Blindfold: A System to "See No Evil" in Content Discovery

Ryan S. Peterson Bernard Wong Emin Gün Sirer
{ryanp,bwong,egs}@cs.cornell.edu
Department of Computer Science, Cornell University
United Networks, L.L.C.

Existing content aggregators provide fast and efficient access to large volumes of shared data and serve as critical centralized components of many peer-to-peer systems, including content discovery for BitTorrent. These aggregators' operators are tasked to spend significant human resources to manually vet uploaded data to ensure compliance with copyright laws. This task does not scale with today's increasing demand for such services. In this paper, we introduce Blindfold, a scheme to ensure that the operators of content aggregators are completely blind to the content that they are storing and serving, thereby eliminating the possibility to censor content at the servers. It works by partitioning the search and upload operations into a series of dependent key-value operations across servers under different administrative domains, with the connection between servers obfuscated using captchas. We have implemented a prototype of Blindfold to show that it is a simple, feasible, and efficient system for serving content that is opaque to the storage servers.

1 Introduction

BitTorrent is one of the most popular peer-to-peer protocols, yet it still relies on centralized components. These centralized components simplify difficult problems related to trust and management, resulting in a system that is easy to deploy and understand. Centralized components for content discovery, such as The Pirate Bay [4] and Mininova [2], collect and provide a searchable index of available content through a web frontend. However, large centralized systems that rely on user contributions face the daunting task of vetting the voluminous content, a process that demands extensive human resources, or risk subjecting themselves to copyright infringement litigation.

This paper presents Blindfold, a novel system that enables users to upload to and search a public key-value storage server without revealing the true keys or values to the server or third parties. The goal of Blindfold is to empower key-value storage operators to be oblivious to how their services are used, allowing them to operate under the same model as public utility providers. Blindfold ensures that storage operators are blind to the content that they handle, keeping keys and values encrypted and opaque to the servers.

A system that provides honest clients unrestricted access to a corpus without revealing any information about that corpus to the storage server or attackers is infeasible. Instead, Blindfold provides non-authenticated clients full access to the data through explicit keyword searches; at the same time, it obstructs the ability to efficiently enumerate the stored content.

The key insight behind Blindfold is to partition the search and upload operations into a series of dependent key-value operations that are performed across multiple storage servers under different administrative domains. The servers are unaware of the partitioning and chaining of operations and of each other; a simple in-browser client controls the high-level search and upload protocols and serves as a communication bridge between the servers. Cryptographic functions hide the true content stored on each server, and captchas obfuscate the connection between servers to protect the system from automated, dictionary-based attacks.

Blindfold is fast and efficient, requiring only simple and inexpensive cryptographic operations and a constant number of server queries per search or upload operation. It uses the standard key-value storage interface, allowing the use of public key-value storage services and enabling it to be immediately and widely deployable. We have implemented a prototype of Blindfold and have found it easy to use and unobtrusive to the user.

2 Related Work

There has been much work in private database systems that define cryptographic protocols for searching over encrypted data. Existing protocols fall into two main categories, depending on who owns the data in question. The larger body of work aims to encrypt a database of sensitive data so that an untrusted server can store the database and perform searches from authorized queriers without read or write access to the cleartext. Related systems provide additional properties, such as protecting the search keywords and search results from the server [9], reordering the entries in the encrypted database to prevent statistical attacks based on data accesses [11], and designing logarithmic-time (rather than linear-time) algorithms for searching over encrypted data [5]. Our work also aims to

obscure content from the server, but also enables public access to content matching keyword searches.

Another body of work examines encrypted database systems where the server owns the data and protects it from unauthorized queriers. The Secure Anonymous Database Search system [8] introduces two intermediate servers that together provide client anonymity while ensuring that all queries originate from a set of clients that were authorized out-of-band. In contrast, Blindfold provides a public keyword search interface rather than one based on access control lists. Much of the existing work in encrypted database protocols strives to make searches over encrypted text more efficient. In contrast, our system stores key-value mappings, where inserted values have associated keywords chosen by their content originators, which enable constant-time searches for content.

The most similar work to Blindfold is Peekaboo [12], a key-value store that splits keys and values across multiple servers to preserve the privacy of clients. Unlike Blindfold, Peekaboo assumes that servers do not collude. Moreover, Peekaboo relies on a distributed protocol among servers rather than operating across servers under different administrative domains that can be unaware of each other's existence. Much of Peekaboo's implementation is centered around enforcing access control on content without breaking its privacy properties.

