Diss. ETH No. 22475

Secure Data Deletion

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

JOEL REARDON

Master of Mathematics, University of Waterloo
born 31.07.1983
citizen of Canada

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Prof. Dr. David Basin, coexaminer
Prof. Dr. Ari Juels, coexaminer
Prof. Dr. Paul Van Oorschot, coexaminer
Dr. Alessandro Sorniotti, coexaminer

2014

Abstract

Secure data deletion is the task of deleting data from a physical medium so that the data is irrecoverable. This irrecoverability is what distinguishes *secure* deletion from *regular* deletion, which ostensibly deletes unneeded data only to reclaim resources. We *securely* delete data to prevent an adversary from gaining access to it, and so secure deletion is a natural part of the confidentiality of data.

In this thesis, we examine secure deletion in a variety of different systems and different layers: from the hardware level of ensuring a storage medium can efficiently delete data to the system level of deleting data from unreliable and untrusted servers. We examine related work in detail, identify the deficiencies and unsolved problems, and build our own solutions to advance the state of the art. Our analysis provides a framework to reason about secure deletion in general. We organize existing solutions in terms of their interfaces to physical media and further present a taxonomy of adversaries differing in their capabilities as well as a systematization for the characteristics of secure deletion solutions. We then design a system and adversarial model for secure deletion that encompasses the most challenging aspects that we distill from our survey. Our research contributions are then provided within this model.

We consider secure deletion in the context of two main types of storage media: mobile storage and remote (e.g., cloud) storage. At the time that the research was undertaken, both these computational environments represented a significant shift in how most people accessed their data. The lack of secure deletion is a security concern in both settings. Both store sensitive user data and both are vulnerable to adversarial compromise. Despite the massive difference in the scale of these devices, the challenges of secure deletion shares surprising similarities.

Secure deletion for mobile devices means secure deletion for flash memory, as it is currently ubiquitously used in portable storage devices. Flash memory has the problem where the unit of erasure is much larger than the unit of read and write, and worse, erasure incurs a greater cost that is manifested in power consumption and physical wear.

Our first contribution for flash memory is research into user-level secure deletion for flash memory, that is, what can be done by a user on their portable device by simply adding an application. We use a concrete example of an Android-based mobile phone. We show that it provides no guarantees on data deletion and the time data remains increases with the storage medium's size. We propose two user-level solutions that achieve secure deletion as well as a hybrid of both solutions, which guarantees the periodic, prompt secure deletion of data regardless of the storage medium's size.

Our second contribution for flash memory is the Data Node Encrypted File System (DNEFS), a file system extension that provides fine-grained efficient secure data deletion. We implement DNEFS in the flash file system Unordered Block Images File System (UBIFS) and call our implementation UBIFSec. We further integrate UBIFSec in the Android operating system running on a Google Nexus One smartphone. We show that it is efficient; Android OS and applications (including video and audio playback) run normally on top of UBIFSec.

Secure deletion for remote storage means secure deletion for persistent storage, that is, a storage medium that is unable to delete *any* data. The reason to thus model cloud storage is that once the data has left the users' control, users are unable to themselves ensure that access control and secure deletion are performed correctly. To compensate for the persistent storage medium's inability to perform deletion, we assume that the user has access to a small securely-deleting storage medium to manage the encryption keys that permit the secure deletion of data.

Our first contribution for persistent storage is to present a general solution 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 as a key disclosure graph mutation and prove that the update function achieves secure data deletion; instances of the update function implement the update behaviour of all arborescent data structures. We find that related work fits well in our model and characterize their key disclosure graphs. We then design and implement a B-Tree instance within the space of possible securely-deleting data structures and analyse its performance, finding that its overheads are small.

Our second contribution for remote storage considers the problem of an unreliable securely-deleting storage medium, that is, one that may lose data, expose data, fail to delete data, and fail to be available. We build a robust fault-tolerant system that uses multiple unreliable storage media. The system permits multiple clients to store securely-deletable data and provides a means to control policy aspects of its storage and deletion. We present details on the implementation both of the distributed securely-deleting medium as well as a file system extension that uses it. The solution has low latency at high loads and requires only a small amount of communication among nodes.

Zusammenfassung

Sicheres Löschen von Daten ist der Vorgang des Entfernens von Daten von einem physischen Medium, so dass diese Daten unwiederbringlich zerstört sind. Die Unwiederbringlichkeit unterscheidet *sicheres* von *regulärem* Löschen, welches vorgeblich unnötige Daten löscht, um Ressourcen zurückzugewinnen. Wir löschen Daten *sicher*, um einen Widersacher daran zu hindern Zugang zu ihnen zu bekommen. Und damit stellt sicheres Löschen einen natürlichen Bestandteil der Datenvertraulichkeit dar.

In dieser Arbeit untersuchen wir sicheres Löschen anhand einer Auswahl verschiedener Systeme und Schichten: von der Hardwareschicht, welche sicher stellt, dass Daten effizient gelöscht werden, bis zu der Systemschicht, die Daten von unzuverlässigen und misstrauten Servern löscht. Eingehend untersuchen wir andere themenbezogene Arbeiten, identifizieren deren Mängel und ungelöste Probleme und erstellen unser eigenes Lösungskonzept, um den derzeitigen Stand der Technik anzuheben. Unsere Darlegung liefert einen allgemeinen Rahmen, um vernünftig über sicheres Löschen reden zu können. Wir strukturieren existierende Lösungen bezüglich ihrer Schnittstellen zu physischen Medien. Weiterhin präsentieren wir eine Widersachertaxonomie, aufgeschlüsselt nach deren Fähigkeiten, sowie eine Systematisierung der Charakteristika die sicheres Löschen auszeichnen. Danach entwerfen wir ein System- und Feindmodell für sicheres Löschen, welches die anspruchsvollsten Aspekte einschliesst, die wir in unserer Studie extrahieren konnten. Unser wissenschaftlicher Beitrag wird schliesslich in diesem Modell dargestellt.

Wir betrachten sicheres Löschen hinsichtlich von zwei Hauptarten von Speichermedien: mobile Speicher und verteilte (z.B. Cloud-) Speicher. Zum Zeitpunkt dieser Arbeit führten beide Rechenumgebungen zu einer gravierenden Veränderung in der Art und Weise, mit der die

meisten Menschen auf ihre Daten zugreifen. Das Fehlen von sicherem Löschen ist ein Sicherheitsbelang in beiden Umgebungen. Beide verwahren sensitive Nutzerdaten und beide sind verwundbar hinsichtlich feindlicher Übernahme. Trotz der massiven Unterschiede in der Grössenordnung dieser Speichermedien teilen sie sich eine überraschend grosse Anzahl an Gemeinsamkeiten.

Sicheres Löschen auf mobilen Geräten bedeutet sicheres Löschen auf Flash-Speichern, welche derzeit überall auf tragbaren Speichergeräten eingesetzt werden. Flash-Speicher haben das Problem, dass die Einheiten, die gelöscht werden, viel grösser sind als die Einheiten, die gelesen oder geschrieben werden. Dadurch fallen grössere Kosten an, die sich in erhöhtem Energieverbrauch und physischer Abnutzung äussern.

Unser erster Beitrag in Bezug auf Flash-Speicher ist die Erforschung des sicheren Löschen von Flash-Speichern auf Benutzerebene, d.h. was alleiniges Hinzufügen einer Anwendung einem Benutzern auf seinem tragbaren Gerät diesbezüglich ermöglichen kann. Wir benutzen hierfür ein konkretes Beispiel eines auf Android basierenden Mobiltelefons. Wir zeigen, dass keine Garantien für das Löschen von Daten gewährleistet werden können und dass die Dauer, wie lange Daten auf dem Speichermedium bestehen bleiben, mit dessen Grösse des Mediums ansteigt. Wir schlagen zwei Lösungskonzepte auf Benutzerebene vor, welche sicheres Löschen ermöglichen, und weiterhin eine Hybridlösung von beiden, die periodisches und sofortiges, sicheres Löschen unabhängig von der Grösse des Speichermediums garantiert.

Unser zweiter Beitrag hinsichtlich Flash-Speicher ist DNEFS, eine Anpassung am Dateisystem, die ein präzises und effizientes, sicheres Löschen von Daten ermöglicht. Wir implementieren DNEFS in dem Flash-Dateisystem UBIFS und nennen es UBIFSec. Ausserdem integrieren wir UBIFSec im Android Betriebssystem auf einem Google Nexus One Mobiltelefon. Wir zeigen, dass es effizient ist und dass das Android Betriebssystem zusammen mit Anwendungen (einschliesslich Video- und Audio-Wiedergabe) normal unter UBIFSec laufen.

Sicheres Löschen für verteilte Speicher bedeutet sicheres Löschen für dauerhafte Speicher, d.h. einem Speichermedium, auf dem es unmöglich ist *jegliche* Daten zu löschen. Der Grund verteilte Speicher so zu modellieren besteht darin, dass, wenn sich die Daten einmal der Kontrolle des Benutzers entzogen haben, dieser unfähig ist sicher zu stellen, dass Zugangsberechtigungen und sicheres Löschen weiterhin korrekt ausgeführt werden. Um die Unfähigkeit des Löschens auf dem dauerhaften Speichermediums zu kompensieren, nehmen wir an, dass der Benut-

zer Zugang zu einem kleinen, sicher löschenden Speichermedium hat, auf dem die kryptographischen Schluüssel, welche das sichere Löschen erlauben, verwaltet werden.

Unser erster Beitrag im Bezug auf dauerhafte Speicher besteht darin, dass wir ein allgemeines Lösungskonzept in Form eines Entwurfs und einer Analyse von sicherem Löschen präsentieren, welche auf Verschlüsselung und Schlüsselverpackung basiert. Wir definieren einen Schlüsselenthüllungsgraphen, der das feindliche Wissen über die Vergangenheit der Schlüsselerzeugung und Schlüsselverpackung modelliert. Wir führen eine generische Aktualisierungsfunktion als eine Schlüsselverpackungsmutation ein und beweisen, dass die Aktualisierungsfunktion sicheres Löschen von Daten gewährleistet. Instanzen der Aktualisierungsfunktion implementieren das Aktualisierungsverhalten von allen baumartigen Datenstrukturen. Wir stellen fest, dass andere themenbezogene Arbeiten gut in unser Modell passen und wir charakterisieren deren Schlüsselenthüllungsgraphen. Weiter entwerfen und implementieren wir eine B-Baum Instanz als eine mögliche, sicher löschende Datenstruktur und analysieren ihre Leistung, wobei sich herausstellt, dass der zusätzliche Rechenaufwand gering ist.

Unser zweiter Beitrag hinsichtlich dauerhafter Speicher beleuchtet das Problem eines unzuverlässigen, sicher löschenden Speichermediums, d.h. eines, das Daten verlieren kann, Daten preisgibt, scheitert Daten zu löschen oder zeitweise nicht verfügbar ist. Wir erstellen ein robustes und fehlertolerantes System, das viele unzuverlässige Speichermedien benutzt. Das System erlaubt mehreren Klienten sicher löschbare Daten zu speichern und bietet die Möglichkeit die Art und Weise des Speicherns und Löschens genauer zu kontrollieren. Wir präsentieren Details zu der Implementierung sowohl für das verteilte, sicher löschende Medium als auch für eine Dateisystemerweiterung, die es benutzt. Das Lösungskonzept weist eine geringe Verzögerung bei grossen Datenmengen auf und benötigt nur einen geringen Kommunikationsaufwand zwischen den Knoten.

Resumé

La suppression de données sécurisées est la tâche de supprimer des données d'un support physique tel que les données sont irrécupérables. L'irrécupérabilité est la différence entre la suppression sécurisées et la suppression ordinaire, qui prétend d'effacer des données non nécessaires afin de libérer de l'espace de stockage. Nous supprimons des données d'une manière *sécurisées* afin d'empêcher un adversaire d'y accéder, et donc la suppression sécurisé fait partie naturelle de la confidentialité des données.

Dans cette thèse, nous examinons la suppression sécurisée dans une variété de différents systèmes et différentes couches : à partir du niveau materiel partant d'un support de stockage qui peut supprimer des données de manière efficace jusqu'au niveau système avec la suppression de données sur des serveurs non fiables. Nous examinons en détail les travaux connexes, identifions les lacunes et les problèmes non résolus, et nous construisons nos propres solutions pour faire progresser l'état de l'art. Notre analyse fournit un cadre pour raisonner sur l'effacement sécurisé en général. Nous organisons les solutions existantes en termes de leurs interfaces à support physique et en outre présentons une taxonomie d'adversaires différents dans leur capacités ainsi qu'un systématisation des caractéristiques de sécurité des solutions de suppression. Nous concevons ensuite un système et un modèle d'adversaire pour la suppression sécurisé qui englobe les aspects les plus difficiles que nous distillons de notre enquête. Nos contributions de recherche sont ensuite fournis dans ce modèle.

Nous considérons la suppression sécurisée dans le cadre de deux principaux types de supports de stockage : le stockage mobile et à distance (par exemple, cloud) de stockage. Au moment où la recherche a été entrepris, ces deux environnements informatiques représentaient un important changement dans la faÃğon la plupart des gens accèdent

à leurs données. Le manque de suppression sécurisée est un problème de sécurité dans les deux cas. Les deux stockent des données sensibles de l'utilisateur et les deux sont vulnérables aux compromis d'un adversaire. Malgré l'énorme différence d'échelle de ces dispositifs, les défis de la part de la suppression sécurisée partagent des similitudes surprenantes.

La suppression sécurisée pour appareils mobiles signifie la suppression sécurisée de la mémoire flash, car actuellement ubiquitairement utilisé dans les dispositifs de stockage portables. La mémoire flash a le problème que l'unité d'effacement est beaucoup plus grande que l'unité de lecture et d'écriture, et pire encore, nécessite un coût plus élevé se manifestant dans la consommation d'énergie et de l'usure physique.

Notre première contribution pour la mémoire flash est la recherche au niveau utilisateur pour la suppression sécurisé à base de mémoire flash, ceux qui peut être fait par un utilisateur sur leur dispositif portable en simplement ajoutant une application.

Nous utilisons un exemple concret d'un téléphone mobile basé sur Android. Nous montrons qu'il ne donne aucune garantie sur la suppression de données et que le temps que les données persistent, augmente avec la taille du support de stockage. Nous proposons deux solutions au niveau de l'utilisateur qui permettent d'atteindre l'effacement sécurisé ainsi qu'un solution hybride, qui garantit la suppression périodique, rapide des données, indépendamment de la taille de support de stockage.

Notre deuxième contribution pour la mémoire flash est DNEFS, un changement de système de fichiers qui fournit une suppression de données sécurisé fine et efficace. Nous mettons en œuvre DNEFS dans le système de fichier flash UBIFS et appelons notre mise en œuvre UBIFSec. Nous intégrons UBIFSec dans le système d'exploitation Android fonctionnant sur un Google Nexus One smartphone. Nous montrons qu'il est efficace ; le système d'exploitation et les applications Android (y compris la vidéo et la lecture audio) fonctionnent normalement audessus de UBIFSec.

L'effacement sécurisé pour le stockage à distance est équivalent à la suppression sécurisée pour le stockage permanent, c'est à dire un support de stockage qui n'est pas en mesure de supprimer *n'importe quel* données. Ce modèle doit être choisit car le stockage cloud, une fois que l'utilisateur a perdu le contrôle sur ces données, est incapable de s'assurer que le contrôle d'accès et l'effacement sécurisé sont effectuée correctement. Pour compenser l'incapacité du support de stockage persistant à effectuer la suppression, nous supposons que l'utilisateur a

accès à un petit support de stockage permettant la suppression en toute sécurité afin de gérer les clés de chiffrement permettant la suppression sécurisée de données.

Notre première contribution pour le stockage permanent est de présenter une solution générale pour concevoir et analyser la suppression sécurisée pour le stockage permanent qui repose sur le chiffrement et l'emballage clé. Nous définissons un mécanisme de divulgation de clé à base d'un graphe qui modèle les connaissances de l'adversaire de l'histoire de la génération de clés et de leur emballage. Nous introduisons une fonction de mise à jour générique en forme de divulgation de clé à base de graphe avec mutation et nous prouvons que le mécanisme de mise à jour permet l'effacement sécurisé des données. Nous nous apercevons que les travaux connexes s'intègrent bien dans notre modèle et caractérisent leur graphe de divulgation de clé. Ensuite nous concevons et mettons en œuvre une instance B-Tree dans l'espace des possibles structures de données pour la suppression sécurisé et analysons sa performance, constatant que ces surplus généraux sont faibles.

Notre deuxième contribution pour le stockage à distance considère le problème d'un moyen de stockage à suppression sécurisée non-fiable, qui est celui qui peut perdre des données, exposer des données, ne pas supprimer des données, et ne pas être disponible. Nous construisons un système robuste, tolérant des pannes et qui utilise plusieurs supports de stockage non-fiables. Le système permet à plusieurs clients de stocker des données avec la possibilité de l'effacement sécurisé et fournit un moyen pour contrôler les paramètres de stockage et de suppression. Nous présentons les détails sur la mise en œuvre pour à la fois le moyen de suppression sécurisé distribué ainsi que pour une extension du système de fichiers qu'il utilise. La solution a une faible latence aussi pendant des charges élevées et nécessite peu de la communication entre les nœuds.

Riassunto

L'attività di secure deletion consiste nel cancellare dati da un dispositivo fisico in modo da renderne impossibile il recupero. L'irrecuperabilità dei dati è ciò che distingue la secure deletion dal normale processo di cancellazione dati, che solitamente cancella i dati solo nel caso in cui le risorse da essi occupate diventino necessarie per altri scopi. L'obiettivo della secure deletion è prevenire l'accesso ai dati cancellati da parte di un avversario. Per questo motivo, l'attività di secure deletion è parte integrante parte della gestione della protezione e riservatezza dei dati.

Questa tesi esamina il problema della secure deletion in diversi sistemi e a diversi livelli: dal livello hardware, dove è necessario avere dispositivi di memorizzazione che possano effettuare la secure deletion in modo efficiente, al livello di sistema, in cui è necessario avere soluzioni che offrono capacità di secure deletion utilizzando server non affidabili. Questo studio presenta una dettagliata analisi delle soluzioni di secure deletion esistenti, ne identifica le mancanze ed i problemi non risolti, e presenta nuove tecniche di secure deletion che migliorano lo stato dell'arte. La nostra analisi presenta un framework per comprendere ed analizzare il problema della secure deletion. Le soluzioni esistenti sono organizzate in base a come queste si interfacciano ai dispositivi di memorizzazione fisici ed in base alle loro caratteristiche. Lo studio presenta anche una tassionomia dei vari adversarial model considerati fino ad ora. Infine, proponiamo un nuovo system model ed un nuovo adversarial model per il problema della secure deletion. Questi modelli considerano gli aspetti più complessi del problema. Tutti i risultati e le soluzioni presentate nella tesi suno esaminati nel contesto di questi modelli.

Questo studio considera il problema della secure deletion in due tipi di dispositivi di memoria: i dispositivi mobili ed i dispositivi di memorizzazione remota (per esempio, i servizi di cloud storage). La

mancanza di metodi di secure deletion è un problema a livello di sicurezza in entrambi gli scenari. Infatti, entrambi i dispositivi possono essere utilizzati per memorizzare dati riservati ed entrambi possono essere compromessi da un avversario. Nonostante le molteplici differenze tra questi dispositivi, le sfide da affrontare in termini di secure deletion sono simili.

In ambito mobile, è necessario effettuare la secure deletion sulle memorie flash, in quanto questo tipo di memorie è estremamente diffuso nei dispositivi mobile. Per le memorie flash, nel caso l'unità di cancellazione sia molto più grande dell'unità di lettura e scrittura si hanno maggiori costi in termini di consumo energetico e usura fisica dei dispositivi.

