them, replacing their database management system with Multiple Principal CryptDB would require only 2-8 lines of application code to changed, and 31-111 lines of annotations to be added to the schema.

Chapter 2 presents the encryption schemes CryptDB uses, in the case of only one principal or user. This section is not the focus of this thesis, as it represents work proposed and implemented by Popa et al. [14]. It is presented as background work, as the concepts of CryptDB's encryption schemes are easier to understand when considering the single principal case, without the complexities of the Multiple Principal case. Chapter 3 describes the design and Chapter 4 describes the implementation of Multiple Principal CryptDB, the focus of this thesis, in detail. Chapter 5 discusses one of the additional features for multiple principal CryptDB – a system for recording the keys requested by an application and from which users' credentials they are derived. Chapter 6 describes the tests we ran on Multiple Principal CryptDB to determine its practicality for real world implementations. Chapter 7 concludes.

Multiple Principal CryptDB is also described in [12], on which I am a co-author. The work on Single Principal CryptDB and the idea of key chaining are not my work, and are presented as background work. The implementations, algorithms, and other design choices described in Chapters 3 and 5 are my personal work.

1.1 Terminology

Throughout this thesis we use the following terms to describe specific ideas. We use the phrases *logged on* and *online* to refer to a user who has been authenticated by the web application they are using. We use *logged off* and *offline* to refer to a user who has specifically revoked the authorization for the the server to act on his behalf.

We use *user* or *external principal* to refer to the authentication information associated with a physical user of a web application, such as a username and password. *Principal*, on the other hand, refers to an entity that fields can be encrypted for, and that can speak for other principals or for users. Principals (external and otherwise) are defined by the programmer during schema creation. An *instance* of a principal is the combination of the programmer-defined principal and the data related to it. Each principal instance has encryption keys associated with it. For example, in an online forum, the pseudonym 'alice' and its associated password would be an instance of an external principal. A post in the forum would be an instance of a principal (not an external principal). Text of the post could be encrypted using the principal post's encryption keys. Similarly, the *access graphs* which describe key chains come in two types: *principal access graphs* which show the programmer-designated linkages of the principals, and *instance access graphs* which show the relationships between the instances of principals. We extend *online* to refer to principals as well as users. A principal (or by extension, a principal's keys) is online if there is an online external principal who can chain to the principal in question.

In the access graphs and annotations, we say that a link in an instance access graph is a *speaks_for* relation. When we need to refer to two principals in a speaks_for relation, we differentiate them by calling the principal which speak_for the other principal the *speaks_from principal*. We refer to the other principal (the one which is spoken for) as the *speaks_for principal*. With regards to the access graph, the speaks_from principal is the principal closer to an external principal. The speaks_for principal the principal further from an external principal and closer to an encrypted field.

We also refer to the concept of external principals *owning* data. This is particular to applications with some sort of pre-defined internal access control scheme. If an external principal is able, by the rules of the application's access control scheme, to access a record stored in the database, that external principal is considered to own that data.

For encrypting data, CryptDB uses symmetric key encryption. In key chaining (detailed in Chapter 3), CryptDB uses both symmetric and public key encryption schemes. To avoid terminological confusion, we use the *symmetric key* to reference the shared private key in symmetric encryption, *asymmetric private key* and *asymmetric public key* for the public/private key pair required for a public key encryption scheme.

Chapter 2

Overview of Single Principal CryptDB

This section will discuss CryptDB in what we refer to as the *single principal case.* This is the situation where all the encryption is based on the same master key – there is only one principal. Intuitively, this is a system where there is only one user, and that user is the only principal. Practically speaking, the single principal case is useful for applications where access to data is not user-dependant (ie, applications with no access control policy), or where the primary threats are attacks to the database. We discuss the single principal case to examine how CryptDB (both in the single principal case and in the multiple principal case) handles Threat 1.

Accessing an entire column or table from an encrypted database is fairly straightforward, since the database returns the entire resultset to CryptDB for decryption. The performance concerns discussed in Chapter 1.1 come into play when the resultset requires filtering. The basic primitive restrictions that define SQL filters are: equality, inequality, aggregates, search, and join. Ideally, CryptDB should receive exactly the resultset that the application would have received if the query were being issued to an unencrypted database, but in an encrypted form. To achieve this goal, CryptDB must be able to translate the application generated filters into encrypted filters that can be executed over the encrypted data.

2.1 Encryption Schemes

CryptDB uses a variety of schemes when encrypting queries in order to preserve the maximum security on each column. The following encryption levels provide the specified properties in CryptDB:

2.1.1 Random (RND)

The RND encryption level should be implemented with an encryption scheme with indistinguishability under an adaptive chosen-plaintext attack (IND-CPA). RND must be a probabilistic scheme, meaning that with overwhelming probability, the encryption of equal plaintexts will not result in an equal ciphertext. In our implementation, RND is an AES or Blowfish scheme in CBC mode with a random seed. We assume that the server does not alter results, so we don't need to secure against chosen ciphertext attacks (IDN-CCA2), but in cases where this is an erroneous assumption, RND could be implemented with a more secure scheme, such as the UFE mode [5] of a block cipher. This encryption scheme is not one the database can use for any predicates or filters.

2.1.2 Deterministic (DET)

A deterministic encryption scheme uses a psuedorandom permutation (PRP), [7] so that equal plaintexts will always be encrypted to produce equal ciphertexts. Thus DET leak only the equality of the plaintexts. Our implementation uses AES or Blowfish, making the usual assumption that AES and Blowfish block ciphers are PRPs. Since we do not want to restrict the length of inserted data, for data that is longer than a single 128-bit AES block, the standard CBC mode of operation leaks prefix equality, so we use CMC mode. [9] Since our DET implementation is intended to leak equality, we use a seed (or "tweak" in [9]) of zero. Using the leaked equality, the database can use DET encrypted data for equality predicates and filters, equality joins, GROUP BY, COUNT, DISTINCT, etc.

2.1.3 Order Preserving Encryption (OPE)

An order preserving encryption scheme is a randomized mapping where the ciphertext preserves the ordering of the plaintext values. That is, if $x < y$, then for order preserving encryption scheme OPE, with key K, $OPE_K(x) < OPE_K(y)$. We implemented the OPE scheme described in [1]. OPE leaks the order of a column, and thus can be used for inequality filters, ORDER BY, MIN, MAX, SORT, etc.

2.1.4 Join (JOIN and DETJOIN)

In order to avoid cross-correlation between columns (i.e. if column $f1$ and column $f2$ are both at DET, and both contain the value 5, we don't want $DET_{f1}(5)$ to be equal to $DET_{f2}(5)$ by default), DET and OPE use distinct keys for each column. Thus joining two columns requires an algorithm that can convert two columns encrypted with different keys into the same two columns encrypted with the same key, without ever decrypting the data. CryptDB uses original JOIN and DETJOIN algorithms which are described in full in [13].

2.1.5 Homomorphic Encryption (HOM)

Homomorphic encryption is an IND-CPA secure probabilistic encryption scheme that allows the database to perform computations on encrypted data. While fully homomorphic encryption is prohibitively slow, [3] specific operations can be quite fast. We implement a Paillier cryptosystem for summation. [11] In Paillier, $HOM_K(x) \circ HOM_K(y) = HOM_K(x + y)$. Thus executing a SUM aggregate or a summation filter can be handled by a UDF on the database that executes the multiplication of the encrypted values, rather than the original sum. This scheme in conjunction with DET can also be used to compute averages by returning the sum (using HOM) and count (using DET) separately.

2.1.6 Word Search (SEARCH)

SEARCH is an encryption the protocol that allows words to be matched in an encrypted string. We implemented protocol outlined by Song et al. [15], with some different implementation details than the authors theorized (they did not implement the protocol themselves).

For each column requiring SEARCH, we split the text into keywords using standard deliminators (or a special extraction function, if the programmer choose to specify one). Repetitions of these words are removed, the words' positions are permuted randomly, and each of the words is encrypted using Song et al.'s scheme, padding each word to be the same size. SEARCH does not leak whether words repeat in multiple rows of column, and since it does not keep multiples in a given row, it is nearly as secure as RND. However, it does leak the number of keywords encrypted with SEARCH, which could be used to determined the number of distinct or duplicate words by comparing the size of the SEARCH and the RND ciphertexts. SEARCH allows CryptDB to support a limited implementation of MySQL's LIKE operation. Since SEARCH only works with complete words, CryptDB cannot support arbitrary regular expression searches. Should multiple words in order be a valuable search capability, duplicate removal and reordering could be removed, but based on our evaluation of the large trace of queries from web applications, this implementation of SEARCH was sufficient in most cases.

2.2 SQL-Aware Encryption

CryptDB uses these various encryption schemes to translate queries into an encrypted form. Suppose the application issues the query with an inequality operation: `SELECT field FROM table WHERE field < 10`. Obviously, 10 cannot be compared to an encrypted value in a useful fashion. However, we can encrypt 10 with the same encryption scheme that field is encrypted with, making the query: `SELECT field FROM table WHERE field < ENC(10)`. For this to have useful results, the ENC() function must be an order preserving encryption (OPE) scheme, as described in Section 2.1.3. However, there are two problems with using an OPE scheme for the entire database. First, knowing the order of the elements in the column does not help perform aggregates on the column, so there are filters which this paradigm does not handle. Second, an attacker accessing the encrypted data can still see order for every column. For a column which has no inequality queries on it, there is no reason to leak this information, and doing so could be dangerous in certain situations.

Thus we would ideally have a different type of encryption for each column, depending on what sorts of queries access that column. If inequalities are projected on a column, as in the example, that column requires an OPE scheme. If the only projections on the column are equalities (for example `SELECT * FROM table WHERE field = 10`), then a deterministic (DET) encryption scheme is a more secure choice (order is not leaked), and still allows all the queries to be performed on the database without decryption. Some columns may require more than one scheme. If, for example, a text field that is filter with both `WHERE field = alice` and `LIKE alice`, the DET scheme cannot be used for searches, while the SEARCH scheme cannot be used for equality, so this field would need two encryptions: DET and SEARCH.

If an application's necessary level of security is known, then it is straightforward to specify for each column which security level should be used to encrypt the column. If a column has more than one operator, the column could be duplicated with the different schemes. However, in the case where the query set is not known, there is no way to know which level each column needs to be at to execute the application's entire query set. Regardless, it should be possible for CryptDB to keep each field at the most secure level that is consistent with the queries applied to it.