Decentralized storage systems allow searches over data that is potentially spread across many machines. Freenet [7] implements a peer-to-peer approach to data storage where users add named files that can be retrieved by other users. Freenet's main goal is to prevent censorship by anonymizing queries with sequences of pseudo-random hops from query originators to content location, and to prevent tampering through signatures. These goals are orthogonal to our own, and one could implement Blindfold on top of Freenet to achieve the properties of both systems.

3 Approach

The Blindfold architecture comprises three components: two or more servers, jointly called the aggregator, which store mappings from search keywords to content; a standalone service, which generates image captchas [10] that require human interaction to solve; and clients, which orchestrate uploading new content and searching for existing content. Each content object (e.g., a BitTorrent file in the case of an aggregator) has an associated set of search keywords chosen by the content's originator. A search query, consisting of one or more keywords, yields all content objects that are associated with all the keywords. For simplicity, we assume that the aggregator is made up of two logically centralized servers, providing a service analogous to that of existing BitTorrent aggregators.

3.1 The Blindfold Protocol

The basic Blindfold protocol splits search keywords and content objects across two servers that operate under different administrative domains. The two servers are the *index server* S_I, which stores mappings from keywords to captcha images, and the *content server* S_C, which stores mappings from captcha solutions to content objects. *Clients* are users in the system that issue search queries for content and upload new content. Lastly, Blindfold relies on a *captcha generator* G, a standalone service that issues signed captcha images. Blindfold handles uploading content and processing search queries without requiring S_I, S_C, or G to communicate with each other; in fact, the three servers can be oblivious of each other's existence.

Blindfold requires clients to perform explicit search queries to reveal content at the aggregator. The aggregator stores only hashes of search keywords and stores all content encrypted with keys known only to its originators. These encryption keys are generated from the content's associated keywords. The intuition behind Blindfold is that search keywords are necessary and sufficient for generating both the hashed keywords stored on the aggregator as well as keys that decrypt the content stored under those keywords, obviating the need for trusted third parties.

We begin by specifying our notation. Let h be a well known one-way hash function. Exponentiating h indicates repeated composition: $h^3(x) = h(h(h(x)))$. $\{X\}_K$ denotes the encryption of X under key K, and K_A^{pub} and K_A^{pri} denote agent A's public and private keys of an asymmetric key pair, respectively. All keys are symmetric unless designated public or private. We use $\langle x, y \rangle$ to denote the concatenation of x and y and $x \oplus y$ to denote the bitwise exclusive OR of x and y. We use $\mathsf{hmac}_k(m)$ to denote the HMAC of message m under symmetric key k [6]. Lastly, $\mathsf{captcha}(p)$ represents the solution to captcha p, where $\mathsf{captcha}$ is a one-way function that can be computed easily with human interaction, but is difficult to compute automatically.

To upload new content to the aggregator, a client chooses search keywords to associate with the new content and requests a new captcha for each keyword from the captcha generator. It sends the unsolved image captchas and hashed keywords to the index server and sends solved captchas and encrypted content to the content server. To prevent the index server from tampering with captchas and their mappings from keywords, the client binds keywords to captchas using HMACs with keywords as the secret keys, which it also sends to the index server. The client encrypts a separate copy of the content for each of its keywords. The encryption key for each copy is deterministically computed from a keyword

Client A uploads content C with keywords $w_1, \ldots, w_m$:

1. $G \to A : \langle g_i, \{h(g_i)\}_{K_G^{\text{pri}}} \rangle$ for $i = 1..m$, where each g_i is a new captcha image.
2. $A \to S_I$: $\langle h^\alpha(w_i), g_i, \{h(g_i)\}_{K_G^{\text{pri}}}, h^\alpha(\mathsf{hmac}_{w_i}(g_i)) \rangle$ for $i = 1..m$ for large, globally known integer α.
3. $S_I \to A : B = \langle g_i', h^\alpha(\mathsf{hmac}_{w_i}(g_i')) \rangle$ for $i = 1..m$, where
$$B = \begin{cases} \langle g_i, h^\alpha(\mathsf{hmac}_{w_i}(g_i)) \rangle & \text{if } M_I(h^\alpha(w_i)) = \emptyset \\ M_I(h^\alpha(w_i)) & \text{otherwise} \end{cases}.$$
S_I also verifies the captchas' signatures and ensures that each $h(g_i) \notin H_I$. It then updates M_I with mapping $h^\alpha(w_i) \mapsto \langle g_i, h^\alpha(\mathsf{hmac}_{w_i}(g_i)) \rangle$ for $i = 1..m$ if no mapping exists for that key and adds each $h(g_i)$ to H_I.
4. $A \to S_C : \langle \{C\}_{K_i}, h(\mathsf{captcha}(g_i')) \rangle$ for $i = 1..m$ after solving the captchas, where each encryption key $K_i = h^{\alpha-1}(\mathsf{hmac}_{w_i}(g_i'))$.
5. S_C updates M_C with mapping $h(\mathsf{captcha}(g_i')) \mapsto M_C(h(\mathsf{captcha}(g_i'))) \cup \{\{C\}_{K_i}\}$ for $i = 1..m$.

