# Secure Data Deletion from Persistent Media

Joel Reardon
ETH Zurich, Switzerland
reardonj@inf.ethz.ch

Hubert Ritzdorf
ETH Zurich, Switzerland
rihubert@inf.ethz.ch

David Basin
ETH Zurich, Switzerland
basin@inf.ethz.ch

Srdjan Capkun
ETH Zurich, Switzerland
srdjan.capkun@inf.ethz.ch

## ABSTRACT

Secure deletion is the task of deleting data irrecoverably from a physical medium. In this work, we present a general approach to the design and analysis of secure deletion for persistent storage that relies on encryption and key wrapping. We define a key disclosure graph that models the adversarial knowledge of the history of key generation and wrapping. We introduce a generic update function and prove that it achieves secure deletion of data against a coercive attacker; instances of the update function implement the update behaviour of all arborescent data structures including B-Trees, extendible hash tables, linked lists, and others. We implement a B-Tree instance of our solution. Our implementation is at the block-device layer, allowing any block-based file system to be used on top of it. Using different workloads, we find that the storage and communication overhead required for storing and retrieving B-Tree nodes is small and that this therefore constitutes a viable solution for many applications requiring secure deletion from persistent media.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.2 [**Operating Systems**]: Storage Management

## Keywords

Secure deletion; privacy; persistent storage; B-Tree

## 1. INTRODUCTION

Secure data deletion is the task of deleting data irrecoverably from a physical medium. Persistent media, however, are not amenable to secure deletion. Such media include off-line tape archives, storage media under adversarial control, or media that leave analog remnants [11]. Encryption is a well-known technique to make data irrecoverable to those unauthorized to access it. However, long-term encryption keys are vulnerable to disclosure by coercive adversaries [3].

Consequently, encrypted data is only securely deleted when the *user* is unable to recover it [17]. To securely delete data, it is therefore necessary to securely delete the key, a technique first used by Boneh and Lipton to securely delete data written onto magnetic tape for off-line archiving [2].

Di Crescenzo et al. [6] first explicitly considered secure deletion on a storage medium consisting of two parts: a large *persistent medium* (such as a tape archive) and a small *securely-deleting medium* (such as local storage). In the related work section we describe a variety of other schemes that also securely delete from such mixed media. While the specific media targeted by these solutions vary radically, they all share the common premise that the securely deleting storage is orders of magnitude *smaller* than the persistent storage. Were this not the case, the user could eschew the use of persistent storage altogether. Consequently, data is stored on the persistent storage with the encryption keys required to access it stored on the securely-deleting medium.

Storing all the data keys on the securely-deleting medium is not always possible. In order to efficiently delete one data item while retaining all others, the system must provide an appropriate *deletion granularity* [17]. A high deletion granularity implies storing many keys—a different key for each data unit. In the case of tape archives, the number of keys required to achieve the appropriate deletion granularity may easily overwhelm the capacity of the securely-deleting medium. For instance, the securely-deleting medium may be an expensive trusted platform module for added security, or a portable smartcard so that a user can easily access their data anywhere; in both cases we can expect to store only a limited amount of data on the securely-deleting medium. In Boneh and Lipton's approach, all data keys are encrypted with a single master key; while storing the master key requires a fixed amount of data on the securely-deleting medium, the deletion operation involves generating a new master key and re-encrypting all data that was not deleted.

To overcome the limitations of Boneh and Lipton's approach, Di Crescenzo et al. designed a tree-based approach that also requires storing a fixed amount of data on the securely-deleting medium but can delete with logarithmic cost. Their approach arranges the persistent storage into blocks that are indexed by a static binary tree: the tree always retains its size and shape; only the values associated with the nodes can change. Both the data and the tree nodes are stored (encrypted) on the persistent medium, such that each internal node contains the decryption keys for its children, and the leaf nodes for the data. The root key is then stored on the securely-deleting medium.

This static key structure prevents both adding and removing nodes. To effect such a change, the user must construct a new data structure and re-encrypt all the data into it. An alternative, however, is to employ one of many dynamic data structures, such as a B-Tree [5]. These data structures, ubiquitously deployed in storage systems such as file systems and databases, change their internal structure based on current storage requirements. However, ensuring secure deletion with dynamic structures becomes less straightforward.

In this work, we describe a B-Tree-based dynamic structure that securely deletes data using a combination of a small securely-deleting storage medium and a large persistent storage medium. We prove the security of our solution by proving the security for a broad class of dynamic data structures: those whose underlying structure forms a directed tree (henceforth called an arborescence [21]). This includes self-balancing binary search trees and B-Trees [5], but also linked lists and extendible hash tables [8]. We develop a new approach to reasoning about this problem by modeling adversarial knowledge as a directed graph of keys and verifying the conditions that result in the secure deletion of data. We define a generic shadowing graph mutation that models how the adversary's knowledge grows and prove that after arbitrary sequences of such mutations one can still securely delete data in a simple and straightforward way. We prove that when using such mutations, data is securely deleted against an adversary given full access to the history of the persistent medium as well as access to all data stored on the securely-deleting medium outside of the data's lifetime. This strong adversary subsumes all weaker ones, who may only use coercive attacks once or obtain only recent history of the persistent storage. The generic shadowing mutation can express the update behaviour of any arborescent data structure; in the related work section we illustrate the resulting key disclosure graph's shape for existing approaches. Consequently, when deploying a secure-deletion system that uses an arborescent data structure to index data, the security guarantees proved in this work extend to that data structure, provided that all its update behaviours are instances of our generic shadowing graph mutation.

We implement our B-Tree-based instance of the secure deletion solution and test it in practice. Our implementation offers a virtual block device interface, i.e., it mimics the behaviour of a typical hard drive. This permits any block-based file system to use the device as a virtual medium, and so any medium capable of storing and retrieving data blocks can therefore be used as the persistent storage. We show that our solution achieves secure deletion from persistent media without imposing substantial overhead through increased storage space or communication. We validate this claim by implementing our solution and analyzing its resulting overhead and performance. We examine our design's overhead and B-Tree properties for different caching strategies, block sizes, and file system workloads generated by `filebench` [12]. We show that the caching strategy approximates the theoretical optimal (i.e., Bélády's "clairvoyant" strategy [1]) for many workloads and that the storage and communication costs are typically only a small percentage of the cost to store and retrieve the data itself.

To summarize, our contributions are the following:

- We propose an intuitive model that captures the growth of adversarial knowledge in secure deletion systems.
- We define a generic shadowing graph mutation that adheres to this model and can implement the update behaviour of any arborescent data structure.
- We prove that secure deletion of data is easily accomplished with a single, simple mutation.
- We design a caching B-Tree whose update mechanism is an instance of our generic mutation.
- We analyze different caching strategies and measure the communication and storage overhead of our B-Tree approach for different workloads; we show that caching is quite effective and that the overhead is typically negligible.

The remainder of this paper is organized as follows: Section 2 presents the adversarial model. Section 3 describes an example instantiation of a securely-deleting data structure in the form of a B-Tree. Section 4 presents the graph theoretic details and proves the security of such schemes in general. Section 5 provides implementation details for the B-Tree. Section 6 then experimentally evaluates the B-Tree approach. Finally, Section 7 discusses related work and Section 8 draws conclusions.

## 2. SYSTEM AND ADVERSARIAL MODEL

Our system model consists of a user who stores data on storage media such that the data can be retrieved during the data's lifetime but cannot be retrieved outside its lifetime. A data's *lifetime* is the time inclusively between two events: the data's initial creation and its subsequent secure deletion. We assume that the user divides the data to store into discrete *data units* that share a lifetime. These can be binary data objects, files, or individual blocks of a block-based file system. The user retains a sortable *handle* to recall these objects (e.g., an object name or a block address). These handles may also be stored on the persistent storage (e.g., a master list of all objects, or by using a file system on top of the block-based storage.)

The user has access to two storage media: a small securely-deleting medium and a large persistent medium. We assume that the securely-deleting medium automatically securely deletes any data that is removed and correctly handles analog remnants and other nuances of secure deletion. We also assume that the persistent medium does not securely delete any data; once (encrypted) data is written onto it, the data cannot be removed. In practice, the persistent medium can correspond to a wide range of use cases, such as sending data over a channel with an eavesdropper, giving another entity access to the storage medium through outsourced archiving or remote storage, storing data on media that do not facilitate secure deletion, or simply neglecting to take appropriate measures to ensure secure deletion. We assume that the securely-deleting medium is orders of magnitude smaller than the persistent storage; in particular, the data being stored does not fit onto the securely-deleting medium. It may even be the case that both media are the same physical device—it is only because ensuring secure deletion is sufficiently expensive (e.g., on flash memory) that it must be done at a smaller scale than the stored data.

We consider a computationally-bounded coercive adversary who can compromise both the securely-deleting medium and the persistent medium. The adversary has access to the history of all data previously written onto the persistent medium. The adversary has full knowledge of the algorithms and implementation of the system and both the persistent and securely-deleting media. The adversary, being coercive, may perform multiple compromises of the user's current secret keys (those keys that the user holds at the time of compromise) as well as the contents of both storage media. As the adversary is computationally bounded, it cannot recover the plain-text message from a cipher-text message without the corresponding encryption key. This strong attacker subsumes all weaker ones, who may be able to coercively attack a single time or may be given a limited amount of data from the persistent storage.

