Breaking Web Applications Built On Top of Encrypted Data

Paul Grubbs
Cornell University
pag225@cornell.edu

Richard McPherson
UT Austin
richard@cs.utexas.edu

Muhammad Naveed
USC
mnaveed@usc.edu

Thomas Ristenpart
Cornell Tech
ristenpart@cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

ABSTRACT

We develop a systematic approach for analyzing client-server applications that aim to hide sensitive user data from untrusted servers. We then apply it to Mylar, a framework that uses multi-key searchable encryption (MKSE) to build Web applications on top of encrypted data.

We demonstrate that (1) the Popa-Zeldovich model for MKSE does not imply security against either passive or active attacks; (2) Mylar-based Web applications reveal users' data and queries to passive and active adversarial servers; and (3) Mylar is generically insecure against active attacks due to system design flaws. Our results show that the problem of securing client-server applications against actively malicious servers is challenging and still unsolved.

We conclude with general lessons for the designers of systems that rely on property-preserving or searchable encryption to protect data from untrusted servers.

1. INTRODUCTION

Many modern Web and mobile applications are built using the client-server architecture: users interact with the application's clients in users' browsers or devices, while the server is responsible for centralized storage and management of users' data. This design offers attractive scalability and performance but if the server is compromised, the attacker gains access to every user's data. Even if this data is encrypted at rest but decrypted when the server operates on it, it is potentially exposed to a persistent or lucky attacker.

Client-side encryption can mitigate the damage from server compromise by ensuring that the server only sees encrypted users' data and never decrypts it. Unfortunately, if the server acts as a "dumb" storage and communication medium, all operations on the data must be performed by the clients, sacrificing most of the advantages of the client-server model.

A new class of client-server systems aims to solve this conundrum [16,28,33,34,47,48,51,54,59]. We call these systems BoPETs ("Building on Property-revealing EncrypTion"). The main idea behind BoPETs is to encrypt users' data before uploading it to the server using special property-revealing encryption (PRE). The server can then execute its part of the application's functionality on encrypted data.

Our contributions. We develop a new approach for systematic security analysis of BoPETs. Unlike previous work which focused on PRE schemes in isolation [9,26,30,42], we take a more holistic approach to analyzing these schemes in the context of the systems where they are deployed.

We first define a taxonomy of real-world threats to the server: snapshot passive (a one-time, "smash and grab" snapshot of the server's state), persistent passive (observing all activity on the server but not interfering with its operations), and active attacks involving arbitrary malicious behavior. We then work backwards from these adversarial capabilities to models. This approach uncovers significant challenges and security-critical decisions faced by the designers of BoPETs: how to partition functionality between the clients and the server, which data to encrypt, which access patterns can leak sensitive information, and more.

We then apply our methodology to a recent BoPET called Mylar [48]. Mylar is an extension to a popular Web application framework called Meteor [39]. Unlike similar commercial systems [59], Mylar is open-source, enabling a thorough analysis. In Mylar, client data deemed sensitive is encrypted using multi-key searchable encryption (MKSE) [49], a PRE that supports sharing and keyword search at a low performance cost. The MKSE scheme at the heart of Mylar is accompanied by cryptographic proofs based on the formal model of security defined by Popa and Zeldovich in [49]. In [48], Popa et al. explicitly claim that Mylar is secure against actively malicious servers (excluding a particular type of passive attacks, as explained in Section 2.2).

We start by showing that the Popa-Zeldovich security definitions for MKSE do not imply the confidentiality of queries even against a passive server. Adapting a well-known counterexample from [17], we construct an MKSE scheme that meets the Popa-Zeldovich definitions but trivially leaks any queried keyword. This shows that the proofs of security for MKSE do not imply any meaningful level of security, but does not yet mean that the actual scheme fails.

We then go on to assess Mylar itself, using the implementation released by Popa et al. as our reference, along with four diverse Meteor apps: kChat (online chat), MDaisy (medical appointments), OpenDNA (analysis of genetic data), and MeteorShop (e-commerce). kChat was released with Mylar, the other three we ported with minimal changes.

We show that all four applications suffer from one or more attacks within each of our threat models—see the summary in Table 1. Even a "smash-and-grab" attacker can compromise users' privacy by analyzing the names, sizes, static link structure, and other unencrypted metadata associated with encrypted objects. A persistent passive attacker can extract even more information by observing the application's access patterns and objects returned in response to users' encrypted queries. In our case-study applications, this reveals users' medical conditions, genomes, and contents of shopping carts. Even if this leakage is inevitable, our approach helps guide investigation into its implications.

The most damaging attacks are application-independent and involve Mylar's search when the server—and possibly some of the users—are actively malicious. We describe two

Threat type	Description	Found attacks
Snapshot passive	Attacker captures a one-time snapshot of server	• kChat: names of principals leak information about chat room topics • MDaisy: metadata about relationships between documents leaks patients' medical information; size of users' profiles leaks their roles • OpenDNA: size of encrypted DNA leaks which risk groups the user is searching for
Persistent passive	Attacker records server operations over a period of time	• MDaisy: server can cluster patients by their medical procedures; if one patient reveals information, entire cluster is compromised • MeteorShop: users' recently viewed items leak what they added to their encrypted shopping carts
Active	Server can arbitrarily misbehave, collude with users	• **Any Mylar app: malicious server can perform brute-force dictionary attacks on any past, present, or future search query over any server-hosted content** • OpenDNA: malicious server can search users' DNA for arbitrary SNPs

Table 1: Threat models and the corresponding attacks on Mylar apps.

methods that a malicious server can use to obtain a user's keyword search token for a document under the server's control, along with the ability to convert it to other documents.

The first method involves the server forcibly giving the user access to a document. The Mylar paper suggests a mechanism for users to exclude documents from searches, but it's broken in the reference implementation. Fixing this problem runs into challenges that appear, for Mylar's intended collaborative applications, inherent and intractable. The second method involves the server colluding with a malicious user who shares a document with an honest user.

In both cases, an honest user ends up sharing a document with the adversary. If the user's keyword search matches this document, the adversary learns the keyword. This generic attack was described in [48], but not its full impact. In fact, Mylar explicitly claims to protect the confidentiality of the honest user's *other* documents, those that cannot be accessed by the malicious users. But honest users' searches are performed over *all* of their documents, thus the adversary learns partial information about documents to which the malicious users do not have access.

The above attack is generic for MKSE, but we dramatically increase its power by exploiting a basic design flaw in Mylar—the unsafe partitioning of functionality between the client and the server. For efficiency, Mylar trusts the server to convert search tokens. Given a keyword token for a document for which it knows the key, the server can "cancel" the client's secret key from the token, enabling an efficient dictionary attack which is fundamentally different from the generic attack mentioned in [48]. Most importantly, this attack only needs the search token and does not rely on the server's ability to observe matches of the user's queries to the existing documents. Even if Mylar completely hid all access patterns, perhaps by running on top of ORAM [24] or any other oblivious storage, it would remain vulnerable to our query recovery attack.

In our simulations for kChat with real-world chat logs, the dictionary attack recovers all user queries and nearly 70% of the keywords in all chats stored on the server. The attacks were experimentally confirmed using the publicly available Mylar and kChat codebase.

Our results show that the problem of securing client-server systems against persistent passive and active attackers on the server is very challenging and still unsolved. We conclude with general lessons for the designers of BoPET systems and outline open research problems.

2. BACKGROUND

2.1 BoPETs

We use the term BoPETs generically to refer to client-server applications that encrypt clients' data so that the server cannot decrypt it, yet rely on special properties of the encryption scheme to retain some of the application's original server functionality. BoPETs are based on one or more property-revealing encryption (PRE) schemes, which is the term we use for schemes that, in this setting, reveal to servers some plaintext properties to facilitate server-side processing over ciphertexts. PRE schemes include so-called property-preserving encryption (PPE) schemes whose ciphertexts publicly reveal plaintext properties, such as equality [6] and order [2, 7, 8], as well as encryption schemes that reveal plaintext properties only when given both a ciphertext and a token derived from the secret key—e.g., searchable encryption [17, 55] and multi-key searchable encryption [49].

BoPETs gained popularity in industry before they were studied formally by academics. As early as 2009, products from Ciphercloud [16] and Navajo Systems [41] were using hand-crafted encryption schemes to enable searching and sorting in software-as-a-service (SaaS) cloud applications. Newer entrants to this rapidly growing market include Perspecsys [46] and Skyhigh Networks [54]. Overnest's Gitzero [22] is a secure Git repo that stores encrypted code on an untrusted server and enables search using searchable encryption. PreVeil [51] is "a version of Mylar for certain kinds of applications" [58]. Kryptnostic [59] built Kodex, a collaboration and chat platform that supports document sharing and search on shared documents using MKSE. ZeroDB is an encrypted database system which, according to section 3 of their whitepaper [18], uses a proxy re-encryption scheme inspired by Mylar's MKSE to share content encryption keys between users.