Il nostro primo contributo in questo ambito analizza la secure deletion a livello utente per le memorie flash, cioè, come un utente possa acquisire capacità di secure deletion semplicemente installando un'applicazione sul proprio dispositivo mobile. Come esempio concreto, abbiamo analizzato uno smartphone con sistema operativo Android. Esso non da nessuna garanzia sulla cancellazione dei dati, e il tempo di permanenza dei dati nella memoria del dispositivo dipende dalla capacità di memorizzazione dello stesso. Per risolvere questo problema, proponiamo due soluzioni che forniscono secure deletion, oltre ad una terza soluzione che combina le due precendenti e garantisce la periodica cancellazione dei dati indipendentemente dalla dimensione della memoria.

Il nostro secondo contributo nell'ambito delle memorie flash è DNEFS, un'estensione per file system che implementa la secure deletion in modo efficiente ed ad elevato livello di precisione. DNEFS è stato implementato nel file system per memorie flash UBIFS. UBIFSec è la versione di UBIFS che supporta DNEFS. Abbiamo integrato UBIFSec nel sistema operativo Android su di un dispositivo Google Nexus One. Attraverso alcuni esperimenti, mostriamo che UBIFSec è sia efficiente; Android OS e varie applicazioni (incluse applicazioni di riproduzione audio e video) vengono eseguite normalmente.

Per ottenere la secure deletion nell'ambito dei dispositivi di memorizzazione remota è necessario cancellare dati da memorie persistenti, che, per loro stessa natura, non sono in grado di cancellare dati. Il motivo che ci ha spinti a modellare in questo modo la memorizzazione di dati tramite servizi cloud è che gli utenti, una volta che i dati non sono più in loro posseso, non sono in grado di controllarne l'accesso da parte di terzi e la loro cancellazione. Per compensare l'incapacità di

cancellare dati delle memorie persistenti, assumiamo che l'utente abbia accesso a un dispositivo, di capacità limitata, che offre secure deletion, per gestire le chiavi di cifratura.

Il nostro primo contributo nell'ambito delle memorie persistenti è una soluzione generale per la progettazione e l'analisi di tecniche di secure deletion basate sulla cifratura e sul key wrapping. Tramite un key disclosure graph modelliamo la conoscenza, da parte dell'avversario, della sequenza di generazione e wrapping delle chiavi di cifratura. Proponiamo quindi una funzione di update generica, e dimostriamo che essa garantisce la secure deletion. Tale funzione di update è stata poi specializzata ed implementata per tutte le strutture dati ad albero. Tramite questo modello, è possibile rappresentare le altre tecniche di secure deletion esistenti, ed è possibile caratterizare i loro key disclosure graph. Infine, abbiamo implementato una soluzione che offre secure deletion per B-alberi ed abbiamo analizzato le sue prestazioni, trovando che i costi aggiuntivi dovuti alla secure deletion sono minimi.

Il nostro secondo controbuto considera un dispositivo di memoria non affidabile che offre capacità di secure deletion, cioè, un dispositivo che può inavvertitamente cancellare i dati, esporli a terze parti, non cancellari correttamente, o risultare inattivo. Proponiamo un sistema affidabile realizzato utilizzando vari dispositivi non affidabili. Il sisteme permette a diversi utenti di memorizzare dati e da modo ad essi di controllare vari aspetti della memorizzazione e della cancellazione. Illustriamo vari dettagli sia sull'implementazione del sistema di memorizzazione distribuito che su di un'estensione per file system che utilizza tale sistema. La nostra soluzione ha bassa latenza ad alti carichi di lavoro e richiede solo una minima quantità di comunicazione tra i vari nodi.

Acknowledgments

Foremost I wish to give my deepest thanks to my supervisors Srdjan Čapkun and David Basin for their education, support and guidance in developing this work. I have learned a great deal about all facets of the nature of science, in particular how to effectively do and communicate it with both the community and the world at large, and for that I am sincerely grateful. I wish to further extend this gratitude to my Master's advisor Ian Goldberg, who took me as his first student and helped begin my academic career.

I wish to express my appreciation and gratitude for my external committee members—Ari Juels, Paul Van Oorschot, and Alessandro Sorniotti—for taking the time to read through this thesis, providing helpful comments, crucial criticism, and for giving me the opportunity to defend my work.

The work presented in this thesis owes a large debt of gratitude to my collaborators who shared greatly in its creation; this includes both my doctoral advisers as well as Claudio Marforio, Alina Oprea, and Hubert Ritzdorf.

I further wish to acknowledge my funders whose financial support facilitated this work. I was partially supported by the Zurich Information Security Center and by the SNF grant Sinergia Project No. CR-SII2_136318/1. Their generous support is greatly appreciated.

I wish to extend my heartfelt gratitude to all those who commented on drafts of my papers, chapters, and presentations from which this thesis is composed. In alphabetical order (and with great apologies if one's mention is neglected), these helpful individuals are Steven Dropsho, Clemens Fruhwirth, Arthur Gervais, Marco Guarnieri, Nikolaos Karapanos, Gregor Kasieczka, Kari Kostiainen, René Kyllingstad, Luka Malisa, Grgur Maretic, Claudio Marforio, Ognjen Maric, Srdjan Marinovic, Ramya Jayaram Masti, Christina Pöpper, Aanjhan Ranganathan, Hubert Ritzdorf, Claudio Soriente, Nils Ole Tippenhauer, Thomas Themel, Petar Tsankov, Thilo Weghorn, and Der-Yeuan Yu.

I want to further thank Arthur Gervais for providing the French translation of my abstract, Thilo Weghorn for providing the German translation, and Marco Guarnieri for providing the Italian translation. I owe thanks to Abel Groenewolt who greatly improved my figures for clarity, consistency, and symmetry. I owe particular thanks to Thomas Themel for editing nearly all the final drafts of my paper submissions.

Deserving of particularly special praise is our research institute's incredible and dynamic group secretary Barbara Pfändner, who so easily and graciously handles the countless details and bureaucratic issues—both academic and governmental—under which I would have easily drowned.

Additionally, I offer thanks and amazement at the free and open movements in software and knowledge. Nearly every aspect of this thesis involves free software, from the compiling the code for experiments to producing the final copy of this text. Wikipedia further provides an invaluable service as a tertiary source—a term I earlier learned and later confirmed from Wikipedia. I am in awe of the overwhelming social benefit of these movements.

Three individuals to whom I wish to extend a deep personal gratitude are my parents Eric and Barbara Reardon, and Joanna Bolechowska, whose love, support, and encouragement during this undertaking were of great influence and importance. I would not have made it this far without such loving parents, and a great deal of the most creative part of this thesis would not have been possible were it not for those moonlit and morning train rides to and from Grindelwald, Grimmialp, and Engelberg.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

The scope of this thesis is secure *data* deletion. We say that *data is securely deleted from a system if an adversary that is given some manner of access to the system is not able to recover the deleted data from the system.* This irrecoverability is what distinguishes *secure* deletion from *regular* deletion, which deletes data for any other reason, e.g., to reclaim wasted storage resources. We *securely* delete data to prevent an adversary from gaining access to the data—to keep the data private—and so secure data deletion is a natural component in the confidential storage of data.

The problem of secure deletion is known in the scientific literature, which has given the problem many names. Thus we hear of data being assuredly deleted [1–3], completely removed [2, 4], deleted [5], destroyed [1,4,6], erased [4–7], expunged [8], forgotten [1,4], permanently erased [9], purged [3,10], reliably removed [6], revoked [2,4], sanitized [6, 10], self destructed [11, 12], and, of course, securely deleted [13, 14]. Whether explicitly stated as a system requirement or implicitly assumed, and however named, the ability to securely delete data in the presence of an adversary is required for the security of many systems.

In the physical world, the importance of secure deletion is well understood: paper shredders are single-purpose devices widely deployed in offices; published government information is selectively redacted; access to top secret documents is managed to ensure all copies can be destroyed. In the digital world, the importance of secure deletion is also well recognized. Legislative or corporate requirements, particularly relating to privileged or confidential communications, may require secure deletion to avoid the disclosure of sensitive data after physical medium disposal [15, 16]. Regulations may change, or new ones enforced, turning data assets into data liabilities. This results in an immediate need to securely delete a significant amount of data, exemplified by Google's illegal collection of wireless network data [17]. This is hard to achieve after the fact if the manner of the data's storage was not designed to accommodate secure deletion.

NIST standardized secure deletion best practices across a wide variety of storage media [10]. NIST's techniques are designed to securely delete the entire storage medium, and some techniques result in the medium's physical destruction: incineration, pulverization, etc. Destruction is an acceptable method if the storage medium need not to be reused and the owner is provided sufficient time and warning to perform the secure deletion before surrendering the storage medium (or its leftover pieces).

NIST's techniques are not a panacea as secure deletion is not limited to one-off events. Instead, it is common that secure deletion is required on a continuous basis while the main system continues functioning. Mobile phones and other portable devices store a wide range of sensitive information, including the timestamped sequence of nearby wireless networks that effectively encodes the user's location history. Mozilla Firefox [18], Google Chrome [19], and Apple Safari [20] all label clearing the web browsing history as a *privacy option.* Mobile phones can delete individual text messages, clear call logs, delete photographs, etc. Corporate e-mails include a boilerplate footnote demanding the immediate deletion of the message if sent to an unintended recipient. Network services with a privacy-focused objective (such as an anonymous message board, mix network [21], or Tor [22] relay) collect log data for intrusion detection or administrative purposes, but wish to securely delete this data as soon as it is longer needed to minimize the privacy risks to users. Network services also need secure deletion simply to comply with regulations regarding their users' private data. Two examples are the European Union's *right to be forgotten* [23] that forces companies that store personal data to do so in a manner that supports the secure deletion of all data about a particular user upon request, and California's legislation that enforces similar requirements for minors [24]. Thus we see a broad need for secure deletion.

Secure deletion seems simple: all modern file systems allow users to "delete" their files. Users of these systems may reasonably, but falsely, assume that when they delete the data, it is thenceforth irrecoverable [25]. Instead, "deletion" is a warning and not a promise. File systems implement deletions by *unlinking* files and *discarding* data blocks. Unlinking a file only removes its reference from the file system to indicate that the file is now "deleted"; the file's full contents remain available. In practice, *secure* deletion is never the default in file system and storage medium design.

Garfinkel and Shelat [26] study secure deletion in practice. They include a forensic analysis of 158 used hard drives bought on the secondary market from 2000–2002. They discovered that even the most basic sanitization is rarely employed. The kinds of data they trivially found includes client documents from a law firm, database of mental health patients, draft manuscripts, financial transactions executed by a bank machine, and plenty of credit card numbers. With relatively little work and cost, they were able to recover an extraordinary amount of personal information.

Garfinkel and Shelat speculate a variety of reasons why this may be the case: a lack of knowledge, of training, of concern for the problem or the data, or of tools to perform secure deletion. It is worth noting that 52 of their hard drives were freshly formatted. It is unclear, however, whether or not this was done as an attempt at secure deletion. The authors note that in "many interviews, users said that they believed DOS and Windows format commands would properly remove all hard drive data." Formatting warns that all data become irrecoverable, but in reality it only writes to a negligible fraction of the storage medium. In our work we aim to provide secure deletion solutions that allows users to easily and efficiently securely delete their data on a continuous basis.

1.1 Scope

Abstractly, the user stores and operates on data items on a physical medium through an interface. *Data items* are addressable units of data; these include data blocks, database records, text messages, file metadata, entire files, entire archives, etc. The *physical medium* is any device capable of storing and retrieving these data items, such as a magnetic hard drive, a USB stick, or a piece of paper. The *interface* is how the user interacts with the physical medium; the interface offers functions to transform the user's data objects into a form suitable for storage on the physical medium. This transformation can also include operations such as encryption, error-correction, duplication, etc. Our work focuses on how such data items are stored on physical media and how to modify the interface and implementations to effect secure data deletion.

Our research contributions focus on secure deletion in the context of two types of storage media: mobile (e.g., smart phone) storage and remote (e.g., cloud) storage. These storage types represent a significant shift in how people access their data, and they have a concerning lack of secure deletion. Both environments store sensitive user data and both are vulnerable to adversarial compromise. Despite the massive difference in scale, the problem and challenges of secure deletion shares surprising similarities, as we shall see. Our goal is to build efficient secure deletion solutions for these settings.

The main assumptions that define our scope are that the contents of data items are independent of where it, and other items, are or were stored, and independent of what is or was stored at other storage positions. As an example, we assume that redacting text from a document is effective at securely deleting the redacted text, and our focus is on developing methods to perform this redaction. This is not always true: file systems store copies of data as temporary files, and functions computed over data content can unexpectedly appear as metadata. The redacted document may be filled in by context or copies. Despite this, secure data deletion as a primitive operation is needed to delete copies and mutations of data or metadata. Our scope is this secure data deletion.

Before beginning on the topics actually covered by this thesis, we wish to list related lines of research that are not covered in this thesis. Some of these are orthogonal to our work, others are higher-level concepts that build on secure data deletion primitive.

- *Information Deletion*: Multiple data items may store the same information; perhaps by being copies or because of a transformation [27]. We do not consider deleting all copies of some information, or finding and deleting all derivative works of data.

- *File Carving, Forensics, and Anti-forensics*: Rebuilding files from a partially-broken disk image is non-trivial [28–31]. We assume that an adversary capable of recovering all pieces of a data item can thus recover the data item itself.

- *Steganographic Storage*: In some environments, users may desire to hide any evidence data was stored at some position [32, 33] or that a secure deletion tool was used to delete data. We assume that knowledge that a position once stored data or that secure deletion tools were used provides no information about the data it contained.

- *History Independence*: Data structures, and file systems that use them, can store data in different positions depending on the state of the system when the data was written. History independence requires that at all times, the storage location of some data depends only on the current valid data and is therefore independent of the history (e.g., previously deleted data) [34]. We do not consider history independence for storage locations.

- *Metadata*: Metadata is data about data, which in many cases must be securely deleted as well. In addition to file system metadata (e.g., file names and access times), other metadata may include functions computed from the data itself: a list of frequent keywords, a hash of the file's data, or a reverse index of a document's words [35]. Metadata can itself be the target of secure data deletion and we assume that it is possible to identify relevant metadata when deleting data.

- *Usable Deletion*: Unlike paper files, digital files often store much more information than what is presented to the user. User misuse of tools may result in data appearing to be deleted when printed but remaining hidden in the digital copy, e.g., by changing the text's colour to white or placing a rectangle over text [36].

- *Usage Control*: The owner of data may wish to provide access to others while being able to force its subsequent deletion. We do not consider cases where those accessing data attempt to circumvent deletion requirements by taking data out of a controlling system; nor do we consider preventing leaks of data. Note that this problem is similar to digital rights management.

- *Provable Deletion*: A compromised storage medium may fake the execution of secure deletion: a secret reserved area of the storage may contain sensitive data inaccessible to anyone but the adversary. We do not consider methods to have the storage medium prove to a verifier that it has actually deleted the data [37].

- *Conspicuous through Absence*: One can contrive a case where deleting data has the effect of actually exposing it: store all possible values of a given length (e.g., all 16 digit numbers) and then securely delete particular numbers (e.g., valid credit card numbers). We assume that the contents of data items are independent of other data items stored on the storage medium.
- *Malicious Deletion*: Cryptoviral extortion uses malware to attack victims' systems, encrypt their files, and then extort a ransom to return access [38]. Of course, encrypting the files entailed writing a new, encrypted version and discarding the old version; comprehensive, automatic secure data deletion in this case is detrimental to the user. We assume that users' deployments of secure deletion and the data they delete are intentional.

While many of these related topics have interesting results and open questions, this thesis does not explore any of them further.

1.2 Contributions and Publications

The contributions of this thesis are the following:

- We organize the space of adversaries, environmental assumptions, and behavioural properties across the related work in secure deletion to focus requirements for new solutions.
- We show when it is possible to securely delete data on flash memory from user space.
- We design the Data Node Encrypted File System: a generic file system extension to provide efficient secure deletion for flash memory, which we experimentally validate.
- We propose an intuitive model that captures the growth of adversarial knowledge in secure deletion systems and prove the security for a wide space of solutions.
- We design, implement, and analyse a caching B-Tree, which is an instance in this space of solutions.
- We design a distributed securely-deleting storage medium that is robust against partial failures in availability, confidentiality, and integrity of data.

Much of work presented in this thesis is based on the following co-authored articles:

1. Joel Reardon, Claudio Marforio, Srdjan Capkun, and David Basin, "User-level Secure Deletion on Log-structured File Systems", *In Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2012

2. Joel Reardon, Srdjan Capkun, and David Basin, "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory", *In Proceedings of the USENIX Security Symposium*, 2012, pp. 333–348

3. Joel Reardon, David Basin, and Srdjan Capkun, "SoK: Secure Data Deletion", *In Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 301–315

4. Joel Reardon, Hubert Ritzdorf, David Basin, and Srdjan Capkun, "Secure Data Deletion from Persistent Media", *In Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 271–284

5. Joel Reardon, David Basin, and Srdjan Capkun, "On Secure Data Deletion", *In IEEE Security & Privacy Magazine*, May 2014, pp. 37–44

6. Joel Reardon, Alina Oprea, David Basin, and Srdjan Capkun, "Robust Key Management for Secure Data Deletion", 2014

1.3 Organization and Structure

This thesis is organized into four parts and twelve chapters.

Chapter 2 surveys related work and organizes existing solutions in terms of their interfaces. The chapter further presents a taxonomy of adversaries differing in their capabilities as well as a systematization of the characteristics of secure deletion solutions. Characteristics include environmental assumptions and behavioural properties of the solution.

Chapter 3 builds a system and adversarial model based on the survey of related work. It describes the model that we use throughout this thesis. It also presents different types of storage media and illustrates the adversary's abilities and the user's goal.

Chapter 4 opens the part on secure deletion for mobile storage. It first presents details on the characteristics of flash memory, which is currently ubiquitously used in portable storage devices. Flash memory has the problem that the unit of erasure is much larger than the unit of read and write, and worse, erasure is expensive. Related work for flash memory is presented in this chapter as well as generalizations of this erasure asymmetry to other kinds of media.

Chapter 5 then presents our research into user-level secure deletion for flash memory, with a concrete example of an Android-based mobile phone. We show that these systems provide no timely data deletion and the time data remains increases with the storage medium's size. We propose two user-level solutions that achieve secure deletion as well as a hybrid of them, which guarantees the periodic, prompt secure deletion of data regardless of the storage medium's size. We also develop a model of the writing behaviour on a mobile device that we use to quantify our solution's performance.

Chapter 6 presents DNEFS, a file system change that provides fine-grained secure data deletion and is particularly suited to flash memory. DNEFS encrypts each individual data item and colocates all the encryption keys in a densely-packed key storage area. DNEFS is efficient in flash memory erasures because the expensive erasure operation is only needed for the key storage area.

Chapter 7 presents UBIFSec, an implementation of DNEFS with the flash file system UBIFS. We describe our implementation and furthermore integrate UBIFSec in the Android operating system. We measure its erasures and show that it is usable. Android OS and applications run normally on top of UBIFSec.

Chapter 8 begins the part on secure deletion for remote storage. It first presents details on the characteristics of persistent storage, a model of a storage medium that is unable to provide any secure deletion of its stored data. After motivating its suitability for modelling remote storage, the chapter then presents a range of related work on the topic of secure deletion for persistent storage. Because no data can be deleted, work on this topic assumes that the user has access to a secondary securely-deleting storage medium but is unable to use it for storing all of their data.