Figure 1: **The protocol for uploading new content to the aggregator.**

and its corresponding captcha. This enables queriers searching for those keywords to generate the decryption keys, provided that they obtain the captchas from the index server. The client hashes values α times for storage on S_I and encrypts content with keys computed from the $\alpha - 1$ hash for storage on S_C. This ensures that the encryption keys cannot be computed from the values stored on S_I.

The servers that comprise the aggregator, S_I and S_C, are independent key-value stores with key-value mappings M_I and M_C, respectively. At a high level, M_I is a mapping from hashed keywords to unique captchas, and M_C is a one-to-many mapping from hashed captcha solutions to encrypted content. Initially, M_I and M_C map all values to the empty set. When the index server S_I receives a content upload request, it verifies that the captcha image is signed by the captcha generator. S_I maintains a set H_I of the hashes of all captchas stored in M_I, which it uses to reject duplicate captchas. If the captcha is unique and its signature is valid, S_I adds to M_I the mapping from hashed keyword to captcha image for each keyword that is not yet mapped. It returns to the client each keyword's captcha after updating M_I, which the client solves to compute the new content's keys on the content server. The content server updates M_C by adding the new content to the sets mapped from the captchas' solutions. The index server never replaces existing key-value mappings, so the captchas' solutions always refer to the same keys on the content server. This ensures that searches for a keyword result in all content that has been

Client A queries the aggregator for keywords $w_1, \ldots, w_m$:

1. $A \to S_I : h^\alpha(w_i)$ for $i = 1..m$ and large, globally known integer α.
2. $S_I \to A : M_I(h^\alpha(w_i)) = \langle g_i, h^\alpha(\mathsf{hmac}_{w_i}(g_i)) \rangle$ for $i = 1..m$.
3. $A \to S_C : h(\mathsf{captcha}(g_i))$ for $i = 1..m$, after verifying the keyword-captcha HMACs and solving the captchas.
4. $S_C \to A : \bigcup_{i=1}^m M_C(h(\mathsf{captcha}(g_i))) = \{C_j\}_{K_j}$ for $j = 1..n$, where n is the number of search results.
5. A computes $K_j = h^{\alpha-1}(\mathsf{hmac}_w(g))$ for $j = 1..n$, where w is the keyword that S_C mapped to search result j and g is its corresponding captcha, and uses them to decrypt each C_j.

Figure 2: **The protocol for issuing a search query to the aggregator and decrypting the results.**

inserted under that keyword.

Figure 1 lists the full protocol for uploading new content. The protocol prevents malicious users from corrupting the aggregator when uploading new content: the index server's mapping is write-once per key, and extra mappings on either server are of no consequence. The protocol as described in this section assumes that the aggregator servers provide basic key-value storage primitives, enabling Blindfold to operate on existing key-value store services. Section 4 describes how using a specialized many-to-many key-value store on the content server enables it to store only one copy of each encrypted content object instead of a copy per associated keyword.

To search for content, a client hashes each keyword in a query string and sends them to the index server. The index server responds with one captcha per search keyword, each with the HMAC that binds it to its search keyword. The client verifies the HMACs to ensure that the index server did not tamper with its mappings, then solves the captchas and sends their hashed solutions to the content server. The content server responds with a set of encrypted search results. The client computes the content's decryption keys from the HMAC of any keyword from the original query and the keyword's corresponding captcha from the index server. Lastly, the client prunes duplicate search results after decrypting them. Figure 2 lists the full search protocol.

The only link between the two aggregator servers is the captchas, which remain unsolved on the index server, with their hashed solutions indexing the mapping on the content server. Solving captchas efficiently requires human interaction, obscuring the links between related entries on the two servers except when a client searches for those keywords. The result is that keywords, whose hashed values are stored on the index server, are difficult to link to content objects on the content server, which

are encrypted using their associated keywords. Even if a link between entries on the servers were known, the hashed keywords on the index server are insufficient to decrypt the content on the content server.

