Side-Channel Attacks on Shared Search Indexes

Liang Wang*, Paul Grubbs†, Jiahui Lu‡, Vincent Bindschaedler§, David Cash¶, Thomas Ristenpart†
*UW–Madison †Cornell Tech ‡SJTU §UIUC ¶Rutgers University

***Abstract*—Full-text search systems, such as Elasticsearch and Apache Solr, enable document retrieval based on keyword queries. In many deployments these systems are multi-tenant, meaning distinct users' documents reside in, and their queries are answered by, one or more shared search indexes. Large deployments may use hundreds of indexes across which user documents are randomly assigned. The results of a search query are filtered to remove documents to which a client should not have access.**

We show the existence of exploitable side channels in modern multi-tenant search. The starting point for our attacks is a decade-old observation that the TF-IDF scores used to rank search results can potentially leak information about other users' documents. To the best of our knowledge, no attacks have been shown that exploit this side channel in practice, and constructing a working side channel requires overcoming numerous challenges in real deployments. We nevertheless develop a new attack, called STRESS (Search Text RElevance Score Side channel), and in so doing show how an attacker can map out the number of indexes used by a service, obtain placement of a document within each index, and then exploit co-tenancy with all other users to (1) discover the terms in other tenants' documents or (2) determine the number of documents (belonging to other tenants) that contain a term of interest. In controlled experiments, we demonstrate the attacks on popular services such as GitHub and Xen.do. We conclude with a discussion of countermeasures.

***Keywords*-side channels; SaaS security; elasticsearch**

I. Introduction

Modern cloud services provide full-text search interfaces to enable users to easily navigate potentially large document sets. Search systems such as Elasticsearch [14] and Solr [48] are both used by individual enterprises and offered as hosted services for other companies. Databases such as MySQL include similar search interfaces for columns containing unstructured text [35].

The canonical search API allows querying one or more keywords (or *terms* as they are usually called) to obtain an ordered list of matching documents. The response may additionally provide a real-valued score for each document. Which documents to return and their scores are determined using a relevance algorithm, most often term-frequency inverse-document frequency (TF-IDF) [30, 54] or one of its variants such as BM25 [53]. The TF-IDF score of a document is proportional to the ratio of the term frequency (the number of times a term appears in that document) to the logarithm of the total number of documents divided by the document frequency (the number of documents containing the term at least once). To compute these scores quickly, the search system maintains an inverted index that contains precomputed document frequencies for each term and term frequencies for each document-term pair.

Maintaining an index incurs overhead, and so best practice guides [12, 34] suggest configuring multi-tenant search systems to use shared indexes: each index is computed over (many) different users' documents. This configuration can additionally improve search efficacy because the document frequencies of other users' documents may help make relevance scores more accurate. When indexes hold private data, search APIs must be carefully configured to return only results for which the querying user has read privileges. The industry-standard method (see for example [12,22,33,34,59]) for searching with a multi-tenant index works in two steps. First, when user u issues a search query, the system forwards the query to the multi-tenant index, which returns results that may include documents to which u does not have access rights. Next, the systems post-processes the list of results to filter out any documents u should not have access to, and returns the remaining results. This is referred to as filtering; see Figure 1 in §III.

This filtering-based approach includes a side channel: one user may be able to determine the document frequency of a term, thereby potentially inferring if other users' documents include that term. This observation was first made by Büttcher and Clarke [7] in the context of local file systems. But to date no side-channel attack has been demonstrated exploiting the observation, and, as we shall see, doing so requires overcoming a number of significant challenges.

This paper. We provide the first treatment of logical side-channel attacks on modern multi-tenant search services. We begin by investigating representative open-source systems and assessing whether the basic document frequency (DF) side channel mentioned above exists. We setup local installations of systems including Elasticsearch/Solr and MySQL, following best practice guides for multi-tenant search. In all systems surveyed, we confirm that DF leakage can occur.

Despite this, and akin to early work on more well-studied side channels such as those based on CPU caches [4,38,40], it is not a priori clear how an attacker can exploit DF scores in realistic settings. In modern multi-tenant infrastructures, there exist a number of challenges: the precise scoring functions used in real services are proprietary and unknown, a user's documents may be assigned to one of many possible indexes, noise in relevance scores arises due to the number

of files fluctuating frequently over time as users add or remove files, indexes may not remove keywords from an index even after a file is deleted, many APIs rate limit queries to search indexes, and more. It could also be, of course, that some sophisticated enterprise services do deploy proprietary countermeasures.

We develop STRESS[1] attacks, which consist of a multi-step methodology for exploiting DF side channels. Our attacks overcome the challenges mentioned and, ultimately, realize the first demonstrated cross-user side-channel attacks in this setting.

Our framework begins by providing three low-level tools that aid in attacks. First is a new approach that we call *score dipping*. It provides a basic ability to infer, for a single index that includes an attacker document, whether there exists another document on the index containing a specific keyword. The insight is that an attacker can abstract away details of the scoring function, relying only on the assumption that scores decrease with increasing DF. Score dipping improves on prior ideas [7] for how to exploit the side channel because it can be used without precise knowledge of the scoring function used by a service plus, as we will experimentally show, it is robust to noise.

In large-scale systems there will be a large number of shards across which an index is split, and score dipping alone is not effective in this setting. Each shard can be thought of as a logically isolated portion of the index, and a scoring function only takes into account documents assigned to the shard. In targeted attacks against a particular victim, attackers must have the ability to place one or more documents on the same shard as the target's documents. But the search service controls shard assignment, typically randomly load balancing new documents across them. Thus attackers are faced with an analogous issue to the co-location challenge that must be overcome in cross-user side-channel attacks in public infrastructure-as-a-service (IaaS) [23,43,51,55,57] or platform-as-a-service (PaaS) [60] clouds.

As a first step towards attacking a multi-shard system, we show how to use score-dipping to construct our second low-level tool, called *co-shard tests*, against multi-shard systems. Our co-operative co-shard test allows an attacker to determine if two attacker-owned documents have been assigned to the same shard. Specifically, we use score-dipping to build a covert-channel between different documents that are owned by the same user or different, co-operating users, and hence determine if they are on the same shard. This channel however does not on its own achieve co-location on a shard with a victim's documents, since the channel is only between attacker documents.

We next propose a new and different approach to obtaining co-location with a victims' documents, and in the process also learn about the service backend. Instead of trying to just obtain co-location with a target, we use our co-operative co-shard test to build our third low-level tool that we call a *shard map*: A set of documents in which each document is present on a distinct shard. We will show that it is possible even on large-scale services to build complete shard maps, i.e., ones that appear to cover all shards used by the system. A complete shard map already reveals the number of shards, but more damagingly will be useful as a preliminary step for more granular attacks. We show how to do the following using a shard map:

- *DF estimation*: We can reverse-engineer each shard's unknown scoring function using a curve-fitting strategy. This yields a function that maps a term's search score to an estimate of that term's DF on a shard. This allows, among other things, trending: the ability to count the number of (private) documents mentioning a word. For example, if one knows an identifier used by a particular company using GitHub, our technique allows counting the number of private files they have stored on the service.
- *Brute-force term recovery*: We can use our shard map to test if a given term exists anywhere in the system, thereby allowing an attacker to brute-force recover moderately high-entropy values from victim repositories. While the side-channel attack does not reveal to the attacker which repositories contained the term, we propose scenarios that nevertheless allow the extraction of sensitive information such as credit card numbers, social security numbers, passwords, and more.

We evaluate the viability of STRESS attacks in practice with case studies of GitHub, Orchestrate.io, and Xen.do. As a sample of our results, we demonstrate on GitHub (in a responsible way, see discussion in §VI) that one can build a 191-document shard map in 104 hours with a single account. We estimate that it would take about a day to brute force a space of 10^6 possible terms on every shard. For example, if one knows the BIN and last four digits of a credit card number stored in a GitHub repository then the rest of the card's number can be brute-forced in under a day with 191 free accounts (c.f., [18] for discussion of credit card numbers and other information being stored on GitHub). We also discuss how stripping relevance scores (but still ranking documents) is likely to be inadequate.

We conclude by discussing potential countermeasures, suggesting in particular a new countermeasure which replaces actual document frequencies with ones trained from public data. We discuss the merits of this approach and routes to deployment.

II. Background

Ranked keyword search. A fundamental information-retrieval task is finding relevant text documents for keyword search queries. Let D denote a corpus of text documents.

[1] **S**earch **T**ext **RE**levance **S**core **S**ide channel

For our purposes, a document consists of a bag-of-words representation; each word is a string that we refer to as a *term*. Our concern will be search systems that expose an API allowing keyword search, i.e., a client can execute a remote procedure call SEARCH(t) for a term t that returns an ordered list of documents $d_1, \ldots, d_n$ for some (typically fixed, small) n, sorted from most to least relevant. In addition to the list, many APIs also return relevance scores $s_1, \ldots, s_n$ for which $s_i \in \mathbb{R}$ indicates the estimated relevance of d_i to the query. A higher score indicates stronger relevance, and so $s_i \geq s_j$ for $i < j$. Many search routines allow more complex queries such as disjunctions and/or conjunctions of keywords, but we will primarily focus on single-term search.

This work will only consider unstructured document search in which documents have no semantic relationships. This distinguishes it from settings such as web or social network search.

In our unstructured search context, the most prevalent way of ranking is via term frequency/inverse document frequency (TF-IDF) scoring [30, 54]. Let D denote the document corpus, $N = |D|$ the number of documents, t be any term, and d be an arbitrary document in D. Define $\text{df}(t, D) = |\{d \in D \,|\, t \in d\}|$ to be the number of documents in D containing term t. This is referred to as the *document frequency (DF)*. We define the *term frequency* $\text{tf}(t, d)$ as the number of times the term t appears in the document d. We define the *inverse document frequency* by

$$\text{idf}(t, D) = 1 + \log \frac{N}{\text{df}(t, D) + 1} \,.$$

The TF-IDF score for the relevance of document d to the single-term query t is

$$\text{score}_{\text{tf-idf}}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D) \tag{1}$$

The TF-IDF score for a multi-term query $q = (t_1, \ldots, t_m)$ is

$$\text{score}_{\text{tf-idf}}(q, d, D) = \sum_{i=1}^{m} \text{score}_{\text{tf-idf}}(t_i, d, D) \tag{2}$$

We note that the $\text{idf}(t, D)$ term is independent of the document d, and it is intuitively used to weight terms for multi-keyword queries.

There are many variants of the basic TF-IDF score that include other parameters and normalizing terms, and also alternative definitions of term frequency and inverse document frequency. Indeed, the live systems we experimented on used more complicated variants of TF-IDF, but we will use this simple formulation for the time being.

To implement TF-IDF scoring and search, a system generates an *inverted index*. For each potentially-searched keyword t, one stores $(t, \text{idf}(t, D))$ at the head of a list of $(d, \text{tf}(t, d))$ pairs. This allows fast computation of the TF-IDF and the documents that should be returned in response to the query.

TF-IDF scoring has many advantages and has intuitive probabilistic and geometric interpretations (c.f., [54]). However, in applications it is often useful to account for other factors in determining relevance, like the length of a document compared to the average length of all documents in the index. The BM25 scoring method incorporates this additional information [53]. As our eventual attacks will focus on TF-IDF, we omit the details and note that our attacks should extend to use of BM25.

Multi-user indexes and the DF side channel. More than 10 years ago, Büttcher and Clarke [7] pointed out a potential side channel when using TF-IDF scoring on multi-user indexes. A multi-user index is simply one generated over a document corpus D that includes files from different users with different permissions. To perform a search on behalf of a user u, one uses the index to compute a ranked list of documents $(d_1, \ldots, d_n)$ with scores $(s_1, \ldots, s_n)$. Then one post-processes the list to redact documents (and their scores) not accessible by u, resulting in a smaller list $(d'_1, \ldots, d'_n)$ with scores $(s'_1, \ldots, s'_n)$ that are returned to u.