In order to preserve this level of security, we introduce the notion of *onions of encryption*. The goal is to start the entire database at the RND security level, and only decrease the security of encryption when the application require more information (ie, equality or inequality is queried for). However, reducing the security level should never reveal any plaintext to the database. Naturally, we could fetch the entire column, and decrypt, re-encrypt, and replace it, but for large databases this would be slow. The idea of onions of encryptions, as shown in Figure 2-1, is to wrap the plaintext in sequentially higher levels of encryption. For example, in the DET onion, the plaintext of the data is encrypted with the DET-JOIN encryption scheme. The ciphertext that results from that process is then encrypted with a DET scheme, and that result is than encrypted with a RND encryption scheme. The result of the highest encryption scheme is then stored in the database, which means that an attacker cannot access any of the lower levels, since she has no way to decrypt the outermost layer of the onion. For the CryptDB proxy, which holds the encryption keys, decryption is just the reverse of the encryption process: decrypt the RND layer, then decrypt

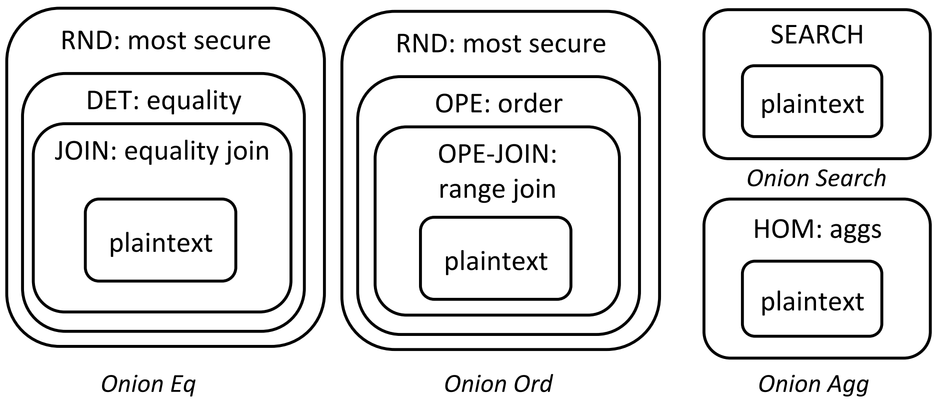


Figure 2-1: Onions. A visual representation of the layers of encryption used by CryptDB. Taken from [12].

the result with DET, then decrypt the result of that with DET-JOIN. Thus, reducing the security level of an entire column simply translates to peeling off onion layers.

Using user defined functions (UDFs) on the database, CryptDB does not even need to fetch the column data to reduce a security level. A UDF is a function loaded on the database that allow queries of the form `UPDATE table SET fieldOPE = DECRYPT_RND(key, field, seed)`. Since UDFs are only declared for going from higher to lower onion layers, but not for decrypting the lowest onion levels, the plaintext remains secure.

Since columns may be acted upon by different combinations of operators, for each column, CryptDB stores three onions per column – one each of DET and OPE, and one of either SEARCH or AGG. Since aggregates do not apply to text fields, and search does not apply to numerical fields, only one of the AGG and SEARCH onions is required.

It may appear that the longer the application uses a database, the lower the security levels will be on the data. In practice, however, this is rarely the case. Most web applications do not issue completely random queries – there is a fixed set of queries that the application code can issue, and it is merely which data inserted, read or modified that is user-defined. Of the six applications examined, four of them left >80% of the sensitive columns at the highest security level.

2.3 Implementation

Our proof of concept of CryptDB implements CryptDB on the MySQL database management system. CryptDB is a library called by a database proxy – our implementation uses mysql-proxy, an open source proxy for the database system MySQL. [16] The proxy calls the functions from the CryptDB library to determine whether, and how to, encrypt and rewrite the queries and resultsets between the application server and the database.

Chapter 3

Multiple Principals

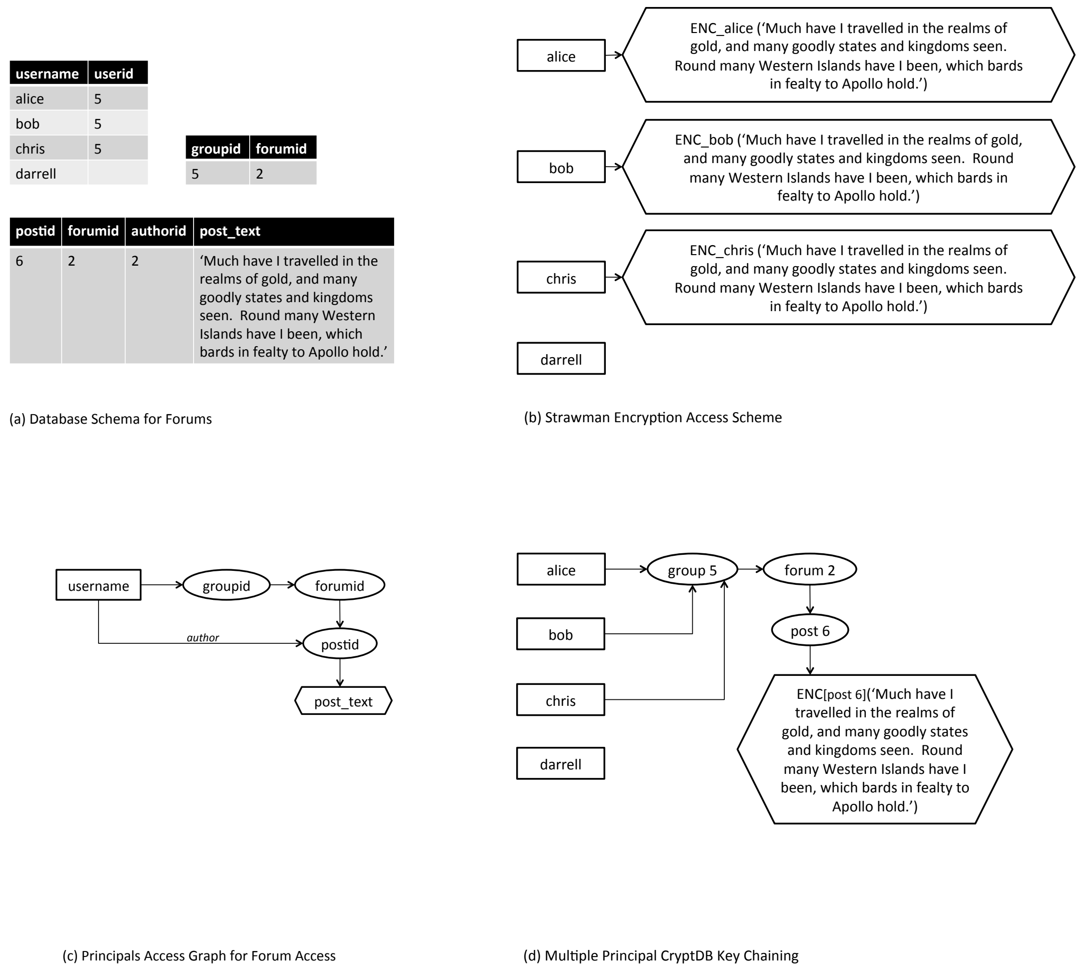
In the single principal case of CryptDB an adversary who compromises the application can cause it to issue arbitrary queries to the database, effectively giving the adversary access to all sensitive information. In an application where users store user-specific sensitive content in the application database, the CryptDB *multiple principle case* can be useful. In this case, data is owned by one or more users, and encrypted with different keys to disallow its decryption unless one of the owners is logged into the application. This type of CryptDB secures the system against Threat 2 from Figure 1-1: in the case of an attack, any data owned exclusively by external principals who are offline for the duration of the attack is secure. To achieve this guarantee, CryptDB must be able to determine which key it needs to use to encrypt and decrypt a field. Since data can be owned by more than one user, CryptDB also needs to be designed to efficiently share data between its owners.

3.1 Key Chaining

A straw-man design that allows sharing would be to duplicate data, encrypting a copy of it with the an owners' password for each owner. Figure 3-1b illustrates an example of this plan with an access controlled forum that has the schema outlined in Figure 3-1a. Even in a such a small example, we can see how this excessive redundancy can create a large space overhead. Removing a user from this group would be time-consuming, as their copy of each forum post must be found and deleted, for all the forums the group owns. Similarly, adding a user to a group would require choosing some other users' copies of all of the forum posts, decrypting these copies, and duplicating and re-encrypting them for the new user. Thus, both adding to and deleting from a group would take $O(|\text{forums groups owns}| \times |\text{posts per forum}|)$ operations. For a large forum, this would be highly impractical.

Consider instead each access control dependency as a set of *principals*. In this model, data is owned, not by a user, but by a principal, which can *speak for* other principals and for users (external principals). When we say *a speaks for b*, we mean that any data owned by b can also be accessed by a. In Figure 3-1, the text of the forum post is owned by the post (a principal), which is turn speaks for the author of the post, which itself speaks for the user who wrote the post. From the point of view of other users who can read the post text, the post also speaks for the forum it is written to, leading to a second access chain ending in the external principals (the users of the forum application). Like the straw-man case above, these access chains allow us to determine information about data ownership. However, instead of using only external principal passwords as keys, we give each principal in the instance access graph a unique encryption key. In the case of the forum post example (Figure 3-1), an instance of post_text will always be encrypted with the encryption key for a post principal instance. Only one copy of the post text is stored, even though it is owned by three separate users. The post key is encrypted with the forum key, which is in turn encrypted with the group key. The group key is encrypted multiple times, once with each of the external principal keys (passwords) who are members of the group. The post key could also be encrypted for external principal who authored the post, but since in this example the author must also be a member of the group, that would be redundant. We call this series of encrypted keys *key chaining*. In Figure 3-1d, alice, bob, and chris can decrypt the group 5 key, using their own passwords, then can decrypt the forum 2 key, using the group 5 key, then the post 6 key using the forum 2 key, and using the post 6 key can read the forum post. Thus, all three of them can transverse the key chain t be able to decrypt the desired post text. On the other hand, darrell has no copy of the group 5 key encrypted with his password, so he has no way of using the key chain to access the post text.

In the key chaining design, unlike the straw-man, the data itself is rarely (never in the applications examined) stored redundantly. The keys themselves are sometimes repeated,



username	userid
alice	5
bob	5
chris	5
darrell	

groupid	forumid
5	2

postid	forumid	authorid	post_text
6	2	2	'Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.'

(a) Database Schema for Forums
alice
ENC_alice ('Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.')
bob
ENC_bob ('Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.')
chris
ENC_chris ('Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.')
darrell
(b) Strawman Encryption Access Scheme
username
groupid
forumid
author
postid
post_text
(c) Principals Access Graph for Forum Access
alice
group 5
forum 2
bob
post 6
chris
ENC[post 6]('Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.')
darrell
(d) Multiple Principal CryptDB Key Chaining

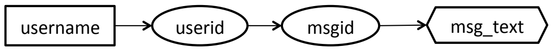
Figure 3-1: A graphical depiction of key chaining. The squares show external principals, the ovals principals, and the hexagons sensitive fields which are encrypted. (a) shows the schema for the this example, without encryption. (b) shows the straw-man design for encrypting owned data. (c) shows the "speaks for" the relationships between the principals, what we refer to as the principal access graph. (d) shows how we can use the instantiated principal access graph (the instance access graph) to come up with a more efficient of encrypting and storing data. In this example, the forum post is not owned simply by the author of the post – if the author is offline, the other members of the forum should still be able read the forum. So this same post also speaks for the forum it is posted in, and the forum speaks for the groups that can view posts there, and the group speaks for users.

but unlike a forum post, keys are a short, known length, minimizing the overhead. Modifying a group also becomes simpler. Deleting a member of a group requires only that that member's copy of the group key be deleted – after which the external user then has no way to chain to the group key, and thus cannot decrypt the forum key. Adding a member to a group requires CryptDB to hold the group key (ie, a member of the group must be online), but this makes sense, as such group modifications will almost certainly be carried out by group members or by some sort of administrative superuser who will be able to chain to all the principals. Given that the group key is accessible, an external principal can be added to a group by encrypting the group key with that external principal's key. Thus, modifying the group requires only one database operation to either insert or delete the group key: $O(1)$.

