Memory Efficient Sanitization of a Deduplicated Storage System

Fabiano C. Botelho Philip Shilane Nitin Garg Windsor Hsu

Backup Recovery Systems Division
EMC Corporation
{fabiano.botelho, philip.shilane}@emc.com
{nitin.garg, windsor.hsu}@emc.com

Abstract

Sanitization is the process of securely erasing sensitive data from a storage system, effectively restoring the system to a state as if the sensitive data had never been stored. Depending on the threat model, sanitization could require erasing all unreferenced blocks. This is particularly challenging in deduplicated storage systems because each piece of data on the physical media could be referred to by multiple namespace objects. For large storage systems, where available memory is a small fraction of storage capacity, standard techniques for tracking data references will not fit in memory, and we discuss multiple sanitization techniques that trade-off I/O and memory requirements. We have three key contributions. First, we provide an understanding of the threat model and what is required to sanitize a deduplicated storage system as compared to a device. Second, we have designed a memory efficient algorithm using perfect hashing that only requires from 2.54 to 2.87 bits per reference (98% savings) while minimizing the amount of I/O. Third, we present a complete sanitization design for EMC Data Domain.

1 Introduction

Deleting data from a storage system is a routine operation. A regular file delete operation makes the file inaccessible via the namespace and frees the underlying data blocks for later reuse, but does not typically erase those blocks. This leaves behind a residual representation of the file that could be recovered. In many systems, merely over-writing the contents of the file first before deleting it will suffice. However, in systems that maintain old histories of objects (via snapshots or log-structured design for example), such a secure delete operation must be implemented with the involvement of the storage system. When disks are repurposed, residual data can often be accessed despite the intentions of the owners [22, 23, 45].

There are several commonly discussed examples of sensitive data being stored on an inappropriate system. A Classified Message Incident (CMI) happens when data at a particular classification level is written to storage not approved for that level of classification. A CMI might occur when a user inadvertently sends an email with "top secret" information to an email system approved for a lower clearance. Another CMI example is that information may be reclassified after it has been stored on a system with a low clearance. When a CMI occurs, the system administrator must take action to restore the system to a state as if the selected data had never been stored, which is how we define sanitization. If a backup takes place before the CMI is rectified, then the backup server must also be sanitized.

Implementing a sanitization process must consider expected threats. Briefly, threats may be as simple as an attacker reading data with root access permissions or as complex as an attacker using laboratory equipment to read the storage media directly. Sanitizing for more complex threats will likely require greater costs either in terms of memory, I/O, or even hardware costs. Guidelines for threats and appropriate sanitization levels have been published by several government agencies [28, 44], which require sanitization when purchasing storage. We discuss threat models in more detail in Section 2.

Sanitizing a storage system has different problems to address than sanitizing a single device such as a hard drive that might be erased with a pattern of overwrites [25]. For an in-place storage system, sanitizing an object (file, record, etc.) consists of following meta data references to the physical location within the storage system, overwriting the values one or more times, and erasing the meta data as well as other locations that have become unreferenced [41, 49]. Our storage system (and many others) is a log-structured file system [37] with large units of writes, which does not support in-place erasure of sub-units. In-

stead, we have to copy forward live data and then erase an earlier region.

A new complexity for sanitization is the growing popularity of deduplication. Deduplication reduces storage requirements by replacing redundant data with references to a unique copy. Data may be referenced by multiple objects, including live and dead (to be sanitized) objects. For these reasons, sanitization should be implemented within the storage system and not solely at a lower level such as the device. After all of the improperly stored data are deleted, our sanitization algorithm is manually started by a storage administrator. Sanitizing individual files is as challenging as sanitizing the entire file system because of the need to track blocks that uniquely belong to the files affected by the CMI. The tracking of references is the main problem to solve to efficiently sanitize a deduplicated storage system.

In the following sections, we present threat models and sanitization requirements (Section 2) and then review deduplicated storage (Section 3). We discuss several alternative techniques to track live data regions for sanitization that make trade-offs in terms of memory requirements, disk I/O, and completeness in Section 4.

For large storage systems, there are multiple orders of magnitude less memory than storage because of cost differences. Also, it is common for deduplicated storage to work with relatively small chunks of data so that duplicates can be identified, such as 4-8KiB average-sized chunks [12, 29, 50]. These chunks tend to be identified with secure hash values such as SHA1, which is 160 bits in size, though other hash sizes are possible. For an 80TiB storage system with 8KiB chunks and 160-bit hashes, 200GiB of memory is required just for references, which is impractical.

Our approach is based on the observation that we know the set of live references that must be tracked while sanitization runs on a snapshot of the entire file system. Instead of needing a dynamic data structure that can handle insertions, we can optimize with a static data structure for our reference set. Our technique is to analyze the set of references and create a perfect hash that uniquely maps keys to values. Our technique for creating a perfect hash data structure requires from 2.54 to 2.87 bits per reference, a 98% savings relative to the original 160-bit SHA1 hashes we use as reference. We briefly review perfect hashing (Section 4.5) and demonstrate how to incorporate this technique into a complete sanitization algorithm (Section 5). To handle the problem of chunk revival from on-going I/Os, we present a technique for check pointing storage, running sanitization while writes happen in parallel, and updating reference status.

We then present performance measurements for our implementation (Section 6), review related work (Section 7), and present our conclusions (Section 8).

Our research has several major contributions: 1) We discuss sanitization requirements in the context of deduplicated storage. 2) We implemented a memory efficient technique for managing data references using perfect hashes. 3) We present a design for efficiently sanitizing deduplicated storage, while writes happen concurrently, which has been commercially available since 2009 as part of the EMC Data Domain product.

2 Sanitization Requirements

In this section, we discuss general properties necessary for a sanitization process, threat models, and specific complexities for deduplicated storage. A production quality sanitization process for a storage system should satisfy the following properties:

P1: All deleted data are erased.

P2: All live data are available.

P3: Sanitization is efficient.

P4: The storage system is usable while sanitization runs.

Properties 1 and 2 are about correctness and completeness. Users should expect that confidential data have been properly erased and that remaining data in the storage system are valid. Property 3 has several meanings. Sanitization should run as quickly as possible, but it also should use system resources responsibly. Property 3 and 4 also interact because sanitization should not require a storage system to be offline or dramatically slow reads and writes due to I/O or memory overheads.

Ideally, sanitization would surgically remove contaminated regions of storage by overwriting the specified file (shredding a file). Unfortunately, this may not be possible in many cases. If a file has already been deleted, then determining which now-unallocated blocks are affected may be difficult. Log-structured storage systems append writes to a log as compared to in-place and use large write units for efficiency, so overwriting small subunits requires valid data to be relocated. Since sanitization must thoroughly erase all potentially contaminated storage, we advise customers to delete all improperly stored data and then initiate sanitization, instead of running sanitization after each individual file is deleted.