Our environment models many different types of persistent storage, which may vary greatly in what kind of data and how much of it is ultimately stored *persistently* and becomes available to the adversary. Since we do not want the security of our solution to rely on the chance that some persistent data was not disclosed to the adversary, we instead assume that all data written onto persistent storage is given to the adversary. This obviates the need for any assumptions about the storage medium's behaviour other than its ability to store and retrieve data. Moreover, while an adversary that coercively attacks the user at all times outside a data's lifetime is hard to conceive in the real world, the additional effort to prove the security against such an adversary is negligibly more than is required to defend against the adversary who coercively attacks only once after some data is deleted. Therefore, in this work we consider a very powerful attacker, but our solution is not significantly more complicated than would be necessary against a much weaker attacker.

For clarity in our presentation, we assume that all keys $k$ have a name $\phi(k) \in \mathbb{Z}^+$, where $\phi$ is an injective one-way function mapping keys to their name. The key's name $\phi(k)$ reveals no information about the key $k$—even to an information-theoretic adversary. For example, the key's name could be the current count of the number of random keys generated by the user. We further assume that the adversary can identify the key used to encrypt data through the use of a *name function*, which maps an encrypted block to the corresponding key's name. Hence, given $E_k(\cdot)$, the adversary can compute a name $\phi(k)$. This permits the adversary to organize blocks by their unknown encryption key and recognize if these keys are later known. We do not concern ourselves with the implementation of such a function, but simply empower the adversary to use it.

Our security goal is to securely delete data, thus preventing the adversary from obtaining it. We want to achieve this goal for all data; though if the adversary coercively attacks the user during the data's lifetime, then the adversary obtains the data, preventing its subsequent secure deletion. Therefore, we aim to securely delete all data that was not compromised during its lifetime. Figure 1 shows example data lifetimes along with an adversarial coercive attack event. Data numbered 1 and 5 are compromised by this event but the remaining data lifetimes (i.e., data numbered 2–4 and 6) are unaffected by it. Hence, our security goal is to prevent the adversary from recovering any of the data items unaffected by the attack.

Note that in our work we also describe the integrity properties of our solution's design and implementation. This is done to consistently describe the actual data format and be-

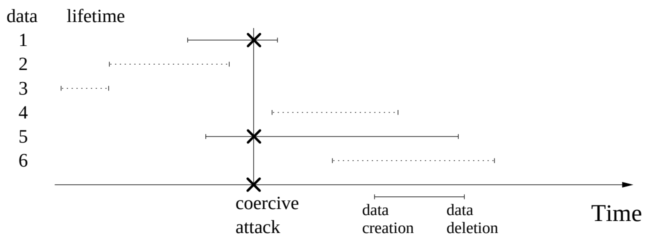


**Figure 1: Secure deletion timeline for different data items. $\times$ marks data compromised by coercive adversarial attacks. Data with lifetimes in dotted lines are not affected by the coercive attack, while data with lifetimes in solid lines are disclosed by the attack.**

haviour of our approach, but the used data integrity scheme is not a novel contribution of this work.

## 3. B-TREE SECURE DELETION

This section details a B-Tree-based design to securely delete data stored on a persistent medium as described in Section 2. It is an example instance from the space of dynamic data structures whose general security we prove in Section 4. The B-Tree implements a securely-deleting key-value map that maps data handles to data units; new pairs can be inserted, existing pairs can be removed, and any stored data unit can be updated. Our map is securely deleting in that data units removed from the map are irrecoverable to an adversary, including old data units that are updated to a new value.

### 3.1 B-Trees

A B-Tree is a self-balancing search tree [5] that implements a key-value map interface. B-Trees are ubiquitously deployed in databases and file systems as they are well-suited to accessing data stored on block devices—devices that impose some non-trivial minimum I/O size.

A B-Tree of order $N$ is a tree where each node has between $\lceil \frac{N}{2} \rceil$ and $N$ child nodes, and every leaf has equal depth [5]. (The root is exceptional as it may have fewer than $\lceil \frac{N}{2} \rceil$ nodes.) The order of a B-Tree node is chosen to fit perfectly into a disk block, which maximizes the benefit of high-latency disk operations that return at minimum a full block of data. B-Trees typically store search keys whose corresponding values are stored elsewhere; leaf nodes store the location where the data can be found. The basic mutating operations `add`, `modify`, and `remove` keys from the tree. Because adding and removing children may violate the balance of children in a node, `rebalance`, `fuse`, and `split` are used to maintain the tree balance property.

#### *B-Tree Storage Operations.*

The `add`, `modify`, and `remove` functions begin with a `lookup` function, which takes a search key and follows a path in the tree from the root node to the leaf node where the search key should be stored. `Add` stores the search key and a reference to the data in the leaf node. `Modify` finds where the data is stored and replaces it with new data; alternatively it can store the new version out-of-place and update the reference. `Remove` removes the reference to data in the leaf node. Both add and remove change the number of children in a leaf node, which can violate the balance property.

### B-Tree Balance Operations.

A B-Tree of order $N$ is balanced when the number of children of each non-root node is inclusively between $\lceil \frac{N}{2} \rceil$ and $N$ and the number of children in each root node is less than or equal to $N$. When there are more or fewer children than these thresholds, the node is overfull or underfull respectively and must be balanced.

Overfull nodes are `split` into two halves and become siblings. This requires an additional index in their parent, which may in turn cause the parent to become overfull. If the root becomes overfull, then a new root is created; this is the only way the height of a B-Tree increases.

Underfull nodes can be either `rebalanced` or `fused` to restore the tree balance property. Rebalancing takes excess children from one of the underfull node's siblings; this causes the parent to reindex the underfull node and its generous sibling and afterwards neither node violates the balance properties. If both the node's siblings have no excess children, then the node is fused with one of its siblings. This means that the sibling is removed and its children are given to the underfull node. This removes one child from their parent, which can cause the parent to become underfull—possibly propagating up to the root. The root node is uniquely allowed to be underfull. If, however, after a fuse operation the root has only one child, then the root is removed and its sole child becomes the new root. This is the only way the height of a B-Tree decreases.

## 3.2 Solution design

We use a B-Tree to organize, access, and securely delete data. We assume that the size of the B-Tree nodes is sufficiently large that the nodes cannot all be stored on the securely-deleting storage—otherwise the client can simply maintain a local list of keys, securely deleting them when the corresponding data should be deleted. Consequently, both data and B-Tree nodes are stored on the persistent storage medium, and they are first encrypted before being stored.

Data blocks are encrypted with a random key. The index for the data block, along with its encryption key, is then stored as a leaf node in the B-Tree. The nodes themselves are encrypted with a random key and stored on the persistent medium. Inner nodes of the B-Tree therefore store the encryption keys required to decrypt their children. The key that decrypts the root node of the B-Tree, however, is never stored on the persistent medium; the root key is only stored on the securely-deleting medium. Only one such key is stored at any time. Old keys are securely deleted and replaced with a new key.

In addition to encryption, each node also stores the *cryptographic digest* (henceforth called *hash*) of its children for integrity in a straightforward application of a Merkle tree [13]. An authentic parent node guarantees the authenticity of its children. The root hash is stored with the key.

To improve efficiency, we keep a cache of nodes available in memory to perform all B-Tree operations, which we call the *skeleton tree*. Figure 2 illustrates an example skeleton tree. The skeleton tree's nodes are loaded lazily from persistent storage and added to the skeleton tree after decryption and verification. To store local changes on the persistent storage, the skeleton tree is periodically *committed*, where all *dirty* nodes (i.e., those that have local modifications) are re-hashed for integrity and re-encrypted with a new key for security, and written onto persistent storage.

To prevent data loss between commit operations, a mechanism must be used for crash safety. Our approach simply keeps a local record of changes on the securely-deleting medium that can be replayed after a crash and is securely deleted once committed. Another option, which does not require using the securely-deleting medium, is DNEFS's approach of writing fresh encryption keys to the persistent storage ahead of time [18]. In particular, a block of fresh keys is directly wrapped with the new root key and stored during the commit operation; these keys are then sequentially used to encrypt data written in the next commit interval.

### Cryptographic Details.

All encrypted data—both the B-Tree's node data and the user's actual data—are encrypted with AES keys in counter mode with a zero IV. All keys are randomly generated using a cryptographically-suitable entropy source. We use each key only once to encrypt data. Therefore, an encryption key's lifetime is the following: it is generated randomly, it is used once to encrypt data, and then it is used arbitrarily many times to decrypt that data until it is securely deleted.

In addition to encryption, we also hash data to ensure its integrity from the persistent storage. We use a cryptographic hash to ensure that even an adversary who directly controls the persistent storage cannot violate the data's integrity. Mykletun et al. [15] propose a variation of Merkle Hash Trees [13] designed specifically for B-Trees. We use a variant of their approach in our scheme: each node is hashed but the hashes of the children are independently stored in their parent (alongside their decryption key). This increases the space required to store each node; however, it allows us to perform B-Tree updates without loading all of a node's siblings from persistent storage.

### Skeleton Tree.

All the B-Tree nodes are stored on the persistent storage. To improve efficiency, however, the actual B-Tree operations are performed on a smaller subset of the B-Tree cached in memory, which is called the *skeleton tree*. The skeleton tree reduces the cost of computing decryption keys for data when the relevant B-Tree nodes are available in memory; this strongly benefits, in particular, sequential data access. It also permits multiple updates to the B-Tree to be batched and committed together, which reduces the total number of B-Tree nodes to update. Finally, it allows the user to control the frequency that the root secret changes on the securely deleting medium; this is useful if updating the securely-deleting medium has a non-trivial cost in latency, wear, or human effort.