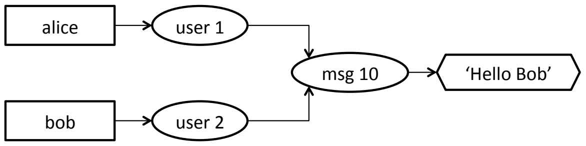
3.1.1 Key Pre-Fetching

External principals then access their keys via these key chains, and can only access their keys when they are logged on. There is then the question of whether a user's keys should be fetched as they are needed, or whether keys should be pre-fetched when the user logs on. Multi-user web applications generate a finite set of queries, but which subset will be called for, and the order in which they will be requested, and thus which keys will be required, for a given user's session are unknown ahead of time. Since there is no way to prefetch only the exact keys an external principal will need we can either pre-fetch all the external principal's keys when they log on, or fetch keys and decrypt the chaining on the fly as keys as needed. Since logging on is a fairly rare event, and in the case of a long key chain, decrypting the entire chain could be time consuming, we choose pre-fetch most of the keys at log-on. Section 4.4 discusses in more detail the prefetching limits – for this discussion we can assume that all keys are pre-fetched. We make the assumption that users are more willing to wait at log-on, rather than when trying to access data. However, in practice, we did not observe any delay to decrypting many keys at logon.

Aside from the usability concerns, pre-fetching has the added advantage of simplifying insertions. In the example above, we note that CryptDB should hold a group key before a member is inserted into that group. If all keys are pre-fetched, then it is immediately obvious



(a) Principals Access Graph for Private Messages



(b) Instances Access Graph for Private Messages

speaks_from	speaks_for	sym key	asym key
alice	user 1	ENC_alice(sym user 1)	
bob	user 2	ENC_bob(sym user 2)	
user 1	msg 10	ENC_sym user 1(sym msg 10)	
user 2	msg 10		ENC_asym user 2(sym msg 10)

(c) Access Keys Table for Private Message

Figure 3-2: Alice sends a message to Bob. The message is owned by both Alice and Bob eventually: the text is encrypted for msgid, which speaks for the userid of the sender and the userid of the receiver. userid speaks for a username, which is an external principal.

if a member of the group is online, since CryptDB will hold all of their keys, including the relevant group key. If keys we not pre-fetched, inserting could be time-consuming, as every online external principal's keys must be fetched until the required group key is found.

3.1.2 Offline Users

There is one more corner case to consider, which is motivated by a slightly different example, shown in Figure 3-2. In this case, there is no reason to suppose that both Alice and Bob are online at the same time, although msgid 10 must speak for both their userids. To handle situations like this (adding a user to a group could have the same problem – while a user who is a member of the group must be online, there is no compelling reason the user who is being added should be), each principal has an asymmetric public key, as well the symmetric key already reasoned about. In the private messages example, Alice writes and sends the message to Bob, who is offline. Since this is a new message, it has a new msgid: 10. The text of the message is encrypted with the msgid 10's symmetric key. Since userid

1 speaks for Alice, and this message speaks for userid 1, the symmetric key of userid 1 is used to encrypt a copy of msgid 10's symmetric key. Since userid 2 speaks for Bob and this message also speaks for userid 2, Bob needs to be able to chain to a copy of msgid 10's symmetric key. Since Bob is offline, however, CryptDB does not hold the symmetric key for userid 2. However, since all principals have a public key, msgid 10's symmetric key is encrypted asymmetrically with Bob's userid's public key. When Bob logs back on, msgid 10's key is prefetched and decrypted using the asymmetric private key for userid 2. The implementation and storage of the various key types is discussed further in Chapter 4.

3.2 Annotations

The access graphs described in Section 3.1 show how CryptDB determines which key to use to decrypt a sensitive field. To generate a principal access graph (such those shown in Figures 3-1a and 3-2a), we extend SQL to include a series of CryptDB specific *annotations*. An application programmer can add our annotations (prefixed with the keyword CRYPTDB to allow our modified SQL parser to filter these and be sure they are passed to CryptDB for processing) to the application schema to create an appropriate principal access graph. Figure 3-3 shows the annotated schema for the examples shown in Figures 3-1 and 3-2.

The PRINCTYPE annotation specifies a principal type. In chained situations like these examples, table keys such as groupid are generally foreign keys in other tables, and CryptDB needs to be aware that groups.groupid refers to the same principal as forums.groupid. The principal types allow that equality identification to take place. PRINCTYPE EXTERNAL annotations specify external users: the roots of the access graphs. Their keys are encrypted with passwords provided by end users. The SPEAKS_FOR annotation outlines a speaks_for link in the access graph; ENC_FOR, a field that needs to be encrypted and the principal it is encrypted for. In both SPEAKS_FOR and ENC_FOR, principals are referred to by field name and principal type, giving CryptDB information about whether or not a certain field is encrypted, and if so by whom.

In Figure 3-3a, it would be possible to add the annotation `CRYPTDB posts.authorid userid SPEAKS_FOR posts.postid postid`, but as previously noted, this would make a

```
CRYPTDB PRINCTYPE username EXTERNAL;
CRYPTDB PRINCTYPE groupid;
CRYPTDB PRINCTYPE forumid;
CRYPTDB PRINCTYPE postid;

CREATE TABLE groups (
    groupid integer,
    username text,
    ...);
CRYPTDB groups.username username SPEAKS_FOR groups.groupid groupid;

CREATE TABLE forums (
    forumid integer,
    groupid integer,
    ...);
CRYPTDB forums.groupid groupid SPEAKS_FOR forums.forumid forumid;

CREATE TABLE posts (
    postid integer,
    forumid integer,
    authorid integer,
    post_text integer,
    ...);
CRYPTDB posts.forumid forumid SPEAKS_FOR posts.postid postid;
CRYPTDB posts.post_text ENC_FOR posts.postid postid;


CRYPTDB PRINCTYPE username EXTERNAL;
CRYPTDB PRINCTYPE userid;
CRYPTDB PRINCTYPE msgid;

CREATE TABLE users (
    username text,
    userid integer);
CRYPTDB users.username username SPEAKS_FOR users.userid userid;

CREATE TABLE priv_msg_info (
    msgid integer,
    authorid integer,
    recipientid integer);
CRYPTDB priv_msg_info.authorid userid SPEAKS_FOR priv_msg_info.msgid msgid;
CRYPTDB priv_msg_info.recipientid userid SPEAKS_FOR priv_msg_info.msgid msgid;

CREATE TABLE msgs (
    msgid integer,
    msgtext text);
CRYPTDB msgs.msgtext ENC_FOR msgs.msgid msgid;
```

Figure 3-3: CryptDB annotations are marked in bold, with the principal types in italics.

redundant link in the access chain. In certain cases, however, if the author of a post had different permissions than its viewers, such an annotation could be useful. This type of decision requires the in-depth knowledge of the programmer and their access control policies and security requirements.

Suffixes to the ENC_FOR annotation allow the programmer to specify a minimum security level (ie, OPE, DET) below which the onions of that column cannot be decrypted. It is also possible to add a predicate to a SPEAKS_FOR annotation, should the access to a certain field be dependant on another column in the table. This is background work, and not within the scope of this thesis, but details can be found in Popa et al. [12].

Once these annotations have been processed when the schema are initialized, the application can run with no other annotations. When the application issues queries such as `INSERT INTO posts VALUES (6, 'Much have I travelled in the realms of gold, and many goodly states and kingdoms seen. Round many Western Islands have I been, which bards in fealty to Apollo hold.', ...)`, CryptDB recognizes these as encrypted fields and if it holds the key for post 6, encrypts post_text before inserting it, using the onion algorithms described in Chapter 2. In cases in the midst of access chains, such as `INSERT INTO forums VALUES (5, 3)`, CryptDB will insert the values into the database without alteration, but will generate and store the necessary keys for principal groupid 5.

Chapter 4

Implementation of Multiple Principal CryptDB

Multiple Principal CryptDB extends Multiple Principal CryptDB to include a class called KeyAccess. KeyAccess is responsible for storing all the principal keys that can be chained to by users who are online. KeyAccess also holds the principal access graph in memory, and is responsible for storing the instance access graph in the database, along with encrypted versions of each principal key.

Our Multiple Principal CryptDB implementation consists of approximately 18,000 lines of C++ code, 150 lines of lua code (used by MySQL-Proxy), with another approximately 10,000 lines of test code. Of these numbers, KeyAccess in a non-distributed system is approximately 2,300 lines of C++ code. Multiple Principal CryptDB has only been implemented on MySQL 5.1, but since KeyAccess has no specialized database access requirements, it should be feasible to port it to other systems.

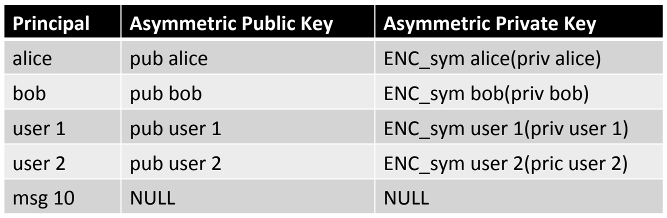
4.1 Annotation Processing and KeyAccess

Processing the annotations requires intercepting queries in MySQL's parser. We specify that a table must exist before an annotation can reference it. This means that when CryptDB sees a CREATE TABLE statement, there is no way to know if the table in question should have onion layers on any of its fields. Since the percentage of sensitive fields was small in the applications we analyzed, we choose to create the table with no onions. If at some future time an ENC_FOR annotation in processed on a field in that table, CryptDB translates it into a series of ALTER TABLE queries that modify the original table to suit requested encryption.

SPEAKS_FOR and PRINCTYPE annotations do not generate any queries. However, all query types are necessary for setting up the data structures that store the access graphs for the application. All references to access keys and encryption keys are processed through a separate class called KeyAccess, which can be decoupled from the rest of CryptDB and run as a separate process or set of client processes, as discussed in Chapter 5. This class records the access graph as a series of links in a special table, which is shown in Figure 3-2c: *access_keys* (speaks_for, speaks_from, sym key, asym key). Each row of *access_keys* contains the principal that is spoken for (labeled speaks_from here), the principal that speaks for the speaks_from principal, and the speaks_for principal's key encrypted for the speaks_from principal. KeyAccess also holds the principal access graph is memory, built from the annotations. The instance access graph (Figures 3-1d and 3-2b) is implicitly stored in *access_keys*.

As annotations are being processed, KeyAccess builds the principal access graph in whatever pattern the annotation queries are issued. However, when instances of the principals are being inserted, the principal access graph must be well-formed: it must be rooted at one or more external principals (well-formed). Otherwise, there would be no user key password chain the encryption from. KeyAccess does not store encrypted fields, only the principals. On the first INSERT query issued to a table in the access graph, KeyAccess checks that the access graph is well-formed and fixes it. If the programmer attempts to alter the graph after values have been inserted into the database, CryptDB issues an error. At this point, the inserted data is already encrypted with the pre-defined key chains, so altering the access graph could corrupt all pre-existing data. Adding to the graph is possible (ie, adding new principals), but as there is no way for CryptDB to predict when data will be inserted into these new principals, these principals must be inserted maintaining the well-formed structure of the access graph.

