SAND2013-8322C

Percival: Blinded Searching of a Secret Share Archive

Joel C. Frank[1], Shayna M. Frank[1], Ian F. Adams[1], Thomas M. Kroeger[2], Ethan L. Miller[1]
[1]*Storage Systems Research Center, University of California, Santa Cruz, CA 95064, USA*
[2]*Sandia National Laboratories, Livermore, CA 94550, USA*[*]

Abstract

The eventuality of keys being lost, stolen, or otherwise revealed to unauthorized parties makes fixed key encryption ill suited for archival storage. Previous research investigated archives based on secret sharing, which removes the issues present in fixed key encryption, but such archives are difficult to search without compromising security.

To address this need, we present Percival, a searchable archive based on secret sharing that leverages pre-indexing, keyed hashing and Bloom filters to enable blinded searching as well as to limit the release of information in the event a key is revealed to an attacker. The addition of chaff during ingestion and searching operations further obfuscates the terms stored in each Bloom filter and prevents correlation across compromised repositories. During search operations, the archive responds with a representational result set that allows the client to quickly determine the actual search results yet the bulk of the computational burden remains with the repository.

1 Introduction

Fixed key encryption is ill suited for archival storage due to the limitations surrounding key management often resulting in lost, stolen, or discovered keys. Secret share archives show promise since the data is split out across multiple repositories, as well as removing the necessity for a key. However searching such an archive is difficult to accomplish without compromising security.

We present Percival, a searchable archive based on Shamir's secret sharing [15]. Percival pre-indexes the data and combines it with a Bloom filter containing keywords ingested via keyed hashing. This design enables blinded searching, which means the data custodians are blinded to the contents of the search as well as remaining blind to the data in the archive. Percival has the additional benefit of key release not resulting in a catastrophic release of data, which is common for standard key-based encryption.

The keywords stored in each Bloom filter are further obfuscated by adding chaff during both data ingestion into the archive and during search operations. This keeps an attacker from learning the relative number of keywords stored in each filter, which could possible reveal something about the underlying data, as well as colluding in order to make correlations across compromised repositories.

Typically in a secret share archive, client side data reconstruction entails retrieving the shards from each repository, requiring high bandwidth and computational load on the client. Percival circumvents that requirement by having the repositories respond to search requests with the header of the shard, instead of the shard itself. This header represents the shard insofar as if the header is able to be reconstructed, then the shard will be as well, which minimizes bandwidth usage, greatly lessens the client's computational load, and improves security by allowing the Bloom filters and headers to be stored physically separate from the shards themselves.

The rest of the paper is organized as follows: We present relevant background material in section 2. Section 3 details the attack scenarios pertaining to this study. Section 4 introduces the basic overall design, its performance evaluation, and threat assessments for each design consideration. Section 5 discusses the ramifications of adding chaff into the system, the benefits of using headers during reconstruction are presented in section 6, future work in Section 7, and we conclude in Section 8.

[*]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

2 Background

In this section, we provide the background and related work relevant to this study and the new searching schemas, starting with the need for information sharing, secret sharing and related terminology, continuing with the benefits and limitations of Bloom Filters, and finishing with related work in the area.

2.1 Information Sharing

There will always be a vital need to protect information, but facilitating the correlation of overlapping data by enabling information sharing can be far more powerful in ways that can enable trusted communication between distrusting parties.

Percival focuses on the ability to make a secret share archive searchable in such a way that the custodian of the data is blinded to the data involved in the search as well as remaining blinded to the data in the archive. While Percival can play a key role in the fundamental utility of a secret share archive, this new unique ability for blinded search can also be a valuable tool in trusted information sharing.

It is not uncommon to have multiple entities with limited mutual trust wanting to share information when there is an overlap. For example, the Federal Bureau of Investigation (FBI) and the Central Intelligence Agency (CIA) have very different jurisdictions; one looks inward while the other looks outward. As a result, on one hand the two organizations have strong legal restrictions on their ability to share information, but on the other hand have a strong need to share information in order to carry out mission critical tasks. Both organizations may have information on a subject that if correlated would enable both to carry out their missions and prevent an incident, but separately both remain impaired. Percival would allow both organizations to share information while maintaining data privacy.

2.2 Secret Sharing

Secret sharing [15] is an approach to information security based on polynomial interpolation whereby a given piece of data is encoded and split into N pieces by creating N polynomial equations, each with order $T-1$, where $T \leq N$. The data can then be reconstructed if at least T of these equations is known.

The strength of this approach is twofold. First, any attacker who gains fewer than T pieces has gained *no* information, as they effectively have an under-constrained set of equations, making all values of the original data equally likely. For example, Figure 1 shows when two points are known, all possible values for the y-intercept, or secret, are still valid.

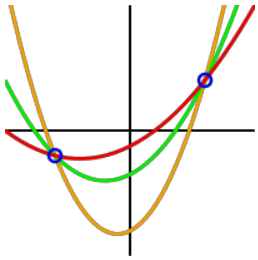

Figure 1: An infinite number of polynomials are equally likely to cross two given points [22].

Second, it has no reliance on keyed encryption, which is ideal for long-term security, as given enough time and computing power, any encryption can be broken. Secret sharing on the other hand is impossible to break without acquiring the requisite number of data pieces [15].

Secret sharing is straightforward to implement and relatively fast to encode, with a worst case running time of $O(n\ log^2 n)$, but it does incur high storage overhead for the security it offers. Secret sharing a piece of data into N pieces necessitates a storage blow-up of N times since each piece is the same size as the original data. Relatedly, there are also trade-offs between ease of access, overhead, and security. A high value for both N and T can make a piece of data very secure, but awkward to access with very high overhead. Conversely, a low value for N or T can reduce the overhead and security.