Initially, the skeleton tree only stores the root of the B-Tree; other node references are loaded lazily. Figure 2 gives an example of this configuration, where the persistent storage has a stale B-Tree and the skeleton tree reflects some combination of addition, removal, and rebalance operations. When a B-Tree operation requires accessing a node missing from the skeleton tree, the corresponding B-Tree node is read from persistent storage and decrypted. Its integrity is checked by using its hash value stored at the parent; if the check passes, then the missing reference is added to the skeleton tree. This new reference now stores the decryption keys and integrity hashes corresponding to all its (missing) chil-

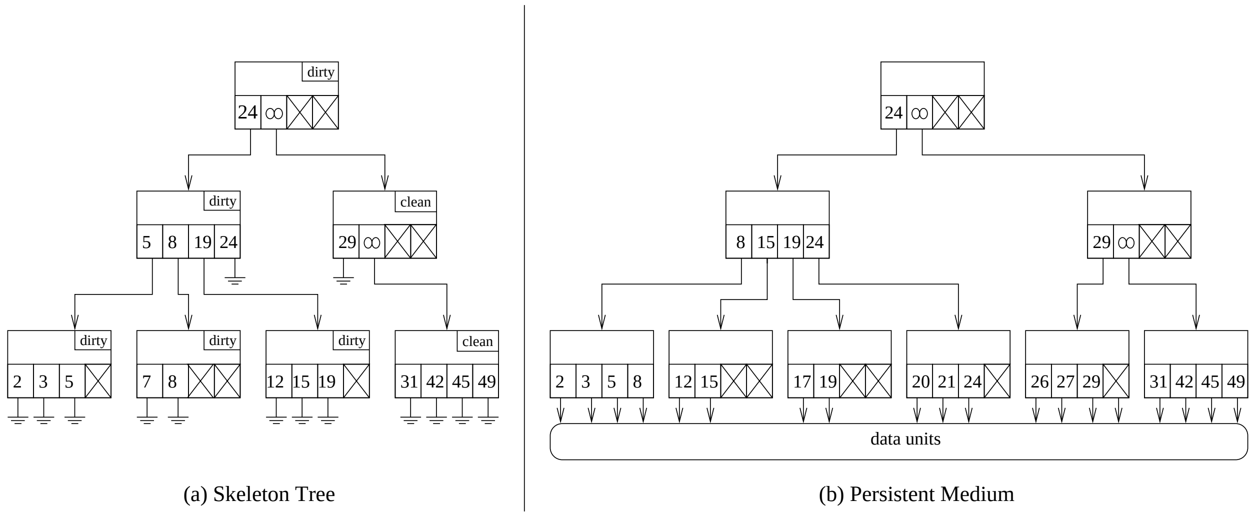


**Figure 2: Example of a B-Tree stored on the persistent medium along with an in-memory skeleton tree. (a) shows the skeleton tree of B-Tree nodes, where node 42 was read and local changes were made: the node 7 was added and the node 17 was deleted, causing a split operation and a fuse operation respectively. (b) shows the persistent medium which stores all the nodes in the tree, some of which are stale. Only the nodes that have been needed are loaded into the skeleton tree.**

dren, allowing the skeleton tree to grow further on request. The size of the skeleton tree is limited: when it reaches its capacity then nodes are evicted from the tree. In Section 6 we present our experimental results with eviction strategies.

All modifications to the B-Tree—e.g., deleting data and rebalancing—are performed on the skeleton tree and periodically committed in batch to persistent storage. A *dirty* marker is kept with the skeleton nodes to indicate which of them have local changes that need committing. Whenever a tree node is mutated—i.e., adding, removing, or modifying a child reference—it is marked as dirty. This includes modifications made due to rebalance operations. B-Tree nodes that are created or deleted—due to splitting or fusing nodes—are also marked as dirty. Finally, dirtiness is propagated up the skeleton tree to the current root.

### *Commitment.*

The B-Tree commit operation writes new versions of all the dirty nodes to persistent storage, thus achieving secure deletion of deleted and overwritten data. Modifications to the B-Tree are first cached and aggregated in the skeleton tree, and then they are simultaneously committed. Therefore, the time that deleted data remains available to the user (and thus the adversary) is based on the frequency that commit operations are performed. Depending on the crash-safety mechanism, an encrypted block of fresh unused encryption keys is written to the persistent storage medium for use in the next commit period. Therefore, the data's lifetime effectively grows in both directions to the nearest commit event; a compromise at one time point in a commit interval is equivalent to a compromise at all time points in that interval. The period between commit events is therefore a trade-off between system performance and deletion latency [17]. Figure 3 builds on Figure 1 by including commit operations.

The commit operation handles two kinds of dirty nodes: *deleted* ones that have been deleted from the B-Tree through

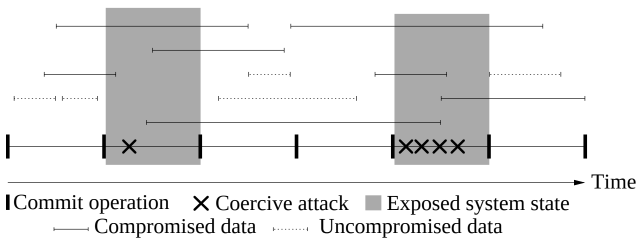


**Figure 3: Secure deletion timeline with commit operations. $\times$ marks coercive attacks. Data with lifetimes in dotted lines are not affected by coercive attacks, while data with lifetimes in solid lines are disclosed by the attacks.**

the fuse operation, and *valid* ones that are still part of the tree. Each valid dirty node is first associated with a fresh randomly-generated encryption key. Because parent nodes store the keys of their children, all parents of dirty valid nodes are updated to store the new keys associated with each child. After this, the sub-tree of valid dirty nodes is traversed in post order to compute each dirty valid node's integrity hash, which is then stored in the parent. The root node's key and integrity hash are stored outside the tree local to the user. The data for each valid dirty node (i.e., the keys, hashes, and search values for its children) is then encrypted with its newly-generated key and stored on persistent storage.

## 4. GRAPH-THEORETIC MODEL OF KEY DISCLOSURE

This section describes the security of a broad class of mutable data structures when used to retrieve and securely delete data stored on persistent storage. It relies heavily on graph theory, which we first briefly review. Afterwards, we present our three theoretical contributions. First, we define a key disclosure graph and show how it models adversarial knowledge. We then prove graph-theoretic conditions under which data is securely deleted against our worst-case adversary. Finally, we define a generic shadowing graph mutation and prove that all instances of it preserve a graph property that simplifies secure deletion.

## 4.1 Graph Theory Background

For completeness, and to commit to a particular nomenclature, we first briefly review the relevant aspects of graph theory. A more detailed treatment can be found elsewhere [21].

### Directed Graphs.

A *directed graph* (henceforth called a *digraph*) is a pair of finite sets $(V, E)$, where $E \subseteq V \times V$. Elements of $V$ are called *vertices* and elements of $E$ are called *edges*. If $G$ is a digraph, then we write $V(G)$ for its vertices and $E(G)$ for its edges.

A digraph's edges are directed. If $(u, v) \in E(G)$, we say the edge goes *from* the *source* $u$ and *to* the *destination* $v$. The edge is called *outgoing* for $u$ and *incoming* for $v$. The *indegree* and *outdegree* of a vertex is the number of all incoming and outgoing edges for that vertex.

### Paths.

A *non-degenerate walk* $W$ of a graph $G$ is a sequence of elements of $E(G)$: $(v_1, u_1), \ldots, (v_n, u_n)$ such that $n \geq 1$ and $\forall i : 1 < i \leq n,\ u_{i-1} = v_i$. The *origin* of $W$ is $v_1$ and the *terminus* is $u_n$. We say $W$ *visits* a vertex $v$ (or equivalently, $v$ is on $W$) if $W$ contains an edge $(v, u)$ or $v$ is the terminus. A *non-degenerate path* $P$ is a non-degenerate walk such that no vertex is visited more than once. Additionally, a graph with $n$ vertices has $n$ *degenerate paths*—zero-length paths that visit no edges and whose origin and terminus are $v \in V(G)$. A *cycle* is a non-degenerate walk $C$ whose origin equals its terminus and all other vertices on the walk are visited once. A directed *acyclic* graph is one with no cycles.

A vertex $v$ is *reachable* from vertex $u$ if there is a directed path from $u$ to $v$. If there is only one such path then we say that $v$ is uniquely reachable from $u$ and use $P^u_v$ to denote this path. The *ancestors* of a vertex $v$, called $\text{anc}_G(v)$, is the largest subset of $V(G)$ such that $v$ is reachable from each element. The *descendants* of a vertex $u$, called $\text{desc}_G(u)$, is the largest subset of $V(G)$ such that each element is reachable from $u$. If $P^u_v$ is a directed path from $u$ to $v$, then $u$ is an *ancestor* of $v$ and $v$ is a *descendant* of $u$. Because of degenerate paths, all vertices are their own ancestors and descendants.

### Subdigraphs.

A *subdigraph* $S$ of a digraph $G$ is a digraph whose vertices are a subset of $G$ and whose edges are a subset of the edges of $G$ with endpoints in $S$. Formally, a subdigraph has vertices $V(S) \subseteq V(G)$ and edges $E(S) \subseteq E(G)|_{V(S)\times V(S)}$. A subdigraph is called *full* if $E(S) = E(G)|_{V(S)\times V(S)}$. A *subdigraph induced by a vertex* $v$, denoted $G_v$, is a full subdigraph whose vertices are $v$ and all vertices reachable from $v$ in $G$. Formally, $V(G_v) = \text{desc}_G(v)$ and $E(G_v) = E(G)|_{V(G_v)\times V(G_v)}$.

