Percival: Blinded Searching of a Secret Split Archive

Joel C. Frank[1], Shayna M. Frank[1], Ian F. Adams[2], Thomas M. Kroeger[3], Ethan L. Miller[1]
[1]Storage Systems Research Center, University of California, Santa Cruz, CA 95064, USA
[2]EVault, San Francisco, CA 94103, USA
[3]Sandia National Laboratories, Livermore, CA 94550, USA[*]

Abstract

Secret splitting across independent sites has been proposed for data storage in archival systems as an approach that removes the issues surrounding key management resulting from fixed key encryption. However, the inherent security of such an archive precludes it from being directly searched; as a result, applications for secret split archives have been limited in the general environment.

In this paper, we present a novel method to perform blinded search across a secret-split archive, which we call Percival. We leverage pre-indexing, keyed hashing and Bloom filters to enable blinded searching, blinding the archive from knowing what terms are being queried. The addition of chaff during ingestion and search operations not only keeps an attacker blind to the data in the archive, but also precludes correlating search results that could potentially reveal which shares are needed for reconstruction. While chaff increases the number of false positives at the archive site, the client can quickly filter most of them. This makes reconstruction of results by the client relatively efficient, keeping the bulk of the computational burden on the repositories.

1 Introduction

Security is a critical issue for long-term storage, particularly given recent incidents such as the Edward Snowden [20] and Bradley Manning disclosures of secret data and the revelation that the National Security Agency may have compromised the design of encryption algorithms. As the volume of health care data, sensor data, personal video and images, and sensitive corporate and government data continues to increase, the threat of compromise at *some* point in the future continues to grow [23]. Much of this risk is due to traditional archival storage having a single point of compromise: the archive repository. If that one point is compromised at any time during the archive's lifespan, the archive's data can be leaked.

Secret-splitting of archival storage can mitigate this problem by distributing data to multiple archive repositories. Secret-split storage was first described by Wylie *et al.* in the PASIS project [30], and adapted to archival storage in the POTSHARDS project [24]. Secret-split archives divide a data object into *shares*, each of which is stored on a separate repository in a distributed environment, making the archive less vulnerable upon the compromise of a single site. However, data protection in such an archive relies on an attacker not being able to determine which shares from different repositories go together, so any long-term archive that uses secret splitting must prevent an attacker from gaining this knowledge.

The need to be able to search long-term archival storage is at odds with the need to keep data secure: queries can reveal information about the data they cover, and can potentially allow an attacker to determine which pieces of data are related, even if the attacker can't immediately identify the data content itself. This is of particular concern for secret-split archives, since the security of the archive depends in part on preventing an attacker from identifying relationships between shares, yet most techniques for searching data allow an attacker to correlate query results, yielding information about which shares are related.

To address this problem, we developed Percival, a archive that uses secret splitting to store data across multiple repositories while providing a mechanism for users to query the archive. Percival pre-indexes the data before splitting it, and combines the resulting shares with a Bloom filter containing terms ingested via keyed hashing. This design enables blinded searching: the data custodians are blinded to the contents of the search as well as remaining blind to the data in the archive. Like all

[*]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.

secret-split archives, Percival has the additional benefit of resistance to key and encryption algorithm compromise and the ability to recover data if sufficiently many repositories agree to do so.

The terms stored in each Bloom filter are further obfuscated by adding chaff during both data ingestion and search operations. This keeps an attacker from learning the relative number of terms stored in each filter, which could possibly reveal something about the underlying data, and makes it more difficult to correlate related shares on a single archive. In addition, the use of chaff obscures correlations between related shares across compromised archives, preventing an attacker from identifying the shares necessary to rebuild a stored data item. The byproduct of adding chaff is an increase in the false hit rate, and subsequently an increase in the number of shares needed to be reconstructed by the client, the cost of which is high bandwidth and computational load on the client [24]. Percival mitigates that requirement by having the repositories respond to search requests with the headers of the shares that meet the search criteria, rather than sending the shares themselves. Each header represents the share: if the header of the object can be reconstructed, then the payload of the object can also be reconstructed. This minimizes bandwidth usage, greatly reduces the client's computational load, and improves security by allowing the Bloom filters and headers to be stored physically separate from the shares themselves. Furthermore, the headers can be secret split following a completely different scheme than the data.