In designing our sanitization technique, we also considered which threat models we can practically address, and we reuse some of the categories presented by Wei et al. [49] and government guidelines [28, 44].

Casual attack: One attempts to gain access to data through interfaces typically exposed by the system such as trying to read a file through NFS. Such attacks may find a version of the contaminated file in an older snapshot even hours or days after the file has been deleted.

Robust keyboard attack: One attempts to gain access through non-typical interfaces such as using root access to read otherwise-hidden data or reading blocks directly from disk when a file system is the expected interface. Deleted files may still be accessible through swap areas or unallocated blocks.

Laboratory attack: One tries to access data using exotic laboratory techniques requiring specific disk format knowledge and specialized hardware such as magnetic force microscopy [14]. Even when storage is overwritten, it may retain magnetic values indicating a previous state.

Most of the techniques available to address these threat models are destructive, and the key challenge is how to preserve clean data. It is especially challenging for deduplicated storage because logical data can be much larger than the post-deduplication data, which we call the deduplication factor. Hence copying the clean data elsewhere is more expensive than processing the post-deduplication data. Alternatively, techniques have been presented that encrypt data and destroy the key to effectively sanitize improperly stored data, but key management becomes a new complexity [2, 36, 40, 43].

We can think of the sanitization problem as the one of sanitizing all "free blocks" of the storage system. For sanitization levels that require a certain number of overwrites be done with different patterns, the in-use blocks also need to be moved and over-written as required. This is because the block may have previously held sensitive data. For a storage system that uses fixed sized blocks and does in place updates, locating all free blocks is relatively straightforward. The main challenge for such systems is that of coordinating the sanitization process with the incoming I/O load. For a storage system that does deduplication, because blocks can be shared by multiple files, determining all free blocks is in itself non-trivial. Ongoing I/Os during sanitization that "revive" a dead block complicate the coordination required. If the system uses variable sized blocks (often called chunks) as units of storage and a log-structured design, the problem is further complicated. The more sophisticated deduplicating storage systems feature all of the above. We present our algorithm for sanitizing a deduplicating storage system that is practical for even the largest systems.

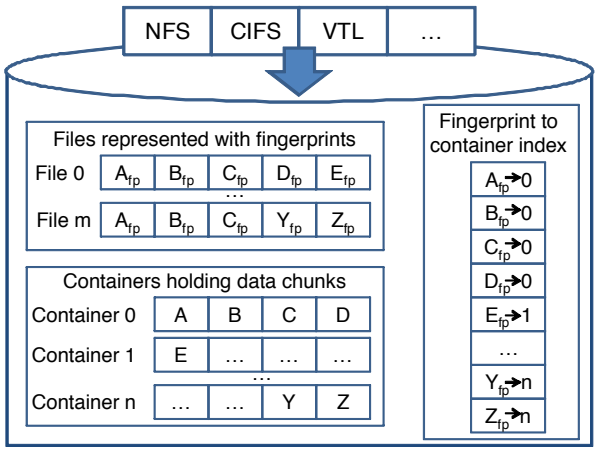

Figure 1: The simplified architecture of a deduplicated storage system includes application visible files that are internally represented by a list of fingerprints, containers holding chunks of data, and an index mapping from fingerprint to container.

3 Deduplicated Storage

Deduplication has become a widespread technique to reduce storage requirements by replacing repeated patterns of data with references to a unique instance [39, 47]. While this paper focuses on sanitization, we briefly review the elements of deduplicated storage that affect our design. Our main implementation is a file system, and we use that terminology throughout the paper, though our sanitization technique can be applied to other deduplicated storage systems.

As shown in Figure 1, when a file is written to the storage system, it is partitioned into chunks that are the unit of deduplication. Chunks may be created with a fixed or variable size, and our system uses variable-sized chunks with an average of 8KB. A secure hash value is calculated over each chunk (SHA1), which we refer to as its fingerprint or chunk reference. A file is represented as a list of fingerprints (file recipe) that can be used to reconstruct the file as shown for files `0` and `m`.

To perform deduplication, a fingerprint is compared against a fingerprint index to determine whether it is duplicate or unique. If a fingerprint is a duplicate, then the current chunk does not need to be stored. If a fingerprint is unique, then the chunk is stored. Identifying duplicates leads to overall space savings for files. Unique chunks are further compressed by using GZ-like schemes, grouped together into 4.5MiB containers, and written out as a unit for efficient I/O. The containers in our system are immutable and form the basis of the log-structured layout. The space savings obtained by eliminating duplicates is called deduplication, and we refer to the dedupli-

cation factor of this system as D defined as *input_bytes / post_deduplication_bytes*. Deduplication factors larger than 1 indicate space savings. Savings from GZ compression of chunks is called "local compression"[1]. The combined effect of the deduplication and local compression is variously called total or overall compression or simply, compression.

In this example, File `0` was written to an empty system, so all of its chunks were unique and were written sequentially to containers `0` and `1`. File `m` was written later, and chunks `A` through `C` (represented by fingerprints A_{fp} through C_{fp}) were duplicates. Chunks `Y` and `Z` correspond to modified regions of the file, and those chunks were written to container `n`. Now suppose File `0` is deleted so that chunks `D` and `E` are unreferenced. For sanitization, it is necessary to erase `D` and `E` from storage. Since containers are immutable, we copy forward live chunks `A`, `B`, and `C` to a new container `n+1` (not shown) and overwrite containers `0` and `1` with zeros, random data, or a specified pattern. This takes care of the need to move live chunks and over-write their old, now free, locations.

4 Managing Chunk References

Before running sanitization, a user deletes unwanted files, if any, from the storage system. Remaining files and their referenced chunks are referred to as *live*, and any unreferenced chunks are referred to as *dead*. The main challenge for implementing sanitization within a deduplicated storage system is managing chunk references so that live chunks can be preserved and dead chunks erased while minimizing memory and I/O requirements. In this section, we discuss several alternatives for managing chunk references and end with our memory efficient approach using perfect hash functions. While there may be simple solutions that can handle a small set of dead (or live) chunks, which can be managed in memory, we are focused on techniques that can scale with the capacity of our storage systems.

4.1 Copy to a Clean System

The first technique to consider is to copy all of the live files from the system being sanitized (call it S_0) to an empty storage system S_1, erase S_0, and potentially copy back the files. This technique has the advantage that it is simple to implement, but the cost of a second storage system is substantial. Copying the live files involves walking the namespace and reading all necessary containers, which could involve an average of D reads of each container if the system has a deduplication factor of D.

A second complete storage system needs to be available and preferable co-located to minimize network replication overhead. This technique is not without its merits: It requires minimal support for sanitization from the storage system, and it is the only viable solution when the level of security needed requires that the original system S_0 be Degaussed [1, 34] or physically destroyed.

Instead, we focus on techniques that involve erasing data by overwriting [28, 44] and that can be implemented within stand-alone storage system S_0.