Büttcher and Clarke pointed out that systems like Apple's filesystem search service Spotlight are multi-user. While permissions models can be rather complex, we will focus our attacks on settings in which users should only be able to read the files they own, and no others.

In this context, Büttcher and Clarke show that $\text{idf}(t, D)$ forms a potentially exploitable side channel that violates document confidentiality, even if a search index properly filters out search results on documents not owned by the user performing the search. This channel will allow an adversary to learn partial information about *document frequency*, so we call this the *DF side channel*.

To demonstrate their observation, consider an adversarial user Eve that wants to determine the number of documents that contain a term t^*. For example, it may be that Eve wants to learn whether another user Alice has a document $d_A = \{t^*\}$ stored on the system. Then, there is a simple attack exploiting the scoring function as a side channel.

Eve generates two documents $d_1 = \{t^*\}$ and $d_2 = \{r\}$ where r is some random term of length sufficient to ensure that it will not appear in any user document. Then Eve issues two search queries: First for SEARCH(t^*), which returns document d_1 with score s_1, and then for SEARCH(r) which returns document d_2 with score s_2. Even though SEARCH only returns results related to documents owned by Eve, Eve can anyway use s_1 and s_2 to infer information about other users' documents. By construction Eve has arranged that $\text{tf}(t^*, d_1) = \text{tf}(r, d_2) = 1$ and $\text{df}(r) = 1$. Thus referring back to (1), Eve knows that

$$\begin{aligned} s_1 &= \text{tf}(t^*, d_1) \cdot \text{idf}(t^*, D) = 1 + \log \frac{N}{\text{df}(t^*, D) + 1} \\ s_2 &= \text{tf}(r, d_2) \cdot \text{idf}(r, D) = 1 + \log N/2 \,. \end{aligned}$$

Thus Eve now has two equations in two unknowns and can solve for N and $\text{df}(t^*, D)$. The latter reveals how many documents in D contain t^*. Under the assumption that t^* would only appear, if at all, in d_A (e.g., because it is rather high entropy), then Eve can conclude that Alice's document contains t^*.

The attack as described requires scores, but Büttcher and Clarke detail another attack that uses only the order of documents returned by a multi-term search to approximately bound $\text{df}(t^*, D)$. They also mention that their techniques could be used to perform brute-force attacks, repeatedly using the side channel for different possible values for the target term t^*.

Büttcher and Clarke conjecture that this side channel could be used to recover information from real multi-user search indexes, but they do not demonstrate any working attacks. So while the DF side channel has been known to exist in theory since 2005, we are unaware of any investigation into its exploitability in practice, despite the widespread deployment of multi-user indexes. As we will discuss in the next section, there appear to be inherent challenges to building real attacks, including some noted by Büttcher and Clarke and others that we uncover related to distributed system design.

Other storage system side channels. Other side channels on search indexes and databases have been developed. Gelernter and Herzberg [17] show how to exploit a cross-site timing side channel to test for the presence of terms in a target search index. Our attack does not require malicious code injection, but does enable term extraction from a search index. Futoransky et al. use a timing side channel on insertions into MySQL and MS SQL databases to extract private information [16]. They observe that insertions take longer if a new virtual memory page is written, and use a divide-and-conquer approach to learn private terms. Their side channel is much harder to exploit than ours because it requires fairly high-precision timing measurements.

III. Survey of Multi-tenant Search Side Channels

The basic DF side channel has only been discussed in theory, and it is unknown what search systems, if any, are vulnerable. We therefore begin by surveying existing open-source multi-tenant search systems, and experimentally confirm that the DF side channel exists in every setting we consider.

Elasticsearch. There are a few prominent systems for implementing full-text search on unstructured documents. Lucene [27] is a Java library which implements the building blocks of a search index, including functionality such as document tokenizers and query parsers. It also implements common data structures used for indexing. Elasticsearch (ES) and Solr are two libraries that implement sharding and cluster management for Lucene indexes. ES and Solr are widely used in industry due to their efficiency and scalability.

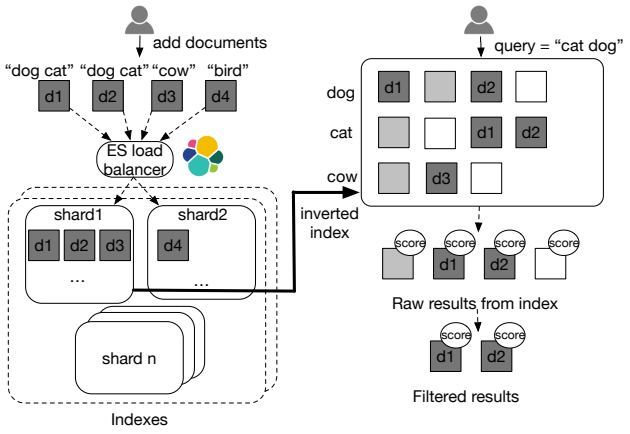

Figure 1: A typical multi-tenant ES deployment consisting of several shards, and an example of inverted indexes and query filtering in ES. Documents from different users are in different colors.

An architectural diagram of a canonical ES deployment is depicted in Figure 1. We assume a multi-tenant setting, in which multiple distinct user accounts have their documents indexed. In large deployments, a single server is insufficient to handle search queries, and so one instead builds separate indexes across multiple *shards*. A common way of load balancing across shards is to assign users at random to a shard, meaning all their files will be in that shard. Should individual users have many files, it may be needed to have more granular load balancing. For example, one can assign each individual document to a shard randomly when the document is uploaded, or there may be other logical groupings of documents. For example in GitHub, users may have multiple git repositories, and as we will see later GitHub load balances across shards at the granularity of repository.

Lucene, Solr, and ES are all open-source projects, and typical configurations for the ranking function can be found online in forums [29,42]. The default ranking used by Lucene (and so, in turn, by ES and Solr) is a variant of TF-IDF given by the equation

$$\text{score}_{\text{es}}(q, d) = \sum_{t \in q} \frac{\rho_{q,d} \cdot \beta_t \cdot \text{tf}(t, d) \cdot \text{idf}(t, D)^2}{\sqrt{|d|} \cdot \sum_{t \in q} \text{idf}(t, D)^2} \quad (3)$$

The *query coordination factor* $\rho_{q,d} = \sum_{t \in q} \mathbb{I}(t \in d)/|q|$ boosts documents that contain more terms matched by the query. It counts the number of query terms matching the document and divides by the total number of terms in the query $|q|$. The *per-term boost function* β_t allows customization of scores based on important application-specific terms. The division by $\sqrt{|d|}$ is what's referred to as the *field-length norm*, and it simply acts to normalize relative to the size of the document. In some configurations, the field-

length norm is combined with an index-time field-level boost, which for our purposes would simply change β_t.

An attacker that retrieves a score $\text{score}_{\text{es}}(q, d, D)$ on their document d will know most of the terms in the right hand side of (3), with the only unknowns being the value N, $\text{df}(t, D)$ for each $t \in q$, and, if the configuration is unknown, the boost function and other factors. When the configuration is known, this is just a (log-linear) equation in two unknowns. In this case the attack applies as in §II.

To test if ES has the DF side channel, we set up a local installation of ES version 2.3.4, and configure it to use one shard with zero replicas. We leave the other configuration options default. Following the suggestions provided by ES [34], we adopt the shared index strategy, create two tenants *alice* and *bob*, and add a *tenant-id* field to a document data structure to specify the document owner. A document data structure is a piece of JSON data that consists of three fields: a *tenant-id* field, a *name* to store the document name, and a *content* field to store the document content.

We implement and test two common search filtering mechanisms to enforce access control: filtering on *tenant-id* in the query [13] and filtered index alias [1]. The former excludes the documents that fail to meet the filtering conditions, e.g., excluding documents whose *tenant-id* $\neq$ *alice* for queries issued by the tenant *alice*. The latter works in the same way as the former, but it makes search filtering easier by allowing a user to create an alias name for a set of filtering conditions.

We first generated a unique term t, added a document $d_a = \{t\}$ as *alice* (the *tenant-id* of d_a is set to *alice*), and got a score $s_a = \text{score}_{\text{es}}(t, d_a)$. Then, we added a document $d_b = \{t\}$ with *tenant-id* = *bob*, and measured $s'_a = \text{score}_{\text{es}}(t, d_a)$. We observed that $s'_a < s_a$, and s'_a decreases as more documents that contain t are added by the tenant *bob*. Finally, we deleted all the documents associated with *bob*, measure again as *alice* to get $s''_a = \text{score}_{\text{es}}(t, d_a)$, and saw s''_a is the same as s_a. We observed the same results under different filtering mechanisms. These observations strongly suggest that one can infer if there are other documents containing a term by examining relevance scores; therefore, the DF side channel exists in ES.

MySQL. We set up a MySQL 5.6 server using its default configurations. In MySQL-based multi-tenant applications, a common design is multi-tenant-per-table, that is, storing all tenant's data in the same table, with a *tenant-id* field to distinguish each tenant's records [32]; then, to get the records only associated with a tenant *alice*, one can issue SQL queries with a condition *tenant-id* = *alice*. We achieve multi-tenancy in MySQL based on this design pattern. Our simple multi-tenant application uses one table, each record in which corresponds to a document. A record has the same three fields as the documents in the ES tests. To enable full-text search in MySQL, we build a `FULLTEXT` index on the *content* field [35]. As a result, all the tenants share the same index.

We conducted the same tests as we do for ES, and observed the same result: the relevance score of a document for a given term will be affected by the documents that contain the same term, even if these documents are owned by other tenants. The result also suggests that the DF side channel exists in MySQL.

Other vulnerable systems. We found five vulnerable cloud-based search services using the similar methodology as in ES and MySQL. A cloud-based search service aims to provide scalable, easy-to-manage full-text search for web or mobile applications. An application can use it to build and maintain indexes on its data, and handle search requests. Such services usually charge the applications based on the amount of storage used or the number of requests processed. All of the systems we considered provide RESTful APIs and reveal relevance scores. Four of the services are built on ES (i.e., hosted-ES services), including AWS Elasticsearch [3], AWS CloudSearch [2], Searchly [45] and bonsai [6]. It's easy to confirm that they inherit the DF side channels from ES. We investigated these four due to their popularity, but there are many other hosted-ES services that could also have the vulnerability. One vulnerable system called Swiftype implements its own search engine [50].

Note that even if the side channel exists in a hosted search service, an application built atop that service will not necessarily have the DF side channel. For example, an application could conceivably assign each of their users to an independent index. However, due to the costs of cloud-based search services, application developers would typically prefer to use shared indexes. In Swiftype, a basic plan ($299 per month) only provides one index for usage, while a business plan ($999 per month) provides up to three indexes. In Searchly, a professional plan ($99 per month) offers 13 indexes. So if a multi-tenant application is built atop the service, the application's users will share the same indexes and might be vulnerable to information leakage. Looking at the case studies advertised by Swiftype [49], we realized some of them are indeed multi-tenant applications. We also noticed that Heroku uses Swiftype and Searchly as its search add-on [21], suggesting the DF side channel might be inherited by Heroku-based applications.

Non-vulnerable systems. We also investigated PostgreSQL [41], CouchBase [9], crate.io [10], Searchify [44] (not to be confused with Search*ly* above), and Google App Engine [20]. Our experimentation suggests that these systems do not exhibit the DF side channel, primarily because they appear to use independent indexes for different tenants.