Of note is the fact that additional pieces may be generated from the same originating data without having to access the ones previously generated or re-encoding the data from scratch, since it only involves adding additional equations to the desired sharing scheme. This scenario is useful when pieces have been lost or compromised, or when a key rotation policy is in place.

2.3 Terminology

Throughout this paper we will use the following terms in regards to Percival and secret sharing:

Shard: The name given to each piece of data generated as a result of splitting a piece of data using the secret sharing algorithm, *e. g.* A 10:6 splitting scheme generates 10 equally sized shards with only 6 of them required for reconstruction.

Sibling Shard: A shard is a sibling of another shard if they are both generated from the same piece of data.

Client: An entity connected to the remove archive that has access to each of the repository keys, key_r, and initiates blinded searches.

Repository: A remote server housing a collection of shards, none of which are sibling shards. No communication occurs between repositories.

Custodian: The entity who manages a repository. Under normal circumstances, a custodian manages no more than a single repository.

Archive: The collection of all repositories in the system.

Stem: Stemming is the process of stripping off all contextual based information from a word, resulting in a stem that is not necessarily an English word, but is identical regardless of contextual use of of the original word. The stem of a word is not the same as its root, *e. g.* The stem of experiment, experiments, and experimentation is experi.

Term: A single stemmed word. Generated during file ingestion as well as during the search process.

Chaff: A set of additional random bits added to a single Bloom filter. Chaff added to one filter will always differ from that added to a different filter.

Bit Group: A group of bits set in a Bloom filter that together represent a single entry or term.

Bit Group Coincidence: The relative number of bits that two bit groups have in common, *e. g.* if two bit groups share three out of four bits, they have a bit group coincidence of 75%.

Bit Group Uniqueness: The inverse of bit group coincidence. The uniqueness of a bit group is the probability that, given a subset of bits from the bit group, the bits can be used to uniquely identify the term represented by the bit group.

2.4 Bloom Filter

A Bloom filter is a probabilistic data structure used for set membership. The key characteristic of Bloom filters is that while false positives are possible, false negatives are not. Furthermore, it is possible to add members to the filter, but once added, they cannot be removed.

Bloom filters are used when pre-indexing each piece of data to be ingested into the archive by attaching a unique Bloom filter to each shard, as well as during the blinded searching process. Both of these uses are discussed in detail in Section 4.

In practice, Bloom filters are implemented as an array of bits. In order to insert a value into the filter, the value is passed through k hash functions, the output of each is an index to be set to 1 in the filter. There are several core factors that directly affect the false positive rate of the filter, which include the following:

- m : the size of the filter
- n : the typical number of values to be stored in the filter
- k : the number of hash functions used to ingest each value

If the size of the filter, m, is too small in comparison to the expected number of values to be stored, n, then the filter will become saturated, *i. e.* most or all of the positions in the filter will be set to 1. When that happens, the false positive rate approaches 100% and the filter becomes useless.

To prevent this, equation 1 is used to find the optimal number of hash functions, k, for a given ratio of filter size, m, to the number of entries in the filter, n, while minimizing the probability false positives, p.

$$k = \frac{m}{n}(\ln 2) \tag{1}$$

$$\ln p = -\frac{m}{n}(\ln 2)^2 \tag{2}$$

However, since at design time that ratio is typically not known, equations 1 and 2 need to be used in conjunction to determine both the filter size, m, and the optimal number of hash functions, k, given only the typical number of entries to be stored in the filter, n. Note that equation 1 assumes optimal p.

The blinded search algorithm depends on both union and intersection operations to be performed on the filters, both of which can be performed using a bitwise OR and AND respectively. Additionally, the union of filters is lossless, *i. e.* the resultant of the union operation is the same as the Bloom filter created by inputing all of the values present in each candidate Bloom filter.

2.5 Related Work

Storer *et al.* developed a system, POTSHARDS [18], which is the current approach for long term storage when information-theoretic security is required. It is based upon Shamir's work with secret sharing [15], and strives to reduce unauthorized exposure of information while maximizing authorized availability in a compromised environment.

The Percival percival project builds on POTSHARDS. The case for our approach focuses on data archives that can operate through system compromises and provide a resilience to insider threat [12]. Some of the first work to support this approach and show the key need for secure operations in spite of adversarial compromise was done by Wylie *et al.* [23].

An alternative to secret share archives was developed by Zage *et al.* [24], in which an algebraic-based encoding solution, Matrix Block Chaining (MBC), is used to "maintain data security and protocol performance when encoding large files. The design of MBC allows for encoding multiple partitions of the original data in parallel

as subsequent encoding operations are not dependent on the output of previous encoding steps. [24]" Their technique was developed specifically for cloud storage, however, and as such does not maintain data availability in a compromised environment.

Due to the rise of privacy issues surrounding data aggregation and commercial storage, there has been a significant amount of work done in recent years regarding encrypted searching [1–7, 9, 11, 16, 21]; all of which rely on the inherent security of the encryption method itself, and since given enough time and computing power most fixed key encryption methods will be broken, they are essentially delayed release. As a result, they are not well suited for applications when data lifetime is measured in decades.

By way of comparison, Octopus, which was developed by Wang *et al.* [20], does not rely on encryption. It is an anonymous way for P2P nodes to communicate via a distributed hash table that provides a mechanism for individual queries to be sent along "multiple anonymous paths, [while introducing] dummy queries to make it difficult for an adversary to learn the eventual target of a lookup. [20]" Our Bloom filter based design precludes the need to use dummy queries as a means of obfuscation since there is no inherent correlation of Bloom filter indexes to search terms.

Chang *et al.* [4] developed an approach using bit masked dictionaries to enable searching of encrypted remote data without revealing information to the data's custodian. The outcome is similar to using a Bloom filter based system where a single bit is used to represent a term stored in the filter. The main difference is that it does not address conjunctive or disjunctive searches, nor does it address mapping multiple terms to the same bit in the dictionary.