4.2 Reference Index

A standard deletion technique is to keep track of reference counts for every chunk. When a unique chunk is written, a record is allocated with a count of 1. For every duplicate written, the count is increased, and for every deletion, the count is decremented. Such a reference count requires a unique reference for every chunk such as a fingerprint along with a counter value. Maintaining correct reference counts is challenging due to complex storage failure cases, and live reference counts are not preferred [24].

A simpler approach is to enumerate the live files during sanitization and record the chunk references at that time. A count is not necessary, since even a single reference to a chunk indicates it should remain alive. Assuming a 160-bit fingerprint for each chunk, the index size is shown in Figure 2 as the number of chunks increases on the horizontal axis. Both the horizontal and vertical axes are log scale. When there are 1 billion chunks, the index is 20 GiB, and a full reference count would require somewhat more space. Our storage systems have tens of billions of chunks and continue to increase in size, so the memory requirements for counts or a fingerprint index are impractical.

Since a full index will not fit in memory, it has to be written to storage. This might be acceptable if the index could be processed sequentially, but fingerprints are inherently random. Inserting index entries requires random I/O and would become the bottleneck of sanitization, so we did not pursue this solution.

Partitioned Reference Index

When an index is too large to fit in memory, a common solution is to partition the index into units that do fit and process each unit. For our system, we would divide containers into units and load the references for a unit of containers into memory. We would then walk the namespace

[1] local compression factor = uncompressed size/compressed size, larger values indicate greater compression.

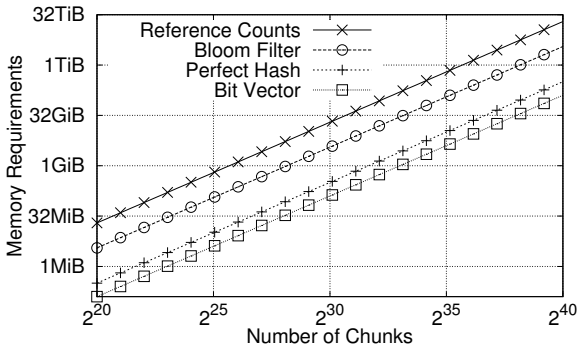

Figure 2: Memory requirements grow linearly with the number of chunks. Perfect hashes uses almost two orders of magnitude less memory than reference counts and only slightly more than a bit vector alternative that has higher I/O overheads.

and determine which references are still alive in the currently processed unit. For containers with a dead chunk, the live chunks are copied forward, and the container is erased.

While this technique allows us to size units to fit our memory requirements, the downside is the amount of I/O involved. Each container is scanned only once but the namespace has to be enumerated once for each partition unit. Depending on the ratio of available memory to storage capacity, this could require dozens of enumerations through the namespace of live files.

4.3 Bloom Filter

While creating a full index of live references requires too much memory, there is a well established index-approximation technique using a Bloom filter [6]. We provide a brief summary of Bloom filters here, but direct the reader to more thorough descriptions [6, 10]. A Bloom filter uses multiple hash functions to map from a key (e.g. our fingerprint) to multiple bit positions within a bit vector. When a key is inserted, all of the mapped bits are set to 1. To check whether a key is contained within a Bloom filter, the corresponding bits are checked, and the response is based on whether all of those bits have value 1. Alternatives to traditional Bloom filters have optimized for memory accesses and flash memory characteristics [4].

Unfortunately, multiple keys can map to the same bit positions, which can lead to false positives when checking the existence of a key. Based on the size of the Bloom filter, number of hash functions, and number of keys to insert, the false positive rate can be set at any arbitrary value, but false positives cannot be eliminated. Figure 2 shows how much memory would be required for a Bloom filter with a false positive rate of $1 : 1,000,000$ for the given number of chunks, which is an order of magnitude memory savings relative to reference counts.

For sanitization with a Bloom filter, we would start by enumerating the live files and inserting the fingerprints into the Bloom filter. Then, while processing containers, only those chunks that have their fingerprints in the Bloom filter are considered to be alive. Because of false positives, though, a small number of dead chunks may incorrectly be considered alive and hence, a Bloom filter does not guarantee sanitization.

4.4 Bit Vector

While the false positive rate of a Bloom filter makes it an impractical solution, it does suggest an alternative. We could allocate a bit vector that is indexed by container number and offset within a container. This only requires a single bit per reference (Figure 2) and is likely the most memory efficient solution. Although not incorporated into the figure, there are small, extra memory overheads to track the number of chunks in a container to enable indexing from container ID to offset in the bit vector.

Unfortunately, the bottleneck has now moved to the construction of the bit-vector. While other solutions only required enumerating references in live files shown in Figure 1, for a bit vector, it is also necessary to check the fingerprint-to-container index, load a container's meta data region, and determine the offset for each reference. Also, since there could be gaps in the container ID range, additional temporary memory is required to quickly map container id to starting offset in the bit-vector. We may achieve some locality in a cache since containers were generated in the order of writes, as noted in previous deduplication papers [5, 24, 29, 50]. Even with potential container reuse, though, we will likely have to load each container D times on average, where D is the deduplication factor. If the system allows concurrent writes during sanitization, then a cache will undergo churn and potentially evict needed container meta data that has to be reloaded. We have not pursued this solution because of I/O overheads.

4.5 Perfect Hash Vector

To motivate our solution, we note that the preceding discussion raises two important problems that we would like to solve. First, we need a compact representation of a set of fingerprints that provides an exact answer for whether a given fingerprint exists in the set or not. Second, this

data structure must be constructed efficiently. This is an instance of the *membership problem*[8, 15]. More specifically, the instance we are going to focus on is a static version of the membership problem where the key space is known beforehand. There will be no dynamic insertion or deletion of keys. Hence the corresponding key for a membership query will always return a value, which usually has a few bits in it. In our context fingerprints identifying the chunks are the keys. This is the perfect scenario to leverage perfect hashing for a static key set [7, 8]. Briefly, perfect hashing is a known technique for minimizing the number of bits needed to represent a mapping from a fixed set of keys to a value.

We are going to use a *Perfect Hash Vector Data Structure*, or simply PH_{vec}, to compactly represent a static set of fingerprints. We denote $[m] = \{0, 1, \ldots, m-1\}$. The data structure has two components: (i) a *Perfect Hash Function* (PHF) or collision free hash function $ph : S \to [m]$ that is specifically computed for an input fingerprint set S of size n and maps S into m buckets, where $m = cn$ for $c > 1$; (ii) a bit vector indexed by the perfect hash function. We denote $|PH_{\text{vec}}|$ as the size in bits of the data structure and $|ph|$ as the number of bits to represent the perfect hash function. Hence

$$|PH_{\text{vec}}| = |ph| + m \text{ bits}. \quad (1)$$

We have tailored a previous algorithm [3] to create a PH_{vec} structure with a payload that ranges from 2.54 to 2.87 bits per fingerprint. Besides the compact representation, one also has to consider two other metrics while choosing a perfect hashing algorithm: (i) *lookup cost*; and (ii) *generation cost*. The lookup cost is usually dominated by the number of random memory accesses needed for each lookup and the generation cost should be a linear function on the number of fingerprints. Hereafter we present an experimental evaluation of those metrics.