### Arborescences and Mangroves.

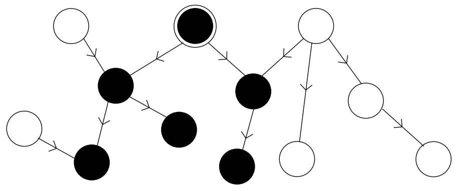


**Figure 4: An example mangrove. Shaded vertices belong to the arborescent subdigraph induced by the circled vertex.**

An *arborescence diverging from a vertex* $r \in V(A)$ (henceforth called arborescence) is a directed acyclic graph $A$ whose edges are all directed away from $r$ and whose underlying graph (i.e., the undirected graph generated by removing the direction of $A$'s edges) is a (graph-theoretic) tree [21]. The vertex $r$ is called the *root* and it is the only vertex in $A$ that has no incoming edges; all other vertices have exactly one incoming edge (Theorem VI.1 [21]). There is no non-degenerate path in $A$ with $r$ as the terminus, and for all other vertices $v \in V(A)$ there is a unique path $P^r_v$ (Theorem VI.8 [21]). To show that a graph $A$ is an arborescence, it is necessary and sufficient to show that $A$ has the following three properties (Theorem VI.26 [21]): (i) $A$ is acyclic (ii) $r$ has indegree 0 (iii) $\forall v \in V(A), v \neq r \Rightarrow v$ has indegree 1.

A directed graph is a *mangrove* if and only if the subdigraph induced by every vertex is an arborescence. This means that, for every pair of vertices, either one is uniquely and unreciprocatedly reachable from the other or neither one is reachable from the other. Observe that an arborescence is also a mangrove, as all its vertices induce arborescences. Figure 4 shows an example mangrove as well as an arborescent subdigraph induced by a vertex.

## 4.2 Key Disclosure Graph

In this section, we characterize the information obtained by the adversary and describe a way to structure it. We begin by limiting the functions the user computes on encryption keys to *wrapping* and *hashing*. Wrapping means that a key $k$ is encrypted with another key $k'$ to create $E_{k'}(k)$. With $k'$ and $E_{k'}(k)$ one can compute $k$, while $E_{k'}(k)$ alone reveals no information about $k$ to a computationally-bounded entity. Hashing means that a key $k$ can be used to compute a one-way digest function $H(k)$ such that $H(k)$ reveals no information about $k$ to a computationally-bounded entity. Furthermore, we require that no plain-text data is ever written onto the persistent medium.

The process of generating keys and using keys to wrap other keys induces a directed graph: nodes correspond to encryption keys and directed edges correspond to the destination key being wrapped by the source key. Knowledge of one key gives access to the data encrypted with it as well as any keys corresponding to its vertex's destinations. Recursively, all keys corresponding to descendants of a vertex are computable when the key corresponding to the ancestor vertex is known. We call this graph the *key disclosure graph*, whose definition follows.

*Definition 1.* Given a set $K$ of encryption keys generated by the user, an injective one-way vertex naming function $\phi : K \rightarrow \mathbb{Z}^+$, and a set of wrapped keys $C$, then the *key disclosure graph* is a directed graph $G$ constructed as follows: $\phi(k) \in V(G) \Leftrightarrow k \in K$ and $(\phi(k), \phi(k')) \in E(G) \Leftrightarrow E_k(k') \in C$.

The user can construct and maintain such a key disclosure graph by adding nodes and edges when performing key generation and wrapping operations respectively. The adversary can also construct this graph using its name function: whenever ciphertext is given to the adversary, the name corresponding to its encryption key is computed and added as a vertex to the graph with the ciphertext stored alongside. The adversary may only learn some parts of the key disclosure graph; we use $G^{adv} \subseteq G$ to represent the subgraph known to the adversary. For instance, the client may not write all the wrapped key values it computes to the persistent storage, or the adversary may not be able to read all data in the persistent storage. In the worst case, however, the adversary gets all wrapped keys and so $G^{adv} = G$; it is this worst case for which we prove our security.

If the adversary later learns an encryption key (e.g., through compromise), then the key's corresponding ciphertext can be decrypted. If the plaintext contains other encryption keys, then the adversary can determine the names of these keys to determine the edges directed away from this vertex. Therefore, the adversary can follow paths in $G^{adv}$ starting from any vertex whose corresponding key it knows, thus deriving unknown keys.

The adversary's ability to follow paths in the key disclosure graph is independent of the age of the nodes and edges. In our scenario and adversarial model, every time data is stored on the persistent medium, the key disclosure graph $G$—and possibly the adversary's key disclosure graph $G^{adv}$—grows. After learning a key, the adversary learns all paths originating from the corresponding vertex in $G^{adv}$. The keys corresponding to vertices descendant to that origin are then known to the adversary along with the data they encrypt. Therefore, the user must perform secure deletion while reasoning about the adversary's key disclosure graph. Moreover, if the user is unaware of the exact value of $G^{adv} \subseteq G$, then they must reason about $G^{adv} = G$.

## 4.3 Secure Deletion

Secure data deletion against a coercive attacker requires that the user who securely deletes data is thereafter unable to recover the deleted data [17]. If the adversary already has an encrypted copy of the data being deleted, then the user must ensure that the corresponding decryption key is securely deleted. The decryption key must be securely deleted even with access to all secret keys managed by the user and all data ever sent to persistent storage. The user must not only securely delete the data's encryption key, but also any encryption key that decrypts any ancestor of the data's corresponding vertex in the adversary's key disclosure graph. This is because a vertex $v$ is reachable from another vertex $u$ in the key disclosure graph if and only if $\phi^{-1}(v)$ is computable from $\phi^{-1}(u)$. Definition 2 now defines secure deletion in terms of paths in the key disclosure graph.

*Definition 2.* Let $G = (V, E)$ be the key disclosure graph for a vertex naming function $\phi$, a set of keys $K$, and a set of ciphertexts $C$, and let $G^{adv} \subseteq G$ be the adversary's subdigraph of the key disclosure graph. Let $R = \{r_1, \ldots, r_n\} \subseteq K$ be the set of keys stored by the user in the securely-deleting medium. Let $D$ be data stored on the persistent medium encrypted with a key $k \in K$. Let $R_{\text{live}} \subseteq R$ be the set of keys stored in the securely-deleting medium at all times when $D$ is alive (i.e., the times between the data's creation and deletion events).

Then $D$ is *securely-deleted against a computationally-bounded coercive adversary* provided that no compromise of the securely-deleting medium occurs when it stores an element of $R_{\text{live}}$ and for all $r \in R \setminus R_{\text{live}}$, there is no path in $G^{adv}$ from $\phi(r)$ to $\phi(k)$.

This definition reflects the following facts: (i) a computationally-bounded adversary cannot recover the data $D$ without the key $k$, (ii) the only way to obtain $k$ is through compromise or through key unwrapping, (iii) an adversary that compromises at all permissible times can only obtain $R \setminus R_{live}$ directly and $\bigcup_{r \in R \setminus R_{live}} \text{desc}(\phi(r))$ through unwrapping, and (iv) $k$ is not within this set.

Observe that this definition requires that no compromise occurs during which time the securely-deleting medium stores an element of $R_{\text{live}}$—the set of keys stored in the secure-deleting storage medium during the lifetime of the data being securely deleted. This is larger than or equal to the data's lifetime, e.g., by extending in both directions to the nearest commit event.

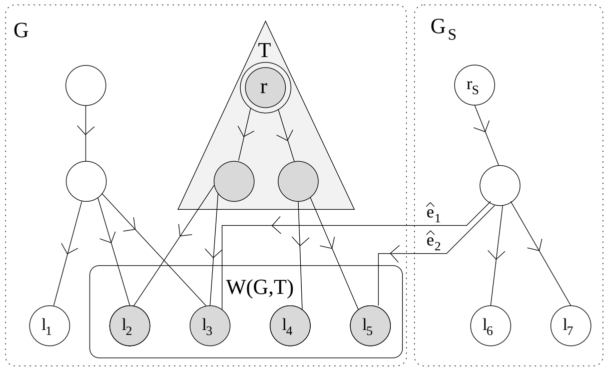
We have shown that to securely delete the data corresponding to a vertex $v$, we must securely delete data corresponding to all ancestors of $v$ that are not already securely deleted. This is burdensome if it requires a full graph traversal, because the adversary's key disclosure graph perpetually grows. We make this efficient by establishing an invariant of the adversary's key disclosure graph: there is at most one path between every pair of vertices (i.e., it is a mangrove). We now define a family of graph mutations that preserves this invariant.

## 4.4 Shadowing Graph Mutations

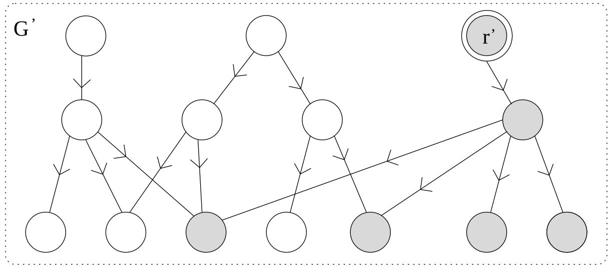
Shadowing is a concept in data structures where updates to elements do not occur in-place. Instead, a new copy of the element is made and references in its parent are updated to reflect this [19]. This results in a new copy of the parent, propagating shadowing to the head of the data structure. We now define a generalized graph mutation, called a *shadowing graph mutation*, and show that if any shadowing graph mutation is applied to a mangrove, then the resulting *mutated graph* is also a mangrove. The mangrove property is therefore maintained throughout all possible histories of shadowing graph mutations.