The academic literature focused on PRE schemes as isolated primitives until BoPET systems such as CryptDB [47], ShadowCrypt [28], Mimesis Aegis [34], and Mylar [48] sought to incorporate PREs into complete client-server systems.

2.2 Mylar

Mylar [48] extends the Meteor Web application framework [39]. Meteor apps include clients and servers, both implemented in JavaScript. Meteor uses MongoDB [40] for server-side storage. MongoDB stores data in *documents* which are organized into *collections*. Each document consists of one or more key-value pairs called *fields*.

Principals. A *principal* in Mylar is a name and a public/private key pair that is used to encrypt and decrypt confidential data. Each principal is thus a unit of access control. In addition to the principals used to encrypt documents, every user has a principal.

The app developer specifies which fields in a MongoDB collection are confidential and which principals are used to encrypt and decrypt these fields. A principal may be used for only one document (e.g., in MDaisy, each appointment is encrypted by a unique principal) or multiple documents (e.g., in kChat, every message sent to a chat room is encrypted with the key of that room's principal). Unencrypted confidential data exists only in users' browsers and is ostensibly never sent to or stored on the server.

Mylar uses certificates to protect the binding between users' identities and their keys. The root of Mylar's certificate graph can be either a certificate from a trusted third-party identity provider (IDP) or a *static principal* whose keys are hard-coded into the Mylar application.

Mylar also includes integrity protections and code verification, but we omit them for brevity. Our attacks do not involve breaking these protections.

Sharing and searching encrypted data. If Alice wants to share encrypted data with Bob, she needs to give him the keys of the corresponding principal. To do this, Alice encrypts these keys with Bob's public key and uploads the resulting *wrapped keys* to the database. By downloading and decrypting them, Bob gains access to the principal.

A user may have access to multiple principals, thus Mylar needs to support keyword search over documents encrypted with different keys. A straightforward approach is to have the user submit a separate search token for each principal it has access to, but this is expensive with many principals.

For efficiency, Mylar relies on the server to generate tokens for searching over multiple principals. The user submits a single token, for documents encrypted with the user's principal. The server then uses a client-generated *delta* value to convert this token to another token for documents encrypted with a different principal. Whenever a user is given access to a new principal, their client automatically sends the delta associated with that principal to the server.

Threat model. Mylar claims to protect confidentiality of users' data and queries against actively malicious servers, including servers that collude with malicious users, under the assumption that honest users *never* have access to any document controlled by the adversary. As we explain in Section 8, this assumption is not enforced by the reference implementation of Mylar, nor do we believe that it can be enforced in realistic collaborative applications.

Mylar explicitly does not hide access or timing patterns, even though in any real-world deployment they are visible to an adversarial server. In Section 7, we show what these patterns reveal to a persistent passive attacker. In Section 8, we show how an active attacker can break the confidentiality of Mylar-protected data *without* exploiting access patterns.

3. SECURITY MODEL FOR MKSE

Mylar is based on a multi-key searchable encryption (MKSE) scheme invented by Popa and Zeldovich [49]. In [48], Popa et al. argue that Mylar is secure by appealing to the cryptographic proofs of security for this scheme.

In this section, we show that the theoretical definitions of security for MKSE proposed by Popa and Zeldovich [49] fail to model security even against a passive adversary. To this end, we construct an artificial scheme that satisfies their definition but trivially leaks all keywords queried by a user, which also reveals the corresponding plaintexts.

This shows that the proofs in [48, 49] are not useful for arguing any meaningful level of security. It does not (yet) imply that the Mylar MKSE scheme is vulnerable to attack. In the rest of this paper, we demonstrate practical attacks on Mylar when using this MKSE scheme as designed and implemented by Popa et al.

3.1 Mylar MKSE

A *multi-key searchable encryption* scheme (MKSE) allows efficient keyword search over encrypted keywords. We focus here on the construction by Popa and Zeldovich [49]. For simplicity assume that all keywords are of the same length, call it ℓ bits. The scheme relies on bilinear pairings. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups, all of the same order p. Let $n \in \mathbb{N}$ be a security parameter. We associate to these groups an efficiently computable pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ that enjoys the property that for all $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2, g_T \in \mathbb{G}_T$ and any $\alpha, \beta \in \mathbb{Z}_p$ it holds that $e(g_1^\alpha, g_2^\beta) = g_T^{\alpha\beta}$. The scheme also uses hash functions $H : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $H_2 : \{0,1\}^n \times \mathbb{G}_T \rightarrow \{0,1\}^{\ell+n}$, modeled as random oracles.

Figure 1 shows the details of the scheme. After generating parameters and keys (MK.Setup and MK.Kg), the client uses MK.Enc to (separately) encrypt each keyword of a document. To enable the server to convert search tokens for documents encrypted under different keys, the client generates an appropriate delta value (MK.Delta) and sends it to the server. To perform keyword search over encrypted documents, the client generates a token (MK.Token) and sends it to the server. The server uses the delta value to adjust the token (MK.Adjust) and determines if an encrypted document matches the keyword (MK.Match).

Note that MK.Match assumes an adjusted token. One can run MK.Delta on $k_1 = k_2$, allowing one to "adjust" a token generated for one key into an "adjusted" token for the same key. Correctness requires that: (1) for all keywords w, MK.Match(tk', c) returns 1 with probability overwhelmingly close to one if tk' is a search token for w and c is an encryption of w, and (2) for all keywords $w \neq w'$, MK.Match(tk', c) returns 0 with probability overwhelmingly close to one if tk' is a search token for w' but c is an encryption of w.

The scheme implemented in Mylar is a variant of the one described above, in which the same randomness r is reused for all keywords from the same document. Only one ciphertext is generated for each unique keyword, and ciphertexts are stored in the order in which the keywords appear in the document. This reuse of randomness does not seem to impact security relative to the Popa-Zeldovich definitions, although analysis would require some change in semantics to accommodate encrypting whole documents at once rather than individual keywords. We omit the details and focus on the simpler scheme shown in the figure. All attacks in the rest of the paper work regardless of which version of the scheme is used, except where mentioned otherwise.

Security definitions. Popa and Zeldovich introduced two notions of security for MKSE: data hiding and token hiding. We will only sketch them informally and refer the interested reader to the whitepaper [49] for the formal definitions.

- MK.Setup(1^n):
  return $\mathsf{pars} = (n, p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, g_T)$
- MK.Kg(pars): return $k \leftarrow \mathbb{Z}_p$
- MK.Delta(k_1, k_2): return $\Delta = g_2^{k_2/k_1} \in \mathbb{G}_2$
- MK.Token(k, w): return $H(w)^k \in \mathbb{G}_1$
- MK.Enc(k, w): Draw a random $r \leftarrow \{0,1\}^n$. Compute $c' = H_2(r, e(H(w), g_2)^k)$. Output $c = \langle r, c' \rangle$.
- MK.Adjust(tk, Δ): return $tk' = e(tk, \Delta) \in \mathbb{G}_T$.
- MK.Match(tk', c): Let $c = \langle r, c' \rangle$. Return 1 if $H_2(r, tk') = c'$ and 0 otherwise.

Figure 1: The MKSE scheme analyzed in [49].

Data hiding is formalized via a game involving a challenger **Ch** and an adversary $\mathcal{A}$. The game starts by having **Ch** run parameter generation. $\mathcal{A}$ then chooses an access graph that specifies which keys can have their search tokens converted to which other keys; $\mathcal{A}$ can choose any keys except one distinguished key k_0 generated honestly by **Ch**. **Ch** generates an adjustment delta for each edge in the graph, and gives these values to $\mathcal{A}$. Then $\mathcal{A}$ picks two keywords w_0, w_1, gives them to **Ch** and receives back encryption MK.Enc(k_0, w_b) for randomly chosen b. $\mathcal{A}$ can make adaptive queries to an encryption oracle (that uses k_0) and a search token oracle (for any key). It cannot, however, make a query to the search token oracle for w_0 or w_1 if it is for a key with an edge to k_0 in the access graph. This restriction is critical, as removing it leads to a vacuous definition (no scheme can meet it), but our counter-example below exploits it. The adversary outputs a guess b' and wins if it equals b.