Figure 2 illustrates the benefit of using the PH_{vec} structure. By bringing the payload per fingerprint down to less than 3 bits we can actually afford putting memory aside to fit all the fingerprints in our systems. The amount of memory will depend on the capacity of the system since the total number of fingerprints also depends on that. While perfect hashing requires more memory than a bit vector solution, we have reduced I/O to a small number of sequential scans, which is discussed in Section 5.

Perfect Hash Algorithm

The algorithm is based on the hash, displace and compress approach [3]. The perfect hash function data structure consists of two levels. A "first level hash function" g maps S into $[r]$, and thus splits S into r "buckets":

$$B_i = \{x \in S | g(x) = i\}, 0 \le i < r.$$

We let $r = n/\lambda$ where $\lambda \ge 1$.

For each bucket i there is a second level hash function mapping to bins within a range of size m, i.e., $h_i : S \to [m]$:

$$h_i(x) = (f_1(x) + d_0 f_2(x) + d_1) \mod m,$$

where $f_1 : S \to [m]$ and $f_2 : S \to [m]$ as well as function g are assumed to be fully random hash functions. The resulting PHF $ph : S \to [m]$ has the following form:

$$ph(x) = h_{g(x)}(x).$$

Function g will map each key to one of the r buckets. Then, for each bucket B_i, we will assign a pair of displacements (d_0, d_1) so that each key $x \in B_i$ is placed in an empty bin given by $h_i(x)$. For each bucket B_i we will try different pairs (d_0, d_1) until one of them successfully places all keys in B_i. In each trial we use a pair from the sequence $\{(0,0), (0,1), ..., (0, m-1), (1,0), (1,1), ..., (1, m-1), ..., (m-1, m-1)\}$. Instead of storing a pair (d_0, d_1) for each bucket B_i, we store the index of the first pair in that sequence that successfully places all keys in B_i, i.e., $d(i)$. The data structure only has to store the sequence $\{d(i) | 0 \le i < r\}$, and make sure that $d(i)$ can be retrieved in $O(1)$ time. Here is the algorithm to generate the PHFs.

Algorithm 1 ***Hash, displace, and compress****[3]*

(1) Split S into buckets $B_i = \{x \in S | g(x) = i\}, 0 \le i < r$;
(2) Sort buckets B_i in falling order according to size $|B_i|$;
(3) Initialize array `T[0...m-1]` with 0's;
(4) for all $i \in [r]$, in the order from (2), do
(5) for $\ell = 0, 1, \ldots$ repeat forming $K_i = \{h_i(x) \mid x \in B_i\}$
(6) until $|K_i| = |B_i|$ and $K_i \cap \{j \mid \texttt{T[}j\texttt{]} = 1\} = \emptyset$;
(7) let $d(i) =$ the successful ℓ;
(8) for all $j \in K_i$ let `T[j]` $= 1$;
(9) Compress $(d(i))_{0 \le i < r}$ and retain $O(1)$ access.

We have used two approaches to compress the sequence $(d(i))_{0 \le i < r}$. In the first one, we use the algorithm from Fredriksson et al. [21] to compress the sequence. Then an update or lookup on the PH_{vec} structure requires 4.5 random memory accesses, on average. If one is willing to give up some compression the algorithm from Vigna [46] can be used to generate PHFs that would require less random memory accesses on average. The resulting data structure is referred to as "Compressed PH_{vec}". In the second approach we just tune the parameters r and m so that each $d(i)$ is upper bounded by 2^{10} with high probability [3]. Hence we represent each $d(i)$ using 10 bits

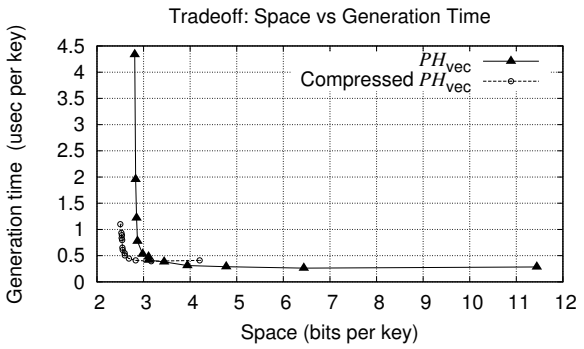

Figure 3: Trade-off between storage space and generation time for the PH_{vec} data structure.

and 2 random memory accesses are required to update or lookup the resulting PH_{vec} structure. The tuning process is discussed later in this section.

We want to minimize Eq. (1) conditioned on being able to efficiently generate the PHFs. There are two parameters to tune: r and m. Belazzougui et al. [3] discussed and analyzed an existent trade-off between the space required for a PHF and the time it takes to generate a PHF. However, there it was not considered that the size of the PHF range also plays a role when the function is being used to index a table storing the values. As the ratio $c = m/n$ increases, the table becomes more sparse and hence more entries are wasted. Conversely, it is easier and faster to generate a PHF. The same thing holds for the number of buckets $r = n/\lambda$.

Figure 3 illustrates the trade-off for both approaches. The experiments were carried out on a 16-core machine running Linux 2.6.23, each Intel Xeon core runs at 2.53GHz and has 8MiB of L2 cache. Here is how we have varied λ and c:

$$\lambda = \{1, 2, 3, 4, 5, 6, 7, 8\} \quad (2)$$

$$c = \{1.37, 1.39, 1.41, 1.43, 1.54, 1.67\}. \quad (3)$$

The set of parameters that has provided a good compromise for the cost to generate a PHF, the space to store it, and the cost to compute it at retrieval time were:

$$m = \lceil 1.43n \rceil \quad (4)$$

$$r = \lceil n/7 \rceil \quad (5)$$

Actually there is a slightly better point for the "Compressed PH_{vec}" but we accept the above values to achieve more compact data structures. Table 1 shows the data structure size for each approach and Table 2 shows the cost to generate the data structure as well as the cost to update it.

Approach	Space (bits/fingerprint)		
	$\|ph\|$	c	Total
Compressed PH_{vec}	1.11	1.43	2.54
PH_{vec}	1.44	1.43	2.87

Table 1: PH_{vec} size in bits per fingerprint.

Approach	Time (microseconds/fingerprint)	
	Generation	Lookup/Update
Compressed PH_{vec}	0.80	0.44
PH_{vec}	0.78	0.26

Table 2: Cost in microseconds per fingerprint to carry out updates or lookups on the PH_{vec} data structure.