Chapter 9 presents a general approach to the design and analysis of secure deletion for persistent storage that relies on encryption and key wrapping. It defines a key disclosure graph that models the adversarial knowledge over a history of key generation and wrapping. We define a generic update function, expressed as a graph mutation for the key disclosure graph, and prove that this update function achieves secure deletion. Instances of the update function implement the update behaviour of all tree-like data structures including B-Trees, extendible hash tables, linked lists, and others.

Chapter 10 presents a securely-deleting data structure using insights from the previous chapter. It uses a B-Tree-based data structure to provide secure deletion. We implement our design in full and analyze its performance, finding that its communication and storage overhead is small.

Chapter 11 considers the problem of an unreliable securely-deleting storage medium, that is, one that may lose data, expose data, fail to delete data, and fail to be available. We build a robust fault-tolerant system that uses multiple unreliable storage media. The system permits multiple clients to store securely-deletable data and provides a means to control policy aspects of its storage and deletion. It presents details on the implementation both of the distributed securely-deleting medium as well as a file system extension that uses it. The solution has low latency at high loads and requires only a small amount of communication among nodes.

Chapter 12 concludes the thesis. We review our contributions and integrate them into our systematization. We then outline avenues for future research. Finally, we draw conclusions and summarize our work.

Chapter 2

Related Work on Secure Deletion

2.1 Introduction

This chapter surveys related work and creates a common language of adversaries, behavioural properties, and environmental assumptions by which to compare and contrast related work. In the next chapter, these concepts are used to design the system and adversarial model that we use throughout this thesis.

The related work presented in this section provides a background. Further related work specific to secure deletion for flash memory and cloud storage are presented in Chapters 4.3 and 8.3, respectively.

2.2 Related Work

In this section, we organize related work by the layers through which they access the physical medium. When deciding on a secure deletion solution, one must consider both the interface given to the physical medium and the behaviour of the operations provided by that interface. For example, overwriting a file with zeros uses the file system interface, while destroying the medium uses the physical interface. The solutions available to achieve deletion depend on one's interface to the medium. Secure deletion is typically not implemented by adding a new interface to the physical medium, but instead it is implemented at some existing system layer (e.g., a file system) that offers an interface to the physical medium provided at that layer (e.g., a device driver). It is possible that an interface to a physical medium does not support an implementation of a secure deletion solution.

Once secure deletion is implemented at one layer, then the higher layers' interfaces can explicitly offer this functionality. Care must still be taken to ensure that the secure deletion solution has acceptable performance characteristics: some solutions can be inefficient, cause significant wear, or delete *all* data on the physical medium. These properties are discussed in greater detail in Section 2.4. For now, we first describe the layers and interfaces involved in accessing magnetic hard drives, flash memory, and network file systems on personal computers. We then explain why there is no one layer that is always the ideal candidate for secure deletion. Afterwards, we present related work in secure deletion organized by the layer in which the solution is integrated.

2.2.1 Layers and Interfaces

Many abstraction layers exist between applications that delete data items and the physical medium that stores the data items. While there is no standard sequence of layers that encompass all interfaces to all physical media, Figure 2.1 shows the typical ways of accessing flash, magnetic, and networked storage media on a personal computer.

Physical. Except for data stored on network file systems, the lowest layer is always the physical medium itself. Its interface is also physical: depending on the medium it can be degaussed, incinerated, or shredded. Additionally, whatever mechanism controls its operation can be replaced with an ad hoc one; for example, flash memory is often accessed through an obfuscating controller, but the raw memory can still be directly accessed by attaching it to a custom reader [39].

Controller. The physical medium is accessed through a controller. The controller is responsible for translating the data format on the physical media (e.g., electrical voltage) into a format suitable for higher layers (e.g., binary values). Controllers offer a standardized, well-defined, hardware interface, such as SCSI or ATA [40], which allow reading and writing to logical fixed-size blocks on the physical medium.

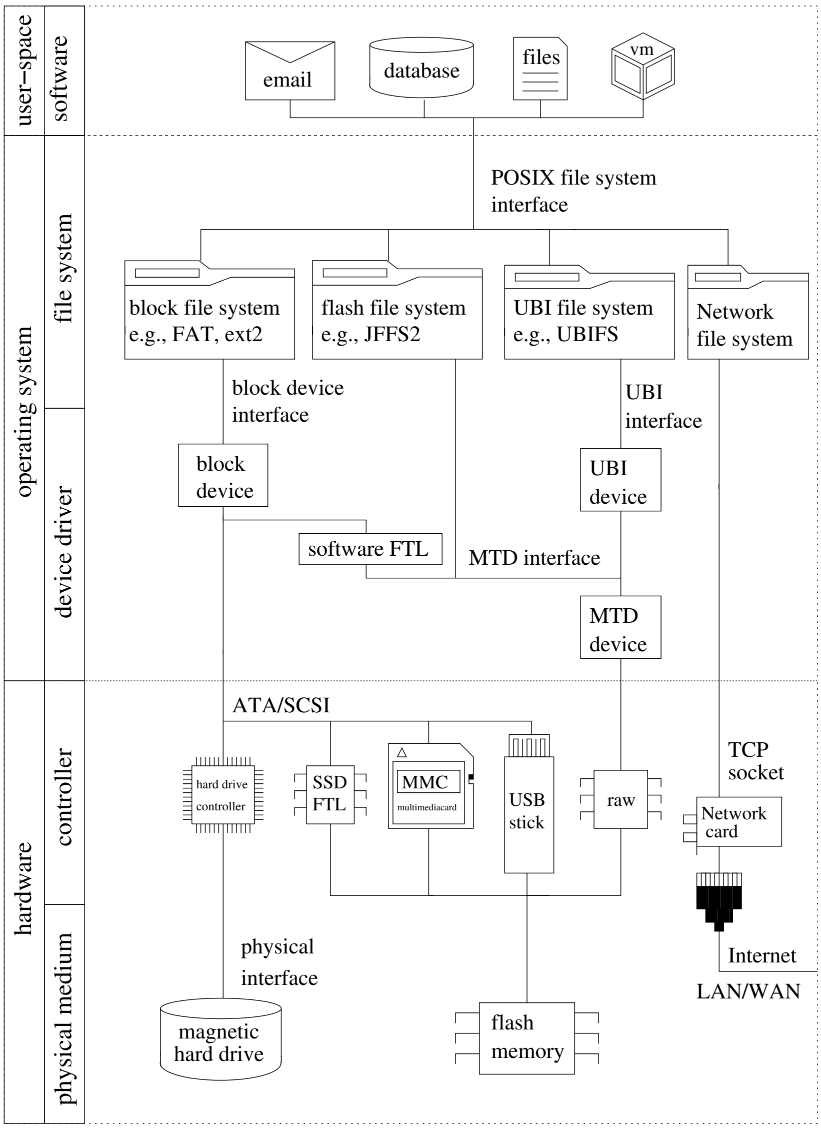


Figure 2.1: Interfaces and layers involved in magnetic hard drives, flash memory, and remote data storage.

They may also offer a secure erase command that securely deletes *all* data on the physical device [41]. Like physical destruction, this command cannot be used to securely delete some data while retaining other data; we revisit secure deletion *granularity* later in this chapter.

While hard disk controllers consistently map each logical block to some storage location on the physical medium, the behaviour of other controllers differs. Flash memory does not permit in-place updates and instead data is logically remapped. When raw flash memory is accessed directly, a different controller interface is exposed. For convenience, flash memory is often accessed through a flash translation layer (FTL) controller, whose interface mimics that of a hard drive. FTLs remap logical block addresses to physical locations such that overwriting an old location does not replace it but instead results in two versions, with obvious complications for secure deletion. FTLs are used in solid-state drives (SSDs), USB sticks, and multimedia cards (MMCs).

Device Driver. Device drivers are software abstractions that consolidate access to different types of hardware by exposing a common simple interface. The block device driver interface allows the reading and writing of logically-addressed blocks. Another device driver—the memory technology device (MTD)—is used to access raw flash memory directly. MTD permits reading and writing, but blocks must be *erased* before being written, and erasing blocks occurs at a large granularity. Unsorted block images (UBI) is another interface for accessing flash memory, which builds upon the MTD interface and simplifies some aspects of using raw flash memory [42].

File System. The device driver interface is used by the file system, which is responsible for organizing logical sequences of data (files) among the available blocks on the physical medium. A file system allows files to be read, written, created, and unlinked. While secure deletion is not a feature of this interface, file systems do keep track of data that is no longer needed. Whenever a file is unlinked, truncated, or overwritten, this is recorded by the file system. The POSIX standard is ubiquitously used as the interface to file systems [43], and the operating system restricts this interface further with access control.

User Interface. Finally, the highest layer is user applications. These offer an interface to the user that is manipulated by devices such as keyboards and mice. Secure deletion at this layer can be integrated into existing applications, such as a database client with a secure deletion option, or it can be a stand-alone tool that securely deletes all deleted data on the file system.

Choosing a Layer. The choice of layer for a secure-deletion solution is a trade off between two factors. At the physical layer, we can ensure that the data is truly irrecoverable. At the user layer, we can easily identify the data item to make irrecoverable. Let us consider these factors in more detail.

Each new abstraction layer impedes direct access to the physical medium, thus complicating secure deletion. The controller may write new data, but the physical medium retains remnants; the file system may overwrite a logical block, but the device driver remaps it physically. The further one's interface is abstracted away from the physical medium, the more difficult it is to ensure that one's actions truly result in the irrecoverability of data.

High-layer solutions most easily identify which data items to delete, e.g., by deleting an email or a file. Indirect information is given to the file system, e.g., by unlinking a file; no information is given to the device driver or controller. Assuming the user cannot identify the physical location of the deleted data item on the medium, then a solution integrated at low layers cannot identify where the deleted data item is located. Solutions implemented in the file system are usually well balanced in this trade off. When this layer is insufficient to achieve secure deletion, it is also possible to pass information on deleted data items from the file system down to lower layers [13, 44].

Organization. In the remainder of this section, we examine secure-deletion solutions at the different layers in Figure 2.1. First, we look at *device-level solutions* and *controller-level solutions*, which have no file system information and therefore securely delete all data on the physical medium. We then move to the other extreme and consider *user-level solutions*, which are easy to install and use, but are limited in their POSIX-level access to the physical medium and are often rendered ineffective by advanced file system features. Next, we look at *file-system-level solutions* for a variety of systems using in-place updates and thus are suitable for magnetic physical media. We conclude with several solutions that extend existing interfaces to allow information on deleted blocks to be sent to lower layers. We defer the survey of techniques suitable for flash memory to Chapter 4, and the survey of encryption-based techniques that are suitable for storing data on persistent storage (e.g., remote “cloud” storage) to Chapter 8.

2.2.2 Physical-Layer and Controller-Layer Sanitization

Physical Layer. The physical layer’s interface is the set of physical actions one can perform on the medium. Secure deletion at this layer often entails physical destruction, but the use of other tools such as degaussers is also feasible. NIST provides physical layer secure-deletion solutions suitable for a variety of physical media [10]. For example, destroying floppy disks requires shredding or incineration; destroying compact discs requires incineration or its subjection to an optical disk grinding device. Of course, not all solutions work for all media types. For example, most media’s physical interfaces permit the media to be put into an NSA/CSS-approved degausser, but this is only a secure deletion solution for particular media types. Magnetic media are securely deleted in this way, while others, such as flash memory, are not.

Controller Layer. Several standardized interfaces exist for controllers that permit reading and writing of fixed-size blocks. Given these interfaces, there are different actions one can take to securely delete data. Either a single block can be overwritten with a new value to displace the old one, or all blocks can be overwritten. As we noted earlier, with knowledge of neither deleted data nor the organization of data items into blocks, sanitizing a single block cannot guarantee that any particular data item is securely deleted. Therefore, the controller must sanitize every block to achieve secure deletion. Indeed, both SCSI and ATA offer such a sanitization command, called either *secure erase* or *security initialize* [41]. They work like a button that erases all data on the device by exhaustively overwriting every block. The use of these commands is encouraged by NIST as the non-destructive way to securely delete magnetic hard drives. The embedded multimedia card (eMMC) specification allows devices to offer a *sanitize* function. If implemented, it must perform the secure deletion of all data (and copies of data) belonging to *unmapped* storage locations, i.e., parts of the file system that have been previously marked for deletion.

An important caveat exists at the physical layer. Controllers translate analog values into binary values such that a range of analog values maps to a single binary value. Gutmann observed that, for magnetic media, the precise analog voltage of a stored bit offers insight into its previously held values [45]. Gutmann's solution to delete this data is also at the controller layer: the controller overwrites every block 35-times with specific patterns designed to ensure analog remnants are removed for all common data encoding formats. In an epilogue to this work, Gutmann remarks that the 35-pass solution had taken on the mystique of a "voodoo incantation to banish evil spirits", and restates that the reason there are so many passes is that it is the concatenation of the passes required for different data encoding formats; it is *never* necessary to perform all 35 for any specific hard drive.

While more recent research was unable to recover overwritten data on modern hard drives [46], it remains safe to say that each additional overwrite does not make the data easier to recover—in the worst case it simply provides no additional benefit [5]. Gutmann's epilogue states that it is unlikely anything can be recovered from modern drives after a couple passes with random data. More generally, Gutmann's results highlight that analog remnants introduced by the controller's use of the physical medium may exist for any storage media type and this must be considered when developing secure-deletion solutions.

2.2.3 User-Level Solutions

Device-level solutions interact at the lowest layer and securely delete all data, serving as a useful starting point in our systematization. Now we move to the other extreme, a securely-deleting user-level application that can only interact with a POSIX-compliant file system. There are three common user-level solutions: (i) ones that call a secure deletion routine in the physical medium's interface, (ii) ones that overwrite data before unlinking, and (iii) ones that first unlink and then fill the empty capacity of the physical medium.

Low-Layer Calls. Device drivers and other low-layer interfaces may expose to user-space special routines for secure deletion. This permits users to easily invoke such functionality without requiring special access to hardware or additional skills. Explicit low-layer calls propagate a secure-deletion solution to a higher-layer interface.

UCSD offers a free Secure Erase utility [41]. It is a user-level application that securely erases all data on a storage medium by invoking the Secure Erase command in the hardware controller's interface.

Similarly, Linux's MMC driver exposes to user-space an `ioctl`[1] that invokes the sanitization routine [47]. Therefore, applications can easily call the ioctl, which—if supported by the hardware—performs the secure deletion of all unmapped data on the storage medium.

File Overwriting Tools. Another class of user-level secure-deletion solutions opens up a file from user-space and *overwrites* its contents with new, insensitive data, e.g., all zeros. When the file is later unlinked, only the contents of the most-recent version are currently stored on the physical medium. To combat analog remnants, overwriting is performed multiple times; various tools [48,49] offer 35-pass overwriting as proposed by Gutmann [45].

Overwriting tools rely on the following file system property: each file block is stored at known locations and when the file block is updated, then all old versions are replaced with the new version. If this assumption is not satisfied, user-level overwriting tools silently fail. Moreover, they do nothing for larger files that were truncated at some time prior to running the tool. They also do nothing for file copies that are not unlinked with this tool.

Overwriting tools may also attempt to overwrite file system metadata, such as its name, size, and access times. The Linux tool `wipe` [49], for instance, also changes the file name and size in an attempt to securely delete this metadata. Note that not all types of metadata may be arbitrarily changed: the operating system's interface to the file system may not allow it, or simply changing the file's name, for example, may not securely delete the old one. The Linux tool `srm` [48] renames the file to a random value and truncates its size to zero after overwriting. Other attributes cannot be easily changed without higher privileges, e.g., the access times or the file's group and owner.

Overwriting tools operate on either an entire file or the entire physical medium. These tools do not handle operations such as overwrites and truncations, which discard data within a file without deleting the file. Though overwriting a file replaces the old data with the new data (or else such an overwriting tool is unsuitable), it does not perform

[1] An input/output control (ioctl) is a operating system function that permits low-level device-specific operations that are not available in a standard interface.

additional sanitization steps such as writing over the location multiple times. While it is possible to write a user-level tool that securely overwrites and truncates a file as well, it becomes the user's burden to ensure that all other applications make use of it.

This leads into the general problem of usability. The user must remember to use the tool whenever a sensitive file must be deleted, and to do this instead of their routine behaviour. Care must be taken to avoid applications that create and delete their own files [14]: a word processor that creates temporary swap files does not securely delete them with the appropriate tool; a near-exact copy is left available. If a file is copied, the copy too must be securely deleted. Neglecting to use the tool when deleting a file results in the inability to securely delete the file's data with this technique.

Free-Space Filling Tools. A file system has both valid and unused blocks. The set of unused blocks is a superset of the blocks containing deleted sensitive data. A third class of user-level secure-deletion tools exploits this fact by *filling* the entire free space of the file system. This ensures that *all* unused blocks of the physical medium no longer contain sensitive information and instead store filler material.

Filling solutions permit users and applications to take no special actions to securely delete data; any discarded data is later securely deleted by an intermittent filling operation. These tools also allow secure deletion for file systems that do not perform in-place updates of file data. Compared to the overwriting solutions, secure deletion through filling allows *per-block level* secure deletion (including truncations) without in-place updates at the cost of a periodic operation. It can only operate at *full scope*—all unused blocks are filled. Examples include Apple's Disk Utility's *erase free space* feature [50] and the open-source tool `scrub` [51].

The correct operation of a filling tool relies on two assumptions: the user who runs the tool must have sufficient privileges to fill the physical medium to capacity, and when the file system reports itself as unwritable it must no longer contain any deleted data. The useful deployment of these solutions therefore requires manual verification that these assumptions do hold.

In contrast to overwriting solutions, these correctness conditions are satisfied more often; this is because modern file systems use journalling when storing new data for crash-recovery purposes. Filling's assumptions do not always hold. Garfinkel and Malan examine secure deletion by filling for a variety of file systems and find mixed results [52]. One observation is that creating many small files helps securely delete data that is not deleted when creating one big file, which may be due to file systems not allocating heavily-fragmented areas for already large files.

The benefits of filling over overwriting are that the user is given secure deletion for all deleted data (including unmarked sensitive files and truncations) that works correctly for a larger set of file systems. Moreover, the user only needs to run the tool periodically to securely delete all accumulated deleted data: applications and user behaviour do not need to change with regards to file management. The trade off is that the filling operation is slow and cannot target specific files. It is a periodic operation that securely deletes all data blocks discarded within the last period. Since deletion is achieved by writing new data instead of overwriting the existing data, it does not perform in-place updates and is therefore suitable for additional file systems and physical medium types that do not permit such operations.

Database Secure Deletion. Databases such as MySQL [53] and SQLite [54] store an entire database as a single file within a file system [55]; databases are analogous to file systems, where records can be added, removed, and updated. This adds a new interface layer for users wanting to delete entries from a database. Database files are long lived on a system, however the data they contain may reside within it very shortly. Many applications store sensitive user data (e.g., emails and text messages) in databases; secure deletion of such data from databases is therefore important.

Both MySQL and SQLite have secure-deletion features. In both cases, the interface for secure deletion is the underlying file system and secure deletion is implemented by overwriting the data with zeros. For MySQL, researchers propose a solution where deleted entries are overwritten with zeros, and the transaction log (used to recover after a crash) is encrypted and securely deleted by deleting the encryption key [55]. For SQLite, there is an optional secure-deletion feature that overwrites deleted database records with zeros [56].

As previously discussed, overwriting blocks with zeros is one way to inform the file system that these blocks are unneeded; necessary, but not sufficient, to achieve secure deletion. SQLite's solution relies on the file system "below" to ensure that overwritten data results in its secure deletion. When the interface does not explicitly offer secure deletion, then it is—at the minimum—necessary to tell the interface that the data is discarded.