Besides the *access_keys* table, KeyAccess stores a second table: *public_keys* (principal



Principal	Asymmetric Public Key	Asymmetric Private Key
alice	pub alice	ENC_sym alice(priv alice)
bob	pub bob	ENC_sym bob(priv bob)
user 1	pub user 1	ENC_sym user 1(priv user 1)
user 2	pub user 2	ENC_sym user 2(pric user 2)
msg 10	NULL	NULL

Figure 4-1: *public_keys*.

type, principal value, asymmetric public key, asymmetric private key, symmetric key). *public_keys* holds all the instances of principals that KeyAccess has seen. The asymmetric private key is stored encrypted with the symmetric private key, and the asymmetric public key is available to all. Figure 4-1 shows the *public_keys* table for the example from Figure 3-2. Note that msg 10 does not have asymmetric keys: asymmetric key pair generation is one of the slowest cryptographic operations in the code. We only use public key cryptography for the case where a key needs to be encrypted for a principal that is currently offline. Since the principal type msg is a leaf on the access tree, there are no principals that have keys that msg would need to chain to, so there is no reason to generate an asymmetric key pair for any of the instances of msg. However, msg 10 still has a row in the *public_tree* table because we use *public_keys* as a reference for the instances of principals that we have seen before.

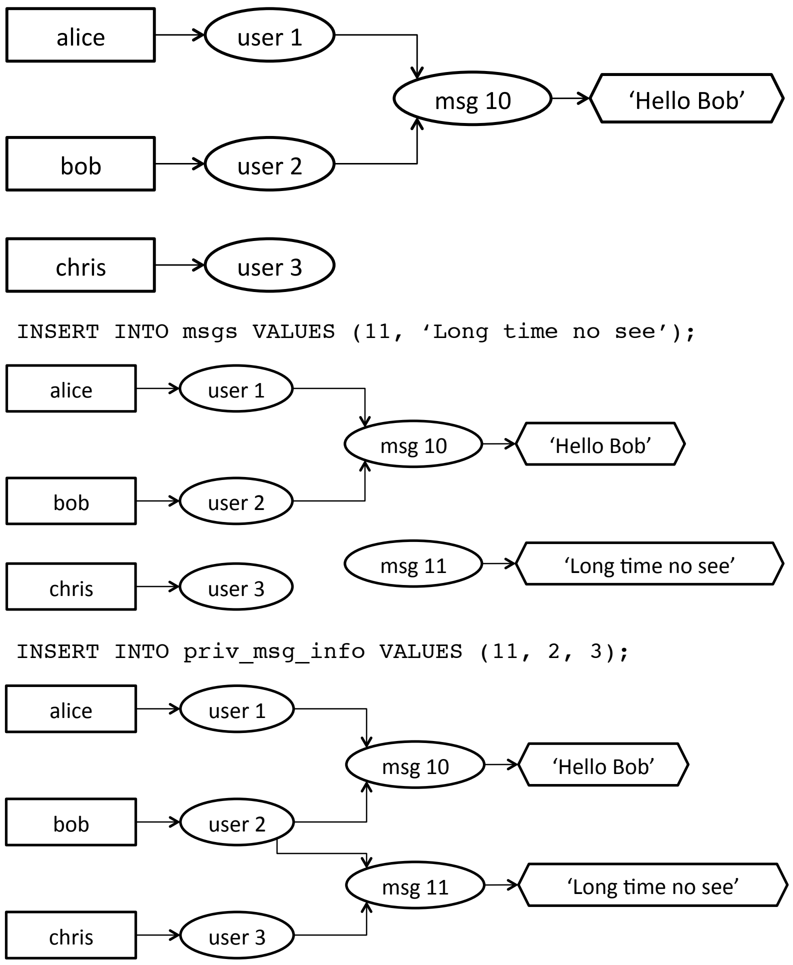
4.2 Login and Logout

External principals are logged on by the application issuing an insert query for a special table *cryptdb_users*. Instead of being an actual table on the database, INSERT and DELETE queries for this table are caught by CryptDB, and send instead insertPsswd and removePsswd calls to CryptDB. The insertPsswd method pre-fetches the user's keys, as described in Section 3.1; removePsswd reverses the process, expunging all keys that the given external principal exclusively chained to. If another external principal who can chain to that key is still logged on, KeyAccess continues to store the key, but does delete the user who is logging off's reference to it.

4.3 Orphaned Keys

The principal access graph must always be well-formed, as discussed above. However, the instance access graph (as exemplified in Figures 3-1b and 3-2b) is determined by the SQL queries issued by the application. Applications are not required to add records to their databases in any order, so CryptDB is designed to handle the insertion of what we refer to as *orphans*. An orphan is the instance of a principal that does not have instances of principals that chain to it. Figure 4-2 expands on Figure 3-2 to show the query progression that could lead to an orphan being created then absorbed back into the instance access graph. This kind of query pattern is fairy common in the applications we investigated, especially if there is an auto_increment on the msgs table.

In the case of orphans, we make the assumption that they will be joined to the instance access graph quickly. In the example in Figure 4-2, CryptDB would create the principal instance for msg 11, generating the necessary keys and holding them in memory like any other principal's keys. Then the message text 'Long time no see' is encrypted with the message 11 keys and inserted in the database. Once the insertion to priv_msg_info gives CryptDB the information that msg 11 is spoken for by userids 2 and 3, msg 11's key is encrypted with userid 2's key and with userid 3's key and inserted into *access_keys*. If userid 2 and userid 3 are both logged off at some point after this insertion occurs, msg 11 is no longer an orphan and thus its key is removed from local memory, and userid 2 or userid 3's keys need to be available to again gain access to msg 11. However, until that insertion into *access_keys* occurs, msg 11 is an orphan, and there is no way to remove it from local memory, so the data it encrypts is vulnerable. The assumption that the instance access graph will generally be well-formed (that is, that orphans will only be orphaned for short periods) is based on the idea that for a normal access control scheme, there would be no reason to leave the orphan inaccessible. If an application does not properly implement a secure access control scheme, and does not reintegrate all its orphans into the access graph, those orphans and any keys they can chain to will remain accessible, since they are not



```
INSERT INTO msgs VALUES (11, 'Long time no see');
```

```
INSERT INTO priv_msg_info VALUES (11, 2, 3);
```

Figure 4-2: Adding an orphan msgid to the private messages instance access graph.

owned by any external user, and thus can never be offline.

4.4 Optimizations

In Section 3.1, we mentioned that keys are pre-fetched when an external principal logs on. While this is the case for the internals of the key chain, there are some situations where a user is unlikely to use certain keys, and it is a waste of time to fetch them, and a waste of space to store them. For example, in the private messages example outlined in Figure 3-2, a user can own thousands of msgid instances, but is unlikely to want to view every single message in a session. Thus we have an additional *threshold* parameter that limits the number of instances of a given principal type that are pre-fetched. If the number of keys of a principal type is greater than the threshold, CryptDB does not fetch those keys, but records that those keys should be online, and the principal who speaks for them. Thus if a key that is online but not pre-fetched is requested, it can be decrypted in a single database query. The threshold only applies to principals that don't speak for any other principals; we don't want to have any chaining required when keys beyond the threshold are fetched, as that would negate the value of pre-fetching keys in the first place.

4.5 Algorithms

KeyAccess has five main functions that maintain the instance access graph and provide keys to the main CryptDB proxy. The algorithms used in KeyAccess are described below. We use the variable type Principal to refer to a principal instance.

insertPsswd(Principal external, password). The purpose of this function is to record that *external* is online, and to pre-fetch *external*'s keys. We add *password* to the saved keys, since this is the symmetric key for the principal *external*. If *external* is a new principal, we merely generate asymmetric keys for *external*, store them in the *public_keys* table, and return. If *external* is not a new principal, all the keys that *external* can chain to need to be loaded. To do this, we do a depth first search of the principal access graph to acquire a list of all the principals is possible for *external* chain to. In most of the systems considered, this will be list of all the principals, but the system is designed to allow for systems that may have multiple types of external principals. We cycle through this list to find all the possible access links (by querying the principal access graph). For each possible access link, we query the database to determine if that link exists in the instance access graph. If so, we check to see how many keys this link will generate. If there are more than our *threshold*, we record that these keys have not been loaded, and move on to the next link. Consider a link in the instance access graph to consist of two principal instances, which we designate speaks_from and speaks_for. In accordance with the convention described in Section 1.1, speaks_from refers to the principal in the link that is closer to *external*, and speaks_for refers to the principal that is further away from *external* in the access graph. If there are fewer keys than *threshold*, we decrypt the key's speaks_for principal's symmetric key and store it. Since the list was generated by a depth first search on the access links, there is no way for a speaks_for principal to be reached before we have stored its speaks_from principal's symmetric key. That way, every speaks_for key required we are attempting to load can be decrypted, either using one of the speaks_from keys we hold already, or acquiring the speaks_from principal's private symmetric key from the database.

insert(Principal speaks_from, Principal speaks_for). This function adds a link to the instance access graph. If *speaks_from* is already online (ie, KeyAccess holds its symmetric key), inserting the new link is straightforward. If *speaks_for* is a new principal, we generate a new symmetric key, and a new asymmetric key pair for it and store the asymmetric pair in the *public_keys* table. If *speaks_for* is not a new principal, then by assumption, its key must be available to make the new link. In either case, we now hold both *speaks_from*'s key, since we are considering the case where *speaks_from* was already online, and *speaks_for*'s key, since it was either already online, or has just been generated. We encrypt the *speaks_for* symmetric key with the *speak_from* symmetric key and store the link and key in the *access_keys* table. If *speaks_from* is not online, but does exist, we acquire or generate the *speaks_for* key, encrypt the *speaks_for* key asymmetrically using the *speaks_from* asymmetric public key, and store that encrypted key and the new link in *access_keys*.

If *speaks_from* does not exist in *public_keys*, it is a new principal and an orphan. We generate and store the symmetric and asymmetric keys for *speaks_from* (the asymmetric keys being stored in the database, the symmetric key in memory since orphans cannot be offline). We acquire or generate the *speaks_for* key, encrypt it with *speaks_from*'s key and store it with the new link in the *access_keys*. Since *speaks_from* is an orphan, any keys it chains to must be online, because there is no way to root its chain. Thus, we also store the *speaks_for* key in memory. Though we make the assumption that orphan keys will not persist, there is no way of knowing how large an access subtree will be connected to them before they are re-integrated into the instance access graph, so we need to store the shape of an orphan-based access tree in memory. In the case where *speaks_from* is an orphan, insert ends with updating these orphan subtree maps.

If *speaks_from* is an already existing orphan, the algorithm is the same as if *speaks_from* was already online, except that at the end, the orphan maps must be updated with the new link. If *speaks_from* is a known principal, and *speaks_for* was an orphan before this link was added (for example, the insert statement in Figure 4-2), we go through the algorithm as described for *speaks_from* being online, but at the end remove *speaks_for* from the orphan maps so that *speaks_for* is only online if *speaks_from* is online.

removePsswd(Principal external). This function logs an external principal off, meaning that we need to forget all the keys which can only be chained to by *external*. To facilitate determining the subtree that needs to be removed, each key is stored in memory with a map to a set of principals that speak for it. To remove *external*, we order the subtree rooted at *external* using a breadth first search, then walk through it, removing *external* and any other principals whose keys will be deleted from the sets of principals that speak for each key. If a key has an empty set of principals that speak for it, the key is marked as deleted. Since the order of the walk through the subtree is determined by a breadth first search, a key will never have a principal that speaks for it deleted after the key is examined. Once we have a marked complete set of keys deleted, we remove those keys from memory.

remove(Principal speaks_from, Principal speaks_for). This function removes an access link from the instance access graph. We remove *speaks_from* from the principals that speak for *speaks_for*. If no other online users can chain to *speaks_for* (that is, *speaks_for*'s principals that speak for set is empty), we call removePsswd(*speaks_for*), which removes the subtree dependant on *speaks_for*.

getKey(Principal principal). This function returns the symmetric key for *principal* if there is an external principal online who can chain to *principal* online. We first check our local key map. If the key is there, it is returned. If the key is not in the local map, we check to see if we have seen *principal* before. If *principal* is unknown, then it is a new orphan, so we generate keys for it, store the asymmetric keys in the database, store the symmetric key in the the local key map, and return the symmetric key. If the key is not an orphan, we check to see if it is online, but has not been pre-fetched (if there were more keys of this principal type than we wanted to store in memory). If the key is one we can chain to, but has not pre-fetched, we fetch it from the database, decrypt it and return it. If there is no external principal online who can chain to the key, we return an error.