This approach not only provides added security, but is also compatible with organizations that consist of many individual sites with limited ability to collaborate. It is not uncommon to have multiple entities with limited mutual trust wanting to share information. 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, the two organizations have strong regulatory restrictions on their ability to share information, but they also 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 improve their ability to carry out their missions, but separately both tend to remain impaired. Percival would allow both organizations to share information while maintaining data privacy.

No security mechanism is perfect, of course, but Percival provides the ability to make a secret split 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. This design prevents a disgruntled insider from successfully disclosing the information under their care in several ways. A single repository contains only a single share from any piece of information and is therefore useless on its own. Rather, in order to reproduce the release of information perpetrated by several recent incidents, many large queries would have to be submitted to multiple repositories in the archive; this unusual usage pattern would easily be detected and reported. While Percival can play a key role in the fundamental utility of a secret split archive, this new unique ability for blinded search can also be a valuable tool in trusted information sharing.

The rest of the paper is organized as follows: We present background material in section 2. Section 3 details the attack scenarios pertaining to this study. Section 4 introduces the basic overall design, including file ingestion, searching, chaff, and the user of search headers. The performance evaluation is presented in Section 5, followed by a threat analysis in Section 6. Related work is discussed in Section 7, future work in Section 8, and we conclude in Section 9.

2 Background

Since Percival relies heavily on secret splitting for data storage and Bloom filters for searching, this section provides background on these concepts and their application to searchable archival storage.

2.1 Secret Splitting

Secret splitting is the act of splitting a piece of data into N *shares* such that only T shares are required for reassembly, where $T \leq N$. Thus, for example, a 6:10 splitting scheme generates ten equally-sized shares, where any 6 shares will enable the reconstruction of the original data. All ten shares are *sibling shares* for one another: a share is a sibling of another share if they are both generated from the same piece of data. A critical property of secret splitting is that, with less than T shares, *no* data is revealed; however.

There are several known techniques to accomplish splitting a secret, all of which have varying levels of information-theoretic security. Shamir first introduced the concept of secret splitting [19]; Shamir secret splitting provides provably information-theoretic security, but requires that a data object of d bytes store N shares, **each** of d bytes, for a total storage of dN. AONT-RS [18] combines an all-or-nothing transform with Reed-Solomon coding, resulting in a secret-splitting algorithm that is more efficient with storage space: each shares is only d/T bytes long, for a total storage of dN/T. However, AONT-RS is only computationally secure, not information-theoretically secure, since an attacker could guess a fixed-size key and recover data from fewer than T shares. We are currently investigating an approach that provides a middle ground between these methods, with security guarantees that can be tuned from varying levels of computational security to information-theoretic security.

While all of these approaches provide varying levels of security, Percival is agnostic with respect to the choice of secret splitting algorithm, treating the algorithm as a black box.

2.2 Bloom Filters

A Bloom filter is a probabilistic data structure used to test set membership [5]. The key characteristic of Bloom filters is that while false positives are possible, false negatives are not. Furthermore, it is possible to add terms to the filter, but once added, they cannot be removed. We define a *term* as a single word, identifier, or other piece of information to be used as an input to a Bloom filter. Percival uses Bloom filters when pre-indexing each piece of data to be ingested into the archive by attaching a unique Bloom filter to each share. Bloom filters are subsequently used during the blinded search process. These uses are discussed in detail in Section 4.

In practice, Bloom filters are implemented as an array of bits. To insert a value into the filter, the value is passed through k hash functions, where the output of each is the index of a bit to be set to 1 in the filter. There are several factors that directly affect the false positive rate of the filter, including 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 result of the union operation is the same as the Bloom filter created by inputing all of the values present in each candidate Bloom filter.

Bloom filters are highly compressible, as shown by Mitzenmacher *et al.* [16]. It is possible to reduce the size of the filter, the false positive rate, as well as the computation required for each lookup operation, at the cost of processing time for each compression and decompression. They are also easily adapted to perform a reverse index search by treating the filter as column-based instead of row-based. These, and similar, optimizations may be used to improve Percival's performance and efficiency.

3 Threat Model

In order to understand how Percival mitigates the risk of data leakage, it is important to specify the assumptions we are making about what attacks may be used against the system and to what resources the attackers have access.