Mangroves have at most one possible path between every pair of vertices. This simplifies secure deletion of data, as illustrated in Figure 5. Computing the set of all ancestors of a vertex—those vertices that must be also securely deleted—is done by taking the union of the unique paths to that vertex from each of the vertices whose corresponding keys are locally stored by the user. Determining the *unique* path to follow to find data is trivial by overlaying a search-tree data structure (e.g., a B-Tree). Moreover, if the user only stores one local key at any time, taking care to securely delete old keys, then data can be securely deleted by just securely deleting the vertices on a single path in the key disclosure graph.

Figure 5 shows an example mutation, where the old key disclosure graph $G$ is combined with $G_S$ and the edges $\hat{e}_1, \hat{e}_2$ to form the new key disclosure graph $G'$. The new nodes and edges correspond to the user generating new random keys and sending wrapped keys to the adversary, respectively.



(a) mutation parameters



(b) post−mutated graph

**Figure 5: An example shadowing mutation. (a) shows the parameters of a shadowing graph mutation and (b) shows the resulting graph. The premutated graph $G$ is combined with the shadowing graph $G_S$ and connecting edges $\hat{E} = \{\hat{e}_1, \hat{e}_2\}$ to form $G'$. Shaded vertices are the vertices reachable from the circled vertex.**

The node $r$ represents the user's current stored secret key; the shaded nodes are $r$'s descendants—those nodes whose corresponding keys are computable by the user. In the resulting graph $G'$, we see that $r'$ corresponds to the new user secret, resulting in a different set of shaded descendant vertices. In particular, the mutation securely deleted the leaves $l_2$ and $l_4$ while adding new leaves $l_6$ and $l_7$.

To perform the mutation, the user prepares $T$—a graph that contains the vertices to shadow. In the post-mutated graph $G'$, no vertex in $T$ is reachable from any vertex in $G_S$. The only vertices in $G$ that are given a new incoming edge from a vertex in $G_S$ are those in the set $W(G,T)$: vertices outside $T$ but that have an incoming edge from a vertex in $T$. Formally, if $G$ is a mangrove, $r \in V(G)$, and $T$ is an arborescent subdigraph of $G_r$ diverging from $r$, then $W(G,T) = \{v \in V(G) \setminus V(T) | \exists\, x \in V(T) \;.\; (x,v) \in E(G)\}$.

To ensure that $G'$ is a mangrove, we must constrain the edges that connect $G_S$ to $G$. We require that any connecting edge $\hat{e}$ goes from $G_S$ to $W(G,T)$ and that each vertex in $W(G,T)$ receives at most one such incoming edge.

Formally, a tuple $(G, r, G_S, T, \hat{E})$ is a *shadowing graph mutation* if it has the following properties:
— $G$ is a mangrove, called the *pre-mutated graph.*
— $r$ is a vertex of $G$.
— $G_S$ is an arborescence diverging from $r_S$ such that $V(G) \cap V(G_S) = \emptyset$. It is called the *shadow graph.*
— $T$ is a subdigraph of $G_r$ such that $T$ is an arborescence diverging from $r$. It is called the *shadowed graph.*
— $\hat{E}$ is a set of directed edges such that
(i) $\forall (i,j) \in \hat{E} \;.\; i \in V(G_S) \wedge j \in W(G,T)$ and
(ii) $\forall (i,j)(i',j') \in \hat{E} \;.\; i \neq i' \Rightarrow j \neq j'$ (i.e., $\hat{E}$ is injective).

A graph mutation contains the initial graph along with the parameters of the mutation. We assume there exists a function $\mu$ that takes as input a graph mutation $(G, r, G_S, T, \hat{E})$ and outputs the mutated graph $G'$, defined by $V(G') \leftarrow V(G) \cup V(G_S)$ and $E(G') \leftarrow E(G) \cup E(G_S) \cup \hat{E}$. Observe that the sets in the unions are all disjoint. Moreover, every resulting path in $G'$ has one of the following forms: $P$, $P_S$, or $(P_S, \hat{e}, P)$, where $P$ is a path visiting only vertices in $V(G)$, $P_S$ is a path visiting only vertices in $V(G_S)$, and $\hat{e} \in \hat{E}$.

## *Mangrove Preservation.*

To simplify the enumeration of a vertex's ancestors in the key disclosure graph, which must be securely deleted in order to delete that vertex, we require as an invariant that the key disclosure graph is always a mangrove. We establish this by showing that, given a shadowing graph mutation, the mutated graph is always a mangrove. Since the graph with a single vertex is a mangrove, all sequences of shadowing mutations beginning from this mangrove preserve this property.

LEMMA 1. *Let $G$ be a mangrove, $r \in V(G)$, and $T$ an arborescent subdigraph of $G_r$ diverging from $r$. Then $\forall i,j \in W(G,T), i \neq j \Rightarrow desc_G(i) \cap desc_G(j) = \emptyset$.*

PROOF. We prove the contrapositive. Suppose that $v \in \mathrm{desc}_G(i) \cap \mathrm{desc}_G(j)$. Then there exist distinct paths $P_v^i$ and $P_v^j$. Since $i, j \in V(G_r)$, there exist distinct paths $P_i^r$ and $P_j^r$. Consequently, $P_i^r P_v^i$ and $P_j^r P_v^j$ are two paths from $r$ to $v$ in $G_r$. Since $G_r$ is an arborescence, these two paths must be equal and so (without loss of generality) $P_v^r = P_i^r P_j^i P_v^j$ and $P_j^r = P_i^r P_j^i$. However, by definition of $W(G,T)$, all edges except the final one in $P_i^r$ and $P_j^r$ are in $E(T)$. If $P_j^i$ is non-degenerate, then $P_i^r P_j^i \neq P_j^r$ as $P_i^r$ has an edge outside of $T$ followed by more than one edge. Therefore, $P_j^i$ is degenerate and $i = j$, as needed. $\square$

LEMMA 2. *If $(G, r, G_S, T, \hat{E})$ is a valid shadowing mutation and $G' \leftarrow \mu(G, r, G_S, T, \hat{E})$, then $G'$ is acyclic.*

PROOF. Since the mutation is valid, $G$ is a mangrove. Suppose to the contrary that $G'$ has a cycle $C$. By construction of $V(G')$, there are three cases:
(i) All of $C$'s vertices are in $V(G)$. Then $C$ is a cycle in $G$, which contradicts $G$ being a mangrove.
(ii) All of $C$'s vertices are in $V(G_S)$. Then $C$ is a cycle in $G_S$, which contradicts $G_S$ being an arborescence.
(iii) $C$'s vertices are a mixture of vertices from $V(G)$ and $V(G_S)$. Suppose $C$ visits $v \in V(G)$ and $u \in V(G_S)$. Then $C$ can be divided into two paths $C = P_u^v P_v^u$, but no such path $P_u^v$ exists. $\square$

THEOREM 1. *If $(G, r, G_S, T, \hat{E})$ is a valid shadowing mutation and $G' \leftarrow \mu(G, r, G_S, T, \hat{E})$, then $G'$ is a mangrove.*

PROOF. By the definition of a mangrove, we must show that all vertices in $G'$ induce arborescences. Suppose to the contrary that there is some $r \in V(G')$ such that $G'_r$ is not an arborescence. Then (at least) one of the three necessary and sufficient conditions of an arborescent graph is violated: (i) $G'_r$ is not acyclic. This implies that $G'$ is not acyclic,

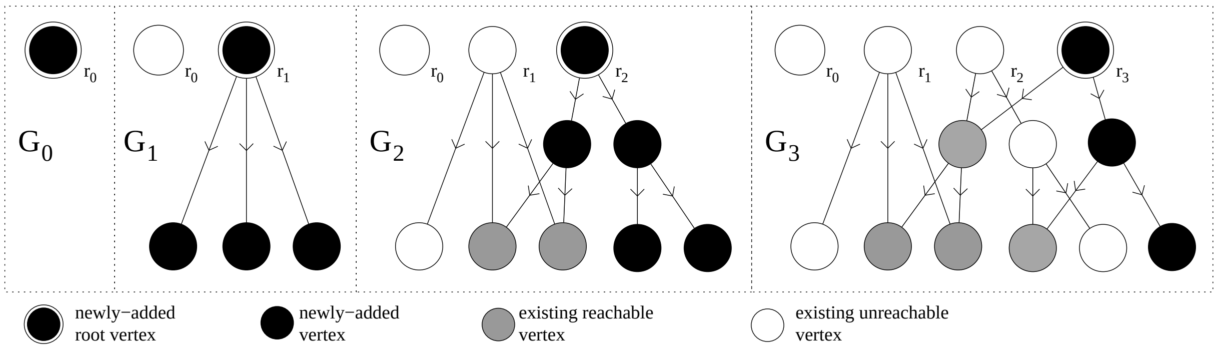


**Figure 6: Example key disclosure graph evolving due to a shadowing graph mutation chain. All graphs except $G_0$ result from applying a shadowing graph mutation on the previous graph. Black nodes are ones added by the most recent mutation with the root node circled; grey nodes are ones from the previous graph that are still reachable from the new root; white nodes are ones from the previous graph that are no longer reachable.**

which contradicts Lemma 2.
(ii) The indegree of $r \neq 0$. Then $r$ must have at least one incoming edge, from a vertex $v$. This results in a cycle, since $v$ is reachable from $r$ by construction of the induced graph $G'_r$, also contradicting Lemma 2.
(iii) There is some $v \in V(G'_r)$ such that $v \neq r$ and indegree of $v \neq 1$.

As the first two conditions lead to immediate contradictions, we assume that the final condition is violated. Moreover, since $v$ is a vertex on an induced graph, there is a path from $r$ to $v$ and thus $v$ must have at least one incoming edge and therefore the indegree of $v \geq 2$. By the induced graph $G'_v$'s construction, both parents of $v$ are reachable from $r$, and so there are two distinct paths $P^r_v$ and $Q^r_v$ in $G'$ from $r$ to $v$. We have two cases: either $r \in V(G)$ or $r \in V(G_S)$.