3 Threat Model

3.1 Attack Scenarios

Information exposure to unauthorized authorities is the primary threat on which this study focuses. It is assumed that an attacker has unlimited computing power and storage, as well as unlimited time to carry out an attack, since Percival's intended use is for long term storage. Furthermore, in applicable attack scenarios, an attacker has the ability to save an unlimited number of past search queries.

The primary attack scenario we will address at length in each of the design sections, since it is the most plausible, is when an attacker controls a single repository. This scenario can also take several forms, including the site system administrator who has an operational need to have access to the repository, the janitorial staff that needs physical access to the repository, or even a disgruntled insider, as seen in recent news events.

The other attack scenario considered in this study is when an attacker has compromised $T-1$ repositories. Even though a base assumption of this work is that an attacker has full access to at most $T-1$ repositories, compromising a repository is not a binary action. As a result, if $T-1$ sibling shards could be correlated, an attacker need not compromise an entire additional repository to obtain the original data, but rather simply steal a single shard that correlates to the other siblings in the secret share group. For this reason, when applicable, we will address the potential for an attacker to correlate shards across $T-1$ repositories, even though that action alone reveals nothing about the underlying data.

3.2 Authentication

Authentication is the linchpin of any security system. While Percival is focused on data archival, the compromise of the authentication can result in data loss in any system. If a user has been authenticated, we assume that the user has full permission to perform searches on all of the data in the archive. Section 7 discusses an approach to implement access levels in Percival.

Using a Percival like architecture for the data store provides numerous advantages:

1. Several authentication systems must be compromised without detection before data can be disclosed.

2. Physical access can't be used to overcome any authentication and gain access to the data.

3. When it is necessary to make adjustments in security policies, the authentication systems can be updated without the necessity to re-encrypt the entire archive.

4. Most authentications systems already have a standard provision for key rotation.

Fundamentally, this approach moves key management for things like rotation and revocation out of the data store and to the authentication system where they belong. Another key issue that this approach deals with is when the primary user associated with the data is no longer available, for example retired and moved to a tropical island. Suppose a new user has a legitimate need to access data stored by such a retired user years earlier. In an encryption based archive, custody of the encryption key for the data would have to be passed on to an appropriate custodian who would later be able to pass these keys on to the new user needing the data. This model has several

single points of failure and leaves the keys exposed to people after they no long have need for the data.

In the Percival architecture, this new user could set up authentication credentials and the appropriate relationship with each repository to enable access to the data. When they no longer need access the authentication domains would remove the relationship and they would no longer have access to the data in the archive. This approach provides a unique property that the system can keep data encoded in a secure way with no one having access to the data until a business need necessitates it.

Moreover the requirements of authentication can be tailored to the specifics of the data archive's security needs. While it might seem overly burdensome for a user to have to authenticate separately to multiple servers, in some cases the nature of the data and in-frequent use of the archive might warrant such a system configuration. Authentication configurations that require a two man or even greater rule could also be implemented.

On the other end of the spectrum one could have a policy where the host on which a user processes data is deemed as an acceptable place to have that data reassembled. In such an environment, the policy could also allow that host to also hold a key-chain that the user could unlock to open authentication credentials to numerous repositories at once. Such an environment would simplify the users interactions at the cost of some limited risk as deemed acceptable by the organization.

4 Design

In this section, we present the specifics of how searching a secret share archive is achieved. We describe both the core file ingestion process as well as how blinded search is achieved. Each section is followed by a detailed threat analysis that specifies the security implications of the particular design choice.

4.1 File Ingestion

The file ingestion algorithm pre-processes each document client-side prior to secret sharing. During preprocessing, the keywords identified for the document are used to populate unique Bloom filters for each repository. The document is then secret split, after which each shard is bundled with a single, unique Bloom filter. These shard-Bloom filter pairs are each sent to a different repository in the archive. Figure 2 depicts an overview of this process.

There are no restrictions on the members of the keyword set, W; however in this experiment, all unique, stemmed words present in the document are used to populate W. The purpose of which is to preclude the need to reindex a document if new keywords become of interest.

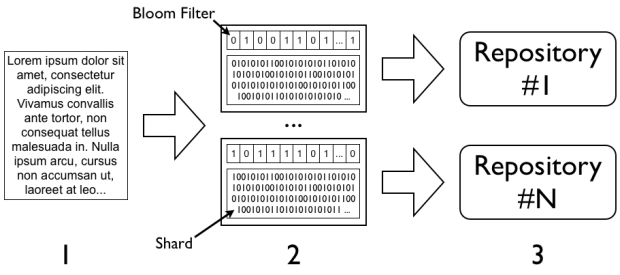

Figure 2: Overview of the file ingestion process. Each file is secret split into N shards (1), each of which is then bundled with a unique Bloom filter, tailored for each repository (2). These shard-Bloom filter pairs are then each distributed to a single repository (3).

During ingestion, each keyword, $w_i \in W$, is used as an input for equation 3 to generate a repository specific unique value, v_i, that represents the keyword.

$$v_i = keyedHash\,(w_i,\, key_r) \tag{3}$$

This study uses HMAC-MD5, a message authentication code utilizing the MD5 cryptographic hash function [14], as the keyed hash function, $keyedHash$. However, since the security of the algorithm is not reliant on the security of the keyed hash function, any other keyed hash function may be used. Each key_r is kept secret from the repositories, and is the *only* secret aspect of this design. Without these repository specific private keys the contents of each Bloom filter, *i. e.* the terms, would be stored in the clear, as well as the Bloom filters for sibling shards being identical across repositories.