3.1 Attack Scenarios

Information exposure to unauthorized users 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 in each of the design sections is that of an attacker controlling a single site storing data for the archive. This is the most plausible attack, since it requires only an adversary with unlimited access to a *repository*—a single site storing shares for the archive. A repository is a site housing a collection of shares, none of which are sibling shares. Under normal circumstances, no communication occurs between repositories. However, an attacker may communicate out-of-band with other repositories as part of an attack. This scenario can also take several forms, including the site system administrator who has an operational need to have access to an entire repository, the janitorial staff that needs physical access to the repository, or even a disgruntled insider such as Edward Snowden [20].

The other attack scenario considered in this study is when an attacker has compromised $T-1$ repositories. A Percival secret split archive contains a total of N repositories, where N is the number of shares into which each piece of data is split; recall that at least T shares are needed in order to reconstruct the data. Therefore, if an attacker compromised T repositories, they would have access to T shares from each piece of data and would subsequently be able to reconstruct the original data. We define an *archive* as the collection of all repositories that together create a system for archiving and operating on sensitive data.

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 shares could be correlated and a remaining sibling share easily identified, an attacker need not compromise an entire additional repository to obtain the original data, but could instead steal the missing sibling share. For this reason, when applicable, we will address the potential for an attacker to correlate shares across repositories 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 archival data, the compromise of 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 across all of the data in the archive. Section 8 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 to *all* repositories can be used to overcome any authentication and gain access to the data; however, access to a single repository cannot.
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 authentication 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. The issue of key management when no user currently requires access, but access will be required in the future is another issue with encryption based archives [24].

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 hold a keychain the user is able to unlock to open authentication credentials to numerous repositories at once. Such an environment would simplify the user's interactions at the cost of some limited risk as deemed acceptable by the organization.

4 Design

We now present the specifics of the design for a searchable secret split archive. We describe both the core file ingestion process as well as our techniques for blinded search. 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 at the client prior to the use of secret splitting. We define a *client* as an entity connected to the remote archive that has access to each of the repository keys, key_r and is capable of generating blinded queries and processing the results. During pre-processing, the terms identified for the document are used to populate unique Bloom filters for each repository. The document is then secret split, after which each share is bundled with a single, unique Bloom filter. Each combination of share and metadata is then sent to a different repository in the archive. Figure 1 depicts an overview of this process.

We define $W = \langle w_1, \ldots, w_k \rangle$ as the set of terms used during file ingestion. The terms can be obtained by any technique, including stemming, keyword selection, user tagging, or more advanced methods such as audio transcription or image recognition; there is no restriction on

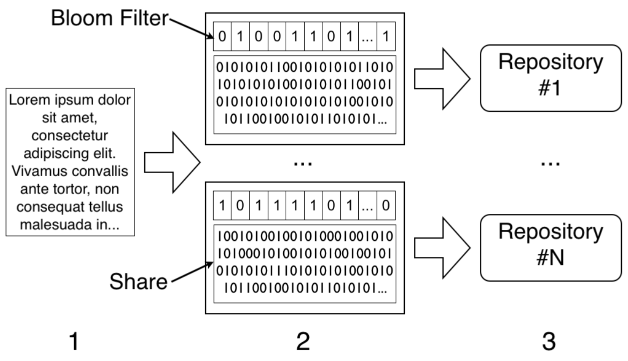
Bloom Filter
0 1 0 0 1 1 0 1 ... 1
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus convallis ante tortor, non consequat tellus malesuada in...
1 0 1 1 1 1 0 1 ... 0
Share
Repository #1
Repository #N
1
2
3

Figure 1: Overview of the file ingestion process. A file is secret split into N shares (1), each of which is then bundled with a unique Bloom filter, tailored for a particular repository (2). The resulting share-Bloom filter pairs are then each distributed to a single repository (3).

the terms inserted into each Bloom filter. During ingestion, each term w_i is used as an input to Equation 3 to generate a repository specific unique value, v_i, that represents the term.

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

The keyed hash function, *keyedHash*, is a message authentication code utilizing a cryptographic hash function [17]. Each key_r is kept secret from the repositories, and is the *only* secret aspect of this design. The repository-specific hash keys protect the contents of each Bloom filter, preventing an attacker from correlating terms to bit groups, as well as ensuring the Bloom filters for sibling shares are different across across repositories. We define a *bit group* as a group of bits set in a Bloom filter that together represent a single term.