5 Sanitization Process

The sanitization process needs to handle the different threat models discussed in Section 2. The National Institute of Standards and Technology [28] and U.S. Department of Defense [44] have both published guidelines that define two levels of security for a sanitization process: (i) the *clearing* level; (ii) the *sanitization* or *purging* level. The clearing level states that a single overwrite of the affected areas is enough to protect against casual attacks and robust keyboard attacks. The purging level states that the devices have to be either Degaussed or destroyed to protect against laboratory attacks.

There are some customers (or data) that only require the clearing level and some that require the purging level. The sanitization process we came up with complies with both levels with one common mechanism. The basic idea is to overwrite data to handle the clearing level. The pattern used for the overwrites can be zeros, random pattern or any user-specified pattern. We have chosen to use zeros. If the purging level is required, we first perform the clearing level which compacts the clean data and allows the clean data to be efficiently migrated to another box by replicating clean post-deduplication data rather than pre-deduplication data.

We first present a version of the sanitization process that requires the file system to be read-only. It is easier to reason about the read-only case because of the static nature of the PH_{vec} data structure described in Section 4.5. All of the fingerprints in the system can be obtained from the on-disk index, and the sanitization process has five phases discussed below and shown in Figure 4.

Algorithm 2 ***Sanitization for a read-only file system***

(1) Merge phase

Setup a marker for the last container to be processed;
Create a consistency point, say CP_0, of the file system.
A consistency point is an in-memory snapshot;
Flush the in-memory fingerprint index buffer and merge it with the on-disk index;

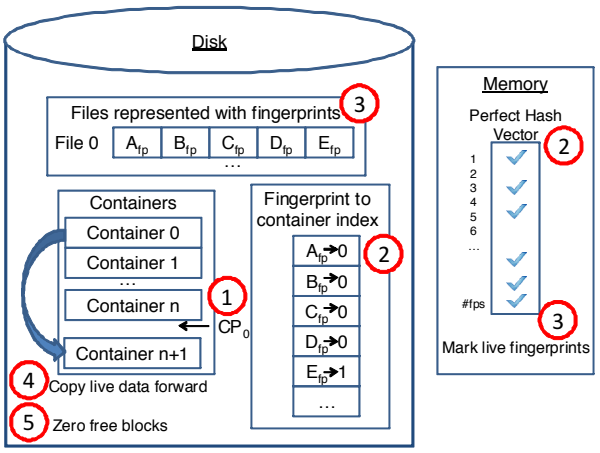

Figure 4: An overview of our five phase sanitization algorithm is shown: 1) Merge, 2) Analysis, 3) Enumeration, 4) Copy, and 5) Zero.

(2) Analysis phase
Traverse the on-disk index for all fingerprints;
Build PH_{vec} for all fingerprints found;
Record the range of containers covered by PH_{vec};
(3) Enumeration phase
Traverse all the files in CP0 (i.e., entire file system);
Mark all fingerprints found as live in PH_{vec};
(4) Copy phase
Select containers with at least one dead chunk;
Copy all live chunks from the selected containers into new containers;
Delete the selected containers;
(5) Zero phase
Zero out the free blocks;
Zero out contaminated areas (NVRAM, Swap, etc.);

Figure 4 illustrates which parts of the system are involved in each phase. The file system keeps a small portion of the fingerprint-to-container index in memory to amortize the I/O cost. The merge phase will then setup a marker that is used as one of the stopping criteria for the copy phase, check point the file system, and flush the in-memory index so it can be merged with the on-disk one. Because the in-memory index has a fixed size, and the merges are done asynchronously, its runtime can be considered a constant.

The analysis phase builds the PH_{vec} structure by scanning the check-pointed on-disk index. Note that the sanitization process operates on a frozen image of the index at a point in time. Hence its runtime depends on the size of the on-disk index, which is no more than 5% of the physical capacity of each system by design.

Building the PH_{vec} structure is complicated because the fingerprint set stored in the index may not fit in internal memory. Hence we need to partition it into smaller buckets and then create a PH_{vec} structure for each bucket. We re-map the fingerprints into variable-sized buckets based on the fingerprint prefix so that buckets will have $16,384$ fingerprints on average. Note that we need to have an offset table that marks the beginning of each bucket in memory so we can have direct access to its PH_{vec} structure. We have chosen $16,384$ as the average size because experiments have shown that it delivers the best trade-off between space overhead for the offset table and building performance for each PH_{vec} structure. We have tried $8K$, $32K$, and $64K$ as well and found either larger memory usage or longer processing time.

The enumeration phase traverses all the files and marks their fingerprints as alive in the PH_{vec} structure. Hence its runtime depends on the logical size of the system.

The copy phase reads the meta data section of each container and, for each fingerprint found, it lookups up the PH_{vec} structure to check its liveness. If a container has at least one dead chunk, it is selected to be copied, live chunks are copied to a new container, and the previous container is marked for deletion. Hence the copy phase runtime is dominated by the I/Os to read the meta data section of all containers and copy the selected ones.

The zero phase is also I/O intensive since it zeroes out (overwrites with the zero pattern) all the free blocks (containers) as well as other potentially contaminated areas: NVRAM, swap partition, core dumps and etc. The zeroing phase could be overlapped with the copy phase, but we implemented separate phases for clarity.

Enabling Read-Write Mode

Removing the read-only restriction breaks the perfect knowledge we had for the key space. In order to leverage the compactness of perfect hashing we need a technique to freeze the key space. Not only that, due to deduplication, an incoming chunk may revive a dead but not yet erased copy of that chunk after enumeration is done. Hence, we need to capture the resurrected chunk in the PH_{vec} structure so as to not corrupt the file system.

We need to change the enumeration phase in order to achieve read-write mode. There are two problems to be addressed: (i) How do we update the PH_{vec} structure for the incoming fingerprints after the analysis phase is done? (ii) How do we update the PH_{vec} structure for fingerprints that came in after CP_0 has been taken but before the PH_{vec} structure was constructed in the analysis phase?

To address the first problem, in the beginning of the enumeration phase, we set up a "notify mechanism". For every incoming chunk that is deduplicated, we notify san-

itization with the pair $(\text{fingerprint}, \text{containerID})$. The sanitization process is not affected by new fingerprints, because at the beginning of the merge phase, the head of the log-structured container set is snapshoted so the current cycle of the sanitization process will not touch any container after that marker: any new data written during current sanitization that itself becomes eligible for sanitization has to be dealt with by the next sanitization operation. By using the notified containerID we are able to check whether the notified fingerprint belongs to the key space used to build the PH_{vec} structure for each bucket. If it does, we can safely record that the notified fingerprint is alive.