Token hiding attempts to capture the desired security for keyword queries. The adversary $\mathcal{A}$ again generates an access graph with a special challenge user with key k_0. $\mathcal{A}$ picks all other keys. The challenger **Ch** then generates delta values for all edges in the graph and gives them to $\mathcal{A}$. Then $\mathcal{A}$ can make adaptive queries to an encryption oracle and search token oracle for any of the keys, as well as output a pair w_0, w_1 of keywords for which **Ch** returns MK.Token(k_0, w_b) for randomly chosen b. Throughout the game $\mathcal{A}$ cannot make an encryption query or search token query on w_0 or w_1 for keys that do not have a path to them from k_0. In words, the adversary can either perform queries on keywords unrelated to the challenge pair, or can query them but only for keys unrelated to k_0 via delta values. Finally, $\mathcal{A}$ outputs b' and wins if $b' = b$.

3.2 Counterexample

Popa and Zeldovich assume that if a scheme is both data-hiding and token-hiding, then no efficient adversary can distinguish encryptions of keywords or distinguish tokens of keywords (i.e., the outputs of MK.Enc and MK.Token, respectively) non-negligibly better than a random guess. In this section, we show that this is false.

The Popa-Zeldovich approach of using two distinct notions for data hiding and token hiding was previously considered for single-key symmetric searchable encryption (SSE) by Curtmola et al. in 2006 [17], building on a data-hiding definition from [23]. Curtmola et al. showed that achieving both data hiding and token hiding does *not* imply that a single-key SSE scheme hides queries. We adapt their counterexample in a straightforward way to the MKSE setting.

The counterexample version of the Mylar MKSE scheme has the same MK.Setup, MK.Kg, MK.Token and MK.Adjust algorithms. We modify encryption and matching as follows:

- MK.Enc$'(k, w)$: Draw a random $r \leftarrow \{0,1\}^n$. Compute $c' = H_2(r, e(H(w), g_2)^k) \oplus (w \| 0^n)$. Output $c = \langle r, c' \rangle$.
- MK.Match$'(tk', c)$: Let $c = \langle r, c' \rangle$. Return 1 if the bit string $H_2(r, tk') \oplus c'$ ends with ℓ zeros and return 0 otherwise.

Below, we prove that this scheme is secure according to the Popa-Zeldovich definitions. Yet, given a search token $H(w)^k$ and an encryption $c = \langle r, c' \rangle$ of a keyword (where $c' = H_2(r, e(H(w), g_2)^{pr}) \oplus (w \| 0^\ell)$), the malicious server can remove the pseudorandom pad and reveal the word w.

We conclude that the proofs of security for Mylar based on the Popa-Zeldovich MKSE model do not imply anything about the actual security of Mylar.

Correctness. The probability that MK.Match$'(tk', c) = 1$ is one when tk' is a token for a keyword w and c is an encryption of w. Now consider the probability of an incorrect match, where $tk' = e(H(w'), g_2)^k$ but $c = \langle r, c' \rangle$ with $c' = H_2(r, e(H(w), g_2)^k) \oplus (w \| 0^n)$. If H is collision resistant, then $H(w') \neq H(w)$ with all but negligible probability and, in turn, the probability that the low n bits of $H_2(r, e(H(w'), g_2)^k) \oplus c'$ are all zero is at most 2^{-n} assuming H_2 is a random oracle.

Data hiding. Intuitively, the modified Mylar scheme is data-hiding because H_2 is a random oracle, so the value c' that is XORed with $w \| 0^n$ acts like a pseudorandom one-time pad. Therefore, the only way to distinguish the two challenge keywords is to run MK.Match and see which challenge keyword-ciphertext it matches. According to the Popa-Zeldovich data-hiding definition, however, the adversary cannot query to obtain a token for either of the challenge keywords because of the restrictions on queries in the game. Thus, the adversary cannot distinguish the challenge keywords.

We now sketch a more detailed proof. Given a data-hiding adversary $\mathcal{A}$ against the counterexample scheme, we construct a data-hiding adversary $\mathcal{B}$ against the original Mylar scheme. $\mathcal{B}$ proceeds through the five stages of the data-hiding game as follows. (1) It runs $\mathcal{A}$ and outputs in the first stage the same access graph G as output by $\mathcal{A}$. (2) $\mathcal{B}$ gets a list of delta values, which it gives to $\mathcal{A}$. (3) When $\mathcal{A}$ outputs two challenge keywords (w_0, w_1), $\mathcal{B}$ chooses a bit d at random and outputs as its challenge (U, w_d) for U drawn uniformly from the message space. It gets back a ciphertext $\langle r, c' \rangle$ and gives back to $\mathcal{A}$ the ciphertext $\langle r, c' \oplus (w_d \| 0^n) \rangle$. (4) $\mathcal{B}$ answers oracle queries by $\mathcal{A}$ using its own oracles. Note that any token queries by $\mathcal{A}$ cannot be on w_0 or w_1 for a user u for which $(u, 0) \in G$ (i.e., for which the delta between user u and 0 is in the graph). Thus $\mathcal{B}$ will never make such a token query on w_d. The probability that one of $\mathcal{A}$'s token queries is on U is negligible for sufficiently large $|U|$. Thus $\mathcal{B}$ satisfies the query restrictions of the game with all but negligible probability. (5) Finally, $\mathcal{A}$ outputs bit b', and $\mathcal{B}$ checks if $b' = d$. If so, it outputs 1 and otherwise it outputs 0.

To complete the proof, it suffices to show that $\mathcal{B}$'s success upper bounds that of $\mathcal{A}$. If $\mathcal{B}$'s challenge is $b = 1$, then the environment that $\mathcal{B}$ simulates for $\mathcal{A}$ is exactly the data-hiding game for the modified Mylar scheme. If $\mathcal{B}$'s challenge is $b = 0$, then the probability that $\mathcal{A}$ outputs d is $\frac{1}{2}$ because $\mathcal{A}$ receives no information about d. Thus twice $\mathcal{B}$'s advantage is an upper bound on $\mathcal{A}$'s advantage.

Combining this with the data-hiding proofs for the original scheme [49] implies the data-hiding security of the modified scheme under the same bilinear pairing assumptions.

Token hiding. Intuitively, token-hiding is satisfied because the adversary receives a token for user key k_0 of exactly one of the challenge keywords and can never request the token of either keyword under k_0 or any other "non-free" user key, meaning the adversary has a delta which can adjust the search token to search over a document whose key it specified in setup. It can also never receive an encryption of either of the challenge keywords under any "non-free" document key, which is a key to a document that is accessible by k_0 or any users that can access any documents also accessible to k_0. Without either of these values, it cannot run MK.Match and distinguish the challenge token.

As with data hiding, we can reduce the token-hiding security of the modified Mylar MKSE scheme to the original one. Because token generation is the same in both games, the reduction simply simulates encryption query responses appropriately (by xor-ing in $w\|0^n$ to values).

3.3 Limitations of formal models

Our counterexample highlights a critical problem with the Popa-Zeldovich formal model for MKSE. In the literature on (single-key) symmetric searchable encryption (SSE), it has long been known that simultaneously meeting separate data-hiding and token-hiding definitions is insufficient. Curtmola et al. [17] give a stronger, all-in-one simulation-based notion of security for SSE, and one could conceivably craft a similar model for MKSE. But designing a model that addresses the full spectrum of attacks from an actively malicious server—which is the explicit goal of Mylar—requires dealing with a number of other, more challenging issues.

One issue is how to formalize **active adversaries**. In the cryptographic literature, "active" is often interpreted as the adversary's ability to make adaptive queries, i.e., choose later encryption or token queries as a function of the earlier ones. Query adaptivity is handled by many simulation-based notions for SSE, starting with [17], and the Popa-Zeldovich definitions allow some query adaptivity as well.

Another issue affecting the modeling of active adversaries is whether the set of documents subject to keyword search is fixed and never changed, or if documents can be deleted, added, or updated. Web applications, including those for which BoPETs such as Mylar are designed, are inherently dynamic. Modern SSE definitions [10, 31, 43] model a *dynamic* document corpus and an adversary that can make dynamic changes. Modeling a dynamic corpus for MKSE would be more complex. An accurate definition must incorporate dynamic changes to the access graph. This includes user nodes *and* document nodes being added, as well as access edges being added between user and document nodes.

In the implementation of Mylar, a malicious server can easily add a document and give any user access to it. The Mylar paper [48] proposes a defense but it has not been implemented, nor is it clear how to implement it in collaborative applications. In Section 8, we show how to exploit this gap to break the confidentiality of data and queries.