Normally, 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. We then extract 24-bit disjoint sections from the hash to generate each index $index_i$, based on the size of the Bloom filter, s_{bf}, as shown in Equation 4. Since all of the bits of SHA-512 are independent, this approach is comparable to traditional approaches that use different hash functions; if additional bits are needed, we can use a longer hash function.

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

Finally, the resulting unique Bloom filters, one for each repository, are each paired with a single share and distributed to different repositories in the archive. It is worth noting that the data protections within the archive

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

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

are 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.

4.1.1 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 examining 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 2.2 that 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 [13] and WikiPedia [29] were used in order to verify the blinded search algorithm against test corpora containing documents with different sizes. Figures 2 and 3 show a word count analysis of these repositories. This is relevant if, when pre-indexing a document, all unique words present are inserted into the Bloom filter, since there is the possibility of overloading the Bloom filter by inserting too many terms. An asymptotic bound for the number of unique words per document was evident; it 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 2 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.

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 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.

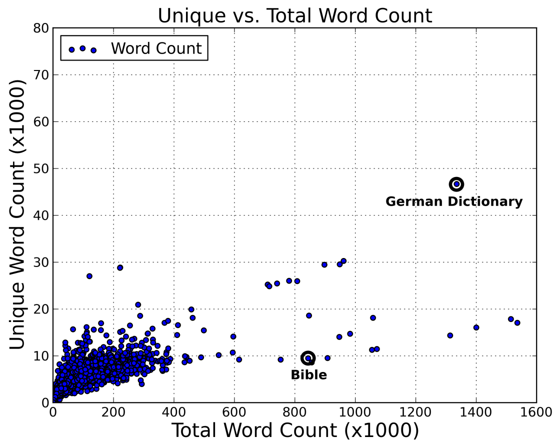

Figure 2: Gutenberg - Comparison of total word count versus the number of unique words per document. The Bible and a German dictionary are shown as reference points.

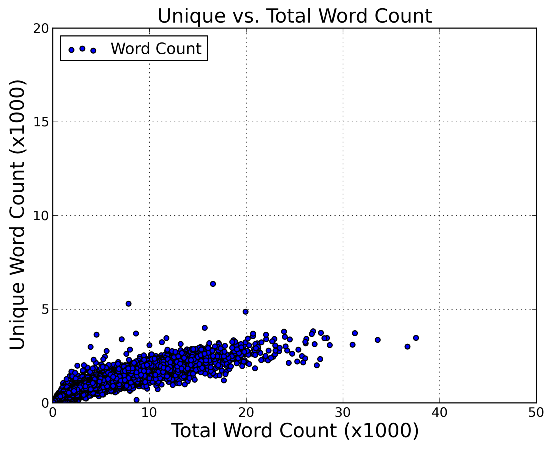

Figure 3: Wikipedia - Comparison of total word count versus the number of unique words per document.

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

	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.

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, it is desirable to detune the Bloom filters below this optimal size as a first step towards obfuscating bit groups from an attacker. *Bit group coincidence* is the relative number of bits that two bit groups have in common: if two bit groups share three out of four bits, they have a bit group coincidence of 75%. The following section addresses this critical design change in the context of a threat assessment.

4.2 Searching

Blinded searching is accomplished by storing the search terms, along with any desired chaff, in a unique Bloom filter tailored for each repository using the appropriate key for that repository. *Chaff* is defined as a set of additional random bits added to a single Bloom filter. These Bloom filters are then sent to the archive, one per repository, for processing. Each repository then stores the filters in a two-way mapping between share ID and resultant Bloom filter, allowing the repository to search Bloom filters for matches to subsequent queries. Searches are done by checking each share's Bloom filter against the query Bloom filter, with the result set consisting of all shares whose Bloom filter contains sufficiently many of the set bits from the query filter. An overview of this process is depicted in Figure 4.

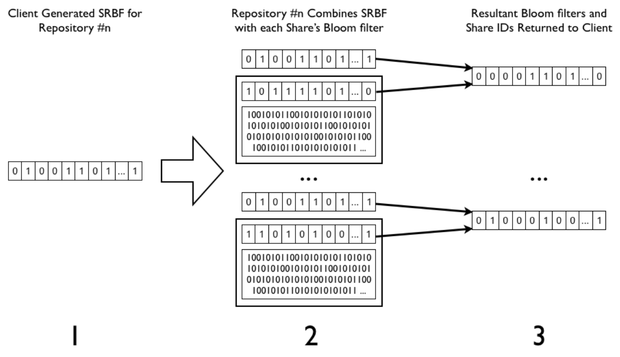
Client Generated SRBF for Repository #n
Repository #n Combines SRBF with each Share's Bloom filter
Resultant Bloom filters and Share IDs Returned to Client
1
2
3

Figure 4: Overview of the searching process. A unique search Bloom filter, SRBF, 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 share-Bloom filter bundles (2), and returns to the client a mapping of share ID to result Bloom filter for client-side processing (3).