IV. The DF Side Channel in Enterprise Systems

In the controlled or partially-controlled settings above, we verified that the DF side channel was present. Enterprise

search systems however introduce a number of complications, and it is at first unclear if the DF side channel can be exploited. In this section we discuss the major issues that must be addressed in understanding if such a system is vulnerable in practice.

Hidden relevance formulae. An adversary may not know which TF-IDF variant is being used. The space of TF-IDF variants is large, with several different possible choices for $\text{tf}(t, d)$ and $\text{idf}(t, D)$ other than what we defined above, as well as different formulas for combining them to compute a score. These choices may use more features than we specified above, such as the length of the document. Scores for multi-term queries may also be computed via more complicated formulae and have constants that can be hand-tuned for a given application. Finally, we found that some services implemented scoring via ad hoc methods that took into account last-touched time or the order of terms in a query (i.e., treating the query (t_1, t_2) differently from (t_2, t_1)).

When the adversary does not know the scoring function it can no longer implement the algebraic attack from the previous section. This issue was cited by Büttcher and Clarke as preventing them from carrying out the attack on Spotlight. Instead, other techniques must be developed that are robust to variations in the scoring function.

Sharding. As mentioned above, enterprise search systems perform load balancing by dividing the document corpus into shards, which are essentially independent indexes. Sharding may be done per-document, per-collection, per-user, or via some other metric like creation time. Replica shards (i.e., copies of shards) are used to increase query throughput.

A side channel will only exist when scores are computed as a function of private documents, which usually means that an adversary's document must be on the same shard as victim data that it hopes to extract. Since search system interfaces do not expose information about sharding, this poses a further challenge for an adversary, who will need to arrange for its documents to be *co-sharded* with victim data, and also not be misled when documents are placed on shards without victim data.

Noise. The production search systems we experimented with displayed noisy behaviors that make attacks more difficult. For instance, in all of our experiments on live systems we observed that relevance scores constantly changed, and issuing the same search multiple times will result in different relevance scores on almost every query (see Figure 2 in §V).

Some of the noise is likely due to variations in the value N, the number of documents in the shard, which is changing constantly as many users write data to the index. This foils the algebraic attack of Büttcher and Clarke, because obtaining two scores computed with same value of N may be difficult or impossible, and anyway one cannot tell when this is the case.

Consistency and deletions. We also observed occasional larger changes in relevance scores likely due to other systems behavior. ES and similar systems have complex mechanisms for propagating newly written data into shards which maintain some form of consistency as segments of data are merged into shards. However, they do not maintain consistent relevance scores when data are merged, causing further difficulties for attacks that depend on fine-grained measurements in score changes. In some services it took up to two minutes for a change in a document to result in a change in relevance scores, slowing possible attacks. We also noticed that searches may be issued in quick succession yet return greatly differing scores, likely due to a segment merge in between the queries.

Deletions are implemented lazily by marking documents for deletion and later expunging them via a background process (c.f., [11]). The DF values are incremented quickly (i.e., after a minute) but apparently only reduced after expunging. Thus an adversary who hopes to delete documents as part of an attack is required to wait until its documents have been expunged before it can observe a change in the score function. Further complication arises because an adversary will not know which shard its document was present on and deleted from.

API restrictions. Search interfaces are rate-limited, both in terms of queries per time period (e.g., 5,000 queries per hour on GitHub) and their total size (e.g., 128 bytes to describe the keywords in the query). A very weak side channel may be mitigated if it requires an infeasible number of queries, or large queries.

Tokenizers and API interfaces often strip special characters, treating them as whitespace. So, for example, a hyphenated number XXXX-YYYY may be tokenized into two terms XXXX and YYYY, which have independent DFs. This affects the information available via the DF side channel.

Bystander data. An adversary who targets a victim or victims will need to carry out the attack in the presence of a possibly large number of *bystander* users whose data are uninteresting to the adversary. These data are not known to the adversary but will be used in relevance score calculations using a formula also unknown to the adversary. These bystanders are also actively writing and deleting data in the index.

The primary effect of bystander data in our work is their effect on *false positives*, which we return to below. The essential issue is that an adversary is *only* able to compute the DF of a term on a given shard. It is thus impossible to distinguish with certainty between cases where a victim document or a bystander document contains the term (and causes its DF to be non-zero). In some cases, we will argue that it is possible to use contextual clues, like the presence of other terms in the same shard, to limit false positives.

V. STRESS ATTACKS

A. Attack Goals and Notation

We consider services that allow an adversary to write data and also use a search interface to retrieve relevant documents with scores. We assume that access control is implemented properly, meaning that other users' private documents are not returned to the adversary, or otherwise leaked directly through the interface.

Our adversary's intermediate goal will be to determine the DF of some given terms $t_1, \ldots, t_q$. (Later we will discuss attacks built on this capability.) We start from simplest case, where each $\mathrm{df}(t_i, D)$ is either 0, meaning no one has a document containing t_i, or is positive, meaning that it appears at least once. The document set D is changing constantly due to bystander activity, but we assume that the terms t_i are not written or deleted in a short period for the attack, and also that the size of the shard does not change dramatically.

In a system with a single shard, the DF of a term is well-defined. But in a multi-shard system, each term will have a shard-specific DF. To define the attack, we model the document set D as being partitioned into sets $D_1, \ldots, D_{n_{\text{SHRDS}}}$, where n_{SHRDS} is the number of shards. In this case, our adversary should determine, for each shard, the tuple $(\mathrm{df}(t_i, D_j))_{i=1}^{q}$ where the shard holds documents D_j. That is, on each of the shards, it should determine an estimate of DF of each term on that shard. We note that this attack will allow an adversary to detect that, say, t_1 and t_2 happen to occur together on the same shard, which is stronger than simply detecting that they occur somewhere in the larger system.

Notation. In this section, we fix a term-sampling algorithm RNDTERM that outputs a fresh random term that is assumed to never appear in bystander documents. In our experiments choosing a uniformly random 16-character alphabetic string was sufficient.

We also fix some notation for documents, terms, and the interface into the search service. Documents will be treated as sets of terms in our notation. In reality they are strings of text but the order of terms does not matter for scoring. We assume the service provides the ability to write documents, which we formalize as $\textsc{write}(d, S)$, where d is a reference to a document and S is a set of terms. This operation will overwrite the entire document to consist of exactly S. Next we will write $\mathrm{score}(t, d)$ to mean the score of document d returned by the service for a search for the term t (note that multiple calls to $\mathrm{score}(t, d)$ may return different scores, and we are not fixing a document set D — $\mathrm{score}(t, d)$ is defined according to the service's response).

B. Basics of Exploiting the Side channel

All of our attacks will be built on a fundamental property of all in-use relevance scoring functions we are aware of: As a term t becomes more common (i.e. its DF increases), the term weight decreases. In the case of basic TF-IDF, the term weight is $\mathrm{idf}(t, D)$, but this is true for all of the variants we have encountered. Indeed it is intentional: A more common term should be given less weight in multi-term queries. An adversary that does not know the scoring function can still take advantage of this property.

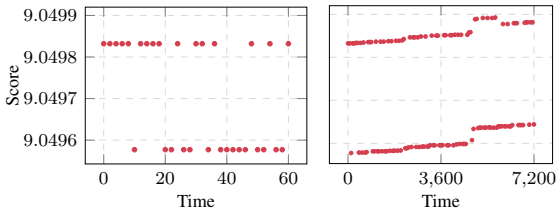

Figure 2: Example relevance scores returned by the GitHub API when searching for the same term several times. **Left** shows the score variations in 60 seconds, and **right** shows the score variations in 2 hours. The time intervals for **left** and **right** are 2 s and 60 s respectively. Y-axis does not start from zero.

Score-dipping. We use this property to build what we call the *score-dipping* attack to determine if a term t appears somewhere in a shard of a system that uses an unknown scoring function. For now, assume that our attack owns a document d on the shard of interest. The attack first writes two terms t and r to d by invoking $\textsc{write}(d, \{t, r\})$, where r is a long random term (not present in another document). Then it requests searches for t and r. The search for t returns only d with some score s, and the search for r returns only d with some score s'. The attack checks if $s < s'$, and if so it guesses that t is present in the shard.

If t was not present in the system originally, then after writing to d, we have both $\mathrm{df}(t, D) = \mathrm{df}(r, D) = 1$ (where D is the document set in the shard) and the searches should return the same score. But if t was already present, then $\mathrm{df}(t, D) > 1$ and thus s should be lower.

To work on a real system this attack must be extended to tolerate noise in the scores returned. Due to bystander activity, we will observe differences in s and s' even when the terms t, r have the same DF. (Indeed just searching for the same term twice will produce different scores. See Figure 2.) Bystander activity that incidentally decreases s or increases s' may cause the attack to output a false positive.

To mitigate this effect, we observed that the effect of changing a DF from 0 to 1 (or some larger number) caused a noticeably larger change in the relevance score than background bystander activity. In our attack, we perform several measurements on a shard to compute the typical variation when searching for terms with DF equal to 1 and 2. We can then determine a threshold for when the score is small enough to indicate that a term's DF is larger than 0.

C. Plan for the Attacks

We now begin building towards an attack on a multi-shard system. In all multi-shard systems, the mapping of documents to shards is handled by some load balancing strategy that is not directly exposed in the interface. (To an outside user, sharding is meant to be transparent, though it does result in variation of relevance scores for the same query across shards.) Thus an adversary cannot directly see in which shards its documents reside, or directly control the shard on which a newly-created document is placed.

The hidden layer of load-balancing creates several difficulties. If we try to repeat the score-dipping attack several times without considering which shard we are on in each run, we will not know when we have explored all of the shards. Some systems might have hundreds of shards, and it may take a minute or more for a write to possibly change a relevance score. API rate-limiting can further slow naive attacks.

A more serious problem is that naively repeating the single-shard attack is not even a correct strategy when a service processes deletions lazily, meaning it only reduces DFs when expunging. In this setting, naive repetition will *detect its own documents*, which artificially increase the DF of terms of interest, during the attack. Concretely, suppose one stage of the attack writes a document $d = \{t, r\}$ containing the term of interest, and that deleting d, or removing t from d does not reduce the DF of t in the shard holding d. Then later stages of the attack that happen to return to the same shard will detect that the DF of t is non-zero, but this will be due to d and not victim documents.

Below we show how to mitigate the difficulty of attacking without deleting via a technique we call *shard mapping* that reverse-engineers the number of shards in the service and also places an adversary-controlled document on each shard. In addition to giving interesting information about a backend, shard mapping helps avoid the issues above, and also improves the efficiency of attacks.

We build two families of attacks using shard mapping: First we show how to quickly test for the presence of terms in other users' documents, allowing for what we call *brute force term extraction*. Second, we present a totally different approach called *DF prediction* that learns the scoring function on a shard and then attempts to predict DFs using the learned function.

D. Tool: Co-Shard Testing

We first build a tool that we will use a sub-routine: *co-shard testing*. This will efficiently determine if an adversary-owned document d_1 resides on the same shard as another adversary-owned document.

Our strategy uses a technique similar to the score-dipping attack and the details are given in Algorithm 1. The routine CoShardTest takes as input references to document d_1, and a set of documents M (not containing d_1), and determines if d_1 was co-sharded with any documents in M (but not which one). It also uses a service-specific constant δ that we set by hand (once for each service). The attack starts by selecting two random terms r and r'. Then it writes r and r' to d_1, and only r to the other documents. After waiting for the writes to propagate, the algorithm issues two searches for r and r', and records the score of d_1 in the searches as s and s' respectively. Finally it outputs true if s' is greater than s by more than δ.