2.2.4 File-System-Level Solutions with In-Place Updates

The utility of user-level solutions is hampered by the lack of direct access to the physical medium. Device-level solutions suffer from being generally unable to distinguish deleted data from valid data given that they lack the semantics of the file system. We now look at secure-deletion solutions integrated in the file system itself, that is, solutions that access the physical medium using the device driver interface.

Here we consider only solutions that use in-place updates to achieve secure deletion. An in-place update means that the device driver replaces a location on the physical medium with new content. Not all device drivers offer this in their interface, primarily because not all physical media support in-place updates. The assumption that in-place updates occur is valid for block device drivers that access magnetic hard drives and floppy disks. Solutions for flash memory cannot use in-place updates, and Section 4.3 discusses this in more detail.

Secure Deletion for ext2. The second extended file system ext2 [57] for Linux offers a sensitive attribute for files and directories to indicate that secure deletion should be used when deleting the file. While the actual feature was never implemented by the core developers, researchers provided a patch that implements it [14].

Their patch changed the functionality that marks a block as free. It passes freed blocks to a kernel daemon that maintains a list of blocks that must be sanitized. If the free block corresponds to a sensitive file, then the block is added to the work queue instead of being returned to the file system as an empty block. The work queue is sorted to minimize seek times imposed by random access on magnetic media.

The sanitization daemon runs asynchronously, performing sanitization when the system is idle, allowing the user to perceive immediate file deletion. The actual sanitization method used is configurable, from a simple overwrite to repeated overwrites in accordance with various standards.

Secure Deletion for ext3. The third extended file system ext3 [58] succeeded ext2 as the main Linux file system and extended it with a write journal: all data is first written into a journal before being committed to main storage. This improves consistent state recovery after unexpected losses of power by only needing to inspect the journal's recent changes.

Joukov et al. [59] provide two secure-deletion solutions for ext3. Their first solution is a small change that provides secure deletion of file data by overwriting it once, which they call *ext3 basic*. Their second solution, *ext3 comprehensive*, provides secure deletion of file data and file metadata by overwriting it using a configurable overwriting scheme, such as the 35-pass Gutmann solution. They both provide secure deletion for all data or just those files whose extended attributes include a sensitive flag.

Secure Deletion via Renaming. Joukov et al. [59] present another secure-deletion solution through a file system extension, which can be integrated into many existing file systems [60]. Their extension intercepts file system events relevant for secure deletion: unlinking a file and truncating a file. (They assume overwrites occur in place and are not influenced by a journal or log-structured file system.) For unlinking, which corresponds to regular file deletion, their solution instead moves the file into a special secure-deletion directory. For truncation, the resulting truncated file is first copied to a new location and the older, larger file is then moved to the special secure-deletion directory. Thus, for truncations, their solution must always process the entire file—not just the truncated component. At regular intervals, a background process runs the user-level tool `shred` [61] on all the files in the secure deletion directory.

Purgefs. Purgefs is another file system extension that adds secure deletion to any block-based file system [62]. It uses block-based overwriting when blocks are returned to the file system's free list, similar to the solution used for ext2. It supports overwriting file data and file metadata for all files or just files marked as sensitive. Purgefs is implemented as a generic file system extension, which can be combined with any block-based file system to create a new file system that offers secure deletion.

Secure Deletion for a Versioning File System. A versioning file system shares file data blocks among many versions of a file; one cannot overwrite the data of a particular block without destroying all versions that share that block. Moreover, user-level solutions such as overwriting the file fail to securely delete data because all file modifications are implemented using a copy-on-write semantics [63]—a copy of the file is made (sharing as many blocks as possible with older versions) with a new version for the block now containing only zeros.

Peterson et al. [64] use a cryptographic solution to optimize secure deletion for versioning file systems. They use an all-or-nothing cryptographic transformation [65] to expand each data block into an encrypted data block along with a small key-sized tag that is required to decrypt the data. If any part of the ciphertext is deleted—either the tag or the message—then the entire message is undecipherable. Each time a block is shared with a new version, a new tag is created and stored for that version. Tags are stored sequentially for each file in a separate area of the file system to simplify sequential access to the file under the assumption that a magnetic-disk drive imposes high seek penalties for random access. A specific version of a file can be quickly deleted by overwriting all of that version's tags. Moreover, *all* versions of a particular data block can easily be securely deleted by overwriting the encrypted data block itself.

2.2.5 Cross-layer Solutions

There are solutions that pass information on discarded data down through the layers, permitting the use of efficient low-layer secure deletion solutions.

Data items contained in a file are discarded from a file system in three ways: by unlinking the file, by truncating the file past the block, and by updating the data item's value. The information about data blocks that are discarded when unlinking or truncating files, however, remains only known by the file system. The device-driver layer can only infer the obsolescence of an old block when its logical address is overwritten with a new value. Here we present two solutions by which the file system passes information on discarded blocks to the device driver: TRIM commands [44] and TrueErase [13]. In both cases, the file system informs the device that particular blocks are discarded, i.e., no longer needed for the file system. With this information, the device driver can implement its own efficient secure deletion without requiring data blocks to be explicitly overwritten by the file system.

TRIM commands are notifications issued from the file system to the device driver to inform the latter about discarded data blocks. TRIM commands were not designed for secure deletion but instead as an efficiency optimization for flash-based physical media. Nevertheless, there is no reason that a device driver cannot use information from TRIM commands to perform secure deletion: TRIM commands indicate every time a block is discarded—there are no false negatives. It is not possible to restrict TRIM commands only to sensitive blocks, which means that it must be an efficient underlying mechanism that securely deletes the data.

Diesburg et al. propose TrueErase [13], which provides similar information as TRIM commands but only for blocks belonging to files specifically marked as sensitive. Users may trivially set all files to sensitive, or use traditional permission semantics to manage file sensitivity. TrueErase adds a new communication channel between the file system and the device driver that forwards from the former to the latter information on sensitive blocks deleted from the file system. Device drivers are modified to implement immediate secure deletion when provided a deleted block; the device driver is thus able to correctly implement secure deletion using its lower-layer interface with the high-layer information on what needs to be deleted; this is more efficient than TRIM commands, which would require deletion for all data. This comes at the risk of false negative in the event that a user neglects to correctly set a file's sensitivity.

2.2.6 Summary

This concludes our survey of selected related work on secure deletion. Further related work specific to flash memory and persistent memory appear in Chapters 4 and 8, respectively. We saw that physical media can be accessed from a variety of layers and that different layers provide different interfaces for secure deletion. In low-layer solutions, fewer assumptions must be made about the interface's behaviour, while in high-layer solutions the user can most clearly mark which data items to delete. For device-level solutions, we discussed different ways the entire device can be sanitized. User-level secure deletion considers how to securely delete data using a POSIX-compliant file system interface. Secure deletion in the file system must use the device driver's interface for the physical medium, and we surveyed solutions that assume the device driver performs in-place updates. For physical media that do not have an erasure operation, physical destruction is the only means to achieve secure deletion.

In the next two sections, we systematize the space of secure-deletion solutions. We first review adversarial models and afterwards compare the characteristics of existing solutions.

2.3 Adversarial Model

Secure-deletion solutions must be evaluated with respect to an adversary. The adversary's goal is to recover deleted data items after being given some access to a physical medium that contained some representation of the data items. In this section, we present the secure-deletion adversaries. We develop our adversarial model by abstracting from real-world situations in which secure deletion is relevant, and identifying the classes of adversarial capabilities characterizing these situations. Table 2.1 then presents a variety of real-world adversaries systematized by their capabilities.

2.3.1 Classes of Adversarial Capabilities

Attack Surface. The attack surface is the physical medium's interface given to the adversary. If deletion is performed securely, data items should be irrecoverable to an adversary who has unlimited use of the provided interface. NIST divides the attack surface into two categories: *robust-keyboard attacks* and *laboratory attacks* [10]. Robust-keyboard attacks are software attacks: the adversary acts as a device driver and accesses the storage medium through the controller. Laboratory attacks are hardware attacks: the adversary accesses the physical medium through its physical interface. As we have seen, the physical layer may have analog remnants of past data inaccessible at any other layer. While these two surfaces are widely considered in related work, we emphasize that any interface to the physical medium can be a valid attack surface for the adversary.

Access Time. The access time is the time when the adversary obtains access to the medium. Many secure-deletion solutions require performing extraordinary sanitization methods before the adversary is given access to the physical medium. If the access time is unpredictable, the user must rely on secure deletion provided by sanitization methods executed as a matter of routine.

The access time is divided in two categories: predictable (or user controlled) and unpredictable (or adversary controlled). If the access time is predictable, then the user can use the physical medium normally and perform as many sanitization procedures as desired before providing it to the adversary. If the access time is unpredictable then we do not permit any extraordinary sanitization methods to be executed: the secure-deletion solution must rely on some immediate or intermittent sanitization operation that limits the duration that deleted data remains available.

Number of Accesses. Nearly all secure-deletion solutions consider an adversary who accesses a physical medium some time after securely deleting the data. One may also consider an adversary who access the storage medium multiple times—accessing the physical medium before the data is written as well as after it is deleted.

We therefore differentiate between single- and multiple-access adversaries. An example single-access adversary corresponds to the scenario when a used device is sold on the market; a multiple-access adversary is someone who, for example, deploys malware on a target machine multiple times because it is discovered and cleaned, or someone who obtains surreptitious periodic access (e.g., nightly access) to a storage medium.

Credential Revelation. Encrypting data makes it immediately irrecoverable to an adversary that neither has the encryption key (or user passphrase) nor can decrypt data without the corresponding key. There are many situations, however, where the adversary is given this information: a legal subpoena, border crossing, or information taken from the user through duress. In these cases, encrypting data is insufficient to achieve secure deletion.

We partition the credential revelation into non-coercive and coercive adversaries. A non-coercive adversary does not obtain the user's passwords and the credentials that protect the data on the physical medium. A coercive adversary, in contrast, obtains this information. It may also be useful to consider a weak-password adversary who can obtain the user's password by guessing, by the device not being in a locked state, or by a cold-boot attack [66]. This adversary, however, is unable to obtain secrets such as the user's long-term signing key or the value stored on a two-factor authentication token.

Computational Bound. Many secure-deletion solutions rely on encrypting data items and only storing their encrypted form on the medium. The data is made irrecoverable by securely deleting the decryption key. The security of such solutions must assume that the adversary is computationally bounded to prevent breaking the cryptography.

We distinguish between computationally bounded and unbounded adversaries. There is a wealth of adversarial bounds corresponding to a spectrum of non-equivalent computational hardness problems, so others may benefit from dividing this spectrum further.

2.3.2 Summary

Adversaries are defined by their capabilities. Table 2.1 presents a subset of the combinatorial space of adversaries that correspond to real-world adversaries. The *name* column gives a name for the adversary, taken from related work when possible; this adversarial name is later used in Table 2.3 (and much later in Table 12.1) when describing the adversary a solution defeats. The second through fifth columns correspond to the classes of capabilities defined in this section. Table 2.2 provides a real-world example where each of the adversaries from Table 2.1 may be found. For instance, while computationally-unbounded adversaries do not really exist, the consideration of such an adversary may reflect a corporate policy on the export of sensitive data or a risk analysis of a potentially-broken cryptographic scheme.

Observe that each class of adversarial capabilities is *ordered* based on adversarial strength: lower-layer adversaries get richer data from the physical medium, coercive adversaries get passwords in addition to the physical medium, and an adversary who controls the disclosure time can prevent the user from performing an additional extraordinary effort to achieve secure deletion. This yields a partial order on adversaries, where an adversary A is *weaker than or equal to* an adversary B if all of A's capabilities are weaker than or equal to B's capabilities. A is strictly *weaker* than B if all of A's capabilities are weaker than or equal to B's and at least one of A's capabilities is weaker than B's. Consequently, a secure-deletion solution that defeats an adversary also

Adversary's Name	Disclosure	Credentials	Bound	Accesses	Surface
internal repurposing	predictable	non-coercive	bounded	sing/mult	controller
external repurposing	predictable	non-coercive	bounded	single	physical
advanced forensic	predictable	non-coercive	unbounded	single	physical
border crossing	predictable	coercive	bounded	sing/mult	physical
unbounded border	predictable	coercive	unbounded	sing/mult	physical
malware	unpredict.	non-coercive	bounded	sing/mult	user-level
compromised OS	unpredict.	either	bounded	sing/mult	block dev
bounded coercive	unpredict.	coercive	bounded	single	physical
unbounded coercive	unpredict.	coercive	unbounded	single	physical
bounded multi-access	unpredict.	coercive	bounded	multiple	physical
unbounded multi-access	unpredict.	coercive	unbounded	multiple	physical

Table 2.1: Taxonomy of secure-deletion adversaries.

Adversary's Name	Example
internal repurposing	loaning hardware
external repurposing	selling old hardware
advanced forensic	unfathomable forensic power
border crossing	perjury to not reveal password
unbounded border crossing	cautious corporate policy on encrypted data
malware	malicious application
compromised OS	operating system malware, passwords perhaps provided
bounded coercive	legal subpoena
unbounded coercive	legal subpoena and broken crypto
bounded multiple-access	legal subpoena with earlier spying
unbounded multiple-access	legal subpoena, spying, broken crypto

Table 2.2: Example of adversaries modelled.

defeats all weaker adversaries, under this partial ordering. Finally, as expected, A is *stronger* than B if B is weaker than A.

2.4 Analysis of Solutions

Secure-deletion solutions have differing characteristics, which we divide into assumptions on the environment and behavioural properties of the solution. *Environmental assumptions* include the expected behaviour of the system underlying the interface; *behavioural properties* include the deletion latency and the wear on the physical medium. If the environmental assumptions are satisfied then the solution's behavioural properties should hold, the most important of which is that secure deletion occurs. No guarantee is provided if the assumptions are violated. It may also be the case that stronger assumptions yield solutions with improved properties.

In this section, we describe standard classes of assumptions and properties. Table 2.3 organizes the solutions from Section 2.2 into this systematization.

2.4.1 Classes of Environmental Assumptions

Adversarial Resistance. An important assumption is the one made on the strength and capabilities of the adversary, as defined in Section 2.3. For instance, a solution may only provide secure deletion for computationally-bounded adversaries; the computational bound is an assumption required for the solution to work. A solution's adversarial resistance is a set of adversaries; adversarial resistance assumes that the solution need not defeat any adversary stronger than an adversary in this set.

System Integration. This chapter organizes the secure-deletion solutions by the interface through which they access the physical medium. The interface that a solution requires is an environmental assumption, which assumes that this interface exists and is available for use. System integration may also include assumptions on the behaviour of the interface with regards to lower layers (e.g., that overwriting a file securely deletes the file at the block layer). For instance, a user-level solution assumes that the user is capable of installing and running the application, while a file-system-level solution assumes that the user can change the operating system that accesses the physical medium. The ability to integrate solutions at lower-layer interfaces is a stronger assumption than at higher layers because higher-layer interfaces can be simulated at a lower layer.

System integration also makes assumptions about the interface's behaviour. For example, various solutions overwrite data with zeros, assuming that this operation actually replaces all versions of the old data with the new version. When such interface assumptions are not satisfied, then the solution does not provide secure deletion. As the in-place update assumption is common among solutions, we mark the ones that require it in Table 2.3 using the label *"in"* after the integration layer name.

2.4.2 Classes of Behavioural Properties

Deletion Granularity. The granularity of a solution is the solution's deletion unit. We divide granularity into three categories: *per physical medium*, *per file*, and *per data item*. A per-physical-medium solution deletes *all* data on a physical medium. Consequently, it is an extraordinary measure that is only useful against a *user-controlled access time* adversary, as otherwise the user is required to completely destroy all data as a matter of routine. At the other extreme is sanitizing deleted data at the smallest granularity offered by the physical medium: e.g., block size, sector size, or page size. Per-data-item solutions securely delete any deleted data from the file system, no matter how small.

Between these extremes lies per-file secure deletion, which targets files as the deletion unit: a file remains available until it is securely deleted. While it is common to reason about secure deletion in the context of files, we caution that the file is not the natural unit of deletion; it often provides similar utility as per-physical-medium deletion. Long-lived files such as databases frequently store user data; the Android phone uses them to store text messages, emails, etc. A virtual machine may store an entire file system within a file: anything deleted from this virtual file system remains until the user deletes the entire virtual machine's storage medium. In such settings, per-file secure deletion requires the deletion of all stored data in the DB or VM, which is an extraordinary measure unsuitable against adversaries who control the disclosure time. In other settings, such as the storage of large media files, file data tends to be stored and deleted at the granularity of an entire file and so per-file solutions may reduce overhead.

Scope. Many secure-deletion solutions use the notion of a sensitive file. Instead of securely deleting all deleted data from the file system in an untargeted way, they only securely delete known sensitive files, and require the user to mark sensitive files as such. We divide the solution's scope into *untargeted* and *targeted*. A targeted solution only securely deletes sensitive files, and can substitute for an untargeted solution simply by marking every file as sensitive.

While targeted solutions are more efficient than untargeted ones, we have some reservations about their usefulness. First, the file is not necessarily the correct unit to classify data's sensitivity; an email database is an example of a large file whose content has varying sensitivity. The benefits of targeting therefore depend on the deployment environment. Second, some solutions do not permit files to be marked as sensitive after their initial creation, such as solutions that must encrypt data items *before* writing them onto a physical medium. Such solutions are not suitable for use cases where, for example, users manually mark emails from the inbox as sensitive so that additional secure-deletion actions are taken when it is later deleted. Finally, targeted solutions introduce usability concerns and consequently false classifications due to user error. Users must take deliberate action to mark files as sensitive. A false positive costs efficiency while a false negative may disclose confidential data. While usability can be improved with a tainting-like strategy for sensitivity [67], this is still prone to erroneous labelling and requires user action. Previous work has shown the difficulty of using security software correctly [68] (even the concept of a deleted items folder retaining data confounds some users [69]) and security features that are too hard to use are often circumvented altogether [5].

A useful middle ground is to broadly partition the storage medium into a securely-deleting user-data partition and a normal operating system partition. Untargeted secure deletion is used on the user-data partition to ensure that there are no false negatives and this requires no change in user behaviour or applications. No secure deletion is used for the OS partition to gain efficiency for files trivially identified as insensitive.

Device Lifetime. Some secure-deletion solutions incur device wear. We divide device lifetime into complete wear, some wear, and unchanged. *Complete wear* means that the solution physically destroys the medium. *Some wear* means that a non-trivial reduction in the medium's expected lifetime occurs, which may be further subdivided with finer granularity based on notions of wear specific to the physical medium. *Unchanged* means that the secure deletion operation has no significant effect on the physical medium's expected lifetime.

Deletion Latency. Secure-deletion latency refers to the timeliness when secure-deletion guarantees are provided. There are many ways to measure this, such as how long one expects to wait before deleted data is securely deleted. Here, we divide latency into immediate and periodic secure deletion.

An *immediate* solution is one whose deletion latency is negligibly small. The user is thus assured that data items are irrecoverable promptly after their deletion. This includes applications that immediately delete data as well as file system solutions that only need to wait until a kernel sanitization thread is scheduled for execution.

A *periodic* solution is one that intermittently executes and provides a larger deletion latency. Such solutions, if run periodically, provide a fixed worst-case upper bound on the deletion latency of all deleted data items. Periodic solutions involve batching: collecting many pieces of deleted data and securely deleting them simultaneously. This is typically for efficiency reasons. An important factor for periodic solutions is crash-recovery. If data items are batched for deletion between executions and power is lost, then either the solution must recover all the data to securely delete when restarted (e.g., using a commit and replay mechanism) or it must securely delete all deleted data without requiring persistent state (e.g., filling the hard drive [49, 51, 70]).