It is worth noting that the key used in this algorithm does not necessarily have to be a machine generated, non-meaningful key. It can just as easily be a user's password concatenated with the identifier for that particular repository, since the goal is simply to require a unique key for each repository. The tradeoff to this type of schema is that even though the data sent to each repository will be different, only a single key is required to reveal the contents of the Bloom filters.

Recall that traditionally, a Bloom filter is populated by passing each input value, v_i, through k hash functions; the outputs of which are the indexes, $index_i$, that are set to 1 to indicate the presence of the value in the filter. In our design, however, each value, v_i, is transformed into a bit string, b, by passing it through a SHA hash; every three bytes of which are used to generate an index, $index_i$, based on the size of the Bloom filter, s_{bf}, as shown in equation 4. For this study, a SHA-512 hash was used to generate the indexes, but any desired hash of sufficient length may be used.

$$index_i = b_{(i*3)-((i*3)+2)} \% s_{bf} \quad (4)$$

b_{0-2}	b_{3-5}	...	$b_{(k-2)-k}$	**Not Used**
24 bits	24 bits	...	24 bits	$\|b\| - (k*24)$ bits

Table 1: Breakdown of the bit string used to generate the indexes set in the Bloom filter for each input value, v_i

Finally, the resulting unique Bloom filters, one for each repository, are paired with a single shard and distributed to different repositories in the archive. It is worth noting that the security of this algorithm is not reliant on the security of the hash functions used. It is assumed that all fixed key cryptographic functions can be broken given enough time and computing power. However, breaking the hash functions used, *i. e.* finding one or more collisions in their outputs, actually strengthens this algorithm since it would result in further obfuscating which terms are represented by a bit group.

The number of hash functions had no significant impact on ingestion time, and it was found that the overhead ingestion rate this design imposes is approximately 50 ms/MB, or 20 MB/sec.

4.2 Bloom Filter Design

The first step in determining the proper Bloom filter size for a given implementation is to calculate the minimum allowable size for the filters. This is accomplished by understanding the corpus to be stored in the archive, and determining the typical and maximum number of keywords, or terms, to be stored in each Bloom filter. The typical number of terms defines the filter parameter, n. Recall from Section 2.4, if the size of the filter, m, is too small relative to the maximum number of terms, the filter will become saturated and no longer useful.

In this study, the Gutenberg Library [10] and WikiPedia [22] were used as the test corpora. Figures 3 and 4 show a word count analysis of these repositories. As expected, an asymptotic bound for the number of unique words per document was evident. This bound occurred at approximately 14,000 unique words per document for the Gutenberg Library, and at approximately 3,000 words for Wikipedia, with an average unique word count of 3,000 and 2,000 respectively. The Bible and a German dictionary are labeled in Figure 3 in order to provide a context for these values. These show that even large documents are usually bounded in their number of unique words, and those that aren't, are extremely specialized *e. g.* dictionaries, etc ....

Given the average unique word count for a corpus, equations 1 and 2 can be used to determine the ideal number of hash functions, assuming a false positive rate

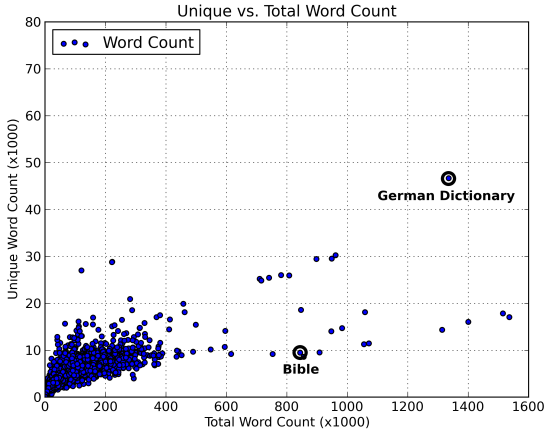

Figure 3: Gutenberg - Comparison of total word count versus the number of unique words per document. The Bible and a German dictionary are shown as reference points.

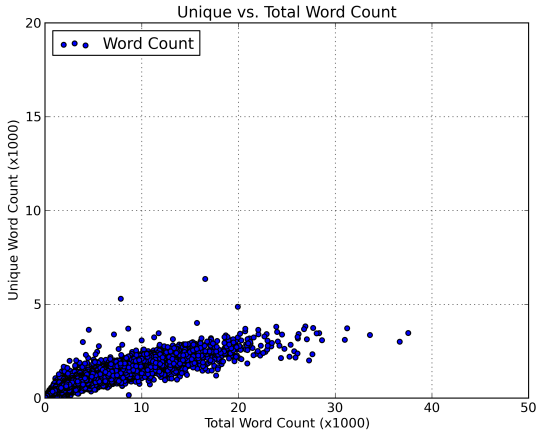

Figure 4: Wikipedia - Comparison of total word count versus the number of unique words per document.

of 0.01%, and defined the ratio of m/n for the Gutenberg Library resulting in a filter size of 100,000 bits. Keeping the same number of hash functions used when processing the Gutenberg corpus, the overall Bloom filter size required for the WikiPedia corpus is 40,000 bits.

Table 2 summarizes the ideal Bloom filter parameters for each corpus.

However, that standard process, which aims to minimize the false positive rate, only serves as a starting point for this design. Since minimizing the false positive rate is analogous to minimizing the bit group coincidence, *i. e.* increase the uniqueness of the bit groups, it is

	Total Size (bits)	Avg. # of Words	# of Hash Functions
Gutenberg	100,000	3,000	13
Wikipedia	40,000	2,000	13

Table 2: Optimal Bloom filter parameters for each test corpus. Chosen sizes result in the same 0.01% false hit rate when using the same number of hash functions. However, these design parameters decrease the bit group coincidence, which potentially exposes the bit groups to an attacker.

desirable to detune the Bloom filters below this optimal size as a first step towards obfuscating bit groups from an attacker. The following section addresses this critical design change in the context of a threat assessment.