Suppose that $r \in V(G)$, and so all vertices of $P^r_v$ and $Q^r_v$ are elements of $V(G)$. Also, by construction, $E(G')|_{V(G)\times V(G)} = E(G)$, and thus all edges of $P^r_v$ and $Q^r_v$ are elements of $E(G)$. Therefore, $P^r_v$ and $Q^r_v$ are distinct paths from $r$ to $v$ in $G$, contradicting $G$ being a mangrove.

Now suppose that $r \in V(G_S)$. If $v \in V(G_S)$, then $P^r_v$ and $Q^r_v$ are distinct paths entirely in $G_S$, which contradicts $G_S$ being an arborescence. So $r \in V(G_S)$ and $v \in V(G)$. We decompose the paths as follows: $P^r_v = P^r_u, (u, w), P^w_v$ and $Q^r_x, (x, y), Q^y_v$, where $(u, w)$ and $(x, y)$ are elements of $\hat{E}$. We know that $P^r_v \neq Q^r_v$, and so there are four different cases based on the edge in $\hat{E}$:
(i) If $(u, w) = (x, y)$, i.e., both paths cross from $G_S$ to $G$ over the same edge in $\hat{E}$, then the two paths must differ elsewhere. Either $P^r_u \neq Q^r_x$ or $P^w_v \neq Q^w_v$. As we have seen before, however, this contradicts either $G_S$ being an arborescence or $G$ being a mangrove respectively.
(ii) If $u \neq x$ and $w = y$, then $(u, w)$ and $(x, w)$ are distinct edges in $\hat{E}$, a violation of its construction. This contradicts $(G, r, G_S, T, \hat{E})$ being a valid shadowing mutation.
(iii), (iv) If $w \neq y$ then we have distinct paths $P^w_v$ and $Q^y_v$ in $G$. Since both paths terminate at the same vertex, either $w$ or $y$ is the ancestor of one of the other's descendants. This contradicts Lemma 1.

In conclusion, such distinct paths $P^r_v$ and $Q^r_v$ cannot exist. Therefore, for all $r \in V(G')$, $G'_r$ is an arborescence and so $G'$ is a mangrove. □

### *Shadowing Graph Mutation Chains.*

Definition 2 tells us that we can achieve secure deletion with appropriate constraints on the shape of the key disclosure graph. We now show that performing a natural sequence of shadowing graph mutations satisfies these constraints, effecting simple secure deletion.

*Definition 3.* A *shadowing graph mutation chain* is a sequence of shadowing graph mutations $\mathcal{M} = (M_0, \ldots, M_p)$ such that: (i) $M_i = (G_i, r_i, G_{S,i}, T_i, \hat{E}_i)$, (ii) $G_0 = (\{\phi(0)\}, \emptyset)$, (iii) $r_0 = \phi(0)$, (iv) $\forall i > 0 \;.\; G_i = \mu(M_{i-1})$, and (v) $\forall i > 0 \;.\; r_i = r_{S,i-1}$.

A shadowing graph mutation chain describes a sequence of mutations applied on a key disclosure graph. Figure 6 shows an example key disclosure graph evolution as the result of three mutations. Each mutation in the chain is applied on the graph that results from the previous mutation, except for the base case. Observe that $r_i$—the root vertex in $T_i$—is always $r_{S,i-1}$ the root vertex added by the shadowing graph in the previous mutation (or the ‘zero’ key for the base of recursion).

We now prove our main result about the interplay of secure deletion and shadowing graph mutation chains. For convenience, given $M = (G, r, G_S, T, \hat{E})$, we say that a vertex $v \in V(G)$ is reachable in $M$ if there exists a path from $r$ to $v$ in $G$.

Lemma 3. *Given a shadowing graph mutation chain $\mathcal{M} = (M_0, \ldots, M_p)$, any vertex $v$ first reachable in $M_i$ and last reachable in $M_{i+k}$ $(k \geq 0)$ is reachable in all intermediate mutations $M_{i+1}, \ldots, M_{i+k-1}$.*

Proof. Suppose to the contrary that there exists a $j$, $i < j < i + k$, such that $v$ is not reachable in $M_j$. By construction of shadowing graph mutations, $v \in V(G_i) \Rightarrow v \in V(G_j) \Rightarrow v \notin V(G_{S,j})$. Select the largest such $j$, so that $v$ is reachable in $M_{j+1}$, and so there exists a path $P^{r_{j+1}}_v$ in $G_{j+1}$. Since $r_{j+1} \in V(G_{S,i})$ and $v \notin V(G_{S,i})$, such a path has the form $P^{r_{j+1}}_{\hat{e}}(\hat{e}, \hat{e}')P^{\hat{e}'}_v$ where $(\hat{e}, \hat{e}') \in \hat{E}_j$ and $P^{\hat{e}'}_v$ is a path in $G_j$. Then $\hat{e}' \in W(G_j, T_j)$ and so $P^{r_j}_{\hat{e}'}$ is a path in $G_j$, implying that $P^{r_j}_{\hat{e}'}P^{\hat{e}'}_v$ is a path from $r_j$ to $v$ in $G_j$, a contradiction. □

Lemma 3 tells us that, when building shadowing graph mutation chains as described, once some reachable vertex becomes unreachable then it remains permanently unreachable. Secure data deletion is achieved by a single mutation that makes the corresponding vertex unreachable from the new root. We now prove our final result on achieving secure deletion with shadowing graph mutation chains.

THEOREM 2. *Let $\mathcal{M} = (M_0, \ldots, M_p)$ be a shadowing graph mutation chain with resulting key disclosure graph $G = \mu(M_p)$. Let $T = (t_0, \ldots, t_p)$ be the (strictly-increasing) sequence of timestamps such that at time $t_i$*
*(i) $\mu(M_i)$ is performed,*
*(ii) the value $k_{i+1} = \phi^{-1}(r_{i+1})$ is stored in the securely-deleting memory, and*
*(iii) all previous values stored are securely deleted.*
*Let $D$ be data encrypted with the key $k$ whose corresponding vertex $v = \phi(k)$ is reachable only in $M_i, \ldots, M_{i+l}$. Then $D$'s lifetime is bounded by $t_i$ and $t_{i+l}$, and $D$ is securely deleted provided no compromise occurs during this time.*

PROOF. The proof is by establishing the premises required for Definition 2. First, $R = \{k_0, \ldots, k_p\}$ and $R_{\text{live}} = \{k_i, \ldots, k_{i+l}\}$ which means that $R_{\text{dead}} = \{k_0, \ldots, k_{i-1}\} \cup \{k_{i+l+1}, \ldots, k_p\}$. Because no compromise occurs from time $t_i$ until $t_{i+l}$, to apply Definition 2 we must only show that for all $k_j \in R_{\text{dead}}$, there is no path from $\phi(k_j)$ to $v$ in $G = \mu(M_p)$.

Assume to the contrary that there is a $k_j = \phi^{-1}(r_j) \in R_{\text{dead}}$ such that there is a path $P_v^{r_j}$ in $G = \mu(M_p)$. Since $v$ is unreachable in $M_j$, $P_v^{r_j}$ is not a path in $G_j$. So there must be an edge $(u, v)$ on $P_v^{r_j}$ such that $u \in V(G_j), (u, v) \in E(G)$, and $(u, v) \notin E(G_j)$. Then $\exists m \geq 0 : (u, v) \notin E(G_{j+m}) \wedge (u, v) \in E(\mu(M_{j+m}))$, that is, some mutation adds $(u, v)$ to the key disclosure graph. By construction, $E(\mu(M_{j+m})) = E(G_{j+m}) \cup E(G_{S,j+m}) \cup \hat{E}_{j+m}$, and since $u \notin V(G_{S,j+m}) \Rightarrow (u, v) \in E(G_{j+m})$, a contradiction. Definition 2 therefore tells us that $D$ is securely deleted. $\square$

## 5. IMPLEMENTATION DETAILS

We have implemented our B-Tree-based solution. We use Linux's network block device (`nbd`), which allows a listening TCP socket to receive and reply to block device I/O requests. In our case, we have our implementation listening on that TCP socket. The `nbd-client` program and `nbd` kernel module—required to connect a device to our implementation and format/mount the resulting device—remain unchanged, ensuring that no modifications to the operating system are required to use our solution. Our implementation includes the encrypted B-Tree as described in Section 3 and interacts with a variety of user-configurable storage backends. Our implementation is written in C++11 and is freely available with a GPL version 2 license.

### *Data Storage.*

Our solution assumes the user somehow divides the data into data units indexed by a handle for storage. There are different ways this can be implemented. Trivially, a key-value storage system can be built where values are binary large objects (blobs) of data, and keys reference this data. The blobs can be entire files, components of files, etc. This allows our solution to implement a simple object storage device (OSD) [14]. If each blob is uniquely indexed by the B-Tree, then modifying the blob requires re-encrypting it entirely with a new key and updating its reference in the B-Tree. This inhibits the ability to efficiently securely delete data from large files such as databases.

Alternatively, data can be divided into fixed-size blocks indexed by the B-Tree. This facilitates random updates as only a fixed-size block must be updated to make any change to data. This is the construction we use in our implementation: a virtual array of data is indexed by offsets of fixed-size blocks and exposed as a block-device interface. This block-device can then be formatted as any block-based file system, which is then overlaid on the B-Tree. Sparse areas of the file system then do not appear as keys in the B-Tree; if written to, the corresponding keys are added to the B-Tree.