Algorithm 1: CoShardTest

```
   Input     : Document d_1 and document set M
   Output    : True iff ∃d ∈ M : d_1, d on same shard
   Parameter: Integer δ > 0
1  r ← RNDTERM; r' ← RNDTERM;
2  WRITE(d_1, {r, r'});
3  foreach d ∈ M do WRITE(d, r);
4  SLEEP;
5  s ← score(r, d_1);
6  s' ← score(r', d_1);
7  if (s' − s) > δ then return True;
8  else return False;
```

This attack works based on the principles described. If d_1 was not co-sharded with any other document, then we have $\text{df}(r, D_j) = \text{df}(r', D_j) = 1$, where D_j is the shard containing d_1. But if d_1 and one (or more) $d \in M$ are on the same shard D_j, then $\text{df}(r, D_j) \geq 2$ and $\text{df}(r', D_j) = 1$, resulting in a noticeable change in the score.

In this algorithm, most of the time is spent in SLEEP on line 4. This is why we have chosen a fast version of co-shard testing, where we can test if a new document d_1 was co-sharded with some $d \in M$ without spending extra time to determine *which* document it was.

The final detail is fixing δ, which must be set so that we distinguish larger changes in the score from random variation. We experimented with each service by repeating several queries over a period of time, and setting δ to more than the maximum observed variation (see Figure 2 for an example of observed random variation).

Co-shard testing on stable services. On some services we noticed that searching for two terms with DF exactly 1 would return results with exactly the same score. On such stable services we can save time when co-shard testing many documents via the following strategy: create many documents, all containing the same random term r. Then request a search for r, which returns all of the created documents, and partition the documents returned by their scores. If the service is stable, the documents on the same shard will have the same scores, and otherwise their scores will likely differ. Thus our test can immediately filter to a subset of documents that are likely co-sharded, and then perform the co-shard test to verify correctness.

E. Tool: Shard Mapping

Our multi-shard attacks will start with a pre-computation phase that we call *shard mapping*, which aims to place

exactly one adversary-owned document on each shard in the system. We call a set M of documents with this property a *shard map*, and the goal of this subsection is to compute a shard map efficiently. Recall that this is non-trivial since the mapping of documents to shards is hidden by the interface. After this somewhat slower pre-computation set the adversary will be able to build attacks efficiently as we describe below.

Our method for computing a shard map M is as follows: Initialize a set M consisting of a single document d_1 (on some shard). Then create another document d_2, and use the co-shard test to check if d_1 and d_2 are on the same shard. If they are, then the attack discards d_2. If d_1 and d_2 are on different shards, then it adds d_2 to the map M. The attack continues creating further documents, except this time it tests for co-sharding with its documents in M before deciding that it has found a new shard and adding the new document to S. After some large number of runs that do not find a new shard, the adversary concludes that the set S consists of exactly one document on each shard of the system.

We denote our method by MAPSHARDS and it is given in detail in Algorithm 2. We repeatedly create a new document and test if it has landed in an "unmapped" shard using COSHARDTEST. If not, we discard the document. If, on the other hand, the document is on a new shard, then we add it to the map M.

Run-time analysis. In Algorithm 2 we assume a service-specific constant n_{MAX} has been fixed. We want to pick an n_{MAX} large enough to ensure that we eventually find every shard without wasting too much time in the attack, since each co-shard test requires a costly sleep to propagate writes.

To analyze the run-time we assume that each newly created document is assigned a uniformly random shard out of n_{SHRDS} possibilities. (Note that the actual shard assignment strategy being used by a target service could be more complex, so n_{SHRDS} estimated by MAPSHARDS would only be a lower bound.) Then the expected number of iterations before we have a document on every shard is given by the well-known *coupon collector problem* with n_{SHRDS} coupons (see for example [5]). A classic analysis tell us that the expected number of tries is close to $n_{\text{SHRDS}} \cdot (\ln(n_{\text{SHRDS}}) + 1.6)$, with tight tail bounds on deviations from the expectation.

Thus one can set n_{MAX} to be slightly larger than the coupon-collector prediction when one knows n_{SHRDS}, say, from technical information the service has released. Alternatively, one can simply guess n_{SHRDS} and run the attack until many iterations fail to find a shard. Let n_{FIND} be the number of shards found after k iterations, and n_{FAIL} be the number of consecutive iterations fail to find a shard after the k^{th} iteration. The probability of seeing n_{FAIL} iterations of failing can be calculated as $(n_{\text{FIND}}/n_{\text{SHRDS}})^{n_{\text{FAIL}}}$. Then, one can stop if the probability is smaller than a certain threshold. We took the latter approach in our attacks.

Algorithm 2: MAPSHARDS

```
   Parameter: Integer n_max > 0
   Output   : Shard map M
1  Create an empty document d_1;
2  M ← {d_1};
3  for j = 2, ..., n_MAX do
4      Create new empty document d_j;
5      if CoShardTest(d_j, M) = False then
6          M ← M ∪ {d_j};
7      else
8          Discard d_j;
9      end
10 end
11 return M
```

Optimizations and multi-mapping. We also implemented a slightly more complicated, but faster, variant of shard mapping. In each iteration of the main loop on line 3, we changed the algorithm to create *two* new empty documents $d_j^{(1)}, d_j^{(2)}$ instead of one. We then execute a version of COSHARDTEST to test if either $d_j^{(1)}, d_j^{(2)}$ (or both) landed on new shards. If neither did, we discard them both. If exactly one did, then we keep it and discard the other. If both landed on new shards, then we must test if they landed on the same new shard via another co-shard test. In principle this could be run with more than two new documents in each iteration but we found that mapping was fast enough with two.

A second optimization is to apply the faster co-shard test when the service returns stable scores. On some services this will increase the speed of the mapping attack substantially.

Later we show that some attacks can be sped up using multiple shard maps $M_1, \ldots, M_m$, where the first documents of all shard maps lie on the same shard, and second lie on the same shard, etc. On some services like GitHub this will be easy to construct due to their sharding policy, which places all files from a repository on the same shard. On others we can run shard mapping multiple times, and then use co-shard testing again to find one document from each map that is on each shard.

F. Attack 1: Brute-Force Term Extraction

We now build our brute-force term extraction attack using a pre-computed shard map M. Our attack will use M to quickly determine the DF of given terms on every shard of the system. More precisely, let B be a (potentially large) set of terms that we are interested in testing for, in a system with n_{SHRDS} shards. Our attack will return a tuple $(B_1, \ldots, B_{n_{\text{SHRDS}}})$ of sets of terms, where $B_i \subseteq B$ consists of the terms from B that are in the i-th shard of the system.

Our attack is given in Algorithm 3. It starts by initializing the sets B_i to be empty, and then iterates over each document in the shard map, writing a random term and all of the terms in B to the document. After waiting for the writes to propagate, it then tests for the presence of each $t \in B$ on the shards using score-dipping again with some threshold δ.

Algorithm 3: TERMEXTRACT

Input : Shard map $M = \{d_1, \ldots, d_{n_{\text{SHRDS}}}\}$, term set B
Output: $(B_1, \ldots, B_{n_{\text{SHRDS}}})$
Param : $\delta > 0$

Initialize all $B_i \leftarrow \emptyset$;
for $i = 1, \ldots, n_{\text{SHRDS}}$ **do**
  $r_i \leftarrow$ RNDTERM;
  WRITE$(d_i, \{r_i\} \cup B)$
end
SLEEP;
foreach $t \in B$ **do**
  for $i = 1, \ldots, n_{\text{SHRDS}}$ **do**
    $s_i \leftarrow \text{score}(r_i, d_i)$;
    $s'_i \leftarrow \text{score}(t, d_i)$;
    if $(s'_i - s_i) > \delta$ **then**
      $B_i \leftarrow B_i \cup \{t\}$
    end
  end
end
return $(B_1, \ldots, B_{n_{\text{SHRDS}}})$

This approach minimizes the number of costly sleep times by writing many terms to each file.

This technique crucially depends on the shard map to avoid incorrectly dipping the score for a term t with the attacker's own write operations. Also we note that if we have multiple shard maps then we can partition B and run independent instances of TERMEXTRACT in parallel.

G. Attack 2: DF Prediction via Score Extrapolation

Natural extensions of our first attack to estimate DFs appeared to work correctly but were slow, as they had to measure and test if the DF was $0, 1, 2, 3, \ldots$ before finding the correct value. Our second attack estimates how many documents contain a given term on each shard of a search service (that is, we estimate $\text{df}(t, D_j)$ for each shard document set D_j). We call this *DF prediction* which is denoted as DFPRED.

At a high level, DF prediction works by collecting data on the behavior of the score function when the DF of a term is known, and then training a model that predicts DF from relevance scores alone. In our attacks we can speculatively guess the class of scoring functions based on knowledge of common implementations, but we still assume that constants and custom modifications to the function are hidden.

The algorithm DFPRED is described in Algorithm 4. It assumes it is given input several documents on the same shard of the service (either from several shard maps or from some other method). Then it performs a data collection step in the loop that estimates the score of a search when a term has DF equal to $1, \ldots, n_{\text{DFE}}$, where n_{DFE} is a parameter of the system (see Figure 3 for example data). After this step it uses a training algorithm to fit a curve f (from some class) that maps integers to reals. This f intuitively is a guess for the mapping from DFs to relevance scores induced by the system.

After computing f we can apply it in attacks. Given a term t of interest, an attack can write t to the document on the target shard and then search and record the score as s. Finally, we produce an estimated DF by computing

$$[f^{-1}(s)] - 1$$

where $[x]$ denotes the closest integer to x. We subtract 1 to account for the document added by the attack that contains t.

Algorithm 4: DFPRED

Input : Documents $d_1, \ldots, d_{n_{\text{DFE}}}$ on same shard
Output : Score-to-DF model f
Params: n_{DFE}, training algorithm TRAIN

$L_{\text{scrs}} \leftarrow \phi$;
for $i = 1 \ldots n_{\text{DFE}}$ **do**
  $r \leftarrow$ RNDTERM;
  for $j = 1 \ldots i$ **do** WRITE(d_j, r);
  SLEEP;
  $s_i \leftarrow \sum_{j=1}^{i} \text{score}(r, d_j)/i$;
  Append $\{\langle i, s_i \rangle\}$ to L_{scrs}
end
$f \leftarrow$ TRAIN(L_{scrs});
return f

Comparison to Brute-Force Term Extraction. Once we have computed the model f we can also use it for brute-force term extraction to get an attack with essentially the same complexity by using f to predict when terms have DF equal to zero. We opted for the first attack above because it does not require the training phase. Note that DF prediction actually recovers more, as it guesses the DF of a term rather than only detecting if the DF is non-zero. As mentioned above, using TERMEXTRACT to decide the DF of a term would be slow.

H. Attack 3: Rank-only Attacks

Our attacks above assumed that the search interface returns relevance scores. Some services however only return the list of ranked results without scores, and here we sketch how to adapt our techniques to this case.

We assume that the service supports multi-term search queries, and that the relevance scoring function assigns weights to terms that decrease with their DF. For now, we also assume there is no noise in the scores on a shard.

When there is no noise in scores, scores will often result in ties, and we start by reverse-engineering how the service breaks ties. In our experience this was done by sorting on the document name, creation time, or some other easily-noticed property of the documents.

Rank-only term extraction. Our rank-only term extraction attack is given in Algorithm 5. It takes as input two documents d_1 and d_2 on the same shard, and a target term set B. Without loss of generality we assume that d_1 is ranked higher than d_2 in the case of a tie.

The algorithm will compute the subset $B' \subseteq B$ of terms present on the shard with d_1 and d_2. The algorithm iterates over each $t \in B$. It writes t into the document d_1, and it writes fresh random terms r_i into the document d_2.