4.2.1 Step : 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, a different set of random chaff bits for each repository may be added in order to further obfuscate the hit patterns across repositories and hide the relationship between bit groups and terms.

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. The hit threshold is the number of bits that need to be in common when comparing the search Bloom filter to the Bloom filter stored with each share; the hit threshold is set to the number of bits in the bit groups corresponding to the search term. For example, a Bloom filter containing three terms with $k = 8$ would require 24 bit matches, even if additional chaff bits were added. Any filters with fewer than 24 bits in common cannot contain all three terms.

4.2.2 Step 2: Repository Side

The goal of the server is to generate a mapping of share 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 shares being stored by the repository. The cardinality of the resulting Bloom filter is then checked against the hit threshold. If it is found to be greater than or equal to the threshold, the share ID and resultant Bloom filter are added to the mapping. Once all shares in the repository have been processed, the resultant mapping is sent back to the client.

4.2.3 Step 3: Client Side

The final step in 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. This can be quickly calculated by *AND*ing the search Bloom filter, without chaff, with the Bloom filters returned by the repository; if any bits are zero, the corresponding share does not contain all of the query terms.

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

4.3 Chaff

Chaff plays a critical role in obfuscating search hit patterns across repositories, as well as concealing the contents of both the Bloom filter stored with each share as well as the filter generated during search operations.

4.3.1 Index Bloom Filters

During file ingestion, the addition of chaff to the index Bloom filters stored with each share has several benefits. First, it obfuscates the bit groups present in each filter. Second, it makes each filter have the same cardinality, which negates the potential attack described in Section 6 in order to estimate the number of terms present in a given Bloom filter. Lastly, it removes an attacker's ability to correlate shares across repositories based on the cardinality of the index Bloom filter.

This is accomplished by adding chaff to each Bloom filter up to a desired loading level, *i. e.* set random bits in the filter to 1 until its cardinality reaches the desired percentage loading. The design tradeoff is that the 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.

4.3.2 Search Bloom Filters

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. The primary purpose of adding chaff to the search Bloom filters is to inhibit an attacker from correlating search results across repositories. It also obscures what the true bit groups are in the filter, since the search Bloom filter is a primary vehicle an attacker can use to aid in bit group definition. In contrast to chaff added to the index Bloom filter, search chaff is easily removed by the client since it can keep a record of the search true search Bloom filter.

4.4 Share Headers

The client can reduce the potentially large result set stemming from file ingestion chaff via the use of *share headers*, each of which is a small representational piece of information that can be returned in the result set instead of the entire share. As described in Section 4.3.1, the client would normally need to request the applicable shares and then attempt their reconstruction, which would filter out all hits in which fewer than T sibling shares were returned by the archive. This assumption relies on the low probability that at least T sibling shares 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, we use share headers, instead of the full share, to dramatically reduce the amount of data that must be rebuilt to verify query results across repositories; reducing the volume of data that must be reconstructed to test results greatly improves performance.

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

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

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

Once generated, the header is secret split and stored with the Bloom filter that would have been stored with each share. 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 shares generated during file ingestion. Furthermore, the headers can be secret split using different parameters, *i. e.* a different number of shares, N', and a different reconstruction threshold, T', relative to 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.

5 Experimental Results

Percival's design was tested using a 64 bit Linux system, with 24GB of RAM and four hyper-threading cores. 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, and were chosen because their differing features required different Bloom filter parameters for each corpus, which allowed for validation of this design under varying conditions. All unique, stemmed words present in the document were used to populate the index Bloom filters.

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

5.1 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.

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 a focused result set from the Gutenberg Library. This reference search was used as the control group for all subsequent searches using the Bloom filter search algorithm.

Figure 5 shows the impact on the false hit rate by varying the number of hash functions and filter sizes compared to a reference search. It can be seen that as the number of hash functions increases, the false hit rate increases. 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.