4.3 Threat Analysis

This section discusses the most common attack scenario, one in which an attacker controls a single repository, and possible vulnerabilities the file ingestion process has exposed.

With a newly ingested archive that hasn't yet been searched, it is not possible for an attacker to uncover the exact mapping from a particular term to its specific bit group. Furthermore, unless a Bloom filter only contains a single term, it is not possible for an attacker to determine which bits form a bit group simply by analyzing the static repository.

An attacker can attempt to determine similarities between the filters attached to shards stored in the compromised repository. For example, Swamidass *et al.* [19] showed that the approximate number of terms present in each Bloom filter can be found using equation 5, where B is the number of bits set to 1 in the Bloom filter, and recall that m is the size of the filter and k is the number of hash functions used during ingestion.

$$\text{Term Count Estimate} = \frac{-m \ln[1 - \frac{B}{m}]}{k} \qquad (5)$$

While this does not inherently reveal anything about the data, it allows an attacker to organize the shards based on an estimate of the number of keywords stored in the Bloom filters.

In classic encryption schemes, an attacker uncovering the key obviously results in a full release of information. While this situation is undesirable for any secure archive, in a Percival system it does not result in catastrophic loss of security since if an attacker is able to uncover the key to their compromised repository, only a small amount of information is revealed. Once an attacker has the key for a repository, they *are* able to correlate all terms to their associated bit group. However, since the Bloom filters only contain the set of unique stemmed words found in the data, all contextual and semantic information remains secure.

As a concrete example, the book Moby Dick [13] contains approximately 200,000 words and has a Shannon entropy of 4.55 [8, 17]. The Shannon entropy is "the average unpredictability in a random variable, which is equivalent to its information content. [22]" In contrast, the book only has approximately 6,800 unique stemmed words, which drops the Shannon entropy to 3.15. This illustrates that the real data is indeed greater than the sum of its parts.

4.4 Searching

Blinded searching is accomplished via the process depicted in Figure 5. The search terms, along with any desired chaff, are stored in a unique Bloom filter tailored for each repository using the appropriate key for that repository. These Bloom filters are then sent to the archive, one per repository, for processing. Each repository then processes the received Bloom filter in order to generate resultant Bloom filters for any hits and stores them in a mapping of shard ID to resultant Bloom filter. The client then correlates the received mappings to determine the true set of shard IDs that match the requested blinded search.

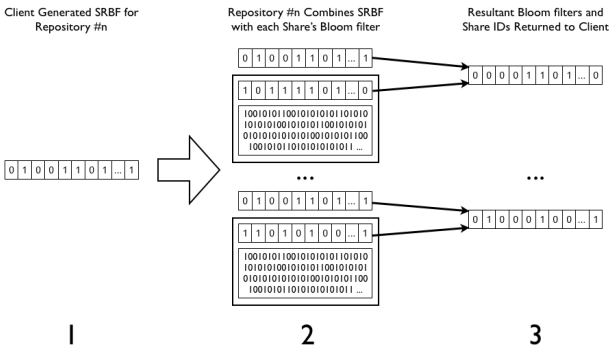

Figure 5: Overview of the searching process. A unique Bloom filter for each repository is generated that contains the search terms and any additional chaff (1). Each repository then processes the received Bloom filter with each of its stored shard-Bloom filter bundles (2), and returns to the client a mapping of shard ID to resultant Bloom filter for client-side processing (3).

4.4.1 Step 1 : Client Side

The first step in processing a search request is to generate a unique Bloom filter for each repository containing the

search terms using the same algorithm described in section 4.1. At this time, random chaff bits unique to each repository may be added as desired in order to further obfuscate the hit patterns across repositories. Section 5 discusses the impact of adding chaff to the Bloom filters stored with each shard.

Once the search Bloom filters are prepared, they are sent to each respective repository along with a hit threshold that allows the repositories to do pre-filtering prior to sending the search results back to the client. Section 4.4.2 details the use of the hit threshold.

4.4.2 Step 2 : Repository Side

The goal of the server is to generate a mapping of shard identifiers to resultant Bloom filters; these filters are created by *and*'ing the search Bloom filter provided by the client with the Bloom filter bundled with each of the shards being stored by the repository.

The cardinality of the resultant Bloom filter is then checked against the hit threshold. If it is found to be greater than or equal to the threshold, the shard ID and resultant Bloom filter are added to the mapping.

Once all shards in repository have been processed, the resultant mapping is sent back to the client.

4.4.3 Step 3 : Client Side

The final step in performing a search begins by removing any chaff bits from each of the Bloom filters in the resultant mappings sent by the repositories. Once the chaff has been removed, the hit threshold is again checked against the cardinality of the resultant Bloom filters to remove any false hits from the mapping.

Since the act of requesting a subset of shard ids can potentially reveal associations between the requested shards, it must be done with care. However, since requesting shard ids is ultimately a secure communication problem, it is outside the scope of this research.

4.5 Searching Performance

Since Bloom filter performance, specifically its false positive rate, is heavily dependent on its core parameters, we now present the empirical data that can be used to tailor the Bloom filter parameters according to the requirements of a specific implementation.

For all experiments, unless otherwise stated, two corpora were used: the Gutenberg Library and Wikipedia. They contained approximately 25,000 and 4 million documents respectively. These were chosen because their differing features required different Bloom filter parameters for each corpus, and therefore allowed for validation of this design under varying conditions.

Bloom filters inherently have a non-zero false positive rate, which is compounded by detuning the filters below optimal in order to improve bit group obfuscation. It is therefore necessary to quantify the impact of this detuning on the accuracy of the search results. As a reference point, it was found that a plain text search for the terms 'motorcycle' and 'Chicago' yielded 26 hits from the Gutenberg Library. This reference search was used as the control group for all subsequent searches using the Bloom filter search algorithm.