Even if the system could be proved secure relative to an all-in-one simulation-based model for MKSE that addresses both query adaptivity and dynamic changes to the document corpus, this is not enough to prevent passive or active attacks based on search access patterns, query [9] or file injection [9, 64], or passive or active attacks against the non-search portions of the BoPET system (such as metadata). Mylar excludes access patterns from its threat model, but this will be small consolation for the users of the applications whose access patterns leak sensitive information to a persistent passive attacker on the server. We show how to exploit metadata and access patterns in Mylar-based applications in Sections 6 and 7, respectively.

4. THREAT MODELS FOR BOPETS

In the rest of this paper, we turn to the security analysis of BoPETs and Mylar in particular. We consider three types of attacks, in the increasing order of attacker capabilities.

A *snapshot passive attack* is a one-time compromise of the server that gives the attacker a complete snapshot of the server's state at the time of the attack. This is the classic "smash and grab" attack that involves an attacker breaking in and stealing all encrypted data, unencrypted data, and metadata stored on the server.

A *persistent passive attack* involves an attacker who can fully observe the server's operations over a period of time. This attacker does not change the server's actions but can watch applications' dynamic behavior, access patterns, and interactions with users. Unlike a snapshot passive attacker, a persistent passive attacker can observe how the server evolves over time in response to interactions with users. We propose the persistent passive attacker as a realistic model for an *honest-but-curious* BoPET server.

An *active attack* involves an arbitrarily malicious attacker who can tamper with messages to and from clients and perform any operation on the server. It can also collude with one or more users in order to compromise confidentiality of the other users' data and can adapt its strategy over time.

Comparison to prior threat models. Commercial BoPETs all encrypt data before uploading to a server but are vague about their adversary models. We believe that they primarily attempt to defend against snapshot passive attackers and network eavesdroppers. (We do not consider the latter in this paper.) We are unable to determine if they claim security against persistent passive or active attackers.

Some academic BoPETs claim security against active (and therefore passive) attacks with the important caveat of excluding attacks based on access patterns or metadata [28, 34, 47, 48]. This restriction stems from the fact that the state-of-the-art PRE schemes upon which BoPETs are based leak this information for the sake of efficiency.

This leakage may be inevitable, but we need methodologies for assessing the damage it can cause. Obviously, in a real-world deployment, a malicious or compromised server can take advantage of all available information, even the information the designers of the system opted to "exclude" from the security model. Our analyses reflect this approach, and the passive attacks against Mylar in Sections 6 and 7 exploit what would be considered out of scope by previous work. We believe similar attacks apply to other BoPETs. On the other hand, the attacks in Section 8 fall squarely within the threat model considered in [48].

5. BUILDING WEB APPLICATIONS ON TOP OF ENCRYPTED DATA

It is difficult to evaluate the security of an application framework in isolation, without considering specific applications, since leakage can vary dramatically from application to application depending on what data they store on the server, what operations they perform on the data, etc.

Mylar can support a wide variety of Web applications, but only a simple chat application called kChat is publicly available. In addition to kChat, we ported three open-source Meteor apps representing different types of functionality (see Section 5.2). We used an "updated" implementation of Mylar [57] linked from the Mylar project website, but all issues we found are present in the original code, too.

5.1 Porting apps to Mylar

Since the main motivation for Mylar is to preserve the structure of the original app, our porting process is parsimonious and follows these principles: (1) maintain user experience of the original app; (2) follow the app's data model unless changes are required to encrypt confidential data; (3) change as few of the relationships between data structures as feasible. We believe that this process reflects what developers would do when porting their Meteor apps to Mylar. Except for a few cases explained in the relevant sections, none of the vulnerabilities uncovered by our analysis arise from the decisions we made while porting our sample apps.

All of Mylar's changes to Meteor are done through plug-and-play modules called *packages*. After adding the Mylar packages to the app, the developer needs to mark which fields to encrypt, which principals to use to encrypt them, and add principal creation to the code.

Creating, viewing, and updating documents with encrypted fields must be handled by the client. If an app accesses the database through *Meteor methods* on the server, this functionality needs to be moved to the client. The developer needs to add code allowing users to share their encrypted data and change the app to use Mylar's encrypted search. This is straightforward but requires creation of *search filters* restricting what encrypted data a given user can search over.

Security decisions. Deciding how to create access-control principals and which field to encrypt with which principal requires an understanding of how data is used and shared within the app. These decisions are the most subtle and critical parts of the porting process.

For apps where multiple documents are encrypted with the same key, we created principals associated with the documents' common feature. For example, all messages in a chat room should be encrypted with the same key, thus principals are associated with chat rooms. If each document is encrypted with its own key, principals correspond to individual documents. For example, each medical appointments in MDaisy has its own principal.

If functionality is moved from the server to the client, the developer may need to update user permissions for the data, which is notoriously challenging [25]. The user must be given enough permissions for the app to work correctly, without enabling him to access another user's private data.

We conjecture that many developers will struggle to make these decisions correctly and that independently developed Mylar apps will contain vulnerabilities caused by developers' mistakes. Even the authors of Mylar made a security error when porting kChat to Mylar (see Section 6.2).

5.2 Sample apps

kChat for Mylar is a chat room app released by the Mylar authors. A user can create rooms, add other users to the rooms they created, send messages, and search for keywords over all messages in all rooms that he belongs to.

To find other case-study apps, we searched the DevPost software project showcase and GitHub for open-source Meteor apps that (1) work with potentially sensitive data, (2) contain non-trivial server functionality, such as searching over sensitive data and/or sharing between multiple users, and (3) are straightforward to port to Mylar.

MDaisy. MDaisy is a medical appointment app. Every user is either a patient or a member of the medical staff. Staff create and manage appointments for patients. Each appointment is associated with a procedure (e.g., MRI, CT scan, etc.). Patients can view information about their appointments and the associated procedures.

Each appointment has its own principal that encrypts all sensitive fields. The staff member creating the appointment grants access to the patient. Information about different types of procedures is stored separately from appointments. Each procedure has its own principal that encrypts its data. A patient is given access to the procedure principal if they have an appointment involving that procedure.

OpenDNA. Single-nucleotide polymorphisms (SNPs) are locations in human DNA that vary between individuals. OpenDNA is a Meteor app that enables users to upload the results from DNA sequencing and testing services such as 23andMe [1] to a server and check them for *risk groups*, i.e., combinations of SNPs that indicate susceptibility to certain conditions or diseases such as Alzheimer's or reveal ancestry.

In the original OpenDNA app, users' DNA is stored unencrypted on disk (not in MongoDB) on the server. Risk groups are crowd-sourced and can be uploaded by any user. Each risk group consists of one or more SNPs-genotype pairs. When a user wants to check their DNA, the server iterates through all risk groups, compares them to the user's DNA, and returns the resulting matches. OpenDNA is an example of an open system, where any user can contribute content to be used in other users' searches.

We modified OpenDNA to encrypt DNA with the user's principal and store it in MongoDB. Risk groups are public and not encrypted. We modified the search functionality to work over the encrypted DNA: the client requests all risk groups from the server, submits encrypted search tokens for each SNP-genotype pair to the server, the server uses these tokens to search over the user's DNA.

MeteorShop. MeteorShop is a sample e-commerce app. A product has an image, description, and price; products are organized into categories and subcategories. A user adds products to a cart, which keeps track of them and the total price. In the ported MeteorShop, every item in the cart is encrypted with the user's principal.

The original MeteorShop uses the potentially insecure `autopublish` package that would push the entire database to client. We modified MeteorShop to only send the products of the subcategory that the user is currently viewing.

6. EXPLOITING METADATA

Client-server applications need to store metadata on the

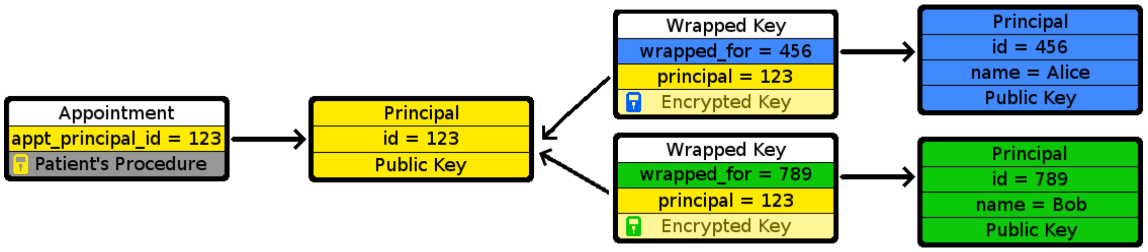

Figure 2: A graph showing the access patterns in MDaisy. Each of the three principals and their associated fields are in a different color. Each encrypted field is marked with a lock the color of the principal that encrypted it.

server. At the very least, this includes information about which keys are used to encrypt the data and which users have access to which principals. In many apps, the data structures created by the app inherently depends on users' secrets, and even the names of data structures may reveal sensitive information. This static metadata is available even to a one-time, snapshot passive attacker.