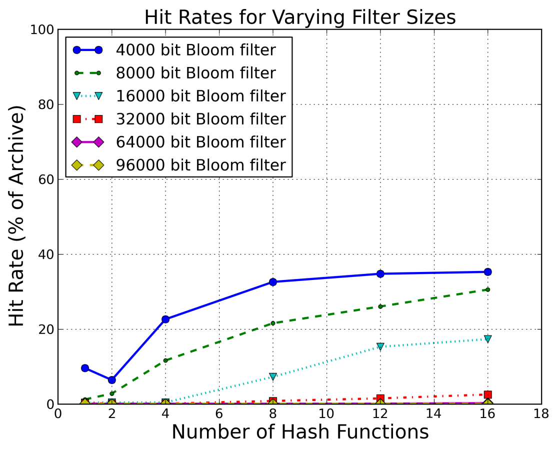

Figure 5: Hit counts for various numbers of hash functions and filter sizes when performing the reference search of 'motorcycle' and 'Chicago'.

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.

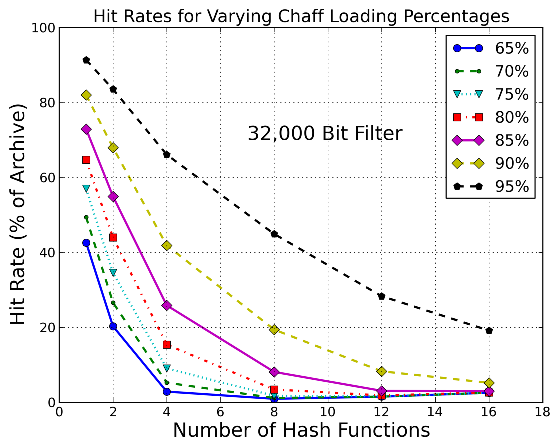

Figure 6: Effect of adding chaff to the Bloom filter stored with each share. Using fewer hash functions greatly amplifies the effect of chaff loading.

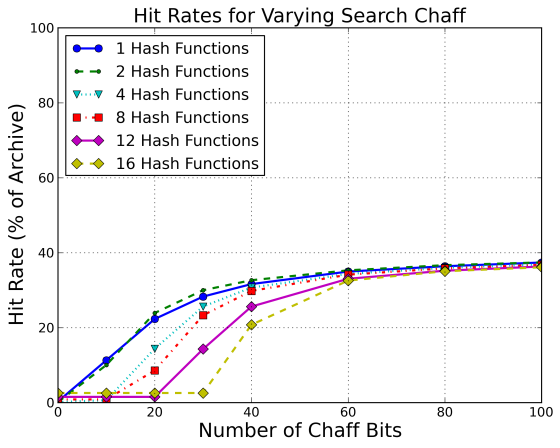

Figure 7: Effect of adding chaff to the search Bloom filter.

Figure 6 shows the effect on the 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 shares 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.

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 6. Second, it takes much less chaff to have a large impact on the size of the result set. 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.

5.2 Header Performance

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

It was found that on average full shares 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 share 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 shares returned from the reference search as the false hit rate increases. The false hit rate is represented by the number of additional shares in the result set returned by each repository. These 'false' shares have no siblings, and as such cannot be successfully matched with any other shares in the result set to form a complete header.

Number of Shares 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 shares in the result set indicates the base 28 shares plus varying amounts of 'false' shares.

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 shares, *e. g.* 100, requires the client to spend an unacceptable amount of time to reconstruct the share headers. This is because a false share must be tested up to $\frac{N!}{T!(N-T)!)}$ times before it can confirmed as invalid. Section 8 discusses a possible solution for this problem.

6 Threat Analysis

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 shares stored in the compromised repository. For example, Swamidass *et al.* [26] 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.

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

While this does not inherently reveal anything about the data, it allows an attacker to organize the shares 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 modest 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 by the secret sharing method.

As a concrete example, the book Moby Dick [15] contains approximately 200,000 words and has a Shannon entropy of 4.55 [11, 22]. The Shannon entropy is "the average unpredictability in a random variable, which is equivalent to its information content [29]." 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.