Algorithm 5: ROTERMEXTRACT

Input : Documents d_1, d_2, term set B
Output : Terms $B' \subseteq B$ present on the shard.

foreach $t \in B$ **do**
  $r \leftarrow$ RNDTERM
  WRITE(d_1, t)
  WRITE(d_2, r)
  SLEEP
  $R \leftarrow$ ROSEARCH$(\{t, r_i\})$
  if d_1 *is ranked below* d_2 *in* R **then**
    $B' \leftarrow B' \cup \{t\}$
  end
end

After waiting for the writes to propagate to the index, the attacker issues a two-term search query for $\{r_i, t\}$, which returns a ranked list R of two results. This list is either $d_1 > d_2$ or $d_2 > d_1$. If it is the latter, the algorithm infers that t is on the i-th shard and adds it to B'.

To see why this attack works, we consider the cases where $\text{df}(t, D)$ is zero or is positive before the attack starts (where D is the document set on the shard). If it is zero, then d_1 and d_2 will have the same score and hence d_1 will be ranked higher. If however $\text{df}(t, D)$ is positive, then d_2 will have a higher idf since the DF of r is exactly 1 and the DF of t is at least 2 after we write the files. Thus, d_2 has a higher score and be ranked first.

Optimizations. This attack can be generalized to test for several terms in each iteration of the main loop instead of 1 (thus reducing the number of sleep operations). This version requires several documents $d_1, \ldots, d_m$ on the same shard, and we assume that ties are broken in the order $d_1 > d_2 > \cdots > d_m$. The attack writes r into the last document d_m, and the terms of interest $t_1, \ldots, t_{m-1}$ in the first $m-1$ documents $d_1, \ldots, d_{m-1}$. Then it issues a search query for $\{t_1, \ldots, t_m, r\}$ and looks at the position of d_m in the list. If a document d_i appears below d_m, then the attack infers that t_i appears on the shard for the same reasons as before.

VI. CASE STUDIES OF MODERN SERVICES

In this section we discuss how an adversary might abuse the attacks we constructed in the previous section. Then we explore three services against which the score-based attacks are effective: GitHub, Orchestrate.io and Xen.do. We report on the performance of our attacks on the services, such as how long they took, how much they would cost to mount at scale, and how often they might fail. Finally, we examine rank-only attacks against GitHub and Orchestrate.io, who provide web interfaces for performing multi-term search without returning relevance scores.

A. Scenarios

To understand possible threats let us abstract the ability that is implied by our brute force term extraction and DF prediction attacks. Let the document sets stored on the shards of the service be $D_1, \ldots, D_{n_{\text{SHRDS}}}$. Term extraction gives us abstractly an oracle $\mathcal{O}_{\text{TE}}$ that takes input (t, i) and returns 0 if $\text{df}(t, D_i) = 0$ and otherwise returns 1. DF estimation provides a richer oracle $\mathcal{O}_{\text{DF}}$ that takes the same inputs (t, i) but returns (approximately) $\text{df}(t, D_i)$ itself.

In this view any abuse will have some fundamental limitations. Short terms are likely to appear by chance in documents, so the first oracle will likely return 1 most of the time. Also, since terms are extracted by a tokenizer, if some text happens to contain periods or hyphens (like a URL, SSN, phone number), then the text will be separated into small terms which may have high false positive rates. Neither of these oracles allows one to test for substrings of terms, so very high-entropy terms like cryptographic keys are, without some side information about them, intractable to guess. We nevertheless identify two types of attack scenarios that are possible within these limitations.

Medium-entropy terms. The first is brute-forcing *medium-entropy terms* that are rare enough to avoid false positives yet drawn from a brute-forcible space. As examples of sensitive medium-entropy data that may be stored within a search service, consider SSNs and phone numbers in the United States. In these cases, and assuming no hyphenation is used (which is desirable for search), an adversary could in principle use the first oracle $\mathcal{O}_{\text{TE}}$ to produce a list of all such numbers and SSNs stored in the shards of the service. This is already a severe violation of the confidentiality expected by users.

A second type of medium-entropy data are (relatively strong) passwords. Note that very weak passwords such as "123456" are likely to generate false positives. An attacker may test a dictionary of common passwords (or their hashes) using $\mathcal{O}_{\text{TE}}$ to determine which ones occur in the services' document set of terms. Passwords could be stored in search services when used as application backends, and there have also been well-publicized incidents of passwords being stored on GitHub repositories. In either case, an attacker could use access to $\mathcal{O}_{\text{TE}}$ to filter the password dictionary to a smaller set that it then uses for online password guessing attacks.

Medium-entropy data targets may also arise when an adversary has partial knowledge on an *a priori* piece of high-entropy data. For instance, someone may store documents that contain terms with adversary-known high-entropy prefixes followed by lower entropy suffixes. The prefixes will lower or remove false positives, allowing for brute-forcing of the rest via the oracle $\mathcal{O}_{\text{TE}}$.

A final type of medium-entropy data may occur when high-entropy data is tokenized into medium-entropy terms. Consider a hypothetical 24-character API key that consists of four 6-character chunks separated by hyphens. These may be tokenized into 6-character terms that could then be found via the oracle $\mathcal{O}_{\text{TE}}$, along with some false positives. This would vastly reduce the space of possible API keys for an attacker

who only needs to try keys formed by combinations taken from the set of 6-character terms found in the index.

Term trending. A second class of attacks uses the richer ability of the $\mathcal{O}_{\mathrm{DF}}$ oracle to estimate DFs rather than simply detect if they are positive. Unlike the previous settings, an attack may profitably query $\mathcal{O}_{\mathrm{DF}}$ for even low-entropy terms to learn about how commonly they are included in documents. For instance, on GitHub, one can use the side channel to learn about the popularity of certain libraries or packages. Or, if separate source code documents include unique identifiers associated to a particular victim (e.g., AWS account IDs), then the $\mathcal{O}_{\mathrm{DF}}$ oracle can be used to count the number of documents in that victim's private repositories. Since we are able to extract per-shard DF estimations, an adversary may be able to guess if it has found the shard of a particular user by looking for a shard that contains, with high DFs, terms associated with that user. One can then focus searches on that shard in order to reduce false positive rates in, e.g., a brute-force attack.

B. Performing Responsible Experiments

We would like to validate the feasibility of attack scenarios as discussed above. However, the nature of the side channel is such that we could, if careless, end up spying on actual user data in these services (e.g., if we simply started querying for passwords). We therefore took care to ensure that our experiments would not expose private information about their users or otherwise cause undue burdens on services.

Our experiments will only target simulated victims, i.e., accounts under our control with documents that we generate. This will give us ground truth. Except for estimating false positive rates, we apply the DF side channel only to long, random unstructured terms that are exponentially unlikely to appear in any bystander's document (given the number of such terms we use the side channel upon). Put another way, we explicitly avoid learning anything about other users' data from the side channel. We refer to users other than our simulated attack and victim users as bystanders (i.e., everyone is a bystander except for our accounts).

False positive rates caused by bystander data are important for understanding the efficacy of possible attacks, as we expect false positives to be a significant limitation to the attacks in practice. The rates however depend on bystanders' potentially private data. We therefore perform carefully limited false positive measurements in which we infer only whether or not we get the right answer from our side channel for random terms of given lengths. Even here we minimize any perceived risk to other users, only searching for random unstructured data with no semantic value. We only report summary statistics and never what random values may have resided in one or more bystander's documents, and we will not make these false-positive datasets public.

Attacks could involve making a large number of queries to the service. We rate-limit our queries appropriately and show how to extrapolate from our experiments to attackers with no qualms about submitting as many queries as possible per unit time.

C. GitHub

GitHub is one of the most popular source code hosting platforms, with 14 million users and 35 million repositories as of April 2016 according to Wikipedia. GitHub has two types of repositories: public repositories and private repositories. Users can register for a free plan and set up unlimited numbers of public repositories, but no documents or repositories can be marked as private. To enable use of private repositories, one can choose a 7 USD per month plan. In a private repository, documents can be accessed and searched by their owner or authorized users. Non-authorized users should not be able to learn anything about the repository's contents, such as the number and type of documents, the contents of those documents, etc.

GitHub search API and basic experiments. GitHub uses ES (hosted by Elastic.co) as its search engine for full-text search [8]. A user could use a web-based interface or RESTful APIs to search for a term of interest. A search request will return with all the documents containing this term in both the public repositories as well as in private repositories to which the requesting user has access. The RESTful search API returns relevance scores to facilitate application development, which our attacks will exploit, while the web-interface returns ranked results without scores (we discuss attacking this setting in the Appendix). Based on public documentation [8, 19], we know that GitHub load balances across shards at the granularity of an individual repository: at the time the repository is created it is assigned to a shard. All documents in that repository are indexed within the assigned shard.

We first performed some manual experimentation using our score-dipping attack to both confirm the DF side channel and reverse engineer some undocumented aspects of the GitHub search service. We found that public repositories and private repositories use the same indexes. This means that, looking ahead, a malicious user could use (free) public repositories and the search API to extract sensitive terms from a victim's private repositories. We also observed that the index update time, i.e., the time between inserting a document into a repository and it being added to an index, is less than 1 minute in most cases.

Search queries emanating from a particular user account are limited to 5,000 per hour. There is a public interface for search as well, which does not require an account, and only searches public documents (which suffices for our attacks should an attacker use public repositories). This is rate limited to 60 per IP per hour. The GitHub search API allows queries with size less than or equal to 128 bytes. In our experiments, we primarily used private repositories for our simulated attacker, and found that pausing at least

two seconds between two consecutive API requests avoids triggering rate limits. Therefore, in all our experiments, we pause for 2 seconds after search query and 60 seconds after creating/updating a document. Using longer pause times might be better for handling outliers (i.e., the index update time can be up to 2 minutes in rare cases), but would significantly increase the experiment running time.

Shard mapping. As mentioned GitHub hosts millions of repositories across many users, and therefore uses a large number of ES shards. We apply our shard mapping tool to determine how many, and to place an attacker document on each of the discovered shards.

We ran the shard mapping algorithm variant as described in §V-E, creating two new repositories each with one document over 513 rounds for a total of 1,026 repositories. The δ in CoShardTest was set to 0.05. We stopped after 50 consecutive rounds (100 repositories) failed to find a new shard. It took 104 hours and we discovered 191 shards. We might have missed some small number of shards. For example, assuming random assignment of repositories to shards, the probability of 100 consecutive failings if there were in fact 200 shards is 1%. Nevertheless, our shard map ends up sufficient for all experiments — all subsequent simulated victims ended up on one of the 191 shards that we discovered. We note an Elastic.co use-case description states that GitHub has 128 shards [19], suggesting this information is out of date.

After creating one repository on each shard at GitHub, we can generate many shard map sets $M_1, M_2, \ldots$ simply by creating one document on each repository.

We note that using shard mapping it would seem possible to track, over time, the number of shards used by GitHub. This could already be a hypothetical confidentiality issue for services that want to keep their infrastructure configuration secret.

Note that the consistency issues mentioned in §IV might produce false positives in CoShardTest; i.e., the difference in scores for two documents is greater than a threshold even though the documents are not on the same shard. To handle this issue, we double check after CoShardTest returns a positive result: we run CoShardTest again and accept the result if both rounds of tests give positive results. We adopt this false positive identification method in the CoShardTest on all examined services.

DF prediction. As mentioned above, the documents in the same repository are assigned to the same shard. Doing more manual tests, we confirmed this, and leverage it in the design of our experiments. We use one account *AcctA* as the account for a simulated attacker and another account *AcctV* for a simulated victim.