4.6 Changes to Application Code

CryptDB is intended to be minimally invasive to the application code. As discussed in section Section 3.2, the programmer must add annotations to the schema creation. There is also the question of logging in and out. Multiple Principal CryptDB is intended for the multi-user application with some sort of internal access control policy. As such, it is assumed that application code will include some sort of user authentication section. To properly use CryptDB, the programmer must add to the authentication code an additional database query which inserts the external principal and its secret key into the *cryptdb_users* table.

Chapter 5

Auditing

We have seen that Multiple Principal CryptDB protects the data of offline users during an attack. However, the data owned by users who were online for any portion of the attack is vulnerable, and there is no way to determine if it was compromised. Thus in the system described, after an attack, we must assume any data owned by users who were online for the attack has been compromised. For a large system, where there are thousands or millions of users, the set of data considered compromised in this model would massive. To constrain the set of compromised data and allow a useful post-attack analysis, we propose to extend KeyAccess to keep a log of all the keys that it provides to CrytpDB. In this way, if an attacker causes the application to issue arbitrary queries, KeyAccess will record all the keys used during the duration of the attack. After the system has been repaired, an examination of the log will allow the application maintainers to determine which keys were accessed, and thus narrow down the set of compromised data.

Having KeyAccess as described in Chapter 3 log key accesses is only a useful solution to this problem if the attacker gains control only of the application (through cross-site scripting, or falsifying its credentials, or some other method), without compromising the machine on which CryptDB is running. However, we wish to allow this kind of useful auditing to be performed after any sort of attack, even an attack on CryptDB or the machine it is running on. With this goal, CryptDB cannot store the log on its own machine, nor can any logs written by CryptDB in the case on an attack be trusted. In the same way that Multiple Principal secures offline users' data by using the users' private password as the key to accessing their data, we propose a distributed version of KeyAccess. Each user (external principal) runs External KeyAccess as a background process. When the user is logged on to the application, its External KeyAccess communicates with the main Server KeyAccess. Keys owned by an external principal are stored by its External KeyAccess, and Server KeyAccess must request every key from the relevant External KeyAccess. The External KeyAccess programs are responsible for logging all accesses to the keys they store, so an adversary will not be able to corrupt the KeyAccess log unless all the users' machines are also attacked.

5.1 Design

For our implementation of a distributed KeyAccess, we have three types of KeyAccess communication with each other: CryptDB, Server, and External. The CryptDB KeyAccess is simply a client stub that does little processing. The Server KeyAccess does most of the CryptDB-side processing. These two are separated so that in a system where it is not possible to run a fully distributed KeyAccess, the Server KeyAccess can be run on a separate machine from the main part of CryptDB, at least partially sandboxing the log from any attack on the application or CryptDB.

Figure 5-1 shows an overview of the distributed KeyAccess design. Keys of online external principals are stored by the External KeyAccess running on that principal's machine. The principal access graph, which is not sensitive, and which all the External KeyAccesses need to access, is stored by the Server KeyAccess. The database does not need to be on the same machine as the Server, but the Server provides wrapper functions for all database accesses, so their users do not need to be aware of any of the details on the database implementation. As described in Section 3.1, we know that more than one external principal can own data and thus an access key can also be owned by more than one external principal. In distributed KeyAccess, each external can hold the entire instance access subgraph that is rooted at that external, and all the keys for the principals in that instance access subgraph. While this increases overall duplication and thus the local memory required by the entire system, keys are relatively small, so no individual machine should be overwhelmed.

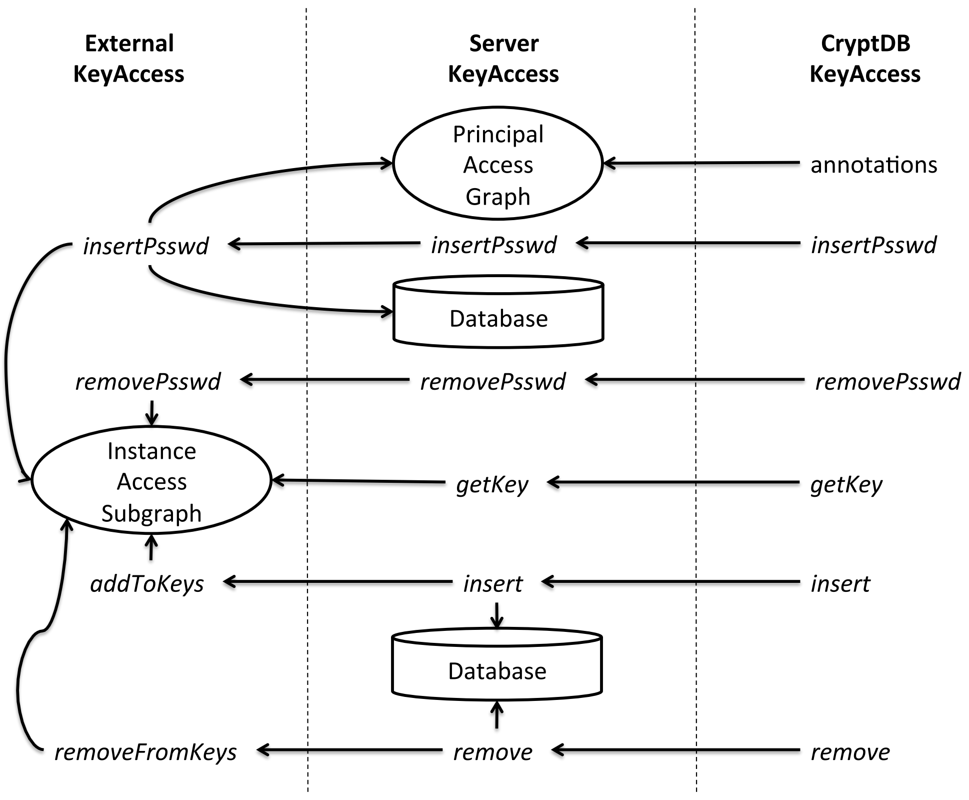


Figure 5-1: Distributed KeyAccess Design. The ovals represent data structures stored in memory for the given program. There is only one database; it is replicated in the image for clarity. The arrows represent the direction of function calls.

If the Server KeyAccess is compromised, as it would be in the case of attack, all logged on keys are still accessible to the attacker, but each access will still be logged with each External KeyAccess. A major change to KeyAccess in the distributed system is that CryptDB KeyAccess and Server KeyAccess insertPsswd no longer take the users' password as an argument. Thus if the Server KeyAccess is compromised, and a user tries to log in, the Server KeyAccess cannot log the external principal in without calling that users' External KeyAccess insertPsswd. If KeyAccess is compromised, the principal access graph information it passes to an External KeyAccess may be erroneous, but since the instance access graph is stored on the database with the keys to each link in an encrypted form, at worst, an adversary creating a false principal access graph will only cause a user who logs in during an attack unable to access a subset of their data. All forms of CryptDB are designed to restrict the data an attacker can view, so the attackers' ability to restrict the data accessible to the external principal is not within the scope of CryptDB's threat model.

5.2 Implementation

For our implementation of distributed KeyAccess, the CryptDB KeyAccess is approximately 200 lines, the Server KeyAccess 2,300 approximately lines, and the External KeyAccess approximately 800 lines. We used Delta V Software's Remote Call Framework [4] to communicate between the various versions of KeyAccess.

5.2.1 Orphaned Keys

Orphaned keys, as described in Section 4-2, remain a challenge in the distributed design. Because they by definition are not owned by any external principal, they cannot be stored in an External KeyAccess. As a result, Server KeyAccess has an additional data structures (not shown the Figure 5-1 for clarity) that hold the orphaned keys and all of their access chains. When orphans are reintegrated into the instance access graph their keys are removed from the Server's local storage and transferred to the relevant External.

5.2.2 Algorithms

In the distributed system, the five main KeyAccess functions described in Section 4.5 are transformed into a series of exchanges between the Server and various External versions of KeyAccess, as shown in Figure 5-1 and described below. Each of these exchanges is triggered by the CryptDB calling one of the normal functions, which the CryptDB KeyAccess forwards to the Server KeyAccess. The rest of CryptDB's behaviour is unchanged regardless of whether it is running the standard or distributed KeyAccess.

insertPsswd(Principal external, ext_address). The Server KeyAccess receives *Principal external*, and *ext_address*, the address of new external, as arguments. It stores the *ext_address*, then forwards the request to the External KeyAccess. The External KeyAccess records the address of the Server, then uses the standard *insertPsswd* algorithm from KeyAccess, querying the Server for database and principal access graph information. Once the External KeyAccess belonging to the newly logged in user has gathered all of the keys that external principal can chain to, it returns the set of principals whose key it now holds to the Server. The Server then records which keys the newly logged in external holds, and wakes up the CryptDB KeyAccess.

Server::removePsswd(Principal external). Removing an external principal is a two stage process. While each External can store the entire instance access subgraph rooted at itself, there is no need for all the keys to be replicated across all the Externals who speak for them, so it is possible for the system to be in a state where the user who is logging off is currently the only external principal holding a given key, but is not the only online user who can chain to it. Figure 5-2 illustrates how such a situation could form and be resolved, using the schema described in Figures 3-1 and 3-3.