Efficiency. Solutions often differ in their efficiency. Wear and deletion latency are two efficiency metrics we explicitly consider. The particular relevant metrics depend on the application scenario and the physical medium. Other metrics include the ratio of bytes written to bytes deleted, battery consumption, storage overhead, execution time, etc. The metric chosen depends on the underlying physical medium and use case.

2.4.3 Summary

Table 2.3 presents the spectrum of secure-deletion solutions systematized into the framework developed in this section. For brevity, we

Solution Name	Target Adversary	Integration	Granularity	Scope
overwrite [48, 49, 61]	unbounded coercive	user-level (in)	per-file	targeted
fill [50, 51, 70]	unbounded coercive	user-level	per-data-item	untargeted
NIST clear [10]	internal repurposing	varies	per-medium	untargeted
NIST purge [10]	external repurposing	varies	per-medium	untargeted
NIST destroy [10]	advanced forensic	physical	per-medium	untargeted
ATA secure erase [41]	external repurposing	controller	per-medium	untargeted
renaming [59]	unbounded coercive	kernel (in)	per-data-item	targeted
ext2 sec del [14]	unbounded coercive	kernel (in)	per-data-item	targeted
ext3 basic [59]	unbounded coercive	kernel (in)	per-data-item	targeted
ext3 comprehensive [59]	unbounded coercive	kernel (in)	per-data-item	targeted
purgefs [62]	unbounded coercive	kernel (in)	per-data-item	targeted
ext3cow sec del [64]	bounded coercive	kernel (in)	per-data-item	untargeted

Solution Name	Lifetime	Latency	Efficiency
overwrite [48, 49, 61]	unchanged	immediate	number of overwrites
fill [50, 51, 70]	unchanged	immediate	depends on medium size
NIST clear [10]	varies	immediate	varies with medium type
NIST purge [10]	varies	immediate	less efficient than clearing
NIST destroy [10]	destroyed	immediate	varies with medium type
ATA secure erase [41]	unchanged	immediate	depends on medium size
renaming [59]	unchanged	immediate	truncations copy the file
ext2 sec del [14]	unchanged	immediate	batches to minimize seek
ext3 basic [59]	unchanged	immediate	batches to minimize seek
ext3 comprehensive [59]	unchanged	immediate	slower then ext3 basic
purgefs [62]	unchanged	immediate	number of overwrites
ext3cow sec del [64]	unchanged	immediate	deletes multiple versions

Table 2.3: Spectrum of secure-deletion solutions. An integration marked with "(in)" means that the integration assumes an in-place update implementation.

do not list all adversaries that a solution can defeat, but instead state what we inferred as the solution's target adversary.

The classes of environmental assumptions and behavioural properties are each ordered based on increased *deployment requirements*. Solutions that cause wear and use in-place updates have stronger deployment requirements (i.e., that wear is permitted and the interface allows and correctly implements in-place updates) than solutions that do not cause wear or use in-place updates. Solutions that defeat weak adversaries have stronger deployment requirements (i.e., that the adversary is weak) than solutions that defeat stronger adversaries. The result is a partial ordering on solutions that reflects substitutability: a solution with weaker deployment requirements can replace one with stronger deployment requirements as it requires less to correctly deploy.

Chapter 3

System Model and Security Goal

3.1 Introduction

This chapter presents the system model, adversarial model, and security goal for the work presented in this thesis. First, the system model describes how the user interacts with storage media to store, read, and delete data. Second, the storage medium models describe a number of different storage media types relevant to this thesis. Third, the adversarial model describes how the adversary is able to gain access to the user's storage media. Finally, the security goal defines secure deletion and describes what data our solutions strive to delete.

3.2 System Model

Our system model consists of a user who stores data on storage media such that the data can be retrieved by the user during the data's lifetime. A data's *lifetime* is the range between two events: the data's initial *creation* and its subsequent *discard*.

We assume that the user divides the data to store into discrete *data items* that share a lifetime. These can be binary data objects, files, or individual blocks of a block-based file system. The user retains a *handle* to recall these items (e.g., an object name or a block address).

The set of handles and the mapping of handles to storage locations may entail the storage of metadata. We do not elevate metadata to require particular concern: metadata is itself stored as data items and can therefore be securely deleted in the same way.

The user continuously stores, reads, and discards data on a storage medium. For example, it may be a mobile phone storing location data continually throughout the day, or a server continually storing sensitive log data needed only for a short time to monitor for malicious behaviour. The user may use multiple storage media when storing and retrieving data. These media may differ in their implementation and interface. We assume that the user can store data using an object store interface (e.g., OSD [71, 72]): the user can *store* data (an object) with a handle (lookup key), *read* the data for a handle, and *discard* the data for a handle. The use of the nomenclature *discard* is intentional to emphasize that discard does not necessarily entail any deletion.

Particular storage media may have expanded interfaces, but we assume that the user stores, reads, and discards data with the object store interface. Updating data is implemented by discarding the old version and storing a new version. Securely deleting data is achieved when discarded data is irrecoverable from a storage medium. Table 3.1 describes how these three storage functions translate into a variety of common storage interfaces.

A fundamental correctness property for a storage medium is that it retrieves stored data. Particularly, from the moment data is stored until it is later discarded, the data must be readable. This property makes no requirements on data not being readable after it is discarded (or for that matter, before it is created); it only requires that data is readable when it is considered valid.

3.3 Storage Media Models

We now describe a suite of storage medium models representative of a variety of real-world systems and which differ greatly in how easily data can be securely deleted. At the extremes, we have the secure-deletion optimal SECDEL model and the secure-deletion near-pessimal PERSISTENT model.

Securely-deleting Model. The SECDEL model is an idealized case where the interface's deletion function performs immediate secure dele-

Storage Interface (IX)	Handles	Contents
Object Store (OSD) [71]	object id	object
Distributed Hash Table (DHT) [73]	key	value
POSIX filesystem (FS) [43]	file and offset	file contents
Block device (blk)	block address	block
ATA device (ATA) [40]	logical block address	sector
Multimedia card (MMC) [74]	data address	block

	Simple storage interface		
IX	Store	Read	Discard
OSD	create, write	read	remove
DHT	store	find value	-
FS	write	read	truncate, unlink
blk	REQ_WRITE	REQ_READ	REQ_DISCARD
ATA	ATA_CMD_WRITE	ATA_CMD_READ	ATA_DSM_TRIM
MMC	MMC_DATA_WRITE	MMC_DATA_READ	MMC_TRIM_ARG

Table 3.1: Mapping the simple storage interface to different storage interfaces. Linux function names or constant definitions for standardized hardware signals are used for the interface.

tion. This models any black-box-like storage system that correctly implements secure deletion. Secure deletion solutions transform other media or ensemble of media into a SECDEL-like model through an explicit construction.

An analog example is a rolling index of numbered cards: data is written onto new cards, which are then numbered by the position and inserted into the index; discarding removes the numbered card and incinerates it.

Clocked Securely-deleting Models. The two clocked models—SECDEL-CLOCK and SECDEL-CLOCK-EXIST—are idealized cases where the interface's deletion function results in secure deletion. Secure deletion, however, is provided in batch through a periodic clock operation. In SECDEL-CLOCK, discarded data remains stored until the next clock edge; in SECDEL-CLOCK-EXIST, discarded data remains stored on the medium until the next clock edge in *both directions*: before its creation and after its discard.

These model any black-box-like storage system that correctly implements a periodic secure deletion operation. We include these idealization because some secure deletion solutions have non-trivial execution costs and are therefore run periodically to compensate. Clocked models have a *clock period*, that is, the time between each *clock edge*. The clock divides time into discrete *deletion epochs*, where all data discarded in one deletion epoch is securely deleted at the clock edge and therefore no longer readable in later deletion epochs.

An analog example of SECDEL-CLOCK is paper recycling in a security-conscious organization. Data written on paper is discarded into recycling bins. Each night, all scrap paper from recycling bins is shredded before recycling.

In-place Update Model. The INPLACE model is does not securely delete data when it is discarded; instead, the corresponding position is marked for deletion, which indicates that its consumed resources can be reclaimed when needed. This models simple filesystems such as FAT and ext2 as well as the block device interface that accesses magnetic storage media such as hard drives and floppy disks. Much of the related work presented in Section 2.2 makes the assumption that data is stored on an in-place update storage medium.

An analog example is a set of blackboards during a lecture. New data is written on a blank board, but old data is not securely deleted until the space it consumes is needed for new data.

Semi-persistent Model. The SEMIPERSISTENT model is a finite-size medium where the interface's concept of a storage position differs from the implementation's concept. An indirection layer maps interface positions to storage positions. The user may store data at logical positions but is given no control over where data is physically stored. The semi-persistent nature of the medium is a consequence of its finite size, necessitating the eventual reuse of physical positions and therefore the secure deletion of previous content. This model corresponds to a variety of log-structured file systems such as JFFS, YAFFS, and UBIFS as well as the effect of accessing flash memory through a flash translation layer (FTL).

An analog example is an inter-office mail envelope's recipient; old recipients' names remain visible but the current valid recipient is the one furthest down. When a name is written in all the slots, the names are erased by discarding the envelop and replacing it.

Persistent Model. The PERSISTENT model is a storage medium that does not ever securely delete data. Once data is stored on the persistent storage it remains stored permanently. This may occur because the nature of the medium is indelible and append-only, such as writing information on optical discs and storing them in an archive. The medium may also not explicitly be append only, but environmental assumptions warrant it to be considered as such; for example, an adversary that obtains continuous access to stored data (i.e., by monitoring the network or controlling the storage medium), or a user who is unable to gauge the adversary's eventual forensic capabilities.

An analog example is publishing data in a newspaper; corrections can be issued for incorrect data, however the published data remains a persistent part of the public record. An example about the concerns over the adversary's eventual forensic capabilities is layers of oil paint on a canvas; while previous layers appear to be deleted, X-ray technology has enabled the discovery of buried paintings, each appended over the previous [75].

3.4 Adversarial Model

We assume the presence of a computationally-bounded unpredictable multiple-access coercive adversary. This means that the adversary gains access to the client's storage medium, at multiple unpredictable points in time. This adversary can perform a *coercive attack* to compromise both the client's storage medium as well as any secret keys, etc., which may be needed to access data stored on the medium. The adversary has full knowledge of the algorithms as well as the implementation of the system and all relevant storage media. We make the assumption that symmetric-key cryptography is perfect, that is, the computationally-bounded adversary cannot recover the plain-text message from a cipher-text message without the corresponding encryption key. This requires that any keys derived from passwords are taken from sufficiently-strong passwords to prohibit offline guessing.

With the exception of the computational bound, this adversary is the strongest one developed in our taxonomy (cf. Table 2.1). Since the time of the attack is unpredictable, no extraordinary sanitization procedure can be performed prior to compromise. Since the user is continually using the storage media, physically destroying it is not possible. Since the attacker is given the user's secret keys, it is insufficient to simply encrypt the storage media [59]. The adversary may also attack multiple times if desired. The solutions we develop defeat this strong adversary and therefore all weaker adversaries as well.

3.5 Security Goal

Secure deletion is used to protect the confidentiality of data. For any particular data item, the security goal is to ensure that an adversary who compromises the storage medium at all times outside the data item's lifetime is unable to recover the data item. This is because we assume that an adversary that coercively attacks during some data's lifetime is trivially able to recover the data; secure deletion aims to protect the confidentiality of data outside its lifetime. Observe that the security goal is realized for all data items if every time the adversary compromises the storage medium, it is *only* able to recover valid data; indeed, this is the behaviour of the optimal SECDEL model.

For clocked models, however, data is only deleted at the next clock edge. An adversary that compromises the storage medium after a data item is discarded but before the next clock edge means that it obtains this discarded data. The time discarded data remains available is called the data item's *deletion latency*. We say that the security goal is achieved for all data items with a *deletion latency* of δ if every time the adversary compromises the storage medium it is *only* able to recover valid data or data discarded within the previous δ time units.

Paradoxically, it is also possible for data to be exposed to an adversary who compromises the storage medium *before* the data is written: this occurs when encryption keys used for data are written in advance of use (and, for example, the adversary trivially obtains encrypted data). Consequently, there is a period of time where adversarial compromises recover future data and we call this period the *existential latency*. We say that the security goal is achieved for all data items with a *existential latency* of ϵ if every time the adversary compromises the storage medium it is *only* able to recover valid data or data that is created within the next ϵ time units.

The behaviour of different models with regards to example adversarial compromises is illustrated with Figures 3.1–3.5. Each figure shows the storage history of seven data items—all with the same data lifetime—as stored by a storage medium that behaves like a particular model. The first, Figure 3.1, is for the optimal SECDEL model, while the last, Figure 3.5, is for the near-pessimal PERSISTENT model. Data lifetimes are visualized with a solid line with the create and discard events indicated with a black circle. A dotted line before or after these events indicate the existential and deletion latencies respectively, that is, times when adversarial compromise obtains the data. The data effectively stored (i.e., compromisable) at each particular time is listed as a set below the life timelines.

Figure 3.1 shows the behaviour of the SECDEL model: only valid data is stored at each time. Figure 3.2 shows the behaviour of the SECDEL-CLOCK model with a clock period of four. Clock edges are shown as thick black lines and each data item has a deletion latency that extends to the next clock edge. Figure 3.3 shows the same for a clocked model that has both existential and deletion latencies, which we call SECDEL-CLOCK-EXIST. The data stored is the same for all times within a particular deletion epoch. Figure 3.4 shows the behaviour of either the INPLACE or SEMIPERSISTENT model. There are three different storage positions, and the dotted line that connects one data item to another indicates that some newly-created data is stored in the same position as the previously-deleted data. Figure 3.4 illustrates both INPLACE and SEMIPERSISTENT models because the main difference between them is whether the user can control the positions used to store new data. Finally, Figure 3.5 shows the behaviour of the PERSISTENT model, which stores all data that was earlier valid: the deletion latency extends to the end.

At the bottom of all these figures are four example adversarial compromises. Each adversary compromises the storage medium at a different set of times, indicated with a black box containing a time. The set of data obtained by each compromise is the union of data stored at each compromise time. The effectiveness of a model with regards to secure deletion can be seen by how well the adversary's *data obtained* approximates the SECDEL model.

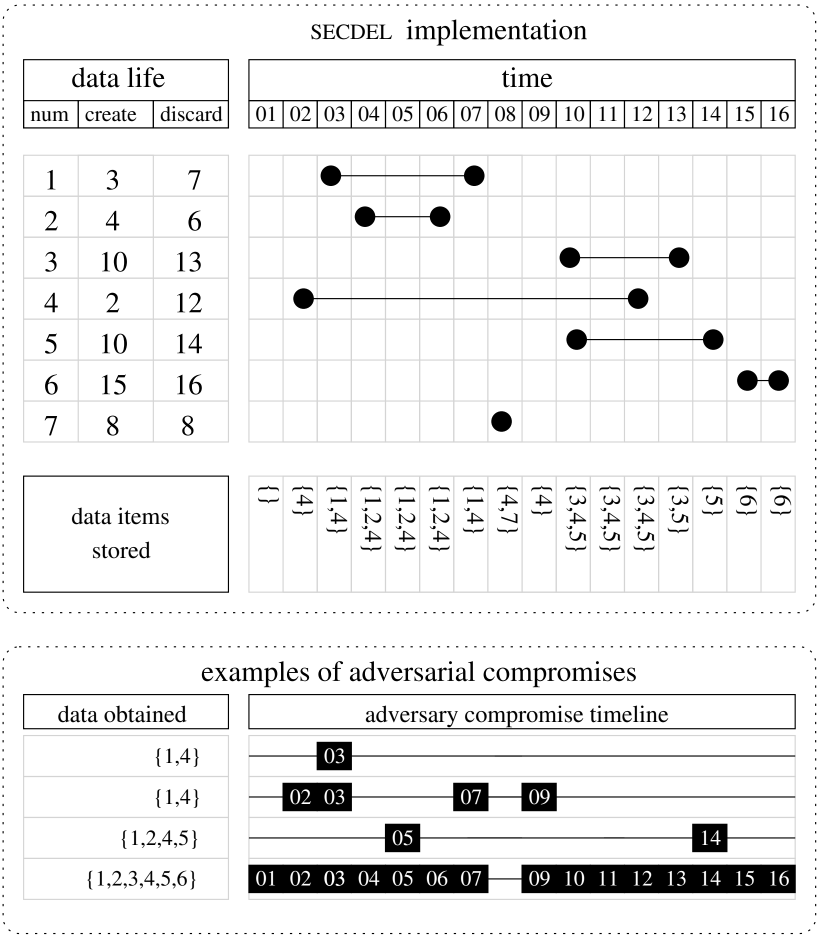


Figure 3.1: Example data lifetimes and adversarial compromises for a SECDEL model. Seven numbered data items have each a lifetime, where time is discretized into 16 points in time. At each point in time, the set of valid data is indicated. Below the data lifetimes are four adversarial timelines. A black box with a number indicates that the adversary performs a coercive attack at that time. The adversary obtains all data valid at each time it compromises. We assume perfect secure deletion: the adversary only obtains valid data.

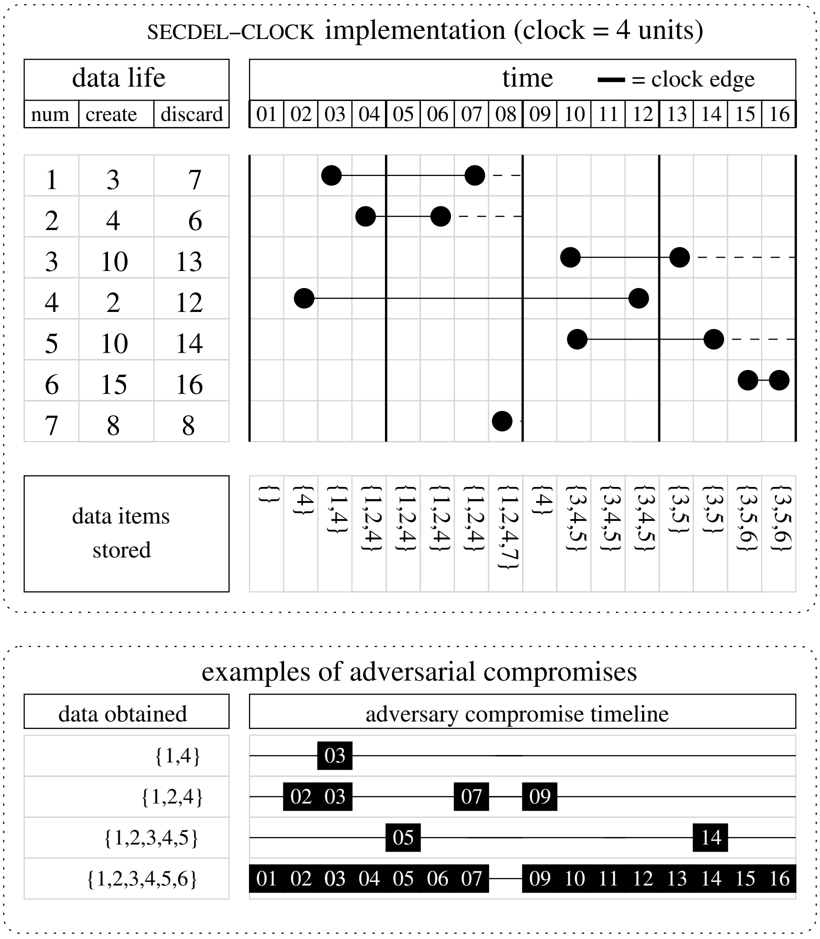


Figure 3.2: Example data lifetimes and adversarial compromises for a SECDEL-CLOCK model. The data lifetimes and adversarial compromises are the same as Figure 3.1. We assume there is a clock operation performed every four time units, which is indicated with a thick black line in the data lifetimes. Data is only deleted at a clock operation, so a dotted line indicates the deletion latency. Adversarial compromises are updated accordingly to obtain deleted data which may still be available.