The other attack scenario addressed is when an attacker controls more than one repository, specifically at most $T-1$, which affords an attacker the possibility of correlating shares across repositories. If no chaff is added to the Bloom filters stored with each share, then sibling shares differ only in which bits are set due to each repository's key. The cardinality of the filter for sibling shares is identical, and as a result greatly limits the search space for an attacker attempting to identify sibling shares. Again, this attack in and of itself is limited because the attacker requires at least T shares to attempt reconstruction, but since compromising a repository is not necessarily a binary action it is worth mentioning.

If chaff is added to the index Bloom filters, it greatly reduces the potential for correlating shares across repositories when an attacker controls more than one, specifically at most $T-1$. Correlation based on cardinality is obviously no longer possible. An attacker *is* able to group shares based on what is *not* in the filters, but this is of limited usefulness due to the chaff. For example, using four hash functions and a filter size of 32,000, the average filter loading for the Gutenberg library due to just the terms is roughly 30%. Therefore, if a significant chaff loading is chosen, *e. g.* $> 60\%$, grouping the shares by what is not in their filter only slightly reduces the search space when trying to correlate shares based on their stored Bloom filter.

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 shares in the repository, and $A\%$ is the percentage of the repository that actually contains the particular term prior to adding chaff.

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

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 8, 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.

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

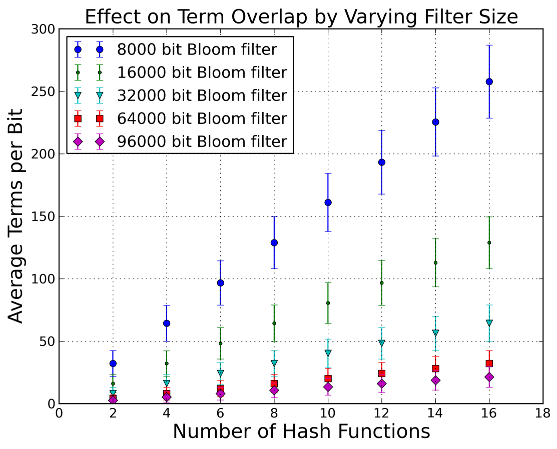

Figure 8: Effect on the average number of terms sharing an index position in a Bloom filter as a result of varying the filter size.

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.

7 Related Work

Our work on Percival builds on two ares of related work: secret split archives and searching encrypted data sets while keeping the custodian blinded to the search.

Secret-split storage was first developed practically in the PASIS project [30], which also contains a good overview of *p-m-n* threshold encoding schemes suitable for use in secret-split archives. In response to several calls for providing archival storage that can operate through system compromises and provide resilience to insider threat [3, 23], this approach was later adapted for archival storage by Storer *et al.* in the POTSHARDS system [24]. Percival's share storage most closely resembles POTSHARDS, though Percival uses a different approach for determining whether shares from different repositories can be combined to rebuild a data object. POTSHARDS also uses an index, stored in POTSHARDS itself, to track the relationships between shares; however, this index cannot be searched without the client first retrieving it.

An alternative to secret split archives was developed by Zage *et al.* [31]. In this approach, 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" [31]. Their technique was developed specifically for cloud storage, however, and as such does not maintain data availability in a compromised environment.

The main concern in Percival is an attacker correlating search results across repositories. There has been a significant amount of work done in recent years regarding encrypted searching [1, 4, 6–10, 12, 14, 21, 28], but these works are all based on a single repository storing all of the user's encrypted data. Since they run on a single repository, they do not need to address the vulnerability of search result correlation, particularly across multiple repositories.

Furthermore, they rely on the inherent security of the encryption method itself since both the data and the search terms simply use fixed key encryption. However, given enough time and computing power, as well as deliberately poor design, it is possible that fixed key encryption methods will be broken; thus, these approaches 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 [27], 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." [27] In contrast to using dummy queries as a means of obfuscation, Percival injects dummy bits into the search queries themselves to accomplish the same goal.

Chang *et al.* [7] 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.

8 Future Work

There are several open areas of research with regard to Percival. The first area is an improved search scheme on each repository, potentially improving archive efficiency. We may be able to use reverse indexing to organize the shares within a repository based on their attached Bloom filters. This could reduce search times could be reduced to linear in the number of documents. Furthermore, while we have seen great successes in data ingestion and look up, we plan to develop experiments that provide more detailed testing and results for complex searches using realistic query workloads.