In order to correctly resolve this situation, the Server first issues a *removePsswdPrep* request to the External KeyAccess for *external* (in the example in Figure 5-2, *external* would be username='alice'). *removePsswdPrep* returns a list of all the principal keys which, from the point of view of *external*, should be deleted in the course of this *removePsswd*. In the example, alice's *removePasswordPrep* would return {username='alice'}. Since the Server KeyAccess knows that alice also chained to {groupid = 5, forumid = 2, forumid = 3}, those

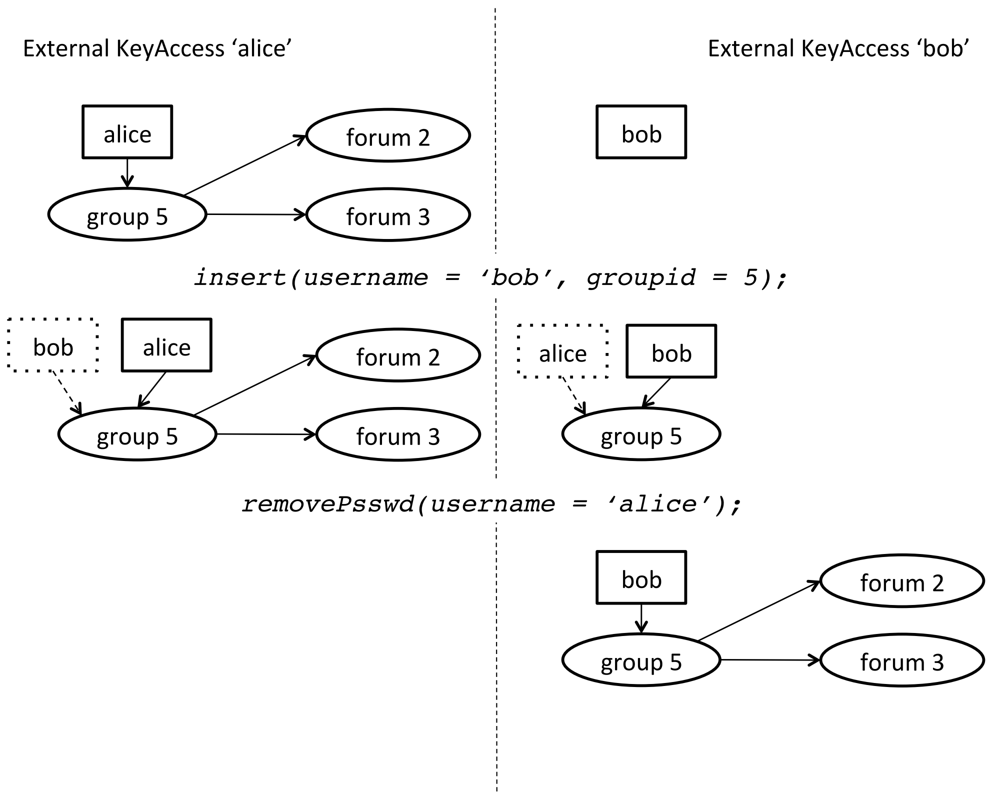


Figure 5-2: Distributed Key Transfer. Principals and links with dotted lines refer to principal keys and links that the External in question is aware of the existence of, but does not hold. For example, in the section between the insert and removePsswd, the External KeyAccess for alice does not how bob's symmetric key, but does record the link from bob to groupid 5.

keys must be accessible by some other user who is online, and they should be transferred to that user. As described in Section 4.5, keys are stored with a set of the principals who speak for them, so using this set, the Server KeyAccess can determine that username='bob' also holds the key for groupid = 5, so the subtree can be transferred to the External KeyAccess for 'bob'. Once the subtree is transferred, and all of the keys that can still be chained to are stored in different Externals, the Server sends *removePsswd* to the External KeyAccess for *external*, which at this point actually deletes all the keys it holds.

insert(Principal speaks_from, Principal speaks_for). The Server processes the *insert* using the same algorithm described in Section 4.5, except that instead of adding keys to a local map, they must be distributed to the appropriate Externals. In the case where *speaks_from* is an orphan (either a new orphan or an already existing one), both *speaks_from* and *speaks_for* keys are stored locally in an orphaned_keys map that is similar to the standard KeyAccess keys map, and the orphan access graphs are stored as usual.

Non-orphaned keys are not stored locally, as we do not want Server KeyAccess to be able to use those keys without the usage being written to an External KeyAccess log. To determine which Externals need to know about the newly inserted link, we take the union of the set of Externals that are currently storing *speaks_from* and the set of Externals that are currently storing *speaks_for*. We need to alert the Externals storing *speaks_from* since they now can chain to the *speaks_for* key, and if that is a new key, it needs to be stored somewhere. We need to alter the Externals storing *speaks_for* to handle a situation such as is shown in Figure 5-2, where *speaks_for* already has a subtree of online keys, which would be lost if its original owners was not aware of this newly inserted link. By making the the Externals that store *speaks_for* are aware of the new link, even though they are not aware of the *speaks_from* key, we can generate the dotted segments of the instance access subgraph shown in Figure 5-2, and thus transfer the keys on removePsswd.

It would be possible to avoid the situation in Figure 5-2 by fetching all the keys *speaks_for* could previous chain to, and ensuring that they are replicated on all of the *speaks_from* External Principals. However, there is no way of predicting how large an instance access subgraph is (ie, how many keys would need to be broadcast to all the *speaks_from* External Principals), and our general paradigm is ensure that the unbounded latency is reserved for logging in and out. Therefore we choose to use the key transfer algorithm described in *removePsswd* rather than chaining and replication in *insert*.

remove(Principal speaks_from, Principal speaks_for). The Server KeyAccess determines which Externals hold the *speaks_for* key, and forwards the *remove* request to each of them. Each of these Externals removes the link from their instance access subgraph, and returns the Server a list of the keys which were deleted as a result of removing the link. The Server updates its own maps to reflect the current lists of which keys each External holds.

getKey(Principal principal). The Server checks to see if the requested *principal* is an orphan. If it is a new orphan, the Server generates the appropriate keys, stores them in the database, initializes the various orphaned keys maps for *principal* and returns the symmetric key. If *principal* is an orphan that already exists, the Server finds *principal*'s symmetric key in the local orphaned_keys map, and returns it. If *principal* is held by one of the Externals who are currently online, the Server queries an External which holds the key. When an External receives a request for a principal's key, it finds the key in its local map, logs the request, and returns the key to the Server, which propagates the key to CryptDB KeyAccess. If *principal* is known, but there is no External online holding its key, the Server returns an error.

5.2.3 Changes to Application Code

Beyond the annotations required for Multiple Principal CryptDB, running a distributed KeyAccess requires each end user to run their own version of External KeyAccess on their machine. In the current implementation, External KeyAccess is assumed to be already running at the time the user logs into the application, and the port and address of each user are hardcoded. Ideally, however, the application would be modified to begin the External KeyAccess process on the local machine, then authenticate with the dynamically determined port and address.

Chapter 6

Evaluation

6.1 Security Analysis

How secure would an application using Multiple Principal CryptDB be? In this section we analyze how CryptDB would be used with various applications, creating the plausible access graphs and determining the eventual encryption level of all the encrypted fields. Since this is the part of CryptDB that in actuality would require input from the programmer, all of the measurements are subjective. To determine sensitive fields, we examined the field names and what we could learn from the source code about their purposes. To build the access graphs we compared the foreign keys in the schema and examined the source code to figure out which columns referred to the same principals, and extrapolated from the sensitive fields and source code and documentation what the access control rules might be.

We analyze multi-user web applications intended for a wide variety purposes, examining whether or not it was possible to run them over CryptDB and what alterations to the application code would be necessary to do so, as Figure 6-1 illustrates. OpenEMR, which we examined, provided enough information in the documentation to make it possible to document the access graph, and the schema could be examined for sensitive fields, but unfortunately, the source code was extremely complex and convoluted, so we cannot state with certainty the number of annotations and code changes that would be require to effectively run it on top of CryptDB.

Figure 6-1 also shows the number of sensitive columns each application has, based on

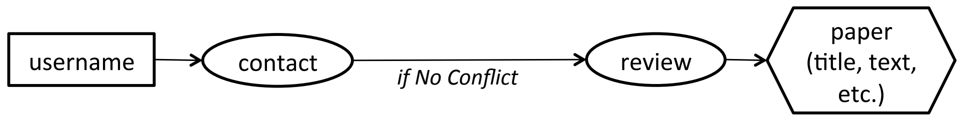
Application	**Annotations**	**Login & Logout Code**	**# Columns Total**	**# Sensitive Columns**	**# Columns at HIGH**
grad-apply	111	2 lines	706	103	95
HotCRP	29	2 lines	704	22	18
OpenEMR	–	–	1,297	566	526
6.02	15	8 lines	15	13	7
PHP-calender	17	2 lines	25	12	3
phpBB	31	7 lines	563	23	21

Figure 6-1: Application Changes and Encrypted Fields. The number of changes required to the application code is split into two types: Annotations and Login & Logout Code. Annotations are the annotations the programmer must add to the database schema, describing the principal access graph. Login & Logout Code refers to the additional query that needs to be sent to CryptDB to trigger *insertPsswd* or *removePsswd*. The sensitivity of fields was determined by an analysis of the source code, schema, and documentation (if it existed). HIGH refers to a security level at RND, HOM, or SEARCH, and applies to the lowest possible security level.

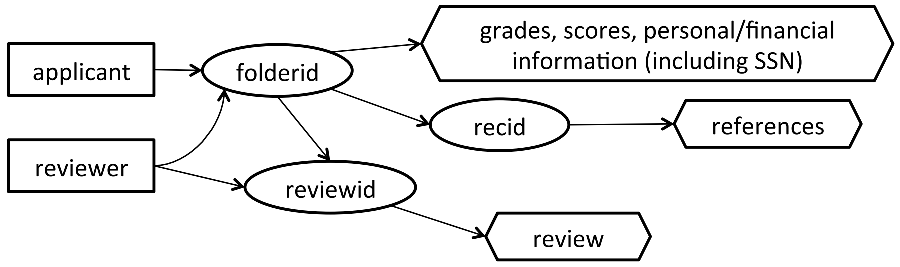
our analysis. Since an application does not issue random queries to an application, but a finite, pre-defined set, we can define the minimum security level each columns' onions could be at, regardless of how long the application runs. The right-most column in Figure 6-1 shows how many of the encrypted columns for each application have a minimum security level of HIGH. A HIGH security level means that the onions that column will always be at their outermost layer.

HotCRP [10] is a popular conference review application. Figure 6-2a shows its the principal access graph. An important parts of the HotCRP access policy is that program committee (PC) members not have the ability to review their own papers, or papers which it would be a conflict to review. However, all PC members can touch any paper in the database and HotCRP has no infrastructure in place to deter a curious or malicious PC member. Using CryptDB, we can build the access graph with the NoConflict predicate on the SPEAKS_FOR annotation. When implementing Multiple Principal CryptDB for HotCRP, the programmer would add the necessary predicate to the SPEAKS_FOR annotation, and provide some SQL function that defined NoConflict.

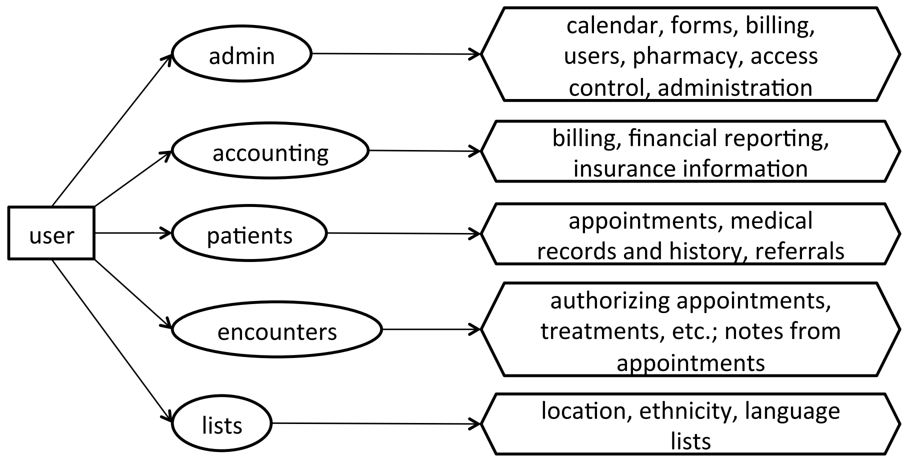
grad-apply is the admissions system for MIT's EECS department. It defines two different types of external principals: applicants and reviewers. It is the only application we examined with multiple external types, which makes it an interesting case, but it is still straightforward to integrate CryptDB in the grad-apply access control policy. Applicants



(a) HotCRP


applicant
reviewer
folderid
grades, scores, personal/financial information (including SSN)
recid
references
reviewid
review

(b) MIT grad-apply


user
admin
calendar, forms, billing, users, pharmacy, access control, administration
accounting
billing, financial reporting, insurance information
patients
appointments, medical records and history, referrals
encounters
authorizing appointments, treatments, etc.; notes from appointments
lists
location, ethnicity, language lists

(c) OpenEMR

Figure 6-2: The principal access graphs for three applications evaluated. In this representation, encrypted fields (hexagons) are shown as single objects for each principal, but in reality a single hexagon can refer to many fields. See Figure 6-1 for details on the numbers of encrypted fields per application.

should only be able to access their own application, whereas reviews need to access exactly the applications they are assigned to review. Figure 6-2b outlines the principal access graph for grad-apply. The duplicate path from reviewer to reviewid would be left out of an actual annotation implementation, since reviewers are only able to view the review the reviews of applications they have been assigned to read.