We tested the accuracy of DF prediction as follows. For a given value of n_{DFE}, we create n_{DFE} training documents in a repository under the attacker's account *AcctA* and run

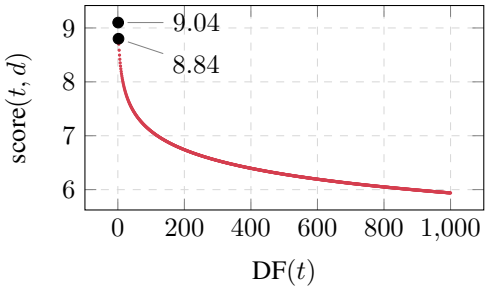

Figure 3: The changes of score(t, d) as df(t, D) increases. The scores when df$(t, D) = 1$ and df$(t, D) = 2$ are highlighted. Y-axis does not start from zero.

DFPred. During the training, we use OriginLab [37] to test the data against all the functions provided, and find without exception the best-fit function is in the form of $f(x) = a - b * \ln(x + c)$, where x is the variable representing the unknown DF and a, b, c are coefficients. This function is consistent with the standard Elasticsearch scoring function in [28].

Using *AcctV* we generate n_{DF} victim documents in a single repository, each document containing the single term $\{t^*\}$ which is chosen as a random 16-byte alphabetic string. We vary n_{DF} and test the accuracy of the attack. We run the CoShardTest attack to place a document $d = \{t^*\}$ from *AcctA* on the same shard with the documents of *AcctV*. We then measure and record the score score(t^*, d) by making a search query from *AcctA*. Then, we calculate $\text{df}_{\text{est}}(t^*) = f^{-1}(\text{score}(t^*, d)) - 1$ as an approximation of DF(t^*) and measure the *relative error rate* (in percentage) and *absolute error* in order to evaluate estimation accuracy. The relative error rate is calculated as $|\text{df}(t^*) - \text{df}_{\text{est}}(t^*)|/\text{df}(t^*) * 100$ and the absolute error is calculated as $|\text{df}(t^*) - \text{df}_{\text{est}}(t^*)|$.

We perform experiments for each $n_{\text{DFE}}, n_{\text{DF}}$ pair for $n_{\text{DFE}} \in \{1, 5, 10, \ldots, 250\}$ and $n_{\text{DF}} \in \{0, 1, \ldots, 999\}$. Figure 3 shows the changes of the relevance score of score(t^*, d) as DF(t^*) increases from 1 to 1,000.

As shown in Table 4, the average relative errors (across all n_{DF}) for any n_{DFE} are all less than 0.5%, and average absolute errors are less than 3. We find that when $n_{\text{DFE}} \geq 50$, the average relative errors and the average absolute errors under different n_{DFE} are similar, i.e., the estimations do not become more accurate as we use more data points during regression analysis. Figure 5 shows a histogram of the errors for the 1,000 experiments (for the 1,000 different n_{DF} values) and for $n_{\text{DFE}} = 50$. As can be seen, the performance of the DF prediction is very good: about 10% of the estimations are correct; less than 14 of the estimations have absolute errors of 5. We note that the attack performs differently on alphabetic and numeric terms, likely due to boosting in the score function.

	Relative error			Absolute error		
	Min	Avg	Max	Min	Avg	Max
Alphabetic	0.07%	0.38%	0.53%	0.52	1.93	2.83
Numerical	0.13%	0.43%	0.58%	0.59	2.15	3.03

Figure 4: An overview of the average relative and absolute errors for DF prediction for all n_{DFE} on GitHub. The first row targets estimation for a random 16-byte alphabetic string and the second row is for random 16-byte number.

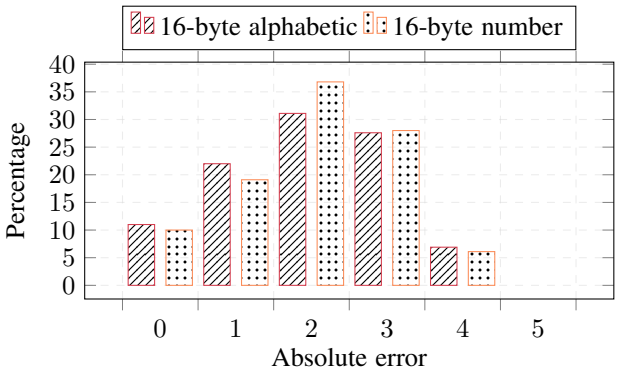

Figure 5: The distribution of the absolute errors when $n_{\text{DFE}} = 50$ (GitHub).

We repeat the experiments again on two further shards and get similarly small error rates: when $n_{\text{DFE}} = 50$, the average relative errors are 0.65% and 0.27%, and the average absolute errors are 3.9 and 1.2, respectively.

One important factor that can affect the estimation accuracy is the time we wait between updating a document and relevance score measurement. We find if waiting only 30 seconds, there is so much noise in the data that we cannot even do a reasonable curve fit to the scoring function. However, sometimes 60 seconds might still not be long enough for an index to reach a stable state: we indeed observed unusual score variations during data collection. While the DF estimation already works well despite this noise, we believe the performance could be improved further with more effort on data collection and processing.

Term extraction attacks. We start by confirming that term extraction works correctly in a controlled setting. We generate a set of 50 victim terms B and a set of 50 control terms B'. We create a victim document $d = B$, and then run our term extraction attack on all the terms in $B \cup B'$ to see if it can properly identify the victim terms. We repeated the experiment 50 times.

To save time, once TERMEXTRACT finds the target shard containing d (i.e., a shard containing any of the terms form $B \cup B'$) we ignore the other shards and only do term exaction on the target shard. The average time for finding the target shard is 1,149 seconds, while the minimum time is 698 seconds and the maximum time is 1,607 seconds. The median

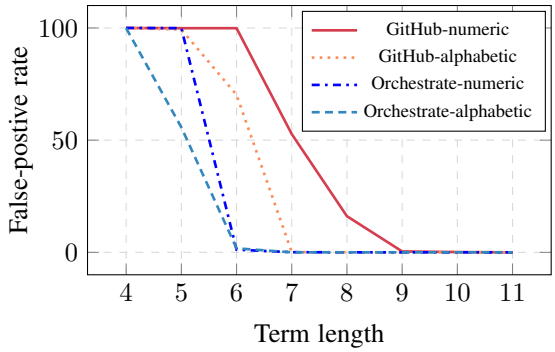

Figure 6: The average false-positive rates for different lengths of alphabetic-character-only term and numeric-character-only terms across three shards in GitHub and Orchestrate.io.

number of tries (i.e., number of shards examined before finding the target index) is 98.

The attack achieves a true-positive rate of 100% and a false-positive rate of 0%. Since we also chose long random terms, excluding any noise due to bystanders, we conclude that the attack solved the experiment perfectly.

False positives on GitHub. The brute-force term extraction attack will encounter false positives due to bystander data. To understand how often terms happen to be contained on GitHub, we estimate the false-positive rate associated with low-entropy terms. We also test two types of terms: alphabetic-character-only terms and numeric-character-only terms. For a given length ℓ, we generate 20,000 terms of length ℓ ($\ell \geq 5$) and of a given type (10^4 terms for numeric-character-only term when $\ell = 4$) to construct B, and randomly select 5% of these terms as B'. We set ℓ to each of $4, 5, ..., 16$. We repeat the test on three different shards, and report on the average false-positive rates across 3 rounds in Figure 6. We can see when $\ell = 4, 5$, the false-positive rates are 100% or near 100% in both services. The false-positive rates are relatively high even when $\ell = 8$, but drops to a very small value ($< 0.5\%$) when $\ell \geq 9$ and zero when $\ell \geq 11$. We can also clearly see that numeric-character-only terms involve more false positives than alphabetic-character-only terms.

Feasibility of brute-force attacks. According to GitHub, developers sometimes leave CCN information in source code [18], and users might also store their own personal information on GitHub [52]. We argue that it is sometimes feasible for an attacker with partial information to harvest this (and other) information via the DF side channel.

Recall that in GitHub one account can send 5,000 requests per hour. Our brute-force attack will write large files containing terms to test and then issue one API call per term. Since writing the file requires a wait time for propagation to the index, one would pipeline the writes while performing the search queries. Assuming this is implemented, in the limit our term extraction needs one API request per term guess (using

a modified version of TERMEXTRACT that uses one random term r_i to generate a score s_i that is then compared against the scores of many victim terms). Each guess checks if the term is on a particular shard. This gives a rough estimate of 120,000 guesses on a shard per day with one account. Creating n additional accounts increases the brute-forcing power by a factor n as the guessing algorithm can be run in parallel.

For a concrete example, if one knows the BIN (bank identification number) and last four digits of a CCN then there are about 10^6 possible CCNs. If an attack has focused on a particular shard the rest of the CCN could be brute-forced with one account in under a day. If the attacker is unsure of the shard, it could create one free account per shard and execute the attack in parallel (which, nicely, would be perfectly load-balanced on GitHub's backend).

D. Orchestrate.io

Orchestrate.io is a database-as-a-service platform for developing web and mobile applications. The information stored on Orchestrate.io is likely different than in GitHub since it is a generic key-value database and is being used to store all types of data. It seems likely that application back-ends store sensitive customer information in Orchestrate.io.

According to Orchestrate.io's official blog, it uses ES as its search engine [36], and has made efforts to secure its search API. However, we found its search API also expose the relevance scores of returned documents. Further tests suggested that the DF side channel also exists in Orchestrate.io.

The Orchestrate.io API does not restrict the number of operators in the query but enforces a maximum query size of 6 KB. The service does not have a specific rate-limiting policy but will throttle a user if her API requests affect their servers' performance. The index update time on Orchestrate.io is faster than GitHub. To avoid burdening on the target server, we decide to pause 30 seconds after creating/updating a document and 2 seconds after each search query.

Attack results. In Orchestrate.io, we use the same experiments as in GitHub to test MAPSHARDS and TERMEXTRACT. MAPSHARDS collects 50 shards in 12 hours, using 128 rounds with 256 documents being created. The δ in COSHARDTEST was set to 0.08. In TERMEXTRACT, the average time for locating the target shard is 324 seconds and the median number of tries is 15. The term extraction attack also achieves a true-positive rate of 100% and a false-positive rate of 0%.

We also conduct the same false-positive tests in Orchestrate.io. The average false-positive rates across three rounds are shown in Figure 6. As the term length increases, the false-positive rates drop to zero more quickly than on GitHub; when $7 \leq l \leq 9$, we only find very few false positives (1 to 3) for a given length.

To perform DFPRED, we need to put multiple documents on the same shard. Unfortunately, unlike GitHub, no features in Orchestrate.io directly facilitate creating documents on the same shard. One solution is to create many documents and use COSHARDTEST to discover the documents that are on the same shard with a target document. However, this is very time-consuming. To speed this process up, we use the aforementioned ad-hoc score-based co-shard test in §V. More specifically, we create 30,000 documents that have the same content in *AcctV*, which is a unique 16-byte term, and measure the relevance scores of these documents. We group the documents by their relevance scores, and keep 500 documents from the largest group. To eliminate false positives, we use COSHARDTEST to confirm these documents are indeed on the same index. We repeat these procedures in *AcctA* and keep 100 documents.

We perform experiments for each $n_{\text{DFE}}, n_{\text{DF}}$ pair for $n_{\text{DFE}} \in \{1, 5, 10, \ldots, 100\}$ and $n_{\text{DF}} \in \{0, 1, \ldots, 499\}$. The scoring function in Orchestrate.io is still in the form of $a - b * ln(x + c)$. The average relative and absolute error rates decrease as n_{DFE} increases. When $n_{\text{DFE}} = 100$, the average relative errors are about 2.2% and the average absolute errors are less than 6.0 for terms being tested. As $\text{df}(t^*)$ increases, the estimations become less accurate. The maximum absolute errors are 15. However, when $\text{df}(t^*) \leq 250$, the attack still performs well, with the maximum absolute error less than or equal to 2.