Figure 6 shows the impact on the false hit rate by varying the number of hash functions and filter sizes compared to a reference plain text search. It can be seen that as the number of hash functions increases, the false hit rate increases. This phenomenon only occurs because the filters are being detuned in an effort to increase bit group obfuscation. It is also worth noting that the apparent decrease in false hit rate for the 4000 bit Bloom filter when changing from one to two hash functions is due to the combination of the high level of saturation in the filter and the decrease in bit group coincidence when adding an additional hash function.

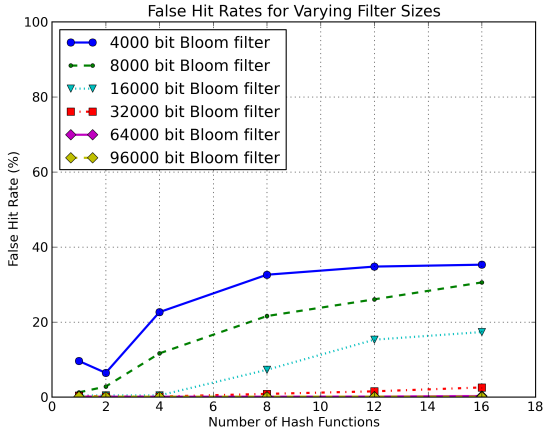

Figure 6: Hit counts for various numbers of hash functions and filter sizes when performing the reference search of 'motorcycle' and 'Chicago'. The true hit count was found to be 26.

A benefit to this design is that search time is not dependent on the size of the files in the archive, nor the number of terms for which the user is searching, since searching simply involves a simple comparison of Bloom filters. As a result, the average time to search a single file was found to be just over 5 μs.

4.6 Threat Analysis

This section addresses potential vulnerabilities introduced by the search algorithm. Again, we begin by focusing on the most prevalent attack scenario; that is when an attacker controls a single repository.

If an attacker is able to obtain the repository key for their compromised repository, the terms contained in the search Bloom filters would be in the clear, assuming no other methods were employed to help obfuscate the search terms, *e. g.* adding chaff and or false search terms. Detuning the Bloom filters while using a low number of hash functions on their own is not enough to have a significant impact on bit group coincidence, *i. e.* with no additional steps taken, an attacker will be able to determine the bit groups present in a search. Additionally, even though there is a significant number of terms that share a single bit, as shown in figure 7, the actual bit group coincidence, which is directly proportional to the number of hash functions and inversely proportional to the filter size, is low enough as to not provide enough obfuscation on its own. For example, in a 4,000 bit filter using 16 hash functions, only 0.000046% of the terms share four bits, which is non-zero but hardly significant.

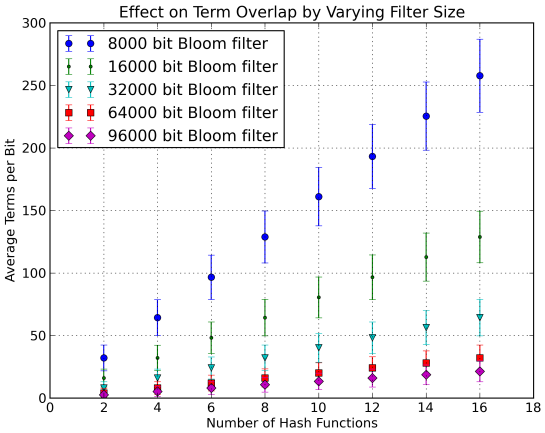

Figure 7: Effect on the average number of terms sharing an index position in a Bloom filter as a result of varying the filter size.

Due to Percival's design, the repository keys are unable to be brute forced without at least one term to bit group correlation. This is because without a correlation there is no way to validate the output when trying candidate keys since every output is *valid and equally likely*. Therefore, a logical first step for an attacker is to uncover at least one correlation.

This could possibly be accomplished using a form of CCA, or chosen cypher-text attack. An attacker could cause a real world event or disaster, and then monitor the search Bloom filters for a shift in the high frequency bit groups. Additionally an attacker could employ a form of adaptive chosen cypher-text attack, CCA2. For example, during the Sturgis motorcycle rally in South Dakota an attacker might assume that the bit groups with the two highest frequencies correlate to 'motorcycle' and 'Sturgis'. The attacker then causes a real world event or disaster involving motorcycles in Chicago, and monitors for the shift in bit group frequencies.

If they are able to uncover a bit group correlation, the upper bound for brute forcing the key is $k! * 2^{|b|-(k*24)}$ possible orderings and values for the bits that form a bit group, which is due to both the unknown ordering of the indexes, $index_i$, and the discarded $|b|$ - $(k * 24)$ bits. Each additional correlation that an attacker is able to discover lowers this upper bound. It is evident that this upper bound can be increased by maximizing the length of b, the output from the hash used for index generation, and by minimizing k, the number of hash functions used.

5 Chaff

This section discusses the impact of adding chaff to the system, both during file ingestion as well as search operations.

5.1 Adding Chaff During File Ingestion

During file ingestion, the addition of chaff to the Bloom filters stored with each shard has several benefits. These include obfuscating the bit groups present in each filter as well as making each filter have the same cardinality, which negates the potential attack previously described in section 4.3 using equation 5 in order to estimate the number of terms present in a given Bloom filter.

This is accomplished by adding chaff to each Bloom filter up to a desired loading level, as opposed to simply adding a fixed amount of chaff. The design tradeoff is that the largest number of terms to be ingested into a filter must be known at design time so that adequate room remain open for chaff, as well as ensuring that no filter's cardinality is already higher than the desired chaff loading level.