OpenEMR is an open source medical database. It stores information about patients' medical histories, insurance policies, and scheduled appointments. Though it has less defined access policies, OpenEMR is designed to store extremely sensitive data, so we consider it a good case to examine for our system. Figure 6-2c shows an outline of the access graph for OpenEMR. This presents an interesting a distinct case from phpBB, as OpenEMR uses phpGACL, a php-based access control program for access control analysis, rather than storing extra access control columns or tables. The implementation of phpGACL that OpenEMR defines creates the five main groups, and specifies which data each group can access. To build OpenEMR on top of Multiple Principal CryptDB, a programmer would need to instrument their embedded phpGAGL version primarily, since the external principal user and the encrypted fields are the only parts of the access graph that are part of the OpenEMR specific part of the code. OpenEMR also has seven fields that our analysis leads us to believe should be encrypted, but which the current implementation of CryptDB could not process. The operations in question where substring searches (not keywords searches) over the encrypted field. The currently SEARCH scheme could not be extended to generically search for any substring, but if the programmer knew what sort of substrings he would be searching for, he could re-implement SEARCH encrypting the specific papers that he is interested in, rather than by the usual key and word deliminators.

6.02 is an application for an MIT course website, that provides an interface to students and course staff. Course staff can update their own status' and students' grades. Students can view the status of course staff, and view only their own grades. While the access graph for 6.02 is not particularly novel (users speak for statuses and grades), 6.02 is an interesting application to examine because it is written using Python's MySQLdb library rather than PHP. As we can see from Figure 6-1, integration is straightforward. Of the fields we considered sensitive, the most sensitive field was that of the students grades, which did remain at HIGH. The other sensitive fields referred to values like the assignment names, grader name, etc.

PHP-calendar is an open source calendar application. Like 6.02, it has a straightforward access graph, with calendars as external principals that speak for various events. Like OpenEMR, PHP-calendar has two fields which our analysis led us to believe should be encrypted, but which had unsupported queries performed on them. In the case of PHP-calendar, both these fields where of the MySQL type DATE, and required type-specific access. The current implementation does not encrypt DATE types in such a way as to allow independent accesses to year, month, and day, as can be done with an unencrypted version of MySQL. However, if such a functionality were important to an application, CryptDB could be extended to handle these specific types and requests. Like 6.02, PHP-calendar has a fairly low percentage of its encrypted fields remaining at HIGH security (only 3/12). However, of the sensitive fields, we consider the most sensitive to be the subject, and descriptions of events, and the user's username and password. With the exception of subject, all of these most sensitive fields are at HIGH.

phpBB is a common open source web forum application, which allows administrators to have extremely fine-grained access control over the permissions. The forum design also includes an apparatus for members to exchange personal messages. The examples in Chapters 3 and 4 are simplifications of phpBB's implementation, with users being able to access posts in forums depending on their groups and private messages being shared between two or more users.

6.2 Performance Analysis

Our evaluations were conducted between two computers. The MySQL 5.1.54 server, the CryptDB proxy, and the phpBB server were all run on a single Intel Xeon 8-core machine with 2.4 GHz CPU and 12 GB of RAM. Although our processor has multiple cores, we enable only one core for most of our experiments to avoid measuring any effects due to parallelism (or lack thereof). The MySQL clients were running on a AMD Opteron Processor 8431 machine with 2.4GHz CPU and 64GB of RAM and connected to the server across a

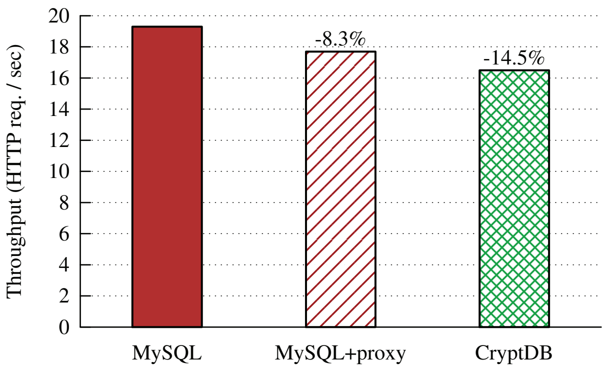


Figure 6-3: Throughput. (Originally appears as Figure 14 in [12]). MySQL refers to the throughput of the application running directly on the MySQL server, MySQL+proxy to the throughput of phpBB connecting to an unencrypted database, and CryptDB to phpBB running fully through CryptDB on the partially encrypted database described in 3-3.

shared Gigabit Ethernet network. The clients, while all running on separate processes, were being run from the same machine. When we evaluated the auditing system, the CryptDB KeyAccess process and the Server KeyAccess process were run on the same machine as the application server, and the External KeyAccess processes were run on the client machine. Besides the clients generated for testing, phpBB requires two users (admin and anonymous), whose respective External KeyAccess processes were also run on the client machine. Private messages between users and forum posts were loaded into the database.

6.2.1 Throughput

To analyze the impact of Multiple Principal CryptDB, we measured the throughput of phpBB for a workload with 10 parallel clients, which ensured 100% CPU load at the application server. Each client continuously issued HTTP requests for reading and writing private messages and forum posts. Figure 6-3 shows the throughput demonstrated by phpBB in three different cases: communicating directly with an unencrypted MySQL database, communicating with an unencrypted database through MySQL-Proxy, and communicating with a partially encrypted database through the CryptDB proxy. We see that phpBB incurs a 14.5% loss of throughout using CryptDB, and that approximately half of that is through inefficiencies in MySQL-Proxy.

The workload each client was performing for the throughput test was: reading five private messages, reading five forum posts, writing one private message, and writing one forum post. The details of each of these requests in discussed in Section 6.2.2.

6.2.2 Latency

Beyond the throughput results, we analyzed the specific impact of Multiple Principal CryptDB using the comparative latency of five types of phpBB actions. The results of this test are shown in Figure 6-4, which shows how the latency for each request type changes as the number previously exists messages and posts in the system varies. As we can see from these charts, the number of keys CryptDB is required to hold does not have a significant performance impact. Multiple Principal CryptDB as described in Chapter 4 adds 5-26ms to the processing of each request. Including a distributed KeyAccess system, requests can take up to an additional 50ms. A further description of each workload is provided below.

Login visits the phpBB's login page, submits the users' username and password. This calls KeyAccess's *insertPsswd* function. Since the tests were run on pre-existing users, *insertPsswd* requires multiple database accesses to fetch all of the users' keys. The overall latency impact was only 2-6ms. Login was not part of the throughput test workloads.

Read Message visits the phpBB User Control Panel, navigates to the users' inbox, and reads the first message in the inbox. For the latency tests, only the final message read was timed. Reading a message calls KeyAccess's *getKey* function, and the requires CryptDB to decrypt the text of the message. The latency increase for reading messages was 19-21ms for standard Multiple Principal CryptDB, which is approximately 30%. A distributed KeyAccess system added an additional 5ms to the latency.

Read Post visits the forum index page, chooses the first forum, views that forums main page, and then views the first post in the forum. Viewing later posts in the forum did not significantly alter the results. For the latency tests, only the final post read was timed. Viewing a post, like reading a message, calls KeyAccess's *getKey*, and requires decryption. The latency increase for reading messages was 16-30ms for standard Multiple Principal CryptDB, which is approximately 44%. A distributed KeyAccess system added an additional 4-39ms to the latency.

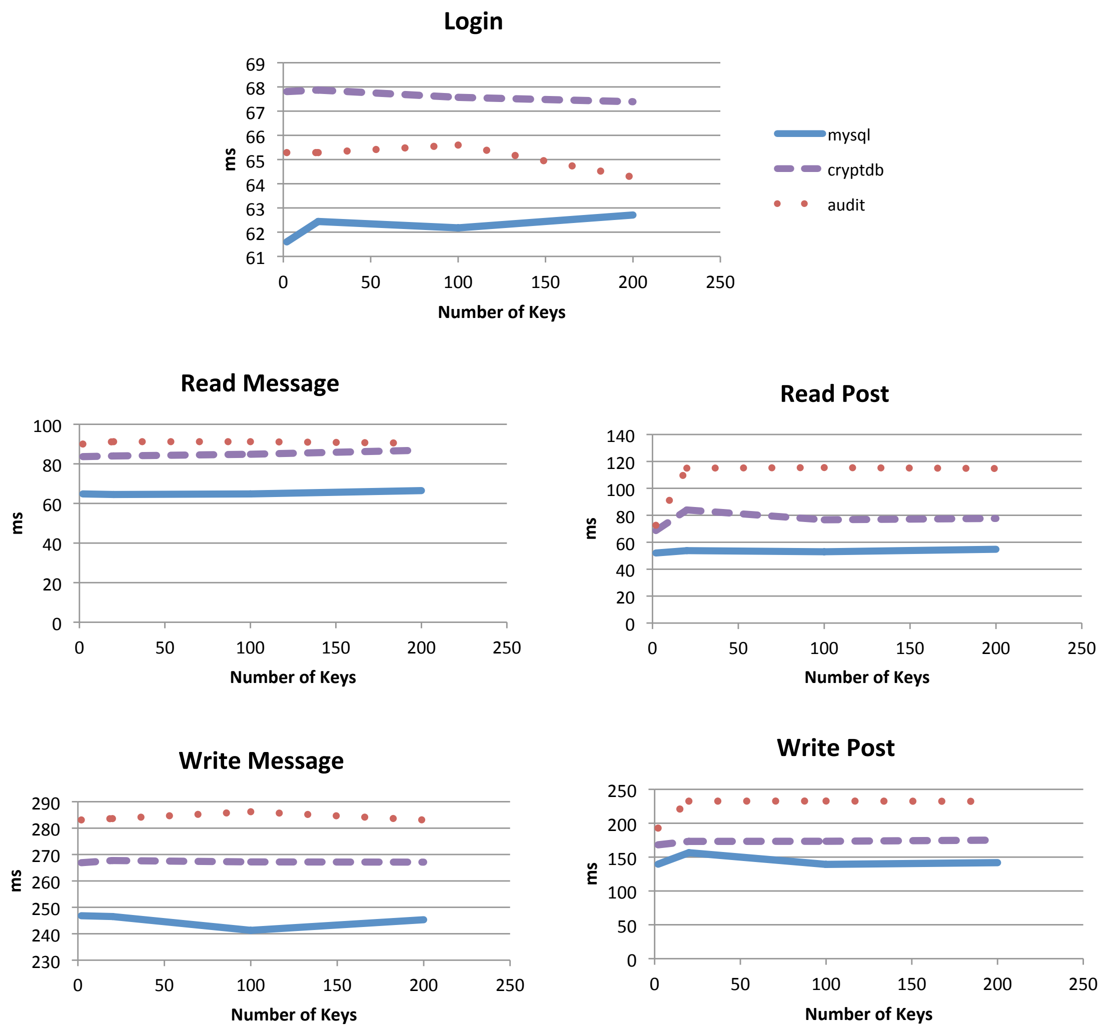


Figure 6-4: Latency. The latency of five common phpBB actions which touch on sensitive fields: logging in, reading a private message, reading a forum post, writing a private message, and writing a forum post. phpBB run directly on a MySQL server is shown in blue, on CryptDB without any auditing functionality in purple, and CryptDB with a distributed KeyAccess in red.