To address the second problem, we take a second consistency point, namely CP_1, of the file system after the PH_{vec} is constructed and the notify mechanism is set up. Note that the sanitization process is operating on two in-memory snapshots (CP_0 and CP_1) and the new writes will not modify them. We have the ability of "diffing" CP_0 and CP_1 to find the modified files in CP_1 relative to CP_0. The main issue here is that there is no guarantee that all the fingerprints coming from the modified files belong to the key space used to build the PH_{vec}, whereas that is guaranteed for all the fingerprints coming from files in CP_0. Let F_0 and F_1 be the set of fingerprints coming from files in CP_0 and from the modified files in CP_1, respectively. Hence, for each fingerprint $f \in F_1$ we need to find out the container storing f before we can update the PH_{vec} that f maps to. That is an expensive operation that requires on-disk index lookups. That is why it is crucial to make F_1 as small as possible. Note that this problem does not exist for the fingerprints in F_0.

Then, our enumeration is actually done in two steps. The first one will traverse all the modified files in CP_1 and carry out on-disk index lookups for all the fingerprints in F_1 before updating the corresponding PH_{vec} structure. The second step will traverse all the files in CP_0 and update the corresponding PH_{vec} structure for all fingerprints in F_0. We will show how data ingestion affects the sanitization process in Section 6.3.

Too Many Fingerprints

The memory sizing for the sanitization process depends on the capacity of the system as well as on the local compression factor we get on the stored data. Our basic algorithm is sized to support a local compression factor up to $4X$. We have never seen a customer case with a higher local compression factor. However, it is possible, and our product must handle that case. We do so by modifying our sanitization process to run from merge phase to copy phase in a loop. Each iteration processes a partition of fingerprints in the index. The fingerprint-to-container index can be thought of as a hash table with a fixed number of buckets. The partition is made on the bucket boundaries. The process eventually finishes because there is a fixed number of buckets and each iteration process a subset of the index buckets (usually, no more than 2 iterations are required). While copying the containers, if a fingerprint maps outside the range of index buckets covered by the current iteration, it is considered alive in that iteration. If the fingerprint identifies a dead chunk, it will be deleted on the following iterations.

6 Performance

In this section we discuss sanitization performance as well as the factors that affect each phase. Our experiments have been performed on a system with six RAID 6 groups attached to it, each using 2 TiB drives for a total of 129.4 TiB of physical usable capacity. Our system has 72 GiB of memory and a 16-core Intel Xeon processor where each runs at 2.53 GHz and has an 8MiB cache. The results are reported with a confidence level of 95%.

For all of our experiments, we use a synthetic backup data set to fill our storage systems. Our tool for generating synthetic data creates random data for the first generation and then modifies each successive generation with deletions, additions, and modifications controlled by parameters. This allows us to control the first backup size, number of generations created, local compression factor and deduplication factor. Unfortunately customers that use the sanitization feature do not provide any real data for us to experiment with and do not reveal statistics about their use of sanitization, due to privacy concerns. However, the synthetic tool has been built to mimic backup workload by leveraging our prior knowledge of such workloads [47].

We have considered three scenarios: (i) non-redundant data is stored; (ii) redundant data is stored; (iii) impact of sanitization on data ingest. The next three sections will give further details on them as well as present the results.

6.1 Without Deduplication

In this section we present experimental results when the data has no deduplication but gets a local compression factor of 2.1X. The goal here is to exclude the deduplication impact on the sanitization process since that is investigated separately so we can use the results presented in this section as a baseline.

We have created file systems with the following logical sizes: 4.5 TiB, 9TiB, 18TiB, 36 TiB and 72 TiB. Then

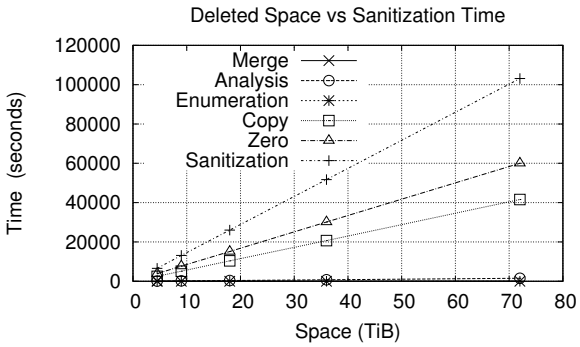

Figure 5: Sanitization time when everything is deleted. We have varied the file system size and the data stored gets a local compression factor of 2.1X. The sanitization throughput is 731 MiB/second.

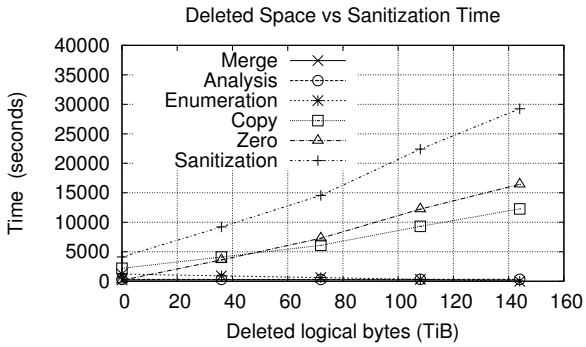

Figure 6: Sanitization time when a portion of the file system is deleted. We have varied how much data is deleted where the data stored gets an overall compression factor of 15.5X (i.e., 7.38X deduplication factor and 2.1X local compression factor). The sanitization throughput is 5.06 GiB/second.

we have deleted the entire file system, run sanitization and measured its performance. We note that it is possible to implement optimizations for the case when the entire file system is deleted but we have not done that because it is not a common use case of a file system.

Figure 5 illustrates the performance for each of the phases as well as for the summation, which corresponds to the total sanitization time (labeled Sanitization). The merge phase always takes about 110 seconds independent of the file system size. The analysis phase runtime grows linearly with the file system size simply because the index also grows linearly, which corresponds to 1.48% of the total Sanitization time. The enumeration phase is always about 6 seconds because there is no logical space to traverse since the entire file system has been deleted. We will show in the next section how the enumeration runtime grows with the logical size of the file system. The runtimes of copy and zero phases grow linearly with the amount of data that has been deleted, and they are the most time-consuming phases. Note that every container that gets copied forward has to be deleted and hence zeroed out. The overall sanitization throughput which is measured by the amount of data we can obliterate per second was 731 MiB/second.

6.2 With Deduplication

In this section we present experimental results when the data gets a deduplication factor of 7.38X and a local compression factor of 2.1X. Ideally the sanitization throughput should have a boost as close to the deduplication factor as possible.

For this experiment we have created a file system with 144 TiB worth of logical data and varied the amount of data deleted. For each experiment, we recreate the entire file system and then delete one of the following values: $\{0, 36, 72, 108, 144\}$ TiB.

Figure 6 illustrates the performance for each of the phases as well as the total sanitization time. The merge phase runtime is again constant at about 230 seconds. In the prior section the merge was faster because the entire file system had been deleted. The analysis phase also remains constant at 320 seconds since the file system size is not varying. We have seen in the prior section that its runtime grows linearly with the size of the file system. The enumeration phase runs at a pace of 122 GiB/second for all the data points. It means that its runtime grows linearly with the logical size of the file system. The copy and zero phase are again the most time-consuming ones but scale linearly with the amount of data that has been deleted. The overall sanitization throughput is 5.06 GiB/second. Note that it went from 731 MiB/second when there is no deduplication to 5.06 GiB/second when the deduplication factor is 7.38X. Sanitization has achieved a throughput boost of 7.1X which is very close to the expected deduplication factor.