6.1 Links between objects

In Mylar, every user, principal, and encrypted item is stored in its own MongoDB document in the app's database on the server. Their relationships (e.g., user Foo has access to principal Bar that was used to encrypt data Baz) form an access graph, which is visible even to a snapshot attacker.

MDaisy. In MDaisy, each appointment is created by a member of the medical staff and shared with only one patient. The details of the appointment are encrypted with its unique principal, but the metadata representing the access graph is not encrypted. Starting with an encrypted appointment, the server can find the appointment's encrypting principal and then, following that principal's wrapped keys, find the patients who can access the appointment. Figure 2 shows a graph of the connections from an appointment to the patients who can access it.

Knowledge of the patients who have access to a particular appointment can leak information about the encrypted data. If patient Bob and doctor Alice have multiple appointments spread over several weeks for a number of months and the attacker knows that Alice is an oncologist, he can form a strong hypothesis that Bob is being treated for cancer.

This leakage is inherent in any application where the semantics of inter-user relationships reveal sensitive information about users. Preventing it requires hiding the access graph from the server—a complicated feat in its own right—as well as hiding users' interactions and data accesses. Techniques like ORAM [24] might help protect users' interactions with the database at the cost of removing BoPET's functionality and reducing it to dumb storage.

6.2 Names of objects

Every developer creating or using a BoPET needs to understand whether users' assumptions align with those of the developer. For example, every principal in Mylar has a `name`, which is used to verify the authenticity of keys and thus intentionally left unencrypted by the Mylar developers. The names of user principals are automatically set to their usernames or email addresses, but developers are responsible for assigning names to all other principals. Using sensitive information when naming principals will leak this information to a snapshot passive attacker.

kChat. kChat's principals are chat rooms and users. Their

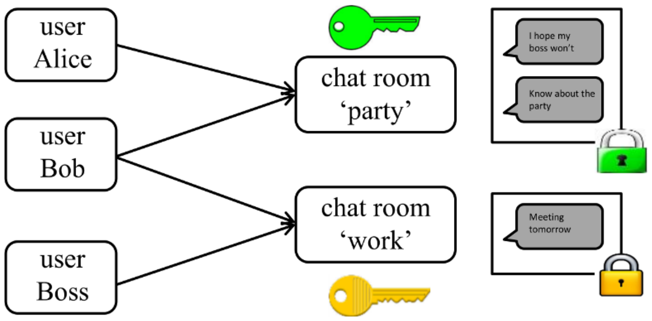

Figure 3: (From [48]) Bob and Alice are discussing a party in the 'party' chat room while Bob and Boss discuss work in the 'work' chat room. Bob doesn't want his boss to find out about the party.

principal names are set to room names and usernames, respectively. In general, principal names are assigned when principals are created and users are not told how they were chosen. Users normally cannot see the names of the rooms that they have not been invited to and they might mistakenly assume that room names are encrypted on the server. This can create a false sense of privacy in users who give their rooms revealing names. In fact, room names are visible via the names of the corresponding principals.

The authors of Mylar intentionally gave human-understandable names to chat rooms in kChat. This breaks the security of their own motivating example. One of the figures in [48] shows a kChat user expressing hope that another user does not learn certain information—see Figure 3). But principal names (i.e., Mylar's unencrypted metadata) are derived from room names (i.e., kChat's metadata). Because Mylar's metadata is visible in plaintext on the server, the server operator can easily learn the name of the chat room and thus the secret that the users want to protect.

This internal inconsistency shows how difficult it is to match user and developer expectations. The authors of Mylar could have chosen nondescript principal names, but this would have broken Mylar's reliance on principal names when verifying keys and identities.

6.3 Features not protected by encryption

Even secure encryption leaks many features of the data, such relative sizes. If the data is searchable, Mylar also reveals the count of unique words for each document because it computes a search token for every word and stores these tokens in the same document as the encrypted data.

OpenDNA. In OpenDNA, the combination of encrypted data and search tokens exceeds the maximum MongoDB document size of 16 MB, thus each user's DNA must be split into multiple MongoDB documents. Mylar provides no guidance to help developers make these decisions.

One simple solution is to split the DNA by chromosomes, with a separate document containing all SNPs for each chromosome. Another simple solution is to split the DNA into n documents, with an equal number of SNPs in each.

Splitting DNA into documents based on chromosomes enables a persistent passive attacker to infer which risk groups the user is searching for. For example, if a risk group matched documents 1, 3, and 8, then the attacker can look for known risk groups that are associated with SNPs in chromosomes 1, 3, and 8. Even if the encrypted chromosomes were stored in random order in the database, the attacker could tell which chromosome matched the risk group as the number of SNPs differs greatly between chromosomes. For example, one DNA test from 23andMe gives 1,700 SNPs in one chromosome and 77,000 in another.[1] These differences result in large discrepancies in the size of the encrypted data and number of encrypted words between documents.

Splitting DNA into equally sized documents also leaks information about the user's queries. In the case of 23andMe, the user is given a file with SNPs ordered by their chromosomes, and the ordering is preserved when the file is uploaded. So if the user's DNA is split into, say, 5 documents, the attacker can guess that chromosomes 1 and 2 are in document 1, chromosome 3 is split between documents 1 and 2, chromosomes 4-6 are in document 2, etc.

MDaisy. In MDaisy, medical staff members have only their names stored in the database, while patients have their name, date of birth, medical record number, and gender stored. Knowing which users are patients and which are staff helps a snapshot passive attacker infer sensitive information (see Section 6.1). If the user's role is encrypted, it can be easily inferred from the number of encrypted fields (four for patients, one for staff). Even if all of the user's information were stored in a single field, the size of that field would distinguish patients from staff members.

Preventing inference from relative ciphertext sizes may very well be impossible in practice. Data can be padded to a pre-defined size, but this introduces large overheads in computation and storage. Without application-specific information about data sizes, a BoPET cannot generically hide the size. Protecting against other attacks such as frequency analysis is tied directly to the cryptography and functionality of the BoPET and may be a fundamental limitation.

7. EXPLOITING ACCESS PATTERNS

BoPETs involve rich client-server interactions. Clients fetch, send, and update documents in the server's database and interact with each other through the server. By design, searchable encryption leaks whether a match succeeded. Furthermore, in systems like Mylar, keywords are encrypted in order, thus the order of the tokens leaks information, too.

A persistent passive or active attacker on the server can infer users' secrets from these access patterns. This leakage is fundamental in any non-trivial BoPET because the services that a BoPET provides to different users depend on these users' inputs into the system.

The designers of Mylar acknowledge that Mylar does not hide access patterns, and in the BoPETs literature it is typical to exclude this leakage [28, 33, 34, 47]. Of course, a malicious server can easily observe these patterns. To fully understand the limits of BoPETs' security in realistic scenarios, we must analyze what sensitive information can be inferred from access patterns in concrete applications.

MDaisy. If two different patients access the same encrypted procedure information, a persistent passive attacker can infer that both are undergoing the same procedure. Given more users and more appointments, this attacker can cluster users and begin to understand how procedures relate to one another (e.g., "if a user goes in for procedure Foo, they will typically come back two weeks later for procedure Bar"). The attacker does not know what the procedures are, but this confidentiality is very brittle. If a single user discloses their procedure (either publicly or by colluding with the server), everyone who underwent the same procedure during the attacker's observations will lose their privacy.

MeteorShop. Our ported MeteorShop app encrypts every item in the user's cart. Items prices stored in the cart are encrypted, too, lest the server identifies the items by solving a knapsack problem given the total price.

A persistent passive attacker can assume that when an encrypted item is added to the user's cart, it came from the list of products most recently requested by the user. This leakage is mitigated somewhat by the fact that MeteorShop does not have individual pages for products. Instead, when a user clicks on a subcategory, the client fetches all products in that subcategory from the server and displays them. The server thus learns only the item's subcategory.

If a shopping app listed each product on its own page, the server would be able to infer the exact item added to the cart. Even if product information were fully encrypted on the server and the attacker were somehow prevented from using the app as a user, the server could infer the products from the images requested by the client.