### *Network Block Device.*

The network block device is a block device offered by Linux. It behaves as a normal block device that can be formatted with any block-based file system and mounted for use. However, it is actually a virtual block device that forwards all block operations over TCP (i.e., reading and writing blocks, as well as trim and flush commands). The listening user-space program is responsible for actually implementing the block device.

### *Virtual Storage Device.*

While the default `nbd-server` program simply serves a local file as a block device, we wrote our own implementation of a virtual block device that interfaces with a variety of back-end storage media. The reading and writing of blocks pass through our shadowing B-Tree implementation. It uses block addresses as indices in the B-Tree; the data's remote storage location in that block address is kept in the leaves of the B-Tree. The user selects how the resulting data is stored, including data blocks for nodes and data (persistent medium) as well as the master key and integrity hash (securely-deleting medium).

## 6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the B-Tree under different workloads and investigate how the performance can be improved through different caching strategies.

### *Workloads.*

We test our implementation's performance on a variety of different file system workloads. We used the `filebench` utility [12] to generate three workloads and we also created our own workload by replaying our research group's version control history. We used filebench's `directio` mode to ensure that all reads and writes are sent directly to the block device and not served by any file system page cache; similarly, we synchronized the file system and flushed all file system buffers after each version update in our version control workload. The workloads we use are summarized as follows:
—`sequential` writes a 25 GiB file and then reads it contiguously. This tests the behaviour when copying very large files to and from storage.
—`random_1KiB` performs random 1 KiB reads and writes on a pre-written 25 GiB file. This tests the performance for a near-worst-case scenario: reads and writes without any temporal or spatial locality.
—`random_1MiB` performs random 1 MiB reads and writes on a pre-written 25 GiB file. This tests the performance for

| | | Cache size: 5 | | | Cache size: 10 | | | Cache size: 20 | | | Cache size: 50 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Workload | block size | LRU | LFU | Belady | LRU | LFU | Belady | LRU | LFU | Belady | LRU | LFU | Belady |
| sequential | 1 KiB | 0.06 | 47.3 | 81.4 | 98.1 | 60.8 | 97.9 | 98.3 | 70.1 | 98.1 | 99.1 | 74.7 | 99 |
| | 16 KiB | 99.6 | 43.2 | 99.7 | 99.7 | 56.6 | 99.8 | 99.7 | 83.3 | 99.8 | 99.8 | 99.5 | 99.8 |
| random (1KiB) | 1 KiB | 0.0023 | 12.2 | 15.9 | 11.8 | 15.3 | 19.8 | 15.6 | 21.3 | 25 | 26.4 | 26.3 | 33.5 |
| | 16 KiB | 47.3 | 32.2 | 52.2 | 49.3 | 40.1 | 58.2 | 53.6 | 56 | 64.5 | 63.4 | 67.2 | 70.3 |
| random (1MiB) | 1 KiB | 0.0068 | 21.5 | 82.1 | 97.5 | 25.4 | 97.7 | 97.7 | 31.6 | 97.8 | 97.9 | 38 | 98 |
| | 16 KiB | 98.1 | 50.5 | 98.5 | 98.6 | 62.6 | 98.7 | 98.7 | 85.2 | 99 | 98.9 | 98.3 | 99.1 |
| svn | 1 KiB | 95 | 35 | 95.6 | 96 | 47.2 | 96.1 | 96.5 | 57.1 | 96.9 | 97.1 | 75.9 | 97.4 |
| | 16 KiB | 97.2 | 68.7 | 97.7 | 97.8 | 81.2 | 97.8 | 98.2 | 88 | 98.2 | 98.6 | 96.2 | 99 |

**Table 1: Caching Hit Ratio (%) for B-Tree Nodes**

random access patterns with a larger block size that provides some spatial locality in accessed data.
—`svn` replays 25 GiB of our research group's version control history by iteratively checking out each version. This test provides an example of a realistic usage scenario for data being stored on a shared persistent storage medium.

We run our implementation behind an `nbd` virtual block device, formatted with the `ext2` file system. We mount the file system with the `discard` option to ensure that the file system identifies deleted blocks through TRIM commands.

### *Caching.*

We experimentally determine the effect of the skeleton tree's cache size and eviction strategy. Using the sequence of block requests characteristic of each workload, we use our B-Tree implementation to output a sequence of B-Tree node requests. A B-Tree node request occurs whenever the skeleton tree visits a node; missing nodes must be fetched from the persistent medium and correspond to cache misses. Observe that for the same workload, the sequence of node requests will vary depending on the B-Tree's block size. We output one B-Tree node request sequence for each block size that we test. With this sequence of node requests, we then simulate various cache sizes and caching behaviours.

We test three different strategies: Bélády's optimal "clairvoyant" strategy [1], least recently used (LRU), and least frequently used (LFU). Bélády's strategy is included as an objective reference point to compare caching strategies. We only maintain cache usage statistics for items currently in the cache.

The results of our experiment are shown in Table 1. We observe that caching nodes is generally quite successful; many of the workloads and configurations have very high hit ratios. This is because contiguous ranges of block address tend to share paths in the B-Tree. Consequently, the cache size itself is not so important; provided it is sufficiently large to hold a complete path then sequential access occurs rapidly.

LRU is generally preferable to LFU. The only exception is very small random writes with a small block size. This is because such writes have no temporal locality and so the frequency-based metric better captures which nodes contain useful data. For random-access patterns, the cache size is far more important than the eviction strategy, a feature also observable from Bélády's optimal strategy. For any form of sequential access, LRU outperforms LFU because LFU unfairly evicts newly cached nodes, which may currently have few visits but are visited frequently after their first caching. We see that LRU often approaches Bélády's optimal strategy, implying that more complicated strategies offer limited potential for improvement.

| | | B-Tree block size | | | |
|---|---|---|---|---|---|
| | | 4 KiB | 16 KiB | 64 KiB | 256 KiB |
| general | total data blocks | 6553600 | 1638400 | 409600 | 102400 |
| | tree height | 5 | 3 | 2 | 2 |
| | cache size (nodes) | 2048 | 512 | 128 | 32 |
| | MiBs sharing path | 0.16 | 2.65 | 42.6 | 682.5 |
| sequent. | cache hits (%) | 99.3 | 99.7 | 99.9 | 1 |
| | storage overhead (%) | 2.4 | 0.6 | 0.1 | 0.03 |
| | comm overhead (%) | 2.4 | 0.6 | 0.1 | 0.03 |
| | block size ovrhd (%) | 0 | 5.3 | 26.3 | 58.1 |
| rand 1k | cache hits (%) | 64.7 | 59 | 43.2 | 73.8 |
| | storage overhead (%) | 2.4 | 0.6 | 0.1 | 0.03 |
| | comm overhead (%) | 1308.5 | 3129 | 8623.5 | 20671.4 |
| | block size ovrhd (%) | 497.9 | 2293.2 | 9473 | 38191.8 |
| rand 1m | cache hits (%) | 99.2 | 98.9 | 96.5 | 95.5 |
| | storage overhead (%) | 2.47 | 0.59 | 0.14 | 0.03 |
| | comm overhead (%) | 4.9 | 3.7 | 7.8 | 17.7 |
| | block size ovrhd (%) | 1 | 7.7 | 34.6 | 82.1 |
| svn | cache hits (%) | 99.2 | 98.9 | 96.5 | 95.5 |
| | storage overhead (%) | 1.74 | 0.42 | 0.1 | 0.02 |
| | comm overhead (%) | 4.4 | 4.9 | 5.4 | 2.6 |
| | block size ovrhd (%) | 0 | 63.4 | 247.9 | 750.2 |

**Table 2: B-Tree Secure Deletion Overhead**

### *B-Tree Properties.*

We investigate our system's overhead with regards to the fetching and storing of nodes that index the data. We now characterize this with regards to different workloads and parameters, expressing the results with the following metrics:
—Cache hits: percentage of B-Tree node visits that do not require fetching.
—Storage overhead: ratio of node storage size to data storage expressed as a percentage.
—Communication overhead: ratio of the persistent medium's communication for fetching and storing nodes compared to the sum of useful data read and written, expressed as a percentage.
—Block size overhead: ratio of additional network traffic (beyond the I/O) for fetching and storing data compared to the sum of data read and written by the file system, expressed as a percentage. (This is based only on the block size / workload and is independent of the use of the secure deletion B-Tree.)

Additionally, we characterize the following B-Tree properties common to all workloads:
—Total data blocks: 25 GiB divided by the block size.
—Tree height: the height of the B-Tree that indexes the number of data blocks.
—Cache size (nodes): the fixed cache size of 8 MiB expressed as nodes that fit into that capacity.
—MiBs sharing path: the size of contiguous data whose blocks all share a unique path, that is, how much data is indexed by a single leaf node.