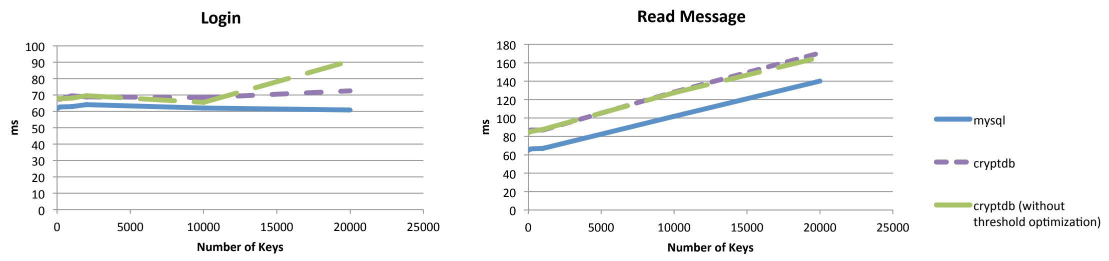


Figure 6-5: Threshold Optimization. The latency of logging in and reading a private message for phpBB run directly on a MySQL server (in blue), for phpBB run over CryptDB as described in Chapter 4 (in purple), and for phpBB run over CryptDB with the threshold optimization removed (in green).

Write Message visits the User Control Panel, navigates to the Compose Message page, fills out and submits the message form, and confirms that the draft is correct. The entire drafting and submission was timed for the latency tests. Writing a private message calls KeyAccess's *getKey* to create the message orphan key, then *insert* to add the new key to the access graph. The latency increase for reading messages was 20-26ms for standard Multiple Principal CryptDB, which is approximately 9%. A distributed KeyAccess system added an additional 16-19ms to the latency.

Write Post visits the first forum from the index page, and adds a new post to it. For the latency tests, only writing the post is timed. Writing a post calls KeyAccess's *getKey* to create the post orphan key, then *insert* to add the new key to the access graph. The latency increase for reading messages was 17-34ms for standard Multiple Principal CryptDB, which is approximately 18%. A distributed KeyAccess system added an additional 25-60ms to the latency.

6.2.3 Threshold

To test the relevance of the optimization proposed in Section 4.4, we ran the Read Message workload on extremely large pre-inserted datasets (up to 20,000 private message keys to be fetched) on phpBB running on MySQL, on CryptDB as described in Chapter 4, and on a modified version of CryptDB which pre-fetches all keys on *insertPsswd*. When running CryptDB with the threshold, we set the threshold variable to be 100. Figure 6-5 shows the results. Since the inbox is stored in reverse chronological order, our workload always reads the last message, for so CryptDB running with a threshold, the Read Message test on datasets with more than 100 messages will require an additional database query in *getKey* to fetch the uncached message key. This additional latency is 0-4ms (>1%). While we make the assumption that users are more willing to wait on login than on other operation, the results show that fetching all the keys (CryptDB without threshold) on login for large datasets can add up to 30ms (49%) latency.

6.2.4 Storage

Multiple Principal CryptDB adds to the data stored on the database in two ways. First, the use of onions replicates encrypted data up to three times, and sometimes ciphertexts are large than their respective plaintexts (depending on the encryption scheme used, and thus on the security level). Second, KeyAccess stores *access_keys*, the instance access graph and associated keys, and *public_keys*, the principal's public keys on the database. A phpBB database after running a workload which generated about 1,000 private messages and 1,000 forum posts was 2.6MB. The database for phpBB run on top of Multiple Principal CryptDB after running the same workload was 3.3MB, which is an increase of about 1.2x. Of the 0.7MB increase, 230KB was for *access_keys*, 276KB was for *public_keys*, and 166KB was from the expansion of encrypted fields.

In distributed KeyAccess, the log for an External principal is the length of the number of keys accessed during the execution of the program. Since External KeyAccess stores the set of keys requested in memory without duplication, and writes to disk when the set is full, a user-defined maximum time has passed (for our tests, we used 5 minutes), or when the *removePsswd* is called, for our tests, log sizes were minimal.

6.3 Analysis of Supported Queries

CryptDB cannot handle all queries, and Multiple Principal key-chaining can put further restrictions on the supported queries, some of which are mentioned in Section 6.1. Figure 6-6, shows an analysis of which queries Single Principal CryptDB supports, and whether any of those queries touch on highly sensitive fields in tables. In some ways, Multiple Principal

Application	**Encrypted columns**	**Encrypted columns touched by unsupported queries**
grad-apply	103	0
HotCRP	22	0
OpenEMR	566	7
6.02	13	0
PHP-calender	12	2
phpBB	23	0
TPC-C	92	0
Trace from sql.mit.edu	128,840	1,094
...col. name contains *pass*	2,029	2
...col. name contains *content*	2,521	0
...col. name contains *priv*	173	0

Figure 6-6: Unsupported Queries. (A subset of Figure 9 in [12].) This table shows the number of columns encrypted, and how many of those column are touched by queries CryptDB cannot currently support. TPC-C refers to the standard benchmark. The trace from sql.mit.edu was provided by MIT's SITB, and represents the queries processed by many of the database-backed applications running on scripts.mit.edu.

CryptDB is less restrictive than fully encrypting a database using Single Principal CryptDB, because the Multiple Principal case encrypts data sparsely, so it puts fewer restrictions on queries that do not touch highly sensitive fields. We see that unsupportable queries rarely touch encrypted fields in the common applications analyzed, and in the large trace that only .85% of all columns are touched by unsupported queries. Cryptographically, CryptDB cannot support complex mathematical queries over encrypted fields, such as sin, log, etc., due to the restrictions of the homomorphic encryption, and cannot support generic substring SEARCH. There are also more complex datatypes (such as DATE) that support for has not yet been implemented in CryptDB.

The Multiple Principal case, while decreasing the number of encrypted fields, and thus the possibility of cryptographic difficulties, also has further restrictions on types of queries it can process on encrypted fields. Unlike Single Principal CryptDB, Multiple Principal CryptDB cannot compute JOINs on encrypted fields. Multiple Principal CryptDB can have a different key for each row of an encrypted field, depending on the principal instance that speaks for the encrypted value, so the JOIN algorithm used in Single Principal CryptDB is not possible, since it requires the two encrypted columns being joined to each be encrypted with a single key. Of course, there is no way for a JOIN between an encrypted and unencrypted column to be performed – to make them comparable, the encrypted column would have to be decrypted to plaintext on the database, which violates our security guarantees. However, as JOINs between encrypted columns, or between encrypted and unencrypted columns seem to be rare (they never occur in the applications we examined), this restriction should not make Multiple Principal CryptDB impractical.

There are also a few restrictions to the types of UPDATEs Multiple Principal CryptDB can handle. Altering the principal access tree annotations provides challenges (as described in Chapter 4), but most fundamentally, altering, rather than inserting or removing, links from the instance access graph is not supported. Consider the private messages example from Figures 3-2 and 3-3. CryptDB cannot correctly process the query `UPDATE msgs SET authorid = 3 WHERE authorid = 2`. Since the key for msgid is currently encrypted for userid 1 and userid 2, to correctly implement this UPDATE, we would need to remove userid 2 SPEAKS_FOR msgid 10 and insert userid 3 SPEAKS_FOR msgid 10. While this is in theory possible, it is time consuming, and based on the applications examined, an unlikely case, so we do not support it. If an application required these kinds of UPDATEs frequently, CryptDB could be modified to transform them in the equivalent DELETE, INSERT pairs, but our current analysis indicates that these UPDATEs are not common.

As mentioned in Chapter 4, in Multiple Principal CryptDB, we make the assumption that when a link in the access chain is inserted, the speaks_for principal (assuming we are not in the case of orphans) is online. From a ideological point of view, this is the assumption that to add a permission to an access control schema, someone with the permissions to access the schema in question will be online. Should there a be an access control scheme that does not reflect this assumption, Multiple Principal CryptDB would probably not be the best choice for data protection.

Chapter 7

Conclusion

In this thesis, we discussed ways to extend the security and auditing capacities of CryptDB, using user passwords as the basis of key chains, and distributed logging of key requests to provide accurate information about compromised data during an attack. With the increasing popularity of third-party data storage in the cloud, cryptographically protected databases are a field that should be explored. CryptDB provided a practical solution to the standard relational database model, which Multiple Principal CryptDB builds on it to tailor security for multi-user web applications with well-defined access control policies. In this thesis, we show that with reasonable overhead and minimal application code alteration, an existing application can be configured to run through a CryptDB proxy that encrypts the sensitive data in such a way that an attacker cannot decrypt the sensitive data unless one of its owners is online. We also provide a secure system for determining a reasonable set of which data could have been compromised in the event of an attack.

Bibliography

[1] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Information Security Practice and Experience*, Sydney, Australia, April 2008.

[2] A. Chen. GCreep: Google engineer stalked teens, spied on chats. *Gawker*, September 2012. http://gawker.com/5637234/.

[3] M. Cooney. IBM touts encryption innovation; new technology performs calculations on encrypted data without decrypting it. *Computer World*, June 2009.

[4] Delta V Software. Remote Call Framework. http://www.deltavsoft.com/index.html.

[5] A. Desai. New paradigms for constructing symmetric encryption schemes against chosen-ciphertext attack. In *Proceedings of the 20th Annual International Concference on Advances in Cryptology*, pages 394-412, August 2000.

[6] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, Bethesda, MD, May-June 2009.

[7] O. Goldreich. *Foundations of Cryptography: Volume I Basic Tools* Cambridge University Press, 2001.

[8] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Usenix Security Symposium*, San Jose, CA, July-August 2008.

[9] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Advances in Cryptography (CRYPTO)*, 2003.

[10] E. Kohler. Hot crap! In *Proceedings of the Workshop of Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.

[11] P. Paillier. Public-key cryptosystem based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Prague, Czech Republic, May 1999.

[12] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with Encrypted Query Processing In *Processings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[13] R. A. Popa and N. Zeldovich. Cryptographic treatment of CryptDB's adjustable join. Technical Report MIT-CSAIL-TR-2012-006, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 2012.

[14] R. A. Popa, N. Zeldovich, and H. Balakrishnan. CryptDB: A practical encrypted relational DBMS. Technical Report MIT-CSAIL-TR-2011-005, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, January 2011.

[15] 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*, Oakland, CA, May 2000.

[16] M. Taylor. MySQL Proxy. https://launchpad.net/mysql-proxy.