OpenDNA. In OpenDNA, if the risk groups being searched for as well as the search order are known to the server, the server learns sensitive information about the user's DNA based on whether a query matched a group or not. If the server knows only which risk groups are associated with diseases, the actual risk group might not matter as the match reveals that the user is at risk for something.

DNA documents in OpenDNA preserve the original order. If a query returns a match, the server will know which encrypted data was matched and can use its location to figure out which SNPs were searched for. From this, the server can infer the SNP values and the risk group searched for.

8. ACTIVE ATTACKS

BoPETs are efficient only insofar as they rely on an untrusted server to execute the application's server functionality. These operations must be verified by the client lest they become an avenue for active attacks by a malicious server. In this section, we use Mylar as a cautionary case study to demonstrate how a malicious server can exploit his unchecked control over security-critical operations to break the confidentiality of users' queries.

First, we show how the server can obtain the delta value that enables searches over a "tainted" principal whose keys are known to the server. Then, we show how the design of Mylar's MKSE allows the server to perform dictionary attacks on *all* search keywords, not just those that occur in the tainted principal. This enables efficient brute-force attacks on all of the victim's queries, past or future, over any principal in the system. Using chat as our case study,

[1]We found similar percentage breakdowns in 1,900 user-released 23andMe reports from openSNP [44].

we demonstrate how this leads to the effective recovery of the information that Mylar was supposed to protect.

8.1 Forcing users into a tainted principal

As explained in Section 2.2, sharing of documents between users in Mylar is implemented using wrapped keys. For example, suppose Alice wants to invite Bob into her new chat room. Alice generates a principal for the room, wraps it with Bob's public key, and adds the wrapped key to the database. Finally, she adds this key's id to the `accessInbox` field in Bob's principal. When Bob's client is informed of a new wrapped key in its `accessInbox`, it immediately and automatically creates the delta for this key, enabling Bob to search over the contents of Alice's chat room.

A malicious server can create its own principal, for which it knows the keys. We call such principals *tainted*. It can then add the tainted principal's wrapped keys to the victim's `accessInbox`, and the victim's client will automatically generate the delta for keyword searches over the tainted principal. To stay stealthy, the server can then immediately remove the principal, wrapped keys, and delta from the database, ensuring that the victim will not notice. This attack has been verified by executing it in Mylar.

Mylar uses a stateless IDP (identity provider) to certify that keys claimed by users and principals actually belong to them. In the above example, it prevents a malicious server from substituting its own keys for other users' and principals' keys. It does not protect the user from being forcibly added to any principal chosen by the server.

Fixes. The problem is fundamental and generic. BoPETs aim to preserve server functionality, and in multi-user applications, the server is responsible for conveying to users that there is a new document that they can now query. To foil this attack, all server actions involving adding access rights to users must be verified or performed by the client. This involves changing the structure of the application and adding non-trivial access-control logic on the client side.

The solution suggested by Popa et al. [48] is to require users to explicitly accept access to shared documents (and, presumably, check the IDP certificate of the document's owner, although this is not mentioned in [48]). The `allowSearch` function in the Mylar implementation is responsible for enforcing this. It contains one line of code, which updates an otherwise unused variable and calls what we believe is an obsolete version of the `MylarCrypto.delta` method without any of the needed parameters. It is never invoked in any publicly available Mylar or application code. For example, in kChat, anyone can add a user to their chat room without that user's consent.

We do not believe that this solution, even if implemented properly, would work for realistic applications. The semantics of `allowSearch` are highly counter-intuitive: when the user is told that someone shared a document with him and asked "Do you want to be able to search this document?," the user must understand that answering "Yes" can compromise all of his queries over his *own* documents, including those that are not shared with anyone. Also, whenever different documents have different access rights, every document must be a separate principal. Asking the user for a confirmation for every document (e.g., email) to which he is given access destroys user experience in many collaborative applications.

Collusion. The `allowSearch` defense does not prevent collusion attacks. If the user voluntarily joins a malicious principal—for example, receives an email from a user he trusts but who is colluding with the server—the result is exactly the same: the server obtains the delta for keyword searches over a tainted principal.

Gap between models and practice. In the searchable encryption literature, it is common to assume for simplicity that the search index is static, i.e., fixed before a security experiment begins and does not change. As explained in Section 3.3, Mylar deals with a more complicated access-controlled search abstraction, where changes to the "index" can take the form of document creation, user creation, or access edge creation when an existing user is given access to a document they did not previously have access to.

In the model claimed to provide the theoretical foundation for Mylar [49], Popa and Zeldovich assume a static access graph. Even if the model properly captured what it means for MKSE to be secure (it does not—see Section 3.2), the security of each user's queries and documents would critically depend on the user *never* touching any other data—voluntarily or by accident—for which the adversary knows the key. This model is not relevant for the practical security of MKSE-based collaborative systems, such as chat or email, where dynamic access graphs are the norm.

8.2 Expanding the attack

Any searchable encryption scheme that leaks access patterns is generically vulnerable to the following attack: if the user searches an adversarially chosen document, the adversary learns whether the query matched one of the keywords in that document. This attack, briefly mentioned in [48, section 5.4], has been known in folklore since before the Mylar paper was published.

In the rest of this section, we explain how the flawed design of Mylar makes it vulnerable to a much more powerful attack. Once the malicious server obtains the token conversion delta for a particular document, he can expand the attack to all of the user's documents (not just those that the user shares with the adversary) and all search keywords, *including keywords that do not occur in any document stored on the server.*

If Mylar fully hid access patterns—for example, by utilizing some kind of oblivious storage or private information retrieval—the generic attack mentioned in [48] would be foiled but Mylar queries would still be vulnerable to the brute-force attack on search keywords described in Section 8.3 (but not to the concomitant plaintext recovery attack, which requires the server to observe which documents matched the user's search query).

All documents. Mylar claims that it "protects a data item's confidentiality in the face of arbitrary server compromises, as long as none of the users with access to that data item use a compromised machine" [48]. This claim is false. In fact, we are not aware of any searchable encryption scheme that can guarantee this property.

For example, suppose Alice keeps her private files on the server. Only she has access, and she is not using a compromised machine. Alice receives a server-hosted message from Bob, who uses a compromised machine. Now, if Alice searches Bob's message for a particular keyword, a malicious server will be able to search all of her private files for that keyword—even though none of the users with access to these files use a compromised machine. This is a generic feature of all searchable encryption schemes, including Mylar's MKSE.

In the implementation of Mylar, the situation is even worse. As we explained in Section 8.1, a malicious server can forcibly add any user to any document, thus expanding the attack to all documents in the system.

All keywords. For efficiency, Mylar trusts the server to convert search tokens between principals. In Section 8.3, we show how this unique feature of Mylar's MKSE enables the server to perform dictionary attacks on **all of the user's queries over any principal**. The state-of-the-art attacks on searchable encryption recover only the query keywords that occur in adversary-controlled documents; increasing recovery ratios requires known-document or file-injection attacks [9,64]. By contrast, Mylar allows a malicious server to recover query keywords regardless of whether they occur in adversary-controlled documents or not.

The basic design flaw (relying on the server to perform a security-critical operation) is exacerbated in Mylar because there is no way for the user to revoke a delta disclosed to the server, nor prevent the server from using the delta to verify his guesses of the user's queries.

8.3 Brute-forcing query keywords

The attack in this section does not require online access. The server can save the victim's queries until the victim is forced into or willingly joins a tainted principal.

Suppose the victim with key k_1 has access to a tainted principal with key k_2. The server has the victim's delta to the principal key, $\Delta_1 = g_2^{k_2/k_1}$. Knowing k_2 and Δ_1 the server can compute $\Delta_1^{1/k_2} = g_2^{k_2/k_1 \cdot 1/k_2} = g_2^{1/k_1}$.

When the victim issues a search query for word w, it is computed as $H(w)^{k_1}$. The server can pair this with g_2^{1/k_1} to get $e(H(w)^{k_1}, g_2^{1/k_1}) = e(H(w), g_2)$. The pairing e, the hash H, and the generator g_2 are all public and known to the server, thus it can **pre-compute** a dictionary of $e(H(w), g_2)$ for every possible search query and, when the victim submits a query, immediately check it against the pre-computed dictionary to uncover the word w.

If the victim has access to another principal k_3, the server can check if a document encrypted with k_3 contains w. This does not require knowing k_3 and can thus be done for *any* document the victim has access to, tainted or not.