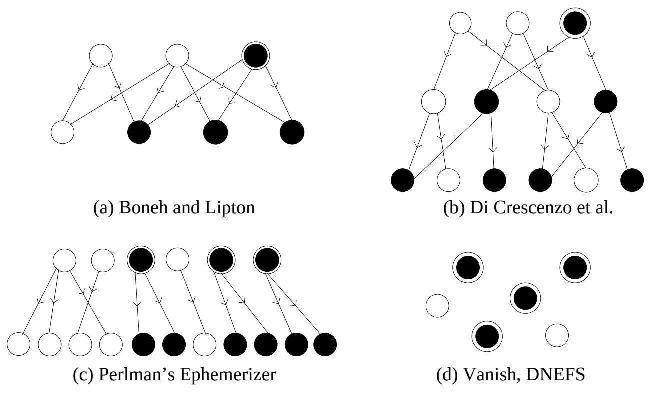

(a) Boneh and Lipton (b) Di Crescenzo et al. (c) Perlman's Ephemerizer (d) Vanish, DNEFS

**Figure 7: Mangrove key disclosure graphs for related work. Circled nodes represent the current key(s) stored in the securely-deleting medium, and shaded nodes represent the keys the adversary would gain with a coercive attack.**

Table 2 shows the results of our experiments. We see that in all cases the storage overhead of the B-Tree nodes is a few percent and decreases with the block size. In all workloads except `random_1KiB`, the communication overhead is also reasonable. Large block sizes most benefit sequential access patterns, because a large block size means more sequential data can be accessed without fetching new nodes (e.g., using a block size of 256 KiB results in half a GiB of data indexed by the same path in the B-Tree). Degenerate performance is observed for our worst-case workload: where data blocks are accessed in a completely random fashion without any spatial or temporal locality. Even the block size overhead resulting from fetching unnecessary data shows a large amount of waste, and we conclude that further engineering effort is needed to optimize this use case.

## 7. RELATED WORK

Secure deletion has been studied in a variety of contexts and has also been extensively surveyed [7, 9, 17]. Moreover, a variety of work considers secure deletion in the context of mixed-media storage devices consisting of a small securely-deleting medium and a large persistent medium [2, 6, 10, 16, 18]. In this section we discuss these works and, when relevant, describe their corresponding key disclosure graphs. For these cases, the update mechanism is a shadowing graph mutation and the resulting key disclosure graph is a mangrove. Hence, we can apply our secure deletion proof to these works as well as our own. Figure 7 shows an example key disclosure graph for each of these works and shows which vertices would be revealed if a coercive attack occurred.

Boneh and Lipton's original use of cryptography to securely delete data considered an off-line large-capacity tape archive and a local low-capacity computer. Data for archiving is encrypted with a random key and the key is added to a list of valid keys. Each key on the list is encrypted with a master key that is periodically randomly regenerated; the new master key is used to re-encrypt the list of valid keys. The master key is then stored locally and securely—the authors propose floppy disks or writing the master key down on paper—such that it can be securely deleted. Data is deleted by removing its corresponding key from the key list and waiting until the next time the master key is regenerated. This scheme's corresponding key disclosure graph is illustrated in Figure 7 (a); it is a mangrove with a maximum path length of one.

Di Crescenzo et al.'s work explicitly assumes a large persistent medium and a small securely-deleting medium. They divide a fixed-size persistent medium into numbered blocks, which are indexed by a pre-allocated binary tree. The keys to decrypt data are stored in the leaves and the tree's internal nodes store the keys to decrypt the children. The root key is stored in a securely-deleting medium. Each change to a data block indexed by the binary tree results in a new key stored in a leaf node and all nodes on the path to the root are rekeyed. This scheme's corresponding key disclosure graph is also a mangrove and is illustrated in Figure 7 (b).

Perlman's Ephemerizer aims to make communication reliably unrecoverable after an expiration time [16]. Exchanged messages are encrypted using ephemeral keys with a predetermined lifetime. Secure deletion is used to ensure keys are irrecoverable after they expire. Perlman's scheme uses a trusted third party—the Ephemerizer—to manage the ephemeral keys and ensure their irrecoverability after expiration. Each message is encrypted with a random key, which is then blinded and sent to the Ephemerizer along with the desired message lifetime. The Ephemerizer encrypts the message key with a corresponding ephemeral key based on the desired lifetime. The message encrypted with the random key, along with the random key encrypted with the ephemeral key, are sent as the message. The recipient uses the Ephemerizer, with blinding, to determine the message key. Once the ephemeral key expires, the Ephemerizer no longer possesses it and is therefore unable to decrypt any keys wrapped with it. In this scheme, the Ephemerizer acts as the securely-deleting medium and the communication channel, being vulnerable to eavesdroppers, is the persistent storage. The resulting key disclosure graph, illustrated in Figure 7 (c) is a mangrove similar to Figure 7 (a) except that there are multiple keys stored in the securely-deleting medium, one key for each future expiration time.

Tang et al.'s Fade extends Perlman's Ephemerizer [20] by explicitly considering cloud storage as the persistent medium and by offering more expressive deletion policies than expiration dates. An Ephemerizer-like entity acts as the securely-deleting medium, but each key that it manages reflects a specific policy that can expire or be revoked. Policies are defined as boolean expressions of attributes.

Cachin et al.'s work also offers policy-based secure deletion with a more expressive policy language [4]. Their system builds a policy graph mapping attributes to policies. Each node is a configurable threshold cryptographic operator; logical 'or' and 'and' are then just special cases. Cachin et al.'s approach assumes a securely-deleting medium to store master keys, and can be implemented in a variety of ways with different retrieval complexities and storage costs.

In both Tang et al. and Cachin et al., data is deleted at the *granularity* of an entire policy, defined as boolean expressions. For instance, one policy may say that *data is not securely deleted* if *its expiration time has not elapsed* and *it has not been specifically redacted*. Each conjunct is associated with a key, both of which are needed to decrypt the message. To achieve this, Tang et al. uses nested key wrapping while Cachin et al. uses threshold cryptography. Consequently, we cannot directly apply our work to these schemes and leave the characterization of the key disclosure graph for these cryptographic concepts as future work.

Geambasu et al.'s Vanish is designed for erasable communication on the Internet [10]. Messages are encrypted with a unique key and published on the Internet (i.e., the persistent medium). Encryption keys are split into shares that are also stored on the Internet in a distributed hash table. The security of their scheme relies on the nodes in the distributed hash table implementing the securely-deleting medium: it requires—albeit contentiously [22]—that the nodes periodically delete the data and that the adversary is overwhelmed by the scale of the distributed hash table to effectively copy all the data therein. As their scheme does not use key wrapping, the corresponding key disclosure graph is trivially a mangrove as it lacks edges (Figure 7 (d)).

Reardon et al.'s data node encrypted file system [18] is designed for flash memory, for which data deletion is an expensive deletion operation. They divide the memory into two areas: a small securely-deleting key storage area and a large main storage which they assume to be persistent. Each atomic unit of data in main storage is encrypted with a unique key from the key storage area. The efficiency of this approach comes from colocating and compressing the key storage area thereby reducing secure deletion's cost. Like Vanish, their scheme does not use key wrapping and so the key disclosure graph is also a mangrove.

## 8. CONCLUSIONS

We developed a general approach to the design and analysis of secure deletion from persistent media. We used graph theory to reason about adversarial knowledge and developed a graph mutation that maintains properties on adversarial knowledge that allows straightforward provable secure deletion. Our mutation subsumes the update behaviour of all arborescent data structures. We designed and implemented a securely-deleting B-Tree based on this mutation. Our analysis showed that the communication and storage overhead is typically negligible and the skeleton tree's caching of B-Tree nodes is very effective.

## 9. ACKNOWLEDGMENTS

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors. We would like to thank our anonymous reviewers as well as Roberto Di Pietro, Thomas Themel, and Nils Ole Tippenhauer for their helpful comments.

## 10. REFERENCES

[1] Laszlo A. Bélády. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[2] Dan Boneh and Richard J. Lipton. A Revocable Backup System. In *USENIX Security Symposium*, pages 91–96, 1996.

[3] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[4] Christian Cachin, Kristiyan Haralambiev, Hsu-Chun Hsiao, and Alessandro Sorniotti. Policy-based Secure Deletion. Cryptology ePrint Archive, Report 152, 2013.

[5] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[6] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to Forget a Secret. In *STACS*, Lecture Notes in Computer Science, pages 500–509. Springer, 1999.

[7] Sarah M. Diesburg and An-I Andy Wang. A survey of confidential data storage and deletion methods. *ACM Computing Surveys*, 43(1):1–37, 2010.

[8] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[9] Simson Garfinkel and Abhi Shelat. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security & Privacy*, pages 17–27, January 2003.

[10] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: increasing data privacy with self-destructing data. In *USENIX Security Symposium*, pages 299–316, 2009.

[11] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium*, pages 77–89, 1996.

[12] McDougall, R. and Mauro, J. FileBench. www.solarisinternals.com/si/tools/filebench/, 2005.

[13] Ralph C. Merkle. A certified digital signature. In *Proceedings on Advances in Cryptology*, CRYPTO '89, pages 218–238. Springer-Verlag New York, Inc., 1989.

[14] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-Based Storage. *Communications Magazine, IEEE*, 41(8):84–90, 2003.

[15] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Providing authentication and integrity in outsourced databases using Merkle hash trees. Technical report, University of California Irvine, 2003.

[16] Radia Perlman. The Ephemerizer: Making Data Disappear. Technical report, Sun Microsystems, 2005.

[17] Joel Reardon, David Basin, and Srdjan Capkun. SoK: Secure Data Deletion. In *IEEE Symposium on Security and Privacy*, 2013.

[18] Joel Reardon, Srdjan Capkun, and David Basin. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. In *USENIX Security Symposium*, pages 333–348, 2012.

[19] Ohad Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):2:1–2:27, 2008.

[20] Yang Tang, Patrick P. C. Lee, John C. S. Lui, and Radia Perlman. FADE: Secure Overlay Cloud Storage with File Assured Deletion. In *SecureComm*, pages 380–397, 2010.

[21] W. T. Tutte. *Graph Theory*. Encyclopedia of Mathematics and its Applications. Addison-Wesley Publishing Company, 1984.

[22] Scott Wolchok, Owen S. Hoffman, Nadia Henninger, Edward W. Felten, J. Alex Haldermann, Christopher J. Rossback, Brent Waters, and Emmet Witchel. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs. In *Proc. 17th Network and Distributed System Security Symposium*, February 2010.