Feasibility of brute-force attacks. In Orchestrate.io, a free-plan user can only send 50,000 requests every month. So to search 10^9 terms, the attacker needs 20,000 accounts. Though this sounds costly, the process can be automated due to the fact that the account registration is very simple — the attacker just needs to fill in an email address and a password — and no captchas are being used.

Another choice is to use Orchestrate.io's professional plan, which is $499 per month, that allows one to send 5 M requests per month and pay $0.01 for 10 K additional requests. Sending 10^9 requests costs an attacker $1,500, but the gain of the attacks could be more than the cost. Of course, smaller spaces can be brute forced much more cheaply and quickly.

E. Xen.do

Xen.do is a hosted search service which aggregates data from a user's accounts on multiple third-party services, builds full-text indexes over the data, and provides interfaces to search the aggregated data. Xen.do supports more than 35 services, including, but not limited to, Google Apps (Gmail, Contacts, Drives, etc.), cloud storage services (Dropbox, OneDrive, etc.), customer relationship management (CRM) systems (Salesforce, ZohoCRM, etc.), and other services (Evernote, Office 365, etc.).

Sensitive information harvesting is particularly threatening on Xen.do since the data are collected from users' personal accounts. Xen.do makes an best effort to guarantee data

security and privacy, and has received high ratings in various security tests such as Skyhigh Networks CloudTrust [56]. Unfortunately, we also find the DF side channel in Xen.do. We found the all the supported services in Xen.do share the same multi-tenant indexes. Therefore, a malicious user can extract the sensitive terms in other users' documents from different sources at the same time.

For Xen.do, its API access is not public and the API key can be only obtained on request. We only obtain a 30-day trial to the beta-test version of the API, which currently only provides basic operations such as full-text search and authentication. One operation — connecting Xen.do to a service — in the attacks must be done manually via the web interface.

The indexes updates on Xen.do are very slow, often taking about 20 to 30 minutes. In our attacks, after creating or updating a document, we query every 10 minutes to see if the document has been indexed. We still pause 2 seconds after each API request.

Attacks results. Using CoShardTest, we confirm that all the services supported by Xen.do are using the same set of shards. We create a document d_1 on a service $serv1$ (e.g., Gmail), and connect *AcctA* to $serv1$; then, we create a document d_2 on another service $serv2$ (e.g., Dropbox), and connect *AcctV* to $serv2$. We then use CoShardTest to test if d_1 and d_2 are on the same shard. If not, we disconnect *AcctA* from $serv1$ and reconnect it again, which forces Xen.do to assign d_1 to a new shard. We did two tests: (1) randomly chose 5 different pairs of $serv1$ and $serv2$, and (2) fix $serv1$ as Dropbox, and 17 different $serv2$. In both tests, CoShardTest usually succeeded in between 4 and 10 tries. The success of CoShardTest indicates that Xen.do uses the same set of shards for all services. The δ in CoShardTest was set to 0.08.

In MapShards, we stop the attack if we can't find more shards in 10 rounds. After 20 rounds, we found 4 shards.

Due to the restrictions of the Xen.do API and slow index propagation, we only collected a small amount of data. We use the ad-hoc score-based method again to put 50 documents on the same shard. Since the index updates are slow, it took us longer to run DFPred (dominated by waiting). We had the best results fitting the scoring function to a curve of the form $f(x) = a - b * ln(x + c)$. We use first the 15 data points to approximate the scoring function and the other data points for evaluation. The absolute errors of 40%, 49%, and 14% of the estimations are 0, 1, and 2, respectively. This preliminary assessment suggests that our attacks will work on Xen.do.

F. Rank-only Attacks on GitHub and Orchestrate.io.

We briefly checked if our rank-only attack works correctly against GitHub and Orchestrate.io, who provide web interfaces for performing multi-term search without returning relevance scores. On GitHub we started with a control experiment with an empty victim document and two attack documents on the same shard (recall that creating several co-resident documents is easy because GitHub shards based at the repository level). Using the web interface for GitHub search (which ranks but does not report scores), we observed that our attack returned a true negative (i.e. the order of the two attacker documents did not change). Next we added the term t (in this case a long random string) to the victim document and re-ran the attack, which swapped the order of the attack documents in the web interface, confirming that the attack works.

Interestingly our attack failed on Orchestrate.io. This appears to be due to their using a non-standard scoring function for multi-term queries. We found that for multi-term queries, Orchestrate.io computed relevance scores that weight terms based on their order in the query. So, for instance, "t_1 t_2" will give different term weights from "t_2 t_1" while TF-IDF and common variants will treat these terms equivalently.

G. Conclusions

The results demonstrate that our score-based attacks can work on the three targets and can be used to extract sensitive data from other tenants' documents. Without relevance scores, one can still exploit the DF-side channel using rank-only attacks. All the services we tested claim protecting data security and data privacy as a priority. Indeed, they make efforts to secure their physical infrastructures, systems, and APIs. However, the DF side channels, hidden in their underlying search engines for years, make the services vulnerable to sensitive data leakage via side-channel attacks.

VII. Countermeasures

Perhaps the most obvious idea for a countermeasure is to simply not return relevance scores in response to searches, instead just providing an ordered list of documents. This might be a hindrance to applications that make use of the API's relevance scoring. But more importantly, while removing relevance scores would prevent our score-based attacks, as shown in §VI, it does not prevent exploitation of the DF side channel via rank-only attacks.

Previously proposed countermeasures. One can remove the side channel by isolating each users' documents within independent indexes. Received wisdom suggests this approach is unsuitable for large-scale systems with many users due to poor performance [12]. Some Elasticsearch deployments have successfully used this architecture via careful tuning and optimization, but it may be too expensive for, e.g., Github to use [25]. Search functionality degradation is also a concern here, since users with small document sets may not provide enough data on their own to have good DF estimates.

Another approach is to retain a multi-tenant index, but compute relevance scores in a way that matches what would have been computed in the independent index case. Büttcher and Clarke were the first to suggest this

countermeasure [7] and called their particular realization of it "query integration". It works by inserting a security manager between the components of the system responsible for query processing and index management. When a user issues a query, the security manager recomputes a user-specific view of the index (and relevance scores) that is consistent with the user's access rights. Subsequent work provided different realizations of this approach [31, 39, 46], focusing on performance improvements by partially pre-computing views.

These approaches were suggested in the context of local file system search. In the multi-tenant cloud services we have primarily focused on, maintaining access control information at every shard will incur a large storage overhead.

Another countermeasure that has been proposed in the literature takes a statistical approach, attempting to add noise or otherwise change the IDF distribution so that an individual user's private information is hidden. Zerr et al. [58] give a countermeasure using a "relevance score transformation function" meeting an ad-hoc statistical notion of confidentiality. It is unclear what guarantee this actually provides.

A. New Countermeasures

All the approaches discussed above seem to have inherent limitations which will impede their usability in large-scale multi-tenant search indexes. We observe that they all preserve the *exact* functionality of TF-IDF scoring (or a slightly noisy variant) over a user's view of the system. This may be unnecessary: approximations that result in similar, but slightly different, scores are likely acceptable in practice.

Below we outline two approaches that eliminate DF side channels more efficiently. We also implement and evaluate one approach. We plan to open-source the relevant code. In both approaches, the searches are no longer scored strictly according to TF-IDF. Instead, the relevance score of a document d is computed as a function *only* of the public documents and of d. In particular, it is no longer a function of other private documents, whether or not d is public or private.

Public-corpus DFs. The first approach is called *public-corpus DFs*. The idea is to train a DF model using public data. In GitHub, for example, this would mean computing a DF model on a subset of public repositories. The model itself would be stored as an auxiliary index in Elasticsearch, enabling nodes to efficiently fetch the current public DF value for a term they have not seen. A default DF (of one) could be used for terms which do not appear in the public data. In settings like Xen.do and Orchestrate where there is no notion of "public" and "private" information, this approach will not work with data on the service. Instead, one could train on suitable public file corpuses, should they exist.

Blind DFs. We call the second approach *blind DFs*. Recall that the search system we consider stores an inverted index that consists of per-term postings lists. At the head of each list, the DF is stored to speed up searching. Typically the DF value is equal to the length of the list.

To implement blind DFs, one augments each posting entry to contain a binary attribute indicating if the document is public (i.e. world readable) or not. We then modify the mechanisms for adding and deleting to maintain a count that we call the *blind DF*, which is now the number of *public* postings in the postings list. This can be achieved, say, by only incrementing or decrementing a posting lists' DF when adding or deleting a public document. Of course one may also store the (true) DF for purposes other than relevance scoring. This metadata must be stored for each document so document deletions can properly decrement the DF.

To process a (public or private) search with blind DFs, one modifies the system to use the blind DFs in place of true DFs, but otherwise leaves it unchanged. In particular, one could compute relevance scores exactly as before, but with a blind DF. To enforce access control one could use the post-processing filtering mechanism as is currently deployed. Since the relevance scores are not a function of private documents, the DFs will contain only public information.

Comparison of the two approaches. Both approaches increase the amount of storage space needed for the index. For blind DFs, the amount of extra space required is on the order of the number of documents in the index, since the public/private attributes for each document must be stored. For public-corpus DFs, the amount of extra space needed is only on the order of the number of unique terms in the index. The amount of space needed for public-corpus DFs does not change as more documents are added to the index, whereas the space overhead of blind DFs does increase over time. Unlike public-corpus DFs, blind DFs can be implemented without any preprocessing. Both approaches will also potentially diminish the utility of DFs because private documents will no longer inform relevance scoring, even when a user is searching her own private documents. Edge cases, such as making a public document private, may be difficult to handle. The main benefit of both approaches is that they are relatively simple to implement. The relevance scoring and other portions of the system would be largely unchanged, including the access control filtering.

B. Evaluation of Public-corpus DFs

Of the two approaches described above, we believe public-corpus DFs will likely be better for large-scale search systems like Github's due to its low space complexity. Here we report on initial experiments to assess the potential practicality of the countermeasure. All experiments were performed on an Ubuntu 16.04 desktop, using Lucene 6.3.0 and Java 8. The machine was equipped with a 512 GB NVMe SSD and 16 GB of DRAM. Microbenchmarks revealed a small latency increase of about 1% due to the countermeasure. We therefore focus our evaluation of public-corpus DFs on two axes: space overhead and search quality.

Corpus	#Docs	#Terms	Size (GB)	TD size (MB)
Reuters	0.8	1.0	0.6	8.7
Wikipedia	3.6	14.5	33.0	200.0
Enron	0.54	0.6	3.0	5.7

Figure 7: The "#Docs" and "#Terms" columns are the total number (in millions) of documents and terms in the corpus respectively. "Size" and "TD size" are the size and the size of the terms dictionary of the corpus respectively. Statistics for the Reuters dataset refer to the pre-processed LYRL2004 version [26].

Space overhead. The space overhead of public-corpus DFs comes from storing the auxiliary index of DFs for each term. It is straightforward to evaluate this by indexing a document corpus using Elasticsearch and measuring the size of the term dictionary in the resulting index. Asymptotically, the term dictionary's size is on the order of the number of unique terms in the index, but we will still measure its size empirically to account for the effect of Elasticsearch's term dictionary compression.

We tested with three datasets: the Reuters RCV1 corpus [26], a dump of the English Wikipedia from April 2015 [15], and the Enron email dataset [24]. Each was parsed, tokenized, stemmed, filtered to remove stop words, and indexed using Lucene. Finally, the statistics in Figure 7 were collected by inspecting the resulting index. The term dictionary size is measured as the sum of the on-disk file sizes of the `.tim` and `.tip` files of the Lucene index. These two files store the compressed term dictionary and an index into it, respectively. Note that the auxiliary data structure would also store the DFs of each term. The size of the DFs in bytes would be about four times the number of unique terms for each corpus.