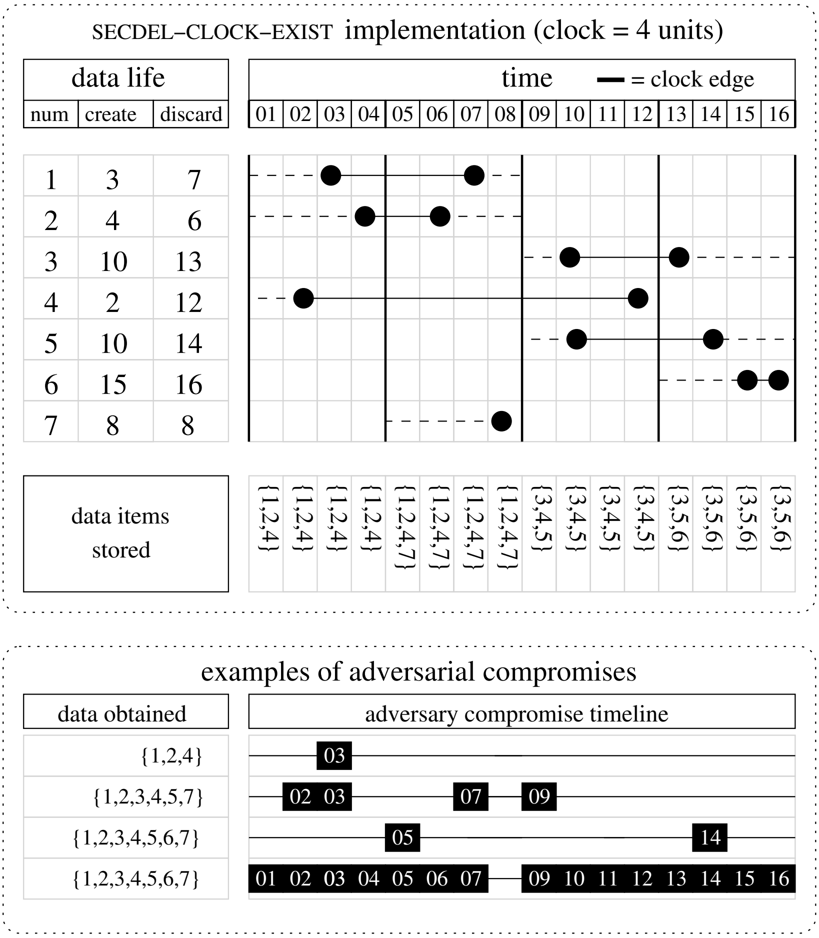
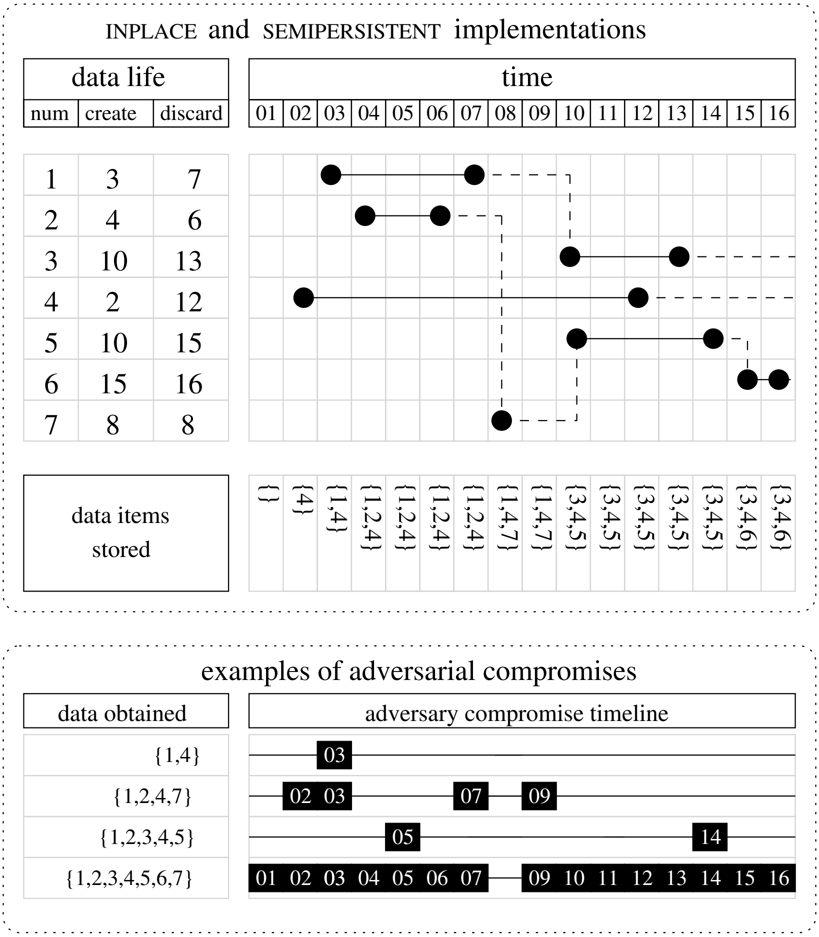


Figure 3.3: Example data lifetimes and adversarial compromises for a SECDEL-CLOCK-EXIST model. The data lifetimes, adversarial compromises, and clock period are the same as Figure 3.2. Not only is data deleted at a clock edge, data can be obtained by compromising the medium before it is written provided it will be written in the same clock period. A dotted line before the data's creation indicates its existential latency. Adversarial compromises are updated accordingly.

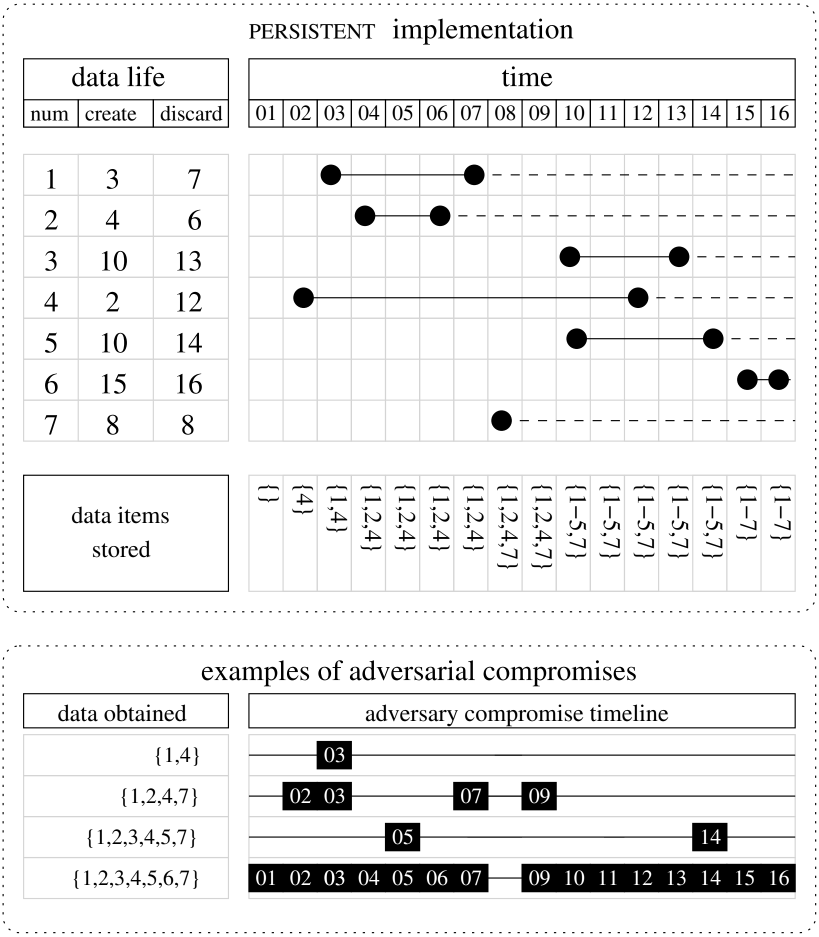

INPLACE and SEMIPERSISTENT implementations
data life
time
data items stored
examples of adversarial compromises
data obtained
adversary compromise timeline

num	create	discard
1	3	7
2	4	6
3	10	13
4	2	12
5	10	15
6	15	16
7	8	8

time	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
data items stored	{}	{4}	{1,4}	{1,2,4}	{1,2,4}	{1,2,4}	{1,2,4}	{1,4,7}	{1,4,7}	{3,4,5}	{3,4,5}	{3,4,5}	{3,4,5}	{3,4,5}	{3,4,6}	{3,4,6}

data obtained	adversary compromise timeline
{1,4}	03
{1,2,4,7}	02 03 07 09
{1,2,3,4,5}	05 14
{1,2,3,4,5,6,7}	01 02 03 04 05 06 07 09 10 11 12 13 14 15 16

Figure 3.4: Example data lifetimes and adversarial compromises for either an INPLACE or SEMIPERSISTENT model. There are three different "physical" storage positions; data remains stored until it is replaced with a new value; a dotted line from the deletion of some data to the creation of another piece of data indicates that the deleted data remained stored until that new piece of data is created. Adversarial compromises are updated accordingly. The difference between in-place updates and semipersistent updates is that in the former the user can select which data is replaced during an update.


PERSISTENT implementation
time
examples of adversarial compromises
adversary compromise timeline

data life		
num	create	discard
1	3	7
2	4	6
3	10	13
4	2	12
5	10	14
6	15	16
7	8	8

time	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
data items stored	{}	{4}	{1,4}	{1,2,4}	{1,2,4}	{1,2,4}	{1,2,4}	{1,2,4,7}	{1,2,4,7}	{1–5,7}	{1–5,7}	{1–5,7}	{1–5,7}	{1–5,7}	{1–7}	{1–7}

data obtained	adversary compromise timeline
{1,4}	03
{1,2,4,7}	02 03 07 09
{1,2,3,4,5,7}	05 14
{1,2,3,4,5,6,7}	01 02 03 04 05 06 07 09 10 11 12 13 14 15 16

Figure 3.5: Example data lifetimes and adversarial compromises for a PERSISTENT model. Once written, data remains stored permanently. Adversarial compromises are updated accordingly.

Part II

Secure Deletion for Mobile Storage

Chapter 4

Flash Memory: Background and Related Work

4.1 Overview

Flash memory is a fast, small, and lightweight type of storage medium that, at the time of writing, is ubiquitously used in mobile, portable devices, including mobile phones, MP3 players, digital cameras, voice recorders, gaming devices, USB sticks, multimedia cards, and solid-state drives. The ability to securely delete data from such devices is important because mobile phones, in particular, store sensitive personal information, such as the timestamped names of nearby wireless networks, personal correspondence, and furthermore, business data, for which company policy or legislation may mandate deletion after some time elapses or at some geographic locations.

We already discussed a variety of ways to securely delete data from magnetic storage, which often involve overwriting the data to replace the old data on the medium. For flash memory, however, this solution does not work since flash memory cannot perform an in-place update—that is, an update that replaces the old version with a new version of data. Instead, all updates to flash memory are performed in a log-structured way: writing the fresh data to a new location and rendering the old version obsolete. Flash memory has the behaviour of a SEMIPERSISTENT implementation.

In the following chapters, we present a detailed examination of secure deletion solutions for flash memory. The remainder of this chapter presents details on flash memory including the interface to access it, flash file systems that use this interface, and flash translation layers that hide the details of flash memory and expose a block-based interface typical for magnetic media. We then present related work for secure deletion on flash memory.

Chapters 5, 6, and 7 provide our contributions. Chapter 5 presents our results on user-level secure deletion for flash memory, that is, what a user can do to securely delete data on their mobile devices without changing their file system, operating system, or hardware. Chapter 6 presents the Data Node Encrypted File System (DNEFS), which augments a file system to provide secure deletion. Chapter 7 validates DNEFS by presenting an implementation for the flash file system UBIFS, analysing its performance, and finding that it is suitable for secure data deletion for flash memory.

4.2 Flash Memory

Flash memory is a non-volatile storage medium consisting of an array of electronic components that store information [76]. A *page* of flash memory is *programmed* to store data, which can thereafter be *read* until the page is *erased* [77]. Flash memory has small mass and volume, does not incur seek penalties for random access, and is energy efficient. As such, portable devices ubiquitously use flash memory.

Figure 4.1 shows how flash memory is divided into two levels of granularity. The first level is called *erase blocks*, which are on the order of 128 KiB [78] in size. Erase blocks are divided into *pages*, which are on the order of 2 KiB in size. Note that different kinds of memory may have different sizes for the erase block and the page, however these values are representative of both typical memory devices as well as the difference in scale between the two levels of granularity.

Erase blocks are the unit of erasure, and pages are the unit of read and write operations [77]. One cannot write data to a flash memory page unless that page has been previously *erased*; only the erasure operation performed on an erase block prepares the pages it contains for writing.

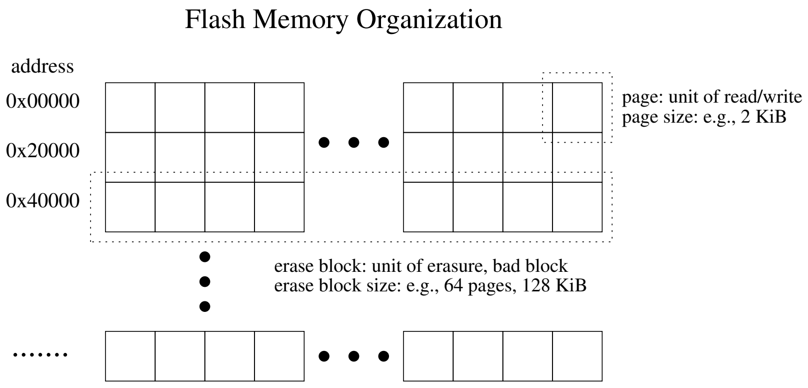

Flash Memory Organization
address
0x00000
0x20000
0x40000
....
page: unit of read/write
page size: e.g., 2 KiB
erase block: unit of erasure, bad block
erase block size: e.g., 64 pages, 128 KiB

Figure 4.1: Flash memory divided into two levels of granularity. Each row of squares represents an erase block and each square represents a page.

Flash erasure is costly: its increased voltage requirement eventually wears out the medium [79]. Each erasure risks turning an erase block into a bad block, which cannot store data. Flash erase blocks tolerate between 10^4 to 10^5 erasures before they become bad blocks. To promote a longer device lifetime, erasures should be evenly levelled over the erase blocks. This is commonly called *wear levelling*.

Flash memory best practices are that the erase block's empty pages are programmed sequentially by their physical layout. This mitigates an issue known as *program disturb*, where programming a flash page affects the data integrity of physically-neighbouring pages. By programming pages sequentially, program disturb is only a concern for the most-recently-programmed page.

In the remainder of this section, we describe how log-structured file systems are used to overcome flash memory's in-place update limitation. We then describe the two main types of log-structured implementations: (i) a flash translation layer that exposes a block device and (ii) special purpose flash file systems that expose POSIX-compliant file system interfaces.

4.2.1 In-Place Updates and Log-structured File Systems

Flash memory's requirement that the entire erase block is erased before data can be written is the reason for flash memory's inability to perform in-place updates. Erase blocks may store a mix of deleted and valid data for a variety of different files. The naive way to update one page on an erase block is to temporarily store the entire erase block elsewhere, erase the original erase block, and finally reprogram all the pages on the original erase block to store the previous data with the exception of the updated page. In fact, Linux provides a simple block device emulator for flash memory, called `mtdblock` [80], which slightly improves on this naive update strategy by buffering multiple changes to a single erase block using a one-item memory cache.

In practice, flash memory's in-place update limitation is managed by using log-structured file systems to store and access data. A log-structured file system differs from a traditional block-based file system (such as FAT [81] or ext2 [57]) in that the entire file system is stored as a chronological record of changes from an initial empty state. As files are written, data is appended to the log indicating the resulting change; each flash page stores a fixed-size block of data. File metadata and data are usually stored without separation. The file system maintains in volatile memory the appropriate data structures to quickly find the newest version of each file header and data page [77, 82].

When a change invalidates an earlier change, then the new, valid data is appended and the erase block containing the invalidated data now contains wasted space. File deletion, for example, may append a log entry that states that file is thenceforth deleted. All the deleted file's data nodes remain on the storage medium but they are now invalid and wasting space. Encrypting a file, for example, appends a new encrypted version of that file with the obsolete plaintext now remaining in the log.

Data is removed from a log-structured file system with a *compaction* process often called *garbage collection* [77],[1] which has the purpose of reclaiming wasted storage resources. The compactor operates at the *erase block* level, which has a larger granularity than a page. While implementation details may vary, in principle the compactor erases any

[1]The term garbage collection comes from its similarity to garbage collection in programming languages where unused memory is found and collected; in this thesis we use the term *compaction* to refer to the "garbage collection" process that specifically collects *non-garbage* data on erase block to recover wasted resources.

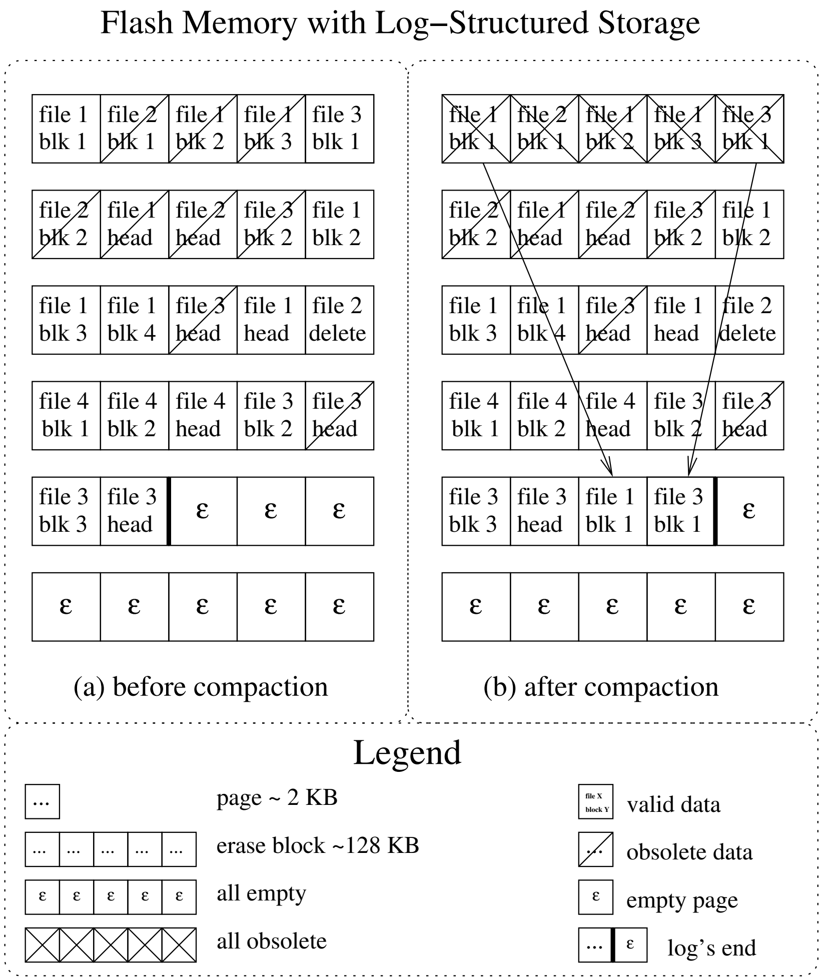


Figure 4.2: Flash memory storing a log-structured file system. Data blocks and headers for different files are stored on the pages; some pages are obsolete as newer versions exist. File 2 is at one point deleted, making all file data except the delete notice obsolete. (a) The state before compacting the first erase block. (b) Two valid pages are copied from the first erase block to the end of the log. The first erase block can now be erased.

erase block that only contains deleted data, and also compacts erase blocks with a significant amount of wasted space by first copying live data to the log's end and then erasing the old erase block. Figure 4.2 shows a flash memory being used to store a log-structured file system's data and the compaction of the first erase block. Figure 4.2 (a) shows the state before compaction where the first erase block contains only two valid blocks; (b) shows the state after copying the valid data to the log's end, resulting in the first erase block storing only obsolete data. The erasure operation can then be performed on it to recover the wasted resources.