This attack is dangerous in any setting, but it is especially severe in typical BoPET applications. Popa et al. use a medical diary application as a case study for multi-key search [48, section 5.4]. In this application, women suffering from endometriosis can record their symptoms for a gynecologist to monitor. Since there are many more patients than gynecologists, all gynecologists are given access to one staff principal which, in turn, has access to all patient information. In other words, there is one high-trust principal shared by the doctors and many low-trust users (the patients). Once the server obtains a delta for the shared principal from *any* gynecologist, the server can perform plaintext recovery attacks on *every* record of encrypted symptoms.

8.4 Experiments

We simulated a brute-force attack on kChat using several years' worth of Ubuntu IRC logs [60] and a dictionary of 354,986 English words [19] (the biggest we could find).

We chose kChat as our target because it is the only application whose code was released by Popa et al. as an example of how to port Web applications to Mylar. The Mylar paper [48] uses kChat extensively as a running example—along with the claims that Mylar is secure against arbitrarily compromised servers—but the release notes of kChat say "The app is secured against passive adversaries (adversaries who read all data at the server, but do not actively change information)." This disclaimer does not appear anywhere in the Mylar paper [48] or Mylar website (see Section 12). In any case, as we showed in Section 6.2, kChat does not protect users' privacy even against a snapshot passive attacker.

All experiments were run on a Dell Optiplex 7020 with 16 gigabytes of RAM and an Intel i7-4790 quad-core CPU, clocked at 3.6GHz.

Setup. Our proof-of-concept code uses the "crypto_mk" and "crypto_server" C++ libraries from Mylar. It accepts a user's search query $H(w)^{k_1}$, a delta g^{k_2/k_1}, and a principal key k_2. This simulates a malicious server obtaining the principal key for a document to which the user has access. Preprocessing every word in our 354,986-word dictionary takes roughly 15 minutes of wall-clock time. Afterwards, recovering the keyword w from $H(w)^{k_1}$ takes less than 15 ms.

Using Java, we downloaded, parsed, and processed Ubuntu IRC logs from 2013, 2014, and 2015. We used Tika [5] to remove non-English words, the Stanford NLP library [56] to create our "bag of words," the Porter stemmer algorithm for stemming [50], and the stopword list from Lucene [4].

Query distribution. We used different years for cross-validation, attacking logs from one year with a keyword query distribution generated from another year's logs. We experiment both with and without stemming (e.g., changing "running" to "run") and stopword removal (e.g., "this", "the", "that"). It is standard in practice to apply stemming and stopword removal, although the kChat implementation does neither. Following [9], queries were sampled with replacement and without removing duplicates to simulate two users searching for the same term. If the chosen query was in our dictionary [19], an adversary would be able to recover the query keyword and all of its occurrences across all documents. The recovery rates were calculated as the number of occurrences of the recovered queried keywords divided by the total number of occurrences of all words in all documents. For the stemming and stopword removal case, the recovery rates were calculated as the number of occurrences of the recovered queried stemmed keywords divided by the total number of occurrences of non-stopwords in all documents.

Results. Figure 4 shows the results of our simulated experiments. The upper curve is without stemming and stopword removal, the lower includes them. While we performed cross-validation, the results of all experiments were nearly identical, so we only present the average recovery rates over 20 trials of the same experiment. The 95% confidence interval was between 4% and 6% in almost all experiments.

We simulated queries in increments of 100, starting with 100 and ending at 2,000. At 100 queries, our recovery rate is 20–25%, growing quickly to above 50% with 800–900 queries. With 2,000 queries, we recover 65–69% of the entire corpus.

The implementation of Mylar does not randomize the order of encrypted keywords. For data encryption, Mylar uses `MylarCrypto.text_encrypt()` which calls the function `tokenize_for_search()` to split strings into lowercase tokens, preserving order. The ordered tokens are then encrypted and sent to the Mylar server. Therefore, in addition to keywords, our attack recovers their order, revealing much more

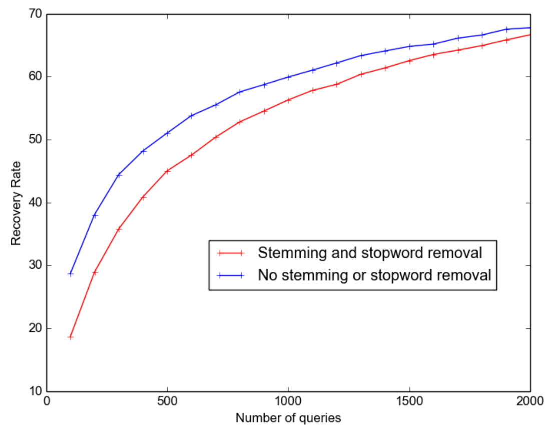

Figure 4: Recovery rates for simulated query brute-force attacks against Ubuntu IRC dataset.

than a bag of words. Figure 5 shows examples of recovered messages without stopword removal or stemming for one of the experiments using 2,000 queries.

The exact query distribution has a large effect on recovery rates. Using uniformly sampled queries means that all but a small fraction of the 180,000 unique keywords only appear once in a given year's log file. This would greatly reduce the efficacy of the attack. Unfortunately, there has been no systematic study of real query distributions in this setting to guide our experiments. Therefore, we chose to sample queries from the keyword frequency distribution, following prior work [9] and because it is unlikely that real user queries would be approximately uniform.

With stemming and stopword removal, the recovery rate for 100 queries is almost 10% lower. This difference in recovery rates becomes smaller as the number of queries increases. While less information is recovered in this case (since we recover only prefixes of words and cannot recover stopwords), it still represents a significant amount of partial leakage about plaintexts.

The effect of stemming on query and plaintext recovery is not well understood. Previous work [9] pointed out that stemming, in some sense, makes the adversary's job easier, because recovering a stemmed query for a word like "run" also recovers the words "running," "runs," and "ran." Stemming also changes the keyword distribution of a corpus of documents, which could make inference attacks easier. It is not known whether stemming affects the relative value of recovering a particular query, because knowing that a stem for, say, "run" exists in a document does not give the adversary information about the exact keyword in that document. We conjecture that recovering a stemmed keyword still reveals valuable information to an adversary since different keywords with the same stem are semantically related.

Measuring the value of recovering a particular keyword is an open problem. Certainly, recovering stopwords gives the adversary little information about a corpus, but quantifying the value of other, less frequent words is difficult and highly distribution-dependent. In information retrieval, there are techniques for quantifying the "relevance" of a keyword to a particular document, but this is mostly in the context of returning the best matches for a particular query and does not capture the value of recovering, for example, the keyword "malignant" or "benign" in a corpus of medical documents.

```
???  ???  don't tell anyone because it might
diminish my manliness but disney rocks
```

```
dad in law is going to the hospital not sure if
he's having an ???  reaction cough medicine
```

Figure 5: Examples of recovered plaintext data from our simulated attacks on [60] . Unrecovered words are marked with "???"

8.5 Chosen-query attacks

In open applications, data contributed by one user can influence other users' queries. An active attacker can then craft queries designed to leak information about the searcher's documents without the need for tainted principals.

For example, risk groups in OpenDNA are crowd-sourced and shared between all users in plaintext. Each risk group contains one or more SNP-genotype pairs. A malicious user or server can upload custom risk groups to the database. These could be as simple as SNP-genotypes pairs that correspond to sensitive genetic information (such as ancestry or predisposition to a certain disease), or as complex as risk groups covering all possible genotypes for a given SNP. If a malicious risk group is successfully found within a user's DNA, the server learns sensitive information. The server can recover a user's entire reported genetic information by brute force with roughly 12 million queries. Admittedly, this is a lot of queries, but note that the server is recovering 760,000 SNPs in this case.

9. RELATED WORK

BoPETs. Searchable encryption (SE) was explored by [11, 17,27,31,45,55]. Islam et al. [30] showed that the access patterns of SE systems can be used to recover queries. Cash et al. [9] investigated plaintext recovery attacks in SE schemes similar to [28, 34].

BoPETs include numerous commercial systems for encrypted cloud computation [16, 22, 46, 51, 54]. In particular, Kryptnostic [59] uses a multi-key searchable encryption scheme very similar to Mylar's. VC3 [53] relies on a trusted processor to run MapReduce in an untrusted cloud.

CryptDB [47] adds encryption to databases while still supporting various client queries. Naveed et al. [42] showed that CryptDB's deterministic (DTE) and order-preserving encryption (OPE) schemes leak information. Mylar does not use OPE and Mylar's encrypted search scheme does not leak word frequencies within documents, making DTE frequency analysis and ℓ_p-optimization attacks impractical.