These results are quite promising: even for the entire English Wikipedia, the public-corpus DFs would only require about 250 MB of storage (the fifth column of Figure 7 plus four times the third column). This is small enough that it could be held entirely in memory on each shard, minimizing the number of slow disk I/O operations.

Search quality. Since the storage overhead of public-corpus DFs is minimal, we can turn our attention to evaluating its impact on search quality. We will use a standard methodology from information retrieval: queries with human-labeled relevance judgments. This measures search quality for a set of synthetic queries on a standard corpus by using human judges to label documents as relevant or non-relevant for each query, then evaluating a search engine's performance in retrieving relevant documents. We built our experiment by modifying Ian Soboroff's `trec-demo` project [47].

Our corpus for the experiment was a pre-built Lucene index consisting of volumes 4 and 5 from NIST's Text Research Collection. These two volumes contain about 530,000 total documents and 4.1 M unique keywords. The index was not stemmed, but common stopwords were removed. We used the relevance judgments from the "ad hoc" track of the sixth, seventh, and eighth sessions of NIST's Text Retrieval Conference (TREC). There were 150 total labeled queries.

		Real DFs	Enron DFs
TF-IDF	MAP	0.17	0.17
	P@5	0.43	0.43
	P@20	0.31	0.31
	P@100	0.17	0.17
BM25	MAP	0.17	0.17
	P@5	0.44	0.43
	P@20	0.31	0.31
	P@100	0.17	0.17

Figure 8: Results of search quality experiment. MAP is "mean average precision". P@n is the precision only considering the top n documents returned for the search, averaged across all queries. Higher scores are better.

Our experiment consisted of a few concrete steps. We performed queries on two versions of the NIST index: an unmodified 'insecure" one which used the actual DFs of the NIST corpus and one which used public DFs from the Enron corpus. For both, we recorded the top 1,000 most relevant documents returned for each query. Finally, with the relevance judgments as ground truth, we computed quality metrics to measure the degradation in quality (if any) caused by our countermeasure.

We used two metrics from information retrieval: "precision" and "mean average precision". Intuitively, precision is the fraction of returned documents that were relevant to the query. If the number of relevant documents returned for a query is r and the total number of returned documents is s, the precision is defined as r/s. The metric P@n is defined as the precision when only considering the top n documents returned for the search. The numbers given in Figure 8 are averaged over all 150 queries.

Mean average precision is a related metric, defined simply as the mean over all queries of the per-query "average precision". The average precision is, importantly, *not* simply the average of an arbitrary set of precision scores. Average precision is defined in our case by measuring the precision at every cutoff point (i.e. n in P@n above) from 1 to 1,000, then summing and dividing by the number of relevant documents.

The results of the experiment are in Figure 8. The results using relevance scores computed with the default implementation is in the column of Figure 8 labelled "Real DFs". The results with the public-corpus DFs countermeasure enabled (using the Enron email corpus) are in the column labeled "Enron DFs".

The results show that using the Enron DFs in place of the real DFs for the corpus has negligible effect on the precision of the searches. Most values are identical when rounded to

the hundredths place. This is true both for TF-IDF and the more modern BM25 scoring function.

Limitations and future work. We believe our evaluation presents good evidence of the practicality of the public-corpus DF countermeasure. Nevertheless, it is limited in a few important ways. First, we only evaluate unstructured English text corpora and queries, and it is unclear if the results generalize to code repositories like Github. Obtaining labeled relevance judgments and corpora for code search is an interesting direction for future work. Since the quality of the search results above is, in an absolute sense, quite low to begin with, an experiment on a better-tuned search system which uses modern IR techniques to increase search quality may yield different results. The final limitation is the small sample size of our experiment. Due to the difficulty of finding appropriate data sets and relevance judgments, we only evaluated search quality for one dataset, and leave a more thorough evaluation as an open problem.

VIII. Vendor Response

We disclosed via email to the three services investigated. Xen.do immediately removed relevance scores from API responses as a preliminary mitigatition and is not vulnerable to the STRESS attack at this time. GitHub forwarded the issue to Elastic.co, their search service provider. Elastic.co suggested several countermeasures in their response. To mitigate our attacks, Elastic.co suggested small deployments could use an index-per-tenant, but they admitted that this could be cost prohibitive for large deployments. In some cases, services can disable scoring and ranking if the resulting functionality loss is acceptable. Another approach is to put sensitive terms in the fields that are not used for ranking, an approach suggested by Alex Brasetvik of Elastic.co. This will prevent the side channel being exploited for those terms, though some services might find reliably identifying sensitive information within tenants' data challenging. We believe the public-corpus DFs countermeasure presented in section VII is the best approach due to its scalability and deployability. Orchestrate.io's parent company, CenturyLink, announced that the service vulnerable to our attack will be shut down on March 17th, 2017.

IX. Conclusion

We presented STRESS attacks. These demonstrate that the industry-standard method for multi-tenant search leads to an exploitable side channel, even in complex distributed systems. We developed efficient attacks on two services, GitHub and Orchestrate, and verified exploitability of another service Xen.do. Using our side channel we estimated the time and cost required to extract information like phone and credit card numbers from private files stored in these services.

Our case studies only hint at the scope of affected systems. As mentioned, we also confirmed that following best practice guides for building multi-tenant search on top of AWS ElasticSearch, AWS CloudSearch, Searchly, bonsai, and Swifttype would lead to a DF side channel. Some of these search-as-a-service systems are in turn used by other cloud services, such as Heroku, which may therefore inherit any side channel. We have not yet performed in-depth experimentation with applications using these services, so it may be that noise or other subtleties prevent, e.g., brute-force term recovery attacks or accurate DF estimation. That said, services would do well to revisit their use of shared search indexes in order to prevent STRESS attacks.

Along another dimension, we have focused on attacks whose search queries include a single term. But many search services allow more sophisticated queries such as phrase or wildcard queries. We began thinking about how to exploit these, but have not yet seen how they could provide attacks better than our single-term ones. Future work may do better.

Based on our experiments we recommend that the implementations move away from the simple filter-based approach to multi-tenancy. We suggested possible countermeasures, such as using document frequencies taken only from public documents, and our preliminary evaluation suggests this approach will be very practical for deployments.

Acknowledgments

We would like to thank all the anonymous reviewers for their comments and suggestions. We would also like to thank the employees at Elastico, GitHub, and Xen.do for their helpful discussions during our disclosure process. This work was supported in part by NSF grants CNS-1558500, CNS-1330308, CNS-1453132, the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and a generous gift from Microsoft.

Ristenpart and Grubbs have large financial stakes in Skyhigh Networks.

References

[1] Index Aliases. https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-aliases.html#filtered, 2016.

[2] Amazon. Amazon Cloudsearch. https://aws.amazon.com/cloudsearch.

[3] Amazon. Amazon Elasticsearch service. https://aws.amazon.com/elasticsearch-service.

[4] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[5] J. K. Blitzstein and J. Hwang. *Introduction to Probability*. Chapman and Hall/CRC, 2014.

[6] Bonsai – Hosted Elasticsearch. https://bonsai.io, 2016.

[7] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, 2005.

[8] A. Cholakian. Elasticsearch at GitHub. http://exploringelasticsearch.com/github_interview.html, 2014.

[9] Couchbase – NoSQL database. http://www.couchbase.com.

[10] Cratedb. https://crate.io.

[11] elastic.co. Updating a whole document. https://www.elastic.co/guide/en/elasticsearch/guide/current/update-doc.html, 2016.

[12] Elasticsearch. Discovering the need for an indexing strategy in multi-tenant applications. https://www.elastic.co/blog/found-multi-tenancy, 2015.

[13] Elasticsearch. Term Filter query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-filtered-query.html, 2016.

[14] Elasticsearch. https://www.elastic.co/products/elasticsearch, 2016.

[15] W. foundation. Wikipedia Dump download. https://dumps.wikimedia.org/enwiki/.

[16] A. Futoransky, D. Saura, and A. Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *WOOT*, 2007.

[17] N. Gelernter and A. Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[18] GitHub. Sensitive data exposure. https://bounty.github.com/classifications/sensitive-data-exposure.html, 2016.

[19] GitHub on Elastic.co case study. https://www.elastic.co/use-cases/github, 2014.

[20] Google. Google app engine. https://cloud.google.com/appengine.

[21] Add-ons - Heroku Elements. https://elements.heroku.com/addons#search, 2016.

[22] Hibernate commmuity documentation, chapter 10.9: Multi-tenancy. https://docs.jboss.org/hibernate/search/5.3/reference/en-US/html/ch10.html#section-multi-tenancy.

[23] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[24] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

[25] K. Kluge. Personal communication.

[26] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research*, 5(Apr):361–397, 2004.

[27] Lucene. https://lucene.apache.org/, 2016.

[28] Lucene Practical Scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html, 2016.

[29] Lucene's scoring function. http://lucene.apache.org/core/3_5_0/api/core/org/apache/lucene/search/Similarity.html.

[30] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[31] E. C. Micheli, G. Margaritis, and S. V. Anastasiadis. Efficient multi-user indexing for secure keyword search. In *EDBT/ICDT Workshops*, pages 390–395, 2014.

[32] Microsoft. Multi-tenant data architecture. https://msdn.microsoft.com/en-us/library/aa479086.aspx, 2006.

[33] How Mingle built its Elasticsearch cluster on AWS. https://www.thoughtworks.com/mingle/news/scaling/2015/01/06/How-Mingle-Built-ElasticSearch-Cluster.html, 2015.

[34] Elasticsearch: the definitive guide. https://www.elastic.co/guide/en/elasticsearch/guide/current/shared-index.html, 2016.

[35] MySQL full text search. http://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html, 2011.

[36] Orchestrate. How we improved elasticsearch indexing. https://www.ctl.io/developers/blog/post/improved-elasticsearch-indexing, 2014.

[37] OriginLab. http://originlab.com/, 2016.

[38] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers" Track at the RSA Conference*, pages 1–20. Springer, 2006.

[39] A. Parker-Wood, C. Strong, E. L. Miller, and D. D. Long. Security aware partitioning for efficient file system search. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2010.

[40] C. Percival. Cache missing for fun and profit, 2005.

[41] PostgreSQL. https://www.postgresql.org.

[42] Lucene's practical scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html.

[43] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[44] Searchify. https://www.searchify.com.

[45] Searchly – Elasticsearch as a service. https://http://www.searchly.com, 2016.

[46] A. Singh, M. Srivatsa, and L. Liu. Efficient and secure search of enterprise file systems. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 18–25. IEEE, 2007.

[47] I. Soboroff. Information retrieval evaluation demo. https://github.com/isoboroff/trec-demo.

[48] Apache Solr. http://lucene.apache.org/solr/, 2016.

[49] Swiftype. Customer case studies. https://swiftype.com/customers, 2016.

[50] Swiftype - site search and enterprise search. https://swiftype.com, 2016.

[51] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security*, pages 913–928, 2015.

[52] Vulnerability.ch. Creative commons: Donors data leak. https://vulnerability.ch/tag/github/, 2014.

[53] Wikipedia. Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25.

[54] Wikipedia. Term frequency-inverse document frequency. https://en.wikipedia.org/wiki/Tf-idf, 2016.

[55] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[56] Xendo. Xendo security blog. https://help.xen.do/hc/en-us/sections/200689704-Security, 2016.

[57] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[58] S. Zerr, D. Olmedilla, W. Nejdl, and W. Siberski. Zerber+r: Top-k retrieval from a confidential index. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 439–449. ACM, 2009.

[59] B. Zhang. A new, experimental approach to implement multi-tenancy with Lucene 4. https://community.jivesoftware.com/community/developer/blog/2013/06/24/a-new-experimental-approach-to-implement-multi-tenancy-with-lucene-4.

[60] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.