Furthermore, in the event a repository's key is revealed to an attacker, it lowers the confidence an attacker will have in the terms actually present in the filter due to the addition of random terms. The probability a particular random term will be added to an individual filter is 0.5^k, which means that the additional percentage of the repository that contains a particular term can be found by equation 6, where $|R|$ is the total number of shards in the repository, and $A\%$ is the percentage of the repository

that actually contains the particular term prior to adding chaff.

$$\text{Additional } \% = |R|(1-A\%)0.5^k \quad (6)$$

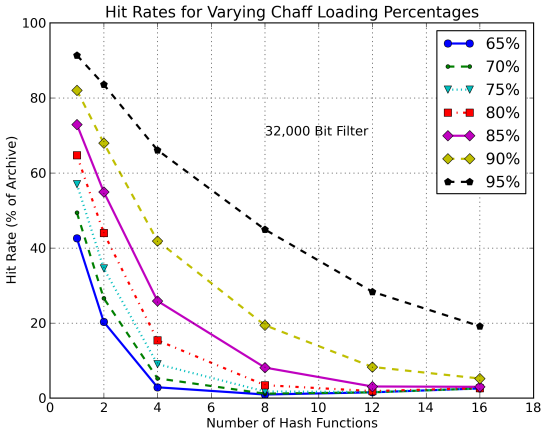

Figure 8: Effect of adding chaff to the Bloom filter stored with each shard. Using fewer hash functions greatly amplifies the effect of chaff loading.

Figure 8 shows the effect on the previous search results of adding varying levels of chaff to the Bloom filters during file ingestion. The cost of this obfuscation is an increase in bandwidth required to handle the responses from each repository, as well as an increase in the post-processing now required by the client in order to separate the false hits from true ones. That process takes the form of requesting applicable shards from each repository and attempting their reconstruction, which has the potential to be a very costly operation depending on the size of the search results. Section 6 discusses this in more detail, as well as presenting a solution.

5.2 Adding Chaff During Search Operations

In addition to adding chaff during file ingestion, it may be added during search operations in either the form of random bits added to the search Bloom filters sent to each repository, or as additional random or deliberately chosen false search terms. As shown in figure 9, even a relatively small amount of chaff added to the search filter greatly adds to the false hit rate.

The effect of adding chaff during searches differs greatly from that of adding it during file ingestion. First, the resulting increase in false hit rate is not dependent on the number of hash functions used, as was shown in figure 8. Second, it takes much less chaff to have a large impact on the size of the result set, *e. g.* the addition of only 100 bits of chaff to the search filter results in roughly a 38% increase in the false hit rate. Lastly, that false hit rate can be easily mitigated by the client by combining each resultant Bloom filter in the result set by the real search filter, *i. e.* the search filter generated prior to the addition of chaff.

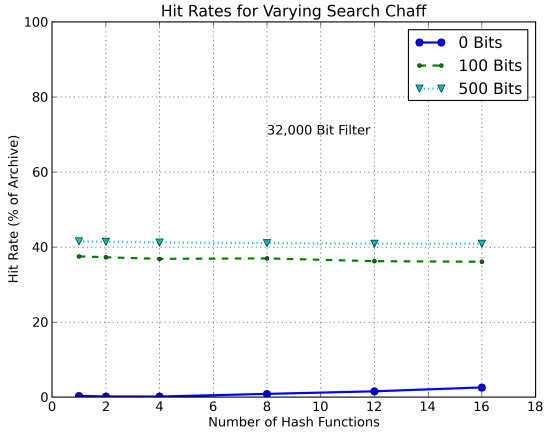

Figure 9: Effect of adding chaff to the search Bloom filter.

6 Shard Headers

This section presents a more efficient manner for the client to mitigate the potentially large result set stemming from file ingestion chaff.

6.1 Header Design

Recall that in section 5.1 it was mentioned that the client would need to request the applicable shards and then attempt their reconstruction, which would filter out all hits in which less than T sibling shards were returned by the archive. This assumption relies on the low probability that at least T sibling shards are falsely returned due to chaff. The main drawback of this process is that it has the potential to be very costly in terms of workload that is now shifted to the client from each repository. In order to reduce this computational burden on the client, the concept of a small header can be introduced into the system, and used during the test reconstruction of the hit results, instead of the full shard, in order to greatly increase performance.

The header is generated during file ingestion, and consists of the shard IDs for each of the sibling shards that together form a document as well as the CRC of those IDs.

ID_0	ID_1	...	ID_{N-1}	CRC

Table 3: Headers consist of the sibling shard IDs and the CRC of those IDs.

Once generated, the header is secret split and stored with the Bloom filter that previously would have been stored with each shard. The introduction of the header not only assists the client with reconstruction, but also has several security related benefits. These newly formed Bloom filter-header pairs can be stored separately from the shards generated during file ingestion. Furthermore, the headers can be secret split using different parameters, *i. e.* a different number of shards, N, and a different reconstruction threshold, T, as those used when secret splitting the data. This allows more flexibility when designing the system, since differing threat models can be addressed at different layers in the system.

6.2 Header Performance

Since the work required to reconstruct a set of sibling shards is proportional to the size of each shard, it isn't feasible to use reconstruction on the shards generated from the data itself as a way to mitigate the false hit rate. This is exponentially compounded when the result set is large.

Performance data was collected using a 64 bit Linux system, with 24GB of RAM and four hyper-threading cores. It was found that on average full shards could be reconstructed at a rate of 2kB per second, which means that it would take approximately an hour and half to reconstruct a 10MB piece of data. By way of comparison, using shard headers is not dependent on the size of the corresponding data they represent, and as a result it was found that a single header could be reconstructed in approximately 2ms.

Table 4 shows the time required to reconstruct the headers for the 28 shards used in the reference search as the false hit rate increases. The false hit rate is represented by the number of additional shards in the result set returned by each repository. These 'false' shards have no siblings, and as such cannot be successfully matched with any other shards in the result set to form a complete header.