As a historical observation, log-structured file systems were first implemented in 1992 by Rosenblum and Ousterhout without anticipating flash memory. Their purpose was to improve efficiency for file systems that perform frequent small writes to different files—i.e., file systems used for logging events—by putting all new data at the end of the log to reduce magnetic media's high seek latency [83]. The notion of erase blocks, then called *segments* was needed for such systems because fragmentation in the sequence of writable positions removed the benefit of seek-free writes. The *segment cleaner* therefore compacted the useful data elsewhere and reclaimed the entire segment for fresh data. This idea ultimately found enormous utility in mitigating the quirks of flash memory—a memory that does not even suffer from seek latency.

4.2.2 Flash Translation Layer.

Flash memory is commonly accessed through a Flash Translation Layer (FTL) [76, 84], which is used in USB sticks, MMC devices such as SD cards, and solid state drives. FTLs access the raw flash memory directly, but expose a block-based interface that is typical for magnetic hard drives. This allows users to use widely-compatible file systems—in particular, FAT—when storing data, allowing easy exchange of data across devices and computers.

FTLs vary in implementation [85, 86], however they all have a simple purpose: to translate logical block addresses to raw physical flash addresses and internally implement a log-structured file system on the memory [85]. New data is written to empty flash memory pages (the end of the log) and a mapping keeps track of what is the most recent version of a particular logical address in the virtual block device.

Most FTLs used for portable devices are implemented in hardware. In this case, software access to the device can only reveal the contents of the virtual block device. For example, the specification for embedded multimedia cards (eMMC) provides no interface functionality to read data from *physically unmapped areas*, that is, parts of the physical memory that does not correspond to data *officially* stored on the virtual block device. By disassembling the hardware and accessing the flash memory directly, however, one can bypass the hardware FTL and easily read the stored data [39]. Linux offers an `ftl` driver as an open-source software-based implementation of an FTL based on Intel's specification [84].

4.2.3 Flash File Systems.

Another solution to accessing flash memory is to use a file system tailored for the purpose. A variety of flash file systems (FFSs) exist; open-source ones for Linux include JFFS [87], YAFFS [82], UBIFS [88], and F2FS [89]. These file systems are log-structured and access the flash memory directly through a Memory Technology Device (MTD). MTDs offer a similar interface as block devices do but are extended to support erasing an erase block, testing if an erase block is bad (i.e., unwritable), and marking an erase block as bad.

Using a tailored flash file system means that a magnetic-storage-targeted file system—which may replicate features such as journalling—need not be mounted above an FTL. Moreover, the lack of an opaque hardware FTL permits greater transparency in how data is stored and deleted as well as easier integration and verification of secure deletion solutions. These file systems, however, are not widely-supported outside Linux systems and therefore are less suitable as external (removable) memory than as internal (non-removable) memory.

4.2.4 Generalizations to Other Media.

The asymmetry between the write and erase granularities is not limited to flash memory, and Table 4.1 summarizes such storage media. It manifests itself in physical media composed of many write-once read-many units; units that are unerasable but replaceable. Examples include a library of write-once optical discs or a stack of punched cards. All write-once media are unerasable—NIST says they must be physically destroyed to achieve any form of secure deletion [10]—but first valid colocated data must be replicated onto a new disc or card and

media	collection	I/O unit	erase unit	erase op.	relevant cost
optical disc	library	track	disc	destroy	blank media
magnetic tape	vault	backup	cassette	tape-over	tape wear, drive time
flash memory	memory	page	erase block	erasure	erase block wear, power
punched cards	stack	column	card	shred	blank media, repunching

Table 4.1: Generalizations of the write and erase granularity asymmetry to other storage media.

then the library updated. Therefore, each erase operation performed on such media destroys one of its constituent storage units.

Similarly, media that can be erased but only with a high asymmetry in granularity also suffer from this problem. For example, a tape archive consists of many magnetic tapes, each storing, say, half a terabyte of data. Tape must be written end-to-end in one operation; data available for archiving is heuristically bundled onto a tape. Later, to securely delete a single backup on the tape, the entire tape is re-written to a new tape with the backup removed or replaced; the old tape is then erased and reused in the tape archive. This operation incurs cost: tapes have a limited erasure lifetime and tape-drive time is an expensive resource for highly-utilized archives.

4.3 Flash Secure Deletion Related Work

In this section, we describe related work on the topic of secure deletion for flash memory.

Secure Erase / Factory Reset. Some flash-based devices offer a *factory reset* feature, which acts like a secure erase feature in their hardware controllers. Such a feature is intended to perform erase block erasure on all the erase blocks that comprise the storage medium. This means that the solution has a per-storage-medium granularity.

A study by Wei et al. [39] observed that solid-state drives' controller-based *secure erase* operation is occasionally incorrectly implemented. In some cases, the device reported a successful operation while the entire file system remained available. In follow-up work, Swanson and Wei [6] describe a solution for verifiable full-device secure deletion that they compare in effectiveness to hard drive degaussing. They propose to encrypt all data written to the physical medium with a key stored only on the hardware controller. To securely delete the device, first the controller's key memory is erased. Every block on the device is then erased, written with a known pattern, and erased again. Finally, the device is reinitialized and a new key given to the flash controller. As with the factory reset, this solution has a per-storage-medium granularity.

The eMMC specification states that the *secure erase* and *secure trim* functions are obsolete as of v4.51 [74]. Instead, a *sanitize* (i.e., secure deletion) function may be optionally provided; the specification states that calling the sanitize function must erase all erase blocks containing unmapped data including all copies; mapped data must remain available. If correctly implemented, this provides secure deletion at a per-data-item granularity, however verifying that it is correctly implemented requires disassembling the physical device. The Linux eMMC device allows this operation to be performed from user-space via an `ioctl` [47].

Compaction. The naive secure deletion solution for physical media with an asymmetry between their write and erase granularities is to immediately compact the erase block that contains the deleted data: copy the valid colocated data elsewhere and execute the erasure operation. This is a costly operation: copying the data costs time and erasing an erase block may additionally cause wear on the physical medium. Nevertheless, there is no other *immediate* secure deletion solution based on erasures that can do better than one erase block erasure per deletion. Any improvement requires batching and thus effects a deletion latency.

Batched Compaction. One obvious improvement over the naive solution is to intermittently perform compaction-based secure deletion on all the erase blocks that have accumulated deleted data since the last secure deletion. This solution is no worse than the naive solution in terms of the time and wear, although the deletion latency—the time the user must wait until data is securely deleted—increases. Each time that deleted data items are colocated on an erase block, the amortized time and wear cost of secure deletion decreases. Indeed, log-structured file systems already perform a similar technique to recover wasted space, where compaction is performed only on erase blocks whose wasted space exceeds a heuristically-computed threshold based on the file system's current need for free space.

Per-File Secure Deletion. Lee et al. [90] propose a secure deletion solution for YAFFS [82]. It performs immediate secure deletion of an entire file at the fixed cost of one erase block compaction. It reduces the erasure cost of secure deletion by only deleting data at the granularity of a file. This solution works at a per-file granularity: Until the file is deleted, it remains entirely available including overwritten and truncated parts. When the file is deleted, a single erase block erasure is sufficient to ensure it is securely deleted.

Their solution encrypts each file with a unique key stored in every version of the file's header. The file system is modified to store all versions of a file's header on the same erase block. Whenever erase blocks storing headers are full, they are compacted to ensure that file encryption keys are only stored on one erase block. To delete a file, the erase block storing the key is compacted for secure deletion, thus deleting all file data under computational assumptions with only one erase block erasure.

Scrubbing. Compaction is the only immediate secure deletion solution that uses erasures. Wei et al. [39] propose a solution for flash memory, called *scrubbing*, which works by draining the electrical charge from flash memory cells—effectively rewriting the memory to contain only zeros.

Scrubbing securely deletes data immediately with the granularity of a page and no erase block erasures must be performed. It does, however, require programming a page multiple times between erasures, which is not appropriate for flash memory [79]. In general, the pages of an erase block must be programmed sequentially [91] and only once. Multiple partial programs per page is permitted provided they occur at different positions in the page and are fewer than the manufacturer's specified limit; multiple overwrites to the same location officially result in undefined behaviour [91]. Flash manufacturers prohibit this due to *program disturb* [92]: bit errors that can be caused in spatially-proximate pages while programming flash memory.

Wei et al. performed experiments to quantify the rate at which such errors occur: they showed that they do exist but their frequency varies widely among flash types, a result also confirmed by Grupp et al. [93]. Wei et al. use the term scrub budget to refer to the number of times that the particular model of flash memory has experimentally allowed multiple overwrites without exhibiting a significant risk of data errors.

When the scrub budget for an erase block is exceeded, then secure deletion is instead performed by compaction: copying all the remaining valid data blocks elsewhere and erasing the block. Wei et al.'s results show that modern densely-packed flash memories are unsuitable for their technique as they allow as few as two scrubs per erase block [39].

4.4 Summary.

While efforts have been made to solving the problem of secure deletion for flash memory, none of these solutions are perfect. Factory reset suffers from problems associated with per-storage-medium solutions. Compaction deletes the data but come at a high cost in terms of flash memory erasures. Lee et al.'s per-file encryption suffers from problems associated with per-file solutions and reduces to naive compaction when dealing with many small files. Scrubbing does not work with all flash memory and, in particular, works less effectively on newer devices. We therefore need novel solutions to this problem.

In the next two chapters, we present our efforts to solve this problem. Chapter 5 presents user-level secure deletion for flash memory. It provides two solutions, as well as a hybrid of them, which provide secure deletion functionality for users without having to modify their operating system or file system. We perform experiments to measure the solutions' costs in terms of erase block erasures and their benefits in terms of deletion latency.

Chapter 6 presents the Data Node Encrypted File System. It is a generic extension to a file system, and therefore a kernel-level solution, which can be used to provide secure data deletion with a configurable deletion latency. It significantly reduces the number or erase block erasures required to achieve secure deletion.

Chapter 7 validates our file system extension by implementing it for the flash file system UBIFS. We measure its performance both in simulation and deployed on a mobile phone and show that it results in a modest increase in the erase block erasure rate and modest decrease in performance.

Chapter 5

User-Level Secure Deletion on Log-Structured File Systems

5.1 Introduction

This chapter addresses the problem of secure data deletion on log-structured file systems. We focus on YAFFS, a file system used on Android smartphones that uses raw flash for the internal memory. We analyse how deletion is performed in YAFFS and show that log-structured file systems in general provide no temporal guarantees on data deletion; the time discarded data persists on a log-structured file system is proportional to the size of the storage medium and related to the writing behaviour of the device using the storage medium. Moreover, discarded data remains stored indefinitely if the storage medium is not used after the data is marked for deletion.

We propose two user-level solutions for secure deletion in log-structured file systems: *purging*, which provides guaranteed time-bounded deletion of all data previously discarded by filling the storage medium, and *ballooning*, which continuously reduces the expected time that any discarded data remains on the medium by occupying a fraction of the capacity. We combine these two solutions into a hybrid, which guarantees the periodic, prompt secure deletion of data regardless of the storage medium's size and with acceptable wear of the memory.

As these solutions require only user-level permissions, they enable the user to securely delete data even if this feature is not supported by the kernel or hardware, over which users typically do not have control. This, for example, allows mobile phone users to achieve secure deletion without violating their warranties or requiring non-trivial technical knowledge to update their firmware with a customized kernel.

We implement these solutions on an Android smartphone (Nexus One [78]) and show that they neither prohibitively reduce the longevity of the flash memory nor noticeably reduce the device's battery lifetime. We simulate our solutions for phones with larger storage capacities than the Nexus One, and show that while purging alone is expensive in time and flash memory wear, when combined with ballooning it becomes feasible and effective. Ballooning provides a trade off between the deletion latency and the resulting wear on the flash memory. It also substantially reduces the deletion latency on large, sparsely-occupied storage media.

5.2 System and Adversarial Model

The user continually stores, reads, and discards sensitive data on a mobile phone.We assume that the user has only user-level access to the mobile phone. This means that the user may not modify the operating system or hardware of the device. The solution can only interact with the file system interface to achieve secure deletion.

We assume that there is an unpredictable multiple-access coercive adversary that can compromise the user's storage medium. Our adversarial model is a slight modification of the main model developed in Chapter 3 in that it is not computationally bounded.

5.3 YAFFS

Yet Another Flash File System (YAFFS) is a log-structured file system designed specifically for flash memory [82]. It is notably used as the file system for the internal memory of some Android mobile phones which store data using raw flash memory.

YAFFS allocates memory by selecting an unused erase block and sequentially allocating the numbered pages (which YAFFS calls chunks) in that erase block. An allocated erase block is freshly erased and therefore devoid of any data. YAFFS searches for empty erase blocks (i.e., ones that contain no valid data) sequentially by the erase block number as defined by the physical layout of memory on the storage medium, wrapping cyclically when necessary. It begins searching from the most-recently allocated erase block and returns the first empty erase block.

YAFFS performs *compaction* (which YAFFS calls garbage collection) to reclaim wasted space on partially-full erase blocks. As illustrated in Figure 4.2, compaction copies all valid (i.e., non-discarded) pages from some partially-full erase block to the log's end; compaction then erases the source erase block, which now contains no valid data. If there is no erase block that can be compacted, that is, there is not a single unneeded page stored on the medium, then YAFFS reports the file system as full and fails to allocate an erase block.

Compaction in YAFFS is either initiated by a thread that performs system maintenance, or takes place during write operations. Usually, a few pages are copied at a time, thus the work to copy an erase block is amortized over many write operations. If the file system contains too few free erase blocks, then a more aggressive compaction is performed. In this case, erase blocks with *any* amount of discarded space are compacted.

YAFFS selects erase blocks for compaction using a greedy strategy based on the ratio of discarded pages on an erase block, however it only searches within a small moving range of erase blocks with a minimum threshold for discarded pages. This cyclic and proactive approach to compaction results in a strong cyclic trend in erase block allocations. When low on free space, YAFFS selects the erase block with the most wasted space by examining all the storage medium's erase blocks.

There are currently two major versions of YAFFS, YAFFS1 and YAFFS2, and among their differences is how file deletion is performed. In YAFFS1, a special *not-deleted* flag in the file's header is set to **1**; when the file is deleted the header is programmed a second time (without first erasing it) to contain the same contents except the flag is set to **0**. Note that this technique is similar to Wei et al.'s *scrubbing* [39]. In YAFFS2, this multiple programming is obviated by writing a new file header instead; this change is to allow YAFFS to support all flash memories, many of which do not permit multiple programmings. We used YAFFS2 for our experiments in this chapter.

5.4 Data Deletion in Existing Log-Structured File Systems

In this section, we investigate data persistence on log-structured file systems by analysing the internal memory of a Nexus One running Android/YAFFS and simulating larger storage media. We instrument the file system at the kernel level to log erase block allocation information. This provides an upper bound on the deletion latency, because allocating an erase block for storage implies that it was previously compacted and erased, and therefore all discarded data previously stored *on that erase block* is securely deleted.

Figure 5.1 shows four data items stored on two erase blocks, each one with two pages. Different data items are indicated with different pattern. Each erase block shows a timeline of what data is stored on which page at which time. Data item create and discard events are indicated. Moreover, when an erase block is reallocated, valid data is copied to another page. The horizontally-striped data item, for example, is twice copied before it is discarded. At the bottom is illustrated the lifetime of data items as well as the time that they are compromisable due to deletion latency. The time between two erase block reallocations is labelled as the erase block reallocation period. All data discarded within this period has its deletion latency bounded by it. Observe that the reallocation period is not fixed for all times and erase blocks; it depends on how the storage medium is used.

In this section we show that modern Android smartphones have large deletion latency, where deleted data can remain indefinitely on the storage media. This motivates the secure deletion solutions in Section 5.5.

5.4.1 Instrumented YAFFS

We built a modified version of the YAFFS Linux kernel module that logs data about the writing behaviour of an Android phone. We log the time and number for every erase block allocation and erasure. This information shows us where YAFFS stores data written at some point in time and when that data becomes irrecoverable. This allows us to compute the deletion latency of data in our simulation.

We used the instrumented phone daily for 670 hours, roughly 27.9 days. Throughout the experiment we recorded 20345 erase block allo-

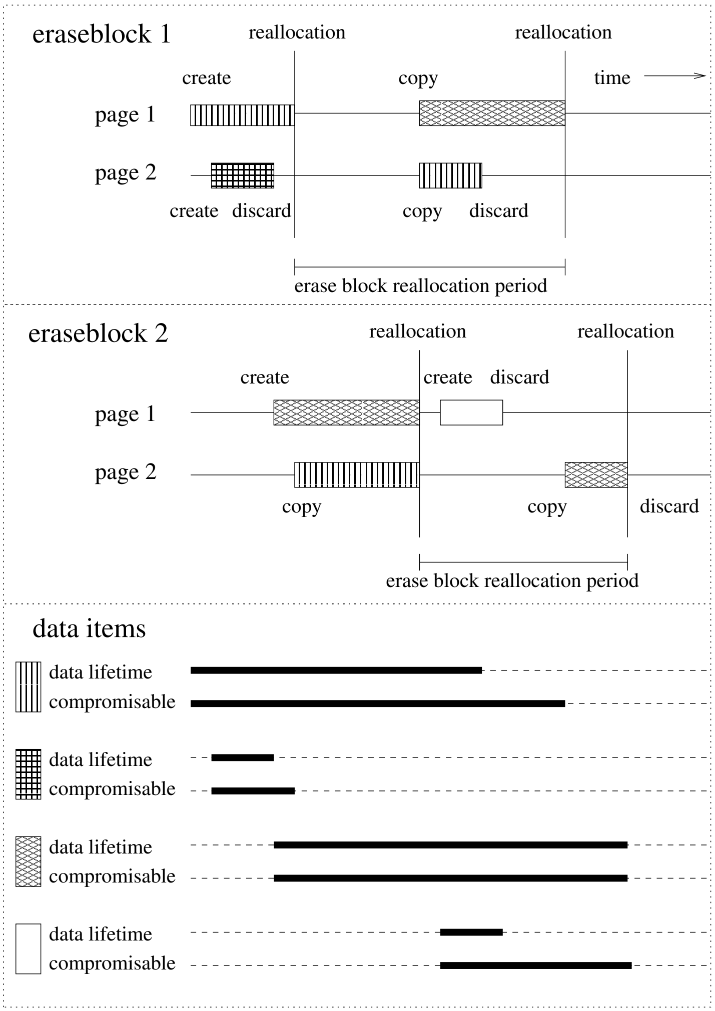


Figure 5.1: Example timeline of data items stored on two erase blocks. Each erase block is twice reallocated and the reallocation period for them is indicated. Different data items have different patterns. The bottom illustrates each data item's lifetime and compromisable time (i.e., lifetime plus deletion latency).

cations initiated by 73 different *writers*. A *writer* is any application, including the Android OS itself or one of its services (e.g., GPS, DHCP, compass, etc.). The experiment's logs show that the median time between erase blocks reallocations is 44.5 hours. The deletion latency is always less than the reallocation period; this means that the median deletion latency is upper bounded by this value.