6.3 Impact on Ingest

In this section we present experimental results when both sanitization and data ingestion run concurrently. We have throttled the sanitization process at 50%. This means that the CPU cycles and the I/O operations are evenly shared between sanitization and data ingestion, though each task can take more that 50% of a resource if it is available.

We have run two sets of experiments related to ingest. In the first experiment, we repeated the experiment described in Section 6.2, but we continue to ingest data

while sanitization runs. The tool generating data was configured to produce a deduplication factor of 7.38X.

Figure 7 illustrates the performance for each of the phases as well as the total sanitization time. The merge phase runtime is again constant but now at about 640 seconds because there is contention with ingest traffic. The analysis phase also remains constant at 570 seconds because of the contention with the data ingest. We note that although the file system size is changing, to sanitization it remains unchanged since it works on a check-pointed file system and that is why the analysis phase runtime stays constant. Both merge and analysis phases are taking more time due to concurrency with the data ingestion.

The enumeration phase is the most affected due to its two steps when data is being ingested. The first step where it needs to validate the fingerprints in the set F_1 (see Section 5) can take up to 82% of the total enumeration time in this experiment. However, that time is bounded by how much data can be ingested from the time where CP_0 is taken to the time where CP_1 is taken considering the maximum throughput of our system. For a fully populated system that is a small fraction of the logical space the system can store, and hence, it does not have much impact on sanitization duration. Note that enumeration time decreases because the logical size shrinks throughout the experiment.

The copy and zero phase are again the most time-consuming ones but their runtimes grow linearly with the amount of data that has been deleted. There is an interesting observation though. The zero phase runs faster than the copy phase, which is exactly opposite of what we saw in Figure 6. This happens because the copy phase contends for both CPU cycles (to uncompress data and recompress the copied data) and I/Os (to write out the copied data) whereas the zero phase only contends for I/Os since it is just zeroing out the free blocks. If we compare Figure 7 with Figure 6, the copy phase runs at about 70% of its maximum pace, the zero phase runs at about 90% of its maximum pace, and data ingestion runs at about 70% of its maximum throughput. The overall sanitization throughput ranges from 2.97 GiB/second to 4.01 GiB/second in this experiment. This means that at 50% throttle, sanitization reaches 59% to 79% of its maximum throughput of 5.06 GiB/second. This is possible because data ingestion is CPU intensive whereas the sanitization process is I/O intensive, and that is why both data ingestion and sanitization can reach more than 50% of their maximum throughput.

The second set of experiments was to investigate how the sanitization process performs when there is no deduplication so both data ingestion and sanitization are I/O intensive. For that we have created a file system with 72

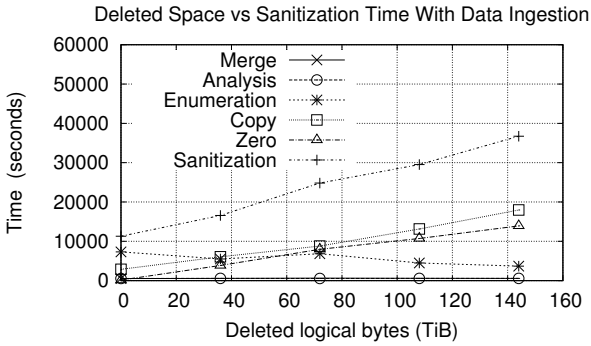

Figure 7: Sanitization time when a portion of the file system is deleted and there is redundant data being ingested by the system. We have varied how much data is deleted where the data stored gets an overall compression factor of 15.5X (i.e., 7.38X deduplication factor and 2.1X local compression factor). The sanitization throughput varied from 2.97 GiB/second to 4.01 GiB/second.

TiB of logical data, then we have deleted the entire file system, started to ingest non-deduplicatable data that can be locally compressed with a factor of 2.1X (the same used in Section 6.1). In contrast to our previous experiment, data ingestion is now both CPU and I/O intensive. In this scenario, sanitization runs at 45% of its maximum throughput, and data can be ingested at 70% of its maximum throughput. Data ingestion is higher than the expected rate of 50% of its peak throughput because the sanitization process does not require much CPU as compared to data ingestion.

7 Related Work

In this section, we briefly review related work that falls into three categories: 1) sanitization of non-deduplicated storage systems, 2) properties of deduplicated storage that affect sanitization, and 3) perfect hashing as a technique to compactly represent a set of references.

The need to sanitize storage to permanently erase user data is a well known problem [27] discussed in multiple governmental guidelines [16, 28, 44]. Investigations of refurbished hard drives show that many contain data a previous owner failed to erase [22, 23, 45]. Several papers [25, 41, 49] and products [19, 30, 34] provide techniques for overwriting storage so that recovering user data is nearly impossible.

Most previous sanitization research discusses erasing the entire storage system, while our customers wish to sanitize individual files. Wei et al. [49] addressed the issue of finding regions of solid state drives that hold previous versions of a file by implementing a full scan of the pages to find physical-to-logical mappings. In our work, determining which storage regions should be erased is

complicated because of the indirect references inherent to deduplicated storage.

Deduplicated storage has become a widespread mechanism to lower storage requirements and costs by replacing repeated data regions with references. Fully surveying deduplication papers [5, 12, 29, 31, 32, 50] and products [18, 35, 42] is beyond the scope of our work, but a selection is provided for reference. All deduplicated storage systems have two features in common that are relevant to sanitization: first is a file recipe that represents a file with a set of references, and second is a set of deduplicated chunks.

Efficiently sanitizing erased files in deduplicated storage requires determining which chunks are currently unreferenced, while minimizing memory and I/O requirements. Previous work for deletion (not sanitization) used reference counts [17, 48], which we believe suffers from large I/O requirements and correctness concerns. Alternatively, Guo et al. [24] proposed a grouped mark-and-sweep, which divided files and containers into sets and tracked which containers were affected by deletions. Since their file recipes had direct storage locations instead of fingerprints, they were unable to copy live chunks forward and erase previous containers unless all chunks in a container were unreferenced. Our technique could be applied to their system to minimize memory requirements for references in each group, though a technique for updating recipes is needed. In general, we believe our sanitization technique using perfect hashes is compatible with other deduplicated storage systems.

An alternative is to encrypt data before it is transmitted to the storage system and then destroy the keys of improperly stored data [2, 36, 43]. There are new complexities for managing the encryption keys, and standard encryption techniques are incompatible with deduplication because encryption effectively randomizes the data to be stored. Storer et al. [40] encrypted each chunk with a key based on its data called convergent encryption, which supports deduplication, while our storage system is compatible with standard storage interfaces.