3.2 Security

Blindfold's primary goal is to protect the aggregator's operators from discovering the content that they are serving. Blindfold achieves this without relying on out-of-band authenticators, allowing any client to perform searches. It is impossible for a public search interface to differentiate between honest clients and malicious clients. Blindfold's security goals, then, are to prevent the aggregator's data from being quickly and systematically discovered while remaining unobtrusive to honest users as they issue targeted searches for content.

The primary defense against attackers is the separation of hashed keywords from the encrypted content. Section 3.1 discussed the motivations for splitting the data across two administrative domains. An attacker that compromises only the index server gleans very little information about the content. The index server does not store any content, limiting the attacker to mounting a dictionary attack to discover the keywords in the system. The index server's mapping can be pre-loaded with captchas whose solutions do not appear on the content server. Pre-loading such a mapping for every English word would effectively hide the real keywords, reducing the viability of this attack. Because each hashed keyword maps to a unique captcha, a snapshot of the server contains no information about the popularities of keywords. A statistical analysis of requests over time or an examination of server request logs would reveal popularities of entries on the servers, but would not reveal the actual keywords or content stored on the aggregator.

An attacker that compromises only the content server would have access to the content objects, each encrypted with one or more keywords and their corresponding captchas on the index server. Without access to the captchas themselves, an attacker cannot decrypt the content. Unfettered access to either server alone leaks no information of the content stored on the aggregator.

Compromising both aggregator servers does not help automated attackers unravel the mapping from keywords to content, as each mapping is protected with a captcha. If the index server is pre-loaded with mappings, as described above, an attacker would have to solve potentially many captchas before discovering a link between an index server mapping and a content server mapping. Even after discovering a link, the keyword hashed on the index server does not reveal the plain-text keyword required to decrypt the corresponding content on the content server. An attack on Blindfold would require significant resources, both computational and human, to decrypt each content object.

4 Implementation

We have implemented a full prototype of Blindfold in three parts. The first is a key-value store service that implements get and put operations. The index and content servers are both instances of this generic key-value store, with parameters that specify how they handle key collisions on insertion. The index server implements a *write-once* mapping, where it discards a put request if a mapping from the key already exists. The content server's one-to-many mapping stores all values inserted under each key by mapping keys to expandable sets of values. Second, we implemented a captcha generator that returns randomly generated captchas and signs each captcha image with its private key. Alternatively, Blindfold can use existing services that generate captchas, which are plentiful [1,3]. Lastly, we implemented the client, which supports upload and search operations, each of which causes the client to interact with the other components. The Blindfold prototype is open source and publicly available at http://www.cs.cornell.edu/∼ryanp/blindfold/.

The Blindfold prototype is surprisingly easy to use. Our experience is that solving one captcha per search keyword is unintrusive and requires little effort. Setting global parameter α to 10^5 provides a reasonable trade-off between search latency and the cost of mounting a dictionary attack to derive content decryption keys; on a modest desktop machine, a search requires approximately two seconds of CPU time per keyword plus the time required to solve the captchas. The delay is negligible to honest clients because clients compute content decryption keys at the same time the user is solving captchas.

The semantics of the index server's write-once and content server's one-to-many stores are typical of key-value stores. However, if the more sophisticated semantics of a one-to-many store are unavailable, a write-once store can be used to implement a one-to-many store with only modifications to the client. If a client attempting to insert mapping $k \mapsto v$ finds that mapping $k \mapsto v'$ already exists, it inserts mapping $h(k) \mapsto v$ instead. To perform a lookup for key k, a client issues requests for $k, h(k), h^2(k), \ldots$ until it receives the empty set as a value, signaling to the client that it has reached the end of the chain. The union of all returned values is equivalent to the intended one-to-many mapping $k \mapsto \{v, v'\}$. The write-once semantics of the underlying store prevent malicious clients from modifying an existing chain.

Our implementation of Blindfold uses two optional mechanisms that prevent clients from hijacking keywords without increasing the number of captchas that queriers must solve. First, it uses a trusted captcha gen-

erator that signs captchas coupled with a modified index server that verifies signatures to ensure that all captchas are solvable and require human interaction. Second, the index server rejects duplicate captchas to prevent multiple keywords from mapping to the same captcha, which would reduce the effort required to enumerate content. Lacking a trusted captcha generator or a modified index server, the correctness of the protocol remains intact if the index server stores a separate captcha for each content object under the same keyword. This requires queriers to solve a captcha for each search result, but the user could stop at any time to view a partial list of results.