It can be seen that the number of reconstruction attempts, and as a result the time required, increases exponentially as the false hit rate increases. Unfortunately, even a result set containing a few false shards, *e. g.* 100, requires the client to spend an unacceptable amount of time to reconstruct the shard headers. This is because a false shard must be tested up to $\frac{N!}{T!(N-T)!)}$ times before it can confirmed as invalid. Section 7 discusses a possible solution for this problem.

Number of Shards in Result Set	Reconstruction Attempts	Elapsed Time (min:sec)
28 + 0	21,952	0:38
28 + 25	148,877	9:59
28 + 50	474,522	40:52
28 + 100	2,097,152	229:21

Table 4: Header reconstruction times as the false hit rate increases. The number of shards in the result set indicates the base 28 shards plus varying amounts of 'false' shards.

7 Future Work

There are several open areas of research with regard to Percival. The first of which is an improved search scheme on each repository, since the development of which was outside the scope of this study. It is hypothesized that reverse indexing can be leveraged to organize the shards within a repository based on their attached Bloom filters. It is in this way that search times could be reduced to linear time. Furthermore, while we have seen great successes in data ingestion and look up it would be informative to develop experiments that provide more detailed testing and results for complex searches using realistic query workloads.

Regarding the high reconstruction time required by the client in order to mitigate the chaff added during file ingestion, a technique similar to that employed in POTSHARDS [18] will be tested. Specifically the use of approximate pointers in order to form rings of shards to greatly narrow down the search space during reconstruction. They were able to achieve performance improvements of up 95% by employing such a method.

The current design of Percival assumes that once a user is authenticated into the system, there is no sense of file ownership. All users authenticated into the system have full access to all files stored within. This leads to another potential way Percival may be expanded. Meta data can be injected into the filter at the time of file ingestion. For example, by automatically injecting the username of the file owner into the filter attached to each shard, and then requiring all valid hit results to have the same k bits set as the username of the current user performing a search, access controls can be enforced. In this way, all levels of access control can be implemented, not just at the user level.

One of the current limitations with Percival's design is that it can only ingest text based data, or data that has been manually tagged with keywords of interest. A use case that does not fit into this model is Sandia's HashNinja project, which is a repository of MD5 hashes. The project's purpose is to provide a standard process for malware triage and analysis while increasing collabora-

tion between trusted organizations that analyze malware. If Percival was extended to ingest MD5 hashes, HashNinja would be able to share its repository in such a way that its custodian is blinded to all searches on the repository.

8 Conclusion

Even though secret sharing removes many of the issues present in fixed key encryption schemes, they are difficult to search without compromising security. This paper presents a highly customizable method to implement blinded search on a secret share archive by utilizing preindexing, Bloom filters, and keyed hashing. It is done in such a way that even if a private repository key is discovered by an attacker, there is no catastrophic release of information.

Chaff is added to the Bloom filters, both during file ingestion as well as during search operations, in order to further obfuscate the keywords stored within each filter. This keeps an attacker from learning the relative number of keywords stored in each filter, which could possible reveal something about the underlying data, as well as colluding in order to make correlations across compromised repositories.

Percival's use of headers during the test reconstruction phase greatly improves the bandwidth requirement when searching, the time required by the client to mitigate the false hit rate introduced by chaff, and the overall security of the system by allowing the Bloom filter-header pairs to be stored physically separate from the shards themselves.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] S. Bellovin and W. Cheswick. Privacy-enhanced searches using encrypted bloom filters. In *Technical Report 2004/022, IACR ePrint Cryptography Archive*, 2004.

[3] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques*, 2004.

[4] Y. C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security Conference*, 2005.

[5] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of Archival Intermemory. In *Proceedings of DL '99*, pages 28–37, 1999.

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered fault tolerance for long-term integrity. In *FAST '09 Proccedings of the 7th conference on File and storage technologies*, 2009.

[7] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[8] R. S. Ellis. *Entropy, Large Deviations, and Statistical Mechanics*. Springer; 1 edition, 1985.

[9] K. M. Greenan, E. L. Miller, T. J. E. Schwarz, and D. D. Long. Disaster recovery codes: increasing reliability with large-stripe erasure correcting codes. In *StorageSS '07*, pages 31–36, New York, NY, USA, 2007. ACM.

[10] Gutengerg. Project gutenberg – project gutenberg literary archive foundation, 2013.

[11] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. USENIX.

[12] T. M. Kroeger, J. C. Frank, and E. L. Miller. The case for distributed data archival using secret splitting with percival. In *1st International Symposium on Resilient Cyber Systems*, San Francisco, California, Aug. 2013.

[13] H. Melville. *Moby-Dick*. Richard Bentley (Britain) and Harper and Brothers (US), 1851.

[14] B. Preneel and P. C. van Oorschot. Mdx-mac and building fast macs from hash functions. In *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963, pages 1–14, 1995.

[15] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

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

[17] M. S. Stoler. *RE-Engineering the Enigma Cipher*. ProQuest, UMI Dissertation Publishing, 2011.

[18] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: secure long-term storage without encryption. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, June 2007.

[19] S. Swamidass and P. Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of Chemical Information and Modeling (ACS Publications)*, 47:952–964, 2007.

[20] Q. Wang and N. Borisov. Octopus: A secure and anonymous dht lookup. *CoRR*, 2012.

[21] H. Weatherspoon. Design and evaluation of distributed wide-are on-line archival storage systems. Technical report, University of California Berkeley, 2006.

[22] Wikipedia. Wikipedia, the free encyclopedia, 2013.

[23] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable storage systems. *IEEE Computer*, pages 61–68, Aug. 2000.

[24] D. Zage and J. Obert. Utilizing linear subspaces to improve cloud security. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.