Studies on perfect hashing started in the early 1980s [20, 33]. For three decades many strong theoretical results have been published [26, 38]. A comprehensive survey till 1997 can be found in [13] and more recent results are surveyed in [8]. However, the gap between theory and practice has been recently bridged [3, 7, 8, 9, 11]. In our products we have tailored the algorithms presented in [3, 8] since each of them trades off lookup cost, generation cost and compactness of the resulting data structure differently. The algorithm in [8] provides the fastest generation (0.45 microseconds/fingerprint), at the cost of three random accesses for the lookup and 3.5 bits per fingerprint of payload for the PH_{vec} structure. The other two algorithms in [3] are discussed in great detail in Section 4.5. The analysis phase of our sanitization process will pick the perfect hashing algorithm dynamically according to number of fingerprints in the system, and the technique that takes 2.87 bits/fingerprint is the default.

8 Conclusions

Sanitization is a critical feature for customers concerned about security, and while sanitizing a device may be relatively straightforward, adding sanitization to a storage system, though complex, is necessary. It is impractical to fully erase a large, expensive storage system when contamination may only affect a small fraction of storage, depending on the threat model. The storage system, not the underlying devices, are able to track which data should be preserved versus erased. Deduplicating storage and log-structured file systems, though, increase the difficulty of determining what data to preserve. We describe these issues and a complete sanitization process that has been commercially available since 2009.

Besides describing our sanitization process, we also explore several technique to manage references for deduplicated storage. We found the best trade-offs using perfect hashes, which minimize memory and I/O requirements. Perfect hashes allow us to represent billions of chunks in memory because only a few bits are required per reference. Using perfect hashes requires a static fingerprint space, which conflicts with our desire to support host writes during sanitization, so we developed a checkpoint-and-update technique that satisfies both requirements.

Our analysis of our sanitization implementation shows nearly linear performance as storage grows, with effective throughput multiplying with the deduplication factor. Most of the processing is devoted to copying live data forward from contaminated containers and zeroing unused regions. We also found the impact on write performance to be acceptable for concurrent host writes during sanitization.

Lastly we want to remark that our sanitization process without the zero phase is also a process to reclaim dead space in the file system. However a process that is meant to reclaim dead space may optimize for performance rather than reclaiming every dead chunk in the file system.

Acknowledgements

We would like to thank our shepherd Pin Zhou and our reviewers for their feedback, Grant Wallace for suggesting the bit vector alternative technique, and the many EMC engineers who continue to improve and support sanitization.

References

[1] A guide to understanding data remanence in automated information systems. http://www.cerberussystems.com/INFOSEC/stds/ncsctg25.htm, May 2012.

[2] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[3] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th Annual European Symposium on Algorithms*, ESA'09, pages 682–693, 2009.

[4] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *Proceedings of the 38th International Conference on Very Large Data Bases*, 2012.

[5] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2009.

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

[7] F. C. Botelho, A. Lacerda, G. V. Menezes, and N. Ziviani. Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences*, 181(13):2608–2625, 2011.

[8] F. C. Botelho, R. Pagh, and N. Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, June 2012. http://dx.doi.org/10.1016/j.is.2012.06.002.

[9] F. C. Botelho, N. C. Wormald, and N. Ziviani. Cores of random r-partite hypergraphs. *Information Processing Letters*, 112(8-9):314–319, 2012.

[10] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[11] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 30–39, 2004.

[12] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[13] Z. Czech, G. Havas, and B. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[14] Z. Deng, E. Yenilmez, J. Leu, J. Hoffman, E. Straver, H. Dai, and K. Moler. Metal-coated carbon nanotube tips for magnetic force microscopy. 85(25), 2004.

[15] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I*, ICALP '08, pages 385–396. Springer-Verlag, 2008.

[16] D. S. Directorate. Australian government information and communications technology security manual. 2006.

[17] C. Dubnicki, L. G. L. Heldt, M. Kaczmarczyk, P. S. Wojciech Kilian, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, 2009.

[18] EMC Corporation. Data Domain. http://www.emc.com/backup-and-recovery/data-domain/data-domain.htm/, 2012.

[19] Eraser. http://eraser.heidi.ie/.

[20] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hashing functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.

[21] K. Fredriksson and F. Nikitin. Simple compression code supporting random access and fast string matching. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 203–216. Springer-Verlag, 2007.

[22] S. Garfinkel. *Security and Usability: Designing Secure Systems That People Can Use*, chapter 15: Sanitization and Usability, pages 293–318. 2005.

[23] S. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *IEEE Security & Privacy*, 1(1):17–27.

[24] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[25] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th conference on USENIX Security Symposium, Fo-*

cusing on Cryptography, July 1996.

[26] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, pages 317–326. Springer LNCS vol. 2010, 2001.

[27] G. Hughes and T. Coughlin. Tutorial on disk drive sanitization. `https://cmrr.ucsd.edu/people/Hughes/DataSanitizationTutorial.pdf`.

[28] R. Kissel, M. Scholl, S. Skolochenko, and X. Li. *Guidelines for Media Sanitization: Special Publication 800-88, Recommendations of the National Institute of Standards and Technology, Computer Security Division*, September 2006.

[29] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 111–123, 2009.

[30] LSoft Technologies Inc. Active@KillDisk. `http://www.killdisk.com/`.

[31] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 1–10, 1994.

[32] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of ACM/IFIP/USENIX Middleware Conference*, 2008.

[33] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.

[34] National Security Agency Central Security Service. *Evaluated Products List - Degausser*, 2012.

[35] NetApp. NetApp ONTAP. `http://www.netapp.com/us/products/platform-os/dedupe.html`, 2012.

[36] R. Perlman. File system design with assured delete. In *Network and Distributed System Security Symposium*, 2007.

[37] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, 1991. Published as Operating Systems Review.

[38] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.

[39] K. Srinivasan, T. Bisson, and G. G. K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[40] M. Storer, K. Greenan, D. Long, and E. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10. ACM, 2008.

[41] S. Swanson and M. Wei. Safe: Fast, verifiable sanitization for ssds. `http://nvsl.ucsd.edu/sanitize/`, 2010.

[42] Symantec. Symantec NetBackup PureDisk. `http://www.symantec.com/netbackup-puredisk`, 2012.

[43] Y. Tang, P. Lee, J. Lui, and R. Perlman. Fade: Secure overlay cloud storage with file assured deletion. *Security and Privacy in Communication Networks*, pages 380–397, 2010.

[44] US Department of Defense National Industrial Security Program. *U.S. National Industrial Security Program Operating Manual (DoD 5220.22-M)*, 2006.

[45] C. Valli and A. Jones. A UK and Australian Study of Hard Disk Disposal. In *3rd Australian Computer, Information and Network Forensics Conference*, 2005.

[46] S. Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 154–168. Springer-Verlag, 2008.

[47] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[48] J. Wei, H. Jiang, K. Zhou, and D. Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[49] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, Feb 2011.

[50] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 269–282, February 2008.