An optimization on the content server enables each encrypted content object to be stored only once instead of once per associated keyword. This requires S_C to expose multiput, which maps multiple keys to a single instance of the value, making M_C a many-to-many mapping. The Blindfold protocol changes accordingly: when a client adds content C under keywords $w_1, \ldots, w_m$ with corresponding captchas $g_1, \ldots, g_m$ according to S_I, it generates just one random encryption key K and constructs vectors $\vec{p} = h(g_1), \ldots, h(g_m)$; $\vec{q} = h(\mathsf{captcha}(g_1)), \ldots, h(\mathsf{captcha}(g_m))$; and $\vec{b} = b_1, \ldots, b_m$, where $b_i = K \oplus h^{\alpha-1}(\mathsf{hmac}_{w_i}(g_i))$. The purpose of b_i is to enable a querier for keyword w_i to compute the key K and decrypt C. The client sends to S_C the value $\langle \{\langle C, \vec{p}\rangle\}_K, \vec{b}, \vec{q}\rangle$. When S_C receives the upload request, it adds to M_C a reference to the value $\langle \{\langle C, \vec{p}\rangle\}_K, \vec{b}\rangle$ under each hashed captcha solution in $\vec{q}$. When S_C processes search queries, it returns the intersection of the requested sets because values for the same content are identical, reducing bandwidth at the server.

When a querier receives a search result, it generates m potential decryption keys $K_i = b_i \oplus h^{\alpha-1}(\mathsf{hmac}_w(g))$ using an arbitrary keyword w from its query and w's corresponding captcha g, and attempts to decrypt $\{\langle C, \vec{p}\rangle\}_K$ with each key K_i until it succeeds. The client treats $\vec{p}$ as a checksum, recognizing a successful decryption when $h(g)$ matches some element of $\vec{p}$. Because each content object will only have a few possible decryption keys, one computed from each element of $\vec{b}$, the additional time required to perform a search is imperceptible to the querier. The key-value store in the Blindfold prototype supports the multiput operation and implements this optimization.

A pragmatic issue in running key-value stores is removing old content to reclaim space. This is particularly important in Blindfold because the security of captchas decays over time as automated attacks become more sophisticated. Removing key-value entries in Blindfold poses a challenge because each entry in the index server is used to service queries for multiple content objects. Blindfold can be extended to use a versioning scheme under which each keyword on the index server maps to two captchas at any one time: an active captcha, under which newly uploaded content is placed, and a legacy captcha, which expires when its last content object expires. Active captchas replace expired legacy captchas, and a new active captcha takes its place on the next content upload for that keyword. This scheme places an upper bound on the age of captchas at the cost of requiring queriers to solve up to twice as many captchas to perform searches.

5 Conclusions

In this paper, we described Blindfold, a system that enables users to upload to and search a public key-value store without revealing the true keys or values to the store or third parties. The system works by partitioning and chaining upload and search operations into a series of key-value operations across servers in different administrative domains. The connection between the servers is obscured and protected by captchas. We showed that the system is simple and feasible with a prototype implementation, and we have found from experience with the system that it is surprisingly unintrusive to the user and easy to use.

References

[1] Captchas.Net. http://captchas.net.

[2] Mininova. http://www.mininova.org.

[3] Recaptcha: Stop Spam, Read Books. http://recaptcha.net.

[4] The Pirate Bay. http://piratebay.org.

[5] A. Boldyareva, M. Bellare, and A. O'Neill. Deterministic And Efficiently Searchable Encryption. *CRYPTO*, Santa Barbara, CA, Aug. 2007.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions For Message Authentication. *CRYPTO*, Santa Barbara, CA, June 1996.

[7] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting Free Expression Online With Freenet. *IEEE Internet Computing*, Jan. 2002.

[8] M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Secure Anonymous Database Search. *Cloud Computing Security Workshop*, Chicago, IL, Nov. 2009.

[9] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques For Searches On Encrypted Data. *IEEE Symposium on Security and Privacy*, Washington, DC, 2000.

[10] L. von Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using Hard Ai Problems For Security. *Eurocrypt*, Warsaw, Poland, May 2003.

[11] P. Williams, R. Sion, and B. Carbunar. Building Castles Out Of Mud: Practical Access Pattern Privacy And Correctness On Untrusted Storage. *CCS*, Alexandria, VA, Oct. 2008.

[12] Y. Xie, D. O'Hallaron, and M. K. Reiter. Protecting Privacy In Key-value Search Systems. *CSAC*, Miami Beach, FL, Dec. 2006.