We also plan to reduce the high reconstruction time required by the client in order to mitigate the chaff added during file ingestion, possibly using a technique similar to that employed in POTSHARDS [24]. By using approximate pointers in order to form rings of shares to greatly narrow down the search space during reconstruction, POTSHARDS was able to achieve performance improvements of up 95%; we will investigate whether similar approaches can help performance in Percival.

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 in it. This leads to another potential way Percival may be expanded. Metadata 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 share, 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. However, the use of chaff may partially defeat this approach, since chaff bits may overlap those used to index a username, potentially allowing a user to gain access to shares she should not be able to access.

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 Hash-Ninja 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 collaboration between trusted organizations that analyze malware. If Percival was extended to ingest MD5 hashes, Hash-Ninja would be able to share its repository in such a way that its custodian is blinded to all searches on the repository. However, it is not clear how terms would be generated and searches done on such an archive.

We are also exploring techniques for building very large-scale, highly-reliable Percival archives, possibly built from low-power clusters of nodes [2, 25]. Ensuring reliability both within an archive and between archives is a difficult challenge, since the repositories themselves do not know which shares belong together, complicating rebuilding in case of device or entire repository failure.

9 Conclusion

Even though secret-splitting archival storage removes many of the issues present in fixed key encryption schemes, such archives are difficult to search without compromising security. We have developed Percival, a system that uses Bloom filters and chaff to facilitate blind searches on secret-split archives without revealing information about the searches to the repositories. Moreover, an attacker who compromises a repository cannot discover correlations between shares on the repository, even using the queries. An attacker also lacks the ability to correlate shares between repositories, making it difficult to identify shares that need to be combined to rebuild a piece of data.

The use of chaff during both file ingestion and query execution obfuscates the bits that correspond to individual terms, foiling attacks that rely on identifying bits set on multiple documents. Chaff also prevents an attacker from calculating the number of terms stored in each filter, further hiding information about the shares stored on each repository.

Percival's use of share headers during the test reconstruction phase greatly improves the bandwidth requirement when searching while reducing the time required by the client to identify the "true" responses to a query. It also improves overall system security by permitting repositories to store the Bloom filter-header pairs separately from the shares themselves, and by allowing different types of secret splitting with different information-theoretic security levels to cover the share headers.

As society needs to store an ever-increasing volume of potentially sensitive data for a long time, the use of secret-split archives will become increasingly necessary to ensure both long-term data security and guard against key loss. The techniques for secure searches developed for Percival will help to make such archives much more usable, ensuring that the long-term data in them will not be simply stored, but rather be available for effective access and use via search queries. By increasing the utility and value of long-term data storage, this approach can make it cost-effective to maintain secure long-term archives.

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] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, MT, Oct. 2009.

[3] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, Apr. 2006.

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

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] 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.

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

[8] 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.

[9] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered fault tolerance for long-term integrity. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[10] 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.

[11] R. S. Ellis. *Entropy, Large Deviations, and Statistical Mechanics*. Springer; 1st Edition, 1985.

[12] 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.

[13] Project Gutenberg – Project Gutenberg Literary Archive Foundation, 2013.

[14] 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.

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

[16] M. Mitzenmacher. Compressed Bloom filters. In *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC '01)*, pages 144–150. ACM, 2001.

[17] B. Preneel and P. C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In *Advances in Cryptology (CRYPTO '95)*, volume 963, pages 1–14, 1995.

[18] J. K. Resch and J. S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2011.

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

[20] Edward Snowden — Wikipedia, The Free Encyclopedia, 2013. [Online; accessed 26-Sep-2013].

[21] 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.

[22] M. S. Stoler. *Re-Engineering the Enigma Cipher*. PhD thesis, University of Louisville, 2011.

[23] M. W. Storer, K. M. Greenan, and E. L. Miller. Long-term threats to secure archives. In *Proceedings of the 2006 ACM Workshop on Storage Security and Survivability*, Alexandria, VA, Oct. 2006.

[24] 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.

[25] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

[26] 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.

[27] Q. Wang and N. Borisov. Octopus: A secure and anonymous DHT lookup. In *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS '12)*, June 2012.

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

[29] Wikipedia, The Free Encyclopedia, 2013. [Online; accessed 26-Sep-2013].

[30] 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.

[31] D. Zage and J. Obert. Utilizing linear subspaces to improve cloud security. In *Dependable Systems and Networks Workshops (DSN-W)*, pages 1–6, June 2012.