5.4.2 Simulating Larger Storage Media

Log-structured file systems favour allocating empty erase blocks before compacting partially-empty erase blocks [82, 86]. We hypothesize that the erase block reallocation period—and consequently the deletion latency—is highly dependent on the file system's size. We tested this hypothesis by writing a discrete event simulator to experiment with the writing behaviour of an Android phone on simulated YAFFS storage media of various sizes. We first describe our experimental setup and then present our results.

Experimental Procedure. To experiment with different flash storage medium sizes, we simulated an Android mobile phone using a flash storage medium in memory. We used our own discrete event simulator that writes, overwrites, and deletes files on a storage medium. This medium is a directory on our computer that simulates accessing flash memory through a flash file system.

We used the collected statistics from our instrumented phone in Section 5.4.1 to determine the writing behaviour for our discrete event simulator. We logged every page that was written to the device for a week, and used this data to compute the period between successive creations of new files, and the characteristics of the files that are created. The characteristics of files are the following:

- The file's lifetime.
- A distribution over the period of time between opening a file for write.
- A distribution over the number of pages to write to a file each time it is opened.
- A distribution over a file's pages where the writes occur.

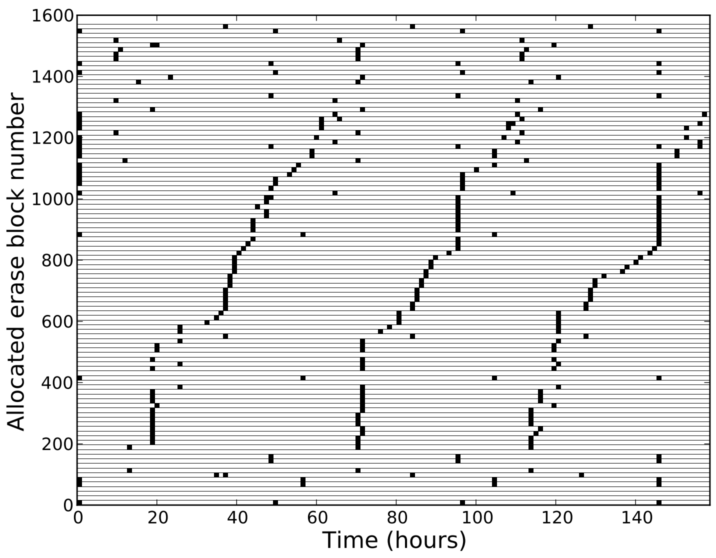


Figure 5.2: Sampled plot of erase block allocation over time for YAFFS on an Android phone. The time between two points on the same horizontal line is the erase block reallocation period.

Additionally, we implemented a pattern writer that operated alongside the simulated writers. It periodically writes a one-page pattern, waits until a new erase block is allocated, and then deletes the pattern. We use the pattern writer to determine the deletion latency for data written at that particular moment in time, but which remains stored; it represents the writing of some sensitive data that is later discarded.

We perform experiments using YAFFS mounted on a virtual flash storage medium created by the kernel module `nandsim`. We use an erase block size of 64 2-KiB pages, consistent with the Nexus One phone [78].

Deletion Latency. Figure 5.2 shows a plot of the storage media's erase block allocations over time to gain an intuition on its behaviour. The horizontal axis is time, and the vertical axis shows the sampled space of sequentially-numbered erase blocks. A black square on the graph means that an erase block was allocated at that time. For clarity,

Partition	Deletion latency (hours)	
size / type	median	95th %ile
200 MB YAFFS	41.5 ± 2.6	46.2 ± 0.5
1 GB YAFFS	163.1 ± 7.1	169.7 ± 7.8
2 GB YAFFS	349.4 ± 11.2	370.3 ± 5.9

Table 5.1: Deletion latency in hours for different configuration parameters.

we compress the space of erase blocks into the rows by sampling every 15th erase block.

We present the results of our experiment in Table 5.1, which gives the median and 95th percentile deletion times in hours for the patterns written onto the storage medium during simulation. The maximum deletion latency is undefined because these systems provide no deletion guarantee and some data remained available after the experiment. Table 5.1 provides results for YAFFS partitions with sizes 200 MiB, 1 GiB and 2 GiB based on our observed access patterns.

We observe the effect of cyclic erase block allocation in YAFFS. There is both a linear growth in deletion latency as the size of the partition increases, and a high percentile observation close to the median. For instance, a YAFFS implementation on a 2 GB partition (e.g., the data partition on the Samsung Galaxy S [94]) with the same access patterns can expect deleted data to remain up to a median of two weeks before actually being erased. In the next section, we present solutions to reduce this data deletion latency.

5.5 User-space Secure Deletion

In this section, we introduce our solutions for secure deletion: purging, ballooning, and a hybrid of both. These solutions all work at user-level, which has a limited interface that can only create, modify, and delete the user's own local files. Such solutions cannot force the file system to perform erase block erasures, prioritize compaction of particular areas in memory, or even know where on the storage medium the user's data is stored.

All of the solutions we present operate with the following principle: they reduce the file system's available free space to encourage more frequent compaction, thereby decreasing the deletion latency for deleted data. Purging consists of filling the storage medium to capacity, thus ensuring that no deleted data can remain on the storage medium. Purging executes intermittently and halts after completion. Ballooning continually occupies some fraction of the storage medium's empty space to ensure it remains below a target threshold, thereby reducing the deletion latency. Ballooning executes continually during the lifetime of the storage medium. The hybrid solution performs ballooning continually, and performs a clock-driven purge operation to guarantee an upper bound on deletion latency.

We implement our solutions and examine their effectiveness for various storage medium sizes. We use deletion latency and storage medium wear as metrics for evaluating their effectiveness. We show that the hybrid solution is well-suited for large storage media, where the deletion latency is a tradeoff with storage medium wear.

5.5.1 Purging

Purging attempts to completely fill the file system's empty space with junk files; if the operation is successful then all partially-filled erase blocks on the storage medium are compacted and therefore all previously discarded data is securely deleted. Importantly, whether completely filling the file system from user space actually completely fills the storage medium depends on the implementation of the actual file system.

After filling the storage medium, the junk files are deleted so that the file system can again store data. Purging must be explicitly executed, which can take the form of automated triggers: when the phone is idle, when the browser cache is cleared, or when particular applications are closed. It is also useful for employees who are contractually obligated to delete customer data, e.g., before crossing a border.

The fact that the storage medium must be completely filled follows from a worst-case analysis of a SEMIPERSISTENT implementation whose allocatable space is the same as the addressable space. Before the storage medium is completely full, there is some area of the medium containing one last piece of unneeded but available data—we must pessimistically assume that is our discarded data. It is important to note that purging's ability to securely delete data is dependent on the implementation of the log-structured file system. In particular, we require the following condition to hold: if the file system reports that it is out of space, then all previously deleted pages are no longer available on the storage medium. This condition holds for YAFFS and Linux's software FTL implementation (version 2.6.36.1), however the implementation of other flash file systems and FTL hardware may differ.

A natural concern for purging's correctness is its behaviour on multithreaded systems. However, using the previous reasoning, purging needs to keep writing to the storage medium until it reports that it is completely full. This ensures that any data that has been deleted prior to purging is irrecoverable as the drive is completely full. Another concern is that, at the moment the storage medium is full, other applications simultaneously writing to the storage medium are told that the storage medium is full. We observe that any ungraceful handling of an unwritable storage medium is a flaw in the application and the storage medium's lack of capacity is a transient condition that is quickly relieved.

We tested purging with the following experiment. We took a pristine memory snapshot of the phone's internal NAND memory by logging into the phone as root, unmounting the flash storage medium, and copying the raw data using `cat` from `/dev/mtd/mtd5` (the device that corresponds to the phone's data partition) to the phone's external memory (SD card). We wrote an arbitrary pattern not yet written on the storage medium, and obtained a memory snapshot to confirm its presence. We then deleted the pattern, obtained a new memory snapshot, and confirmed that the pattern still remained on the flash memory. Finally, we filled the file system to capacity with a junk file, deleted it, and obtained another memory snapshot to confirm that the pattern was no longer on the flash memory.

The time it took to execute purging on the Nexus One was between thirty seconds to a minute. As we soon see, however, this time is highly dependent on the storage medium's size. During execution the system displayed a warning message that it was nearing drive capacity, but the warning disappeared after completion.

Figure 5.3 shows the resulting erase block allocations reported by an instrumented version of YAFFS executing purging. The horizontal axis corresponds to time in hours, and the vertical axis shows the sampled space of numbered erase blocks. A small black square in the graph indicates when each erase block was allocated. For clarity, as with Figure 5.2, only a sampled subset of erase block (every 15th) have their allocations plotted. At the right side of Figure 5.3, we see the near immediate allocation of every erase block on the medium as indicated by

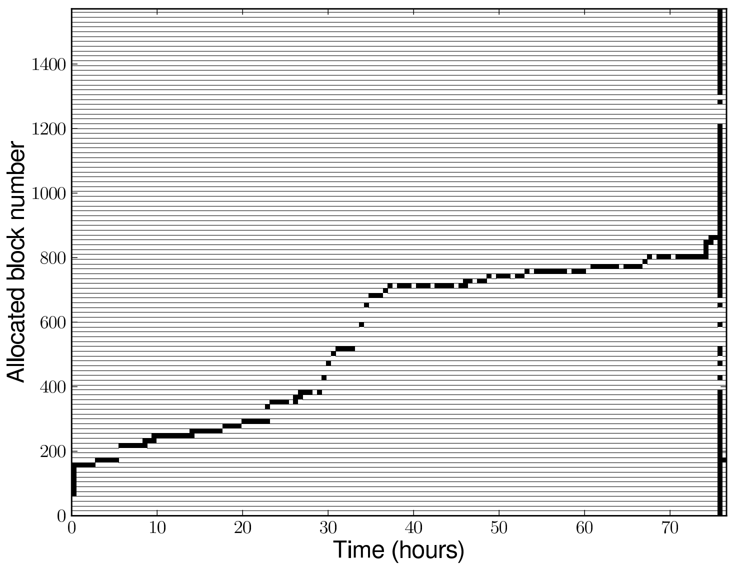


Figure 5.3: Plot of erase block allocation over time for YAFFS (cf. Figure 5.2). After simulating writing for some time, we performed purging, which is visible at the right edges of the plot where many erase blocks are rapidly allocated.

the black squares forming a near vertical line. This is the consequence of filling the storage medium to capacity; a log-structured file system must compact every erase block that contains at least one deleted page.

5.5.2 Ballooning

In contrast to purging, which guarantees secure data deletion with a bounded deletion latency, we now present ballooning, which does not guarantee secure deletion with any bound but does reduce the deletion latency in expectation. Ballooning artificially constrains the file system's available free space. This results in more frequent compaction due to reduced capacity, and therefore reduces the time any deleted data—regardless of *when* it is deleted—remains accessible on a log-structured file system. Ballooning creates junk files to occupy the free

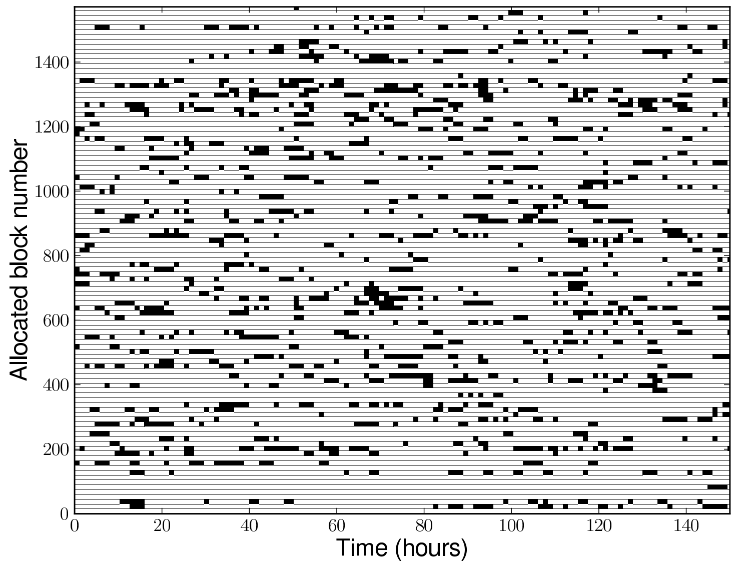


Figure 5.4: Plot of erase block allocation over time for YAFFS while using aggressive ballooning.

space of the storage medium, which reduces the total number of erase blocks available for allocation. This reduces the expected erase block reallocation period, and therefore the expected deletion latency. These ballooning junk files are periodically rotated—new ones written and then old ones deleted—to promote efficient wear levelling.

In Section 5.6, we explore how varying free space thresholds—the aggressiveness of ballooning—affect deletion latency and other measurements. First, however, we visualize evidence that does not refute our hypothesis that ballooning reduces the erase block reallocation period. Figure 5.4 shows the erase block allocations that result from executing ballooning on YAFFS. We see a stark difference when compared with Figure 5.2. As the number of allocatable erase blocks decreases, YAFFS' sequential allocation becomes much more erratic, and the erase block reallocation period decreases. Row segments in Figure 5.4 that contain no allocation activity (i.e., a black square) likely correspond to erase blocks that are now filled with junk files. The figure shows a decrease in the erase block allocation period, which therefore reduces the expected deletion latency.

5.5.3 Hybrid Solution: Ballooning with Purging

The disadvantage of purging is that its cost is dependent on the free space available on the storage medium. In contrast, the disadvantage of ballooning is that it cannot provide a guarantee on when (or indeed if) data is deleted. By combining both these solutions, we create a hybrid scheme that has neither disadvantage. We use periodic purging for secure data deletion, and we use ballooning to ensure that a large storage medium's empty space must not be refilled during every purging operation. The result is a clock-based solution where purging is periodically performed, dividing time into deletion epochs. The deletion latency of all data is therefore bounded by the duration of a deletion epoch. The resulting storage medium has a SECDEL-CLOCK behaviour.

Reducing the number of erase blocks that must be filled during purging mitigates three concerns: purging's wear on the storage medium, its power consumption, and its execution time. Large capacity storage media are particular suitable to this solution: they may have large segments of their capacity empty, which ballooning occupies with junk files to achieve a deletion latency representative of smaller-sized storage media. In the next section we quantify this with experimental results for various storage medium sizes and ballooning aggressiveness settings.

5.6 Experimental Evaluation

We developed an application that implements our hybrid solution. The application periodically examines the file system to determine the free space, and appropriately creates and deletes junk files to maintain the free space within the upper and lower thresholds. The lower threshold is user defined and we set the upper threshold to be 4 MiB larger than the lower threshold to avoid a thrashing effect. The oldest junk file is always deleted before more recent ones to load-balance flash memory wear. Long-lived junk files can also be removed, with new ones written, to perform appropriate wear-levelling if necessary. The purging interval is user-specified, allowing the user to select a tradeoff between the timeliness of secure deletion and the resulting wear on the device.

Our application runs successfully on the Android phone. The only permission it requires is the ability to run while the phone is in a locked state; the application also needs to specify that it runs as a service, meaning execution occurs even when the application is not in the foreground. The application can be installed on the phone without any elevated privileges and operates entirely in user-space. Ballooning must maintain a minimum of 5% of the erase blocks free to avoid perpetual warnings about low free space. Purging triggers a brief warning about low free space that disappears when purging completes.

We now present the experiments we performed using ballooning on simulated flash media of different sizes. We varied the amount of ballooning that was performed and measured the time that discarded data remained on the storage medium to determine ballooning's effectiveness. We measured the ratio of deleted pages on erase blocks, which intuitively captures the amount of ballooning. We also measured the rate of flash erase block allocations, which intuitively captures the added cost of ballooning. After each simulation execution, we performed purging and measured the additional erase block allocations, which is the purging cost for the amount of ballooning used by our hybrid solution.

The erase block allocation rate tells us directly the rate that pages are written to the flash storage medium. Data can be written from two sources: the actual data written by the simulator, and the data copied by the log-structured file system's compactor. Our simulator uses a constant write distribution and therefore the expected rate of writes from the simulator is the same for all experiments. Therefore, the observed disparity in erase block allocation rates reflects exactly the additional writes resulting from the increased compactions caused by our application to achieve secure deletion.

To quantify how promptly secure data deletion occurs, we measure the expected time data remains on the storage medium. We calculate this measurement using our pattern writer that periodically writes one page pattern onto the medium and deletes them. We then compute how long the written pattern remains on the storage medium.

5.6.1 Experimental Results.

Table 5.2 present the results of simulated storage media usage with different ballooning thresholds. The partition size is the full storage capacity of the medium. The fill ratio is the average proportion of valid

Partition type	Free EB	Fill ratio	EB allocs per hour	Life years	Purge cost EB	Deletion latency hours median	95th %ile
200 MB YAFFS	603.8	20%	32.7 ± 2.3	54	1556.8	41.5 ± 2.6	46.2 ± 0.5
	91.8	63%	53.4 ± 4.7	33	705.2	10.8 ± 1.7	14.6 ± 1.3
	21.0	80%	95.0 ± 24.2	18	429.8	4.2 ± 0.6	6.6 ± 0.2
	15.1	84%	166.5 ± 42.5	10	357.8	2.6 ± 0.7	5.4 ± 1.5
1 GB YAFFS	4487.2	7%	26.0 ± 1.0	68	7827.0	163.1 ± 7.1	169.6 ± 7.8
	254.1	40%	35.8 ± 3.4	50	1106.5	28.4 ± 4.1	33.6 ± 2.6
	88.2	64%	59.8 ± 8.4	29	765.0	10.4 ± 0.5	16.1 ± 2.0
	56.2	72%	70.4 ± 0.8	25	692.3	8.2 ± 0.6	12.6 ± 2.6
	26.1	82%	163.6 ± 18.9	10	525.2	4.3 ± 0.4	7.6 ± 0.6
	23.7	83%	232.9 ± 11.4	7	360.8	3.0 ± 0.4	6.1 ± 0.6
2 GB YAFFS	9503.7	4%	25.3 ± 0.8	70	15663.8	349.4 ± 11.2	370.3 ± 5.9
	387.8	43%	36.6 ± 1.5	49	1630.5	34.7 ± 7.5	43.1 ± 8.6
	254.5	48%	41.1 ± 3.7	43	1237.5	28.7 ± 1.5	34.8 ± 6.1
	56.4	76%	87.5 ± 5.8	20	845.8	8.5 ± 0.9	13.0 ± 0.4
	37.2	80%	205.4 ± 24.3	8	484.8	4.7 ± 0.5	9.4 ± 1.9
	36.9	80%	248.2 ± 33.0	7	338.4	3.3 ± 0.7	7.4 ± 1.0

Table 5.2: Erase block (EB) allocations, storage medium lifetimes, and deletion times for the YAFFS file system.

data on erase blocks in the storage medium, ignoring both completely full and completely empty erase blocks. We compute this by taking the periodic average of all fill ratios for eligible erase blocks, and averaging these measurements (weighted by time between observations) over the course of our experiment. The erase block allocations per hour is the rate that erase blocks are allocated on the storage medium, indicating the frequency of writes to the storage medium. We used the erase block allocation rate, along with an expected erase block lifetime of 10^4 erasures before becoming a bad block [79], to compute an expected storage medium lifetime in years assuming even wear levelling. The purge cost is the number of erase blocks that must be allocated to execute purging with this configuration. Two deletion latencies are provided: the median and 95th percentile, which give a good indication of the distribution. The maximum value is undefined, as ballooning provides no guarantee of secure deletion. Each experiment was run four times and we provide 95% confidence intervals for relevant measurements.

Deletion Latency versus Block Allocation Rate. As discussed in Section 5.4.2, without ballooning both the fill ratios and the deletion latency are highly dependent on the size of the storage medium. As ballooning increases the fill ratio, however, the deletion latency similarly