Privacy of Web apps. Client-side encryption in Web apps was explored in [3, 15, 21, 28, 52]. Distribution and revocation of access in encrypted file sharing systems was studied in [61, 62].

SUNDR [35] and Depot [38] address data integrity, SPORC [20] adds confidentiality to SUNDR, Radiatus [14] isolates individual users' data. Verena [32] is a Meteor extension that addresses integrity, not confidentiality.

Side channels in Web apps. Chen et al. [13] demonstrated powerful side-channel attacks on HTTPS-protected Web apps that exploit packet sizes and other features. These vulnerabilities are "fundamental to Web 2.0 applications." Sidebuster [63], SideAuto [29], and the system by Chapam and Evans [12] help discover side-channel vulnerabilities in

Web apps. Side channels can be mitigated by padding or varying network traffic [36, 37]. All of these papers assume a network adversary, as opposed to an untrusted server.

10. LESSONS AND CONCLUSIONS

The Mylar framework for building web applications on top of encrypted data was claimed to be secure against active attacks [48]. Mylar is based on a multi-key searchable encryption (MKSE) scheme, which was proved secure in the formal model proposed by Popa and Zeldovich [49]

Our first conclusion is that the Popa-Zeldovich model for MKSE does not imply security against either passive, or active attacks. Our second conclusion is that the security claims made by Popa et al. in [48] are false: Mylar does not protect a data item's confidentiality if none of the users with access to that data item use a compromised machine. Furthermore, a basic system design flaw at the core of Mylar—relying on an untrusted server to convert clients' tokens—enables an efficient brute-force dictionary attack on users' queries. This attack recovers even the search keywords that do not occur in adversary-controlled documents.

The most important lessons transcend Mylar and apply generically to the entire class of BoPETs. First, we give concrete illustrations of how encryption schemes designed to be secure against snapshot passive adversaries end up being completely insecure when deployed in a system requiring security against an active adversary. Second, the natural process for porting applications—encrypt all data and adapt server operations so they can be performed over ciphertexts—leaves metadata exposed. This reveals a lot of sensitive information even to snapshot passive adversaries.

Another lesson is that BoPETs need to define realistic threat models and then develop formal cryptographic definitions that capture security in those threat models. Building a scheme first and then crafting a cryptographic model in which the scheme can be proved secure can result in schemes whose security breaks down in practice.

The problem of building client-server application frameworks that provide meaningful security against persistent passive and active attackers on the server remains open. To protect against persistent passive attacks, access patterns must be hidden or securely obfuscated. To protect against active attacks, every essential operation performed by the server must be either executed or at least verified on the client side of the application. This runs contrary to the entire premise of BoPETs, since they aim to preserve the existing split of application functionality between clients and servers with all concomitant benefits and rely on clever encryption to protect data from untrusted servers. We conjecture that verifying or moving every server operation to the client involves a substantial re-engineering of application logic and is likely to incur high performance overheads.

Acknowledgments. This work was partially supported by the NSF grants CNS-1223396, CNS-1330308, CNS-1514163, and CNS-1546033 and a generous gift by Microsoft.

11. REFERENCES

[1] 23andMe. https://www.23andme.com.
[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.
[3] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *USENIX Security*, 2012.
[4] Apache Lucene. https://lucene.apache.org/core.
[5] Apache Tika. https://tika.apache.org.
[6] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, 2007.
[7] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.
[8] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *EUROCRYPT*, 2015.
[9] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.
[10] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014.
[11] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In *CRYPTO*, 2013.
[12] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in Web applications. In *CCS*, 2011.
[13] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in Web applications: A reality today, a challenge tomorrow. In *S&P*, 2010.
[14] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, and T. Anderson. Radiatus: Strong user isolation for scalable Web applications. *Univ. Washington Tech. Report*, 2014.
[15] M. Christodorescu. Private use of untrusted Web servers via opportunistic encryption. In *W2SP*, 2008.
[16] Ciphercloud. http://www.ciphercloud.com.
[17] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS*, 2006.
[18] M. Egorov and M. Wilkison. ZeroDB white paper. *CoRR*, abs/1602.07168, 2016.
[19] English-words. https://github.com/dwyl/english-words.
[20] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.
[21] R. Fischer, M. Seltzer, and M. Fischer. Privacy from untrusted Web servers. *Yale Univ. Tech. Report YALEU/DCS/TR-1290*, 2004.
[22] Gitzero. https://www.gitzero.com.
[23] E.-J. Goh. Secure indexes. http://eprint.iacr.org/2003/216.
[24] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 1996.
[25] S. Greif. The allow & deny security challenge: Results. https://www.discovermeteor.com/blog/allow-deny-challenge-results/, 2015.
[26] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. Cryptology ePrint Archive, Report 2016/895, 2016. http://eprint.iacr.org/2016/895.
[27] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *CCS*, 2014.
[28] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. ShadowCrypt: Encrypted Web applications for everyone. In *CCS*, 2014.
[29] X. Huang and P. Malacaria. SideAuto: quantitative information flow for side-channel leakage in Web applications. In *WPES*, 2013.
[30] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[31] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.
[32] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun.

Verena: End-to-end integrity protection for Web applications. In *S&P*, 2016.

[33] S. Keelveedhi, M. Bellare, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *USENIX Security*, 2013.

[34] B. Lau, S. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A mimicry privacy shield–a system's approach to data privacy on public cloud. In *USENIX Security*, 2014.

[35] J. Li, M. N. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[36] W. M. Liu, L. Wang, K. Ren, P. Cheng, and M. Debbabi. k-indistinguishable traffic padding in Web applications. In *PETS*, 2012.

[37] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci. HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows. In *NDSS*, 2011.

[38] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 2011.

[39] Meteor. https://www.meteor.com.

[40] MongoDB. https://www.mongodb.org.

[41] Navajo Systems. https://www.crunchbase.com/organization/navajo-systems.

[42] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.

[43] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *S&P*, 2014.

[44] openSNP. https://opensnp.org.

[45] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind Seer: A scalable private DBMS. In *S&P*, 2014.

[46] Perspecsys: A Blue Coat company. http://perspecsys.com.

[47] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[48] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *NSDI*, 2014.

[49] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. https://eprint.iacr.org/2013/508.

[50] The Porter stemming algorithm. http://tartarus.org/˜martin/PorterStemmer.

[51] Preveil. http://www.preveil.com.

[52] K. P. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *SoCC*, 2011.

[53] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *S&P*, 2015.

[54] Skyhigh Networks. https://www.skyhighnetworks.com.

[55] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *S&P*, 2000.

[56] The Stanford NLP library. http://nlp.stanford.edu/software.

[57] T. Steinhauer. Mylar - ported to Meteor v1.1. https://github.com/strikeout/mylar, 2015.

[58] Stilgherrian. Encryption's holy grail is getting closer, one way or another. http://www.zdnet.com/article/encryptions-holy-grail-is-getting-closer-one-way-or-another/.

[59] M. Tamayo-Rios and N. J. H. Lai. An implementation of KFHE with faster homomorphic bitwise operations. https://github.com/kryptnostic/krypto/blob/develop/krypto-lib/src/main/V2/V2Specification.pdf.

[60] Ubuntu IRC logs. http://irclogs.ubuntu.com.

[61] Virtru Corporation. End-to-end data protection with Virtru encryption as a service (EaaS). *Virtru Tech. Report*, 2015.

[62] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*, 2012.

[63] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated detection and quantification of side-channel leaks in Web application development. In *CCS*, 2010.

[64] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, 2016.

12. POSTSCRIPTUM

The published Mylar paper [48] claims that "Mylar allows users to share keys and data securely in the presence of an active adversary." This statement was repeated on the public Mylar website.[2] At some point between April and July of 2016, while this paper was undergoing peer review and before it became public, all claims that Mylar protects data and queries from active attacks were erased from the Mylar website[3] without any explanation of why they were removed or acknowledgment that they were ever there. As of this writing, the "Security guarantees and use cases" page states that Mylar provides protection for users' queries and data only against a passive attacker. This directly contradicts the published Mylar paper [48].

The page currently asserts that Mylar is secure against active attacks when used in combination with Verena [32]. No evidence is provided for this assertion, and it is not mentioned anywhere in the Verena paper, which aims to protect the integrity of database queries and webpages returned from the server but does not prevent the server from executing other malicious code. Furthermore, neither Mylar, nor Verena hides access patterns, therefore neither is secure against persistent passive attacks.

[2]https://web.archive.org/web/20160422082354/http://css.csail.mit.edu/mylar/
[3]https://css.csail.mit.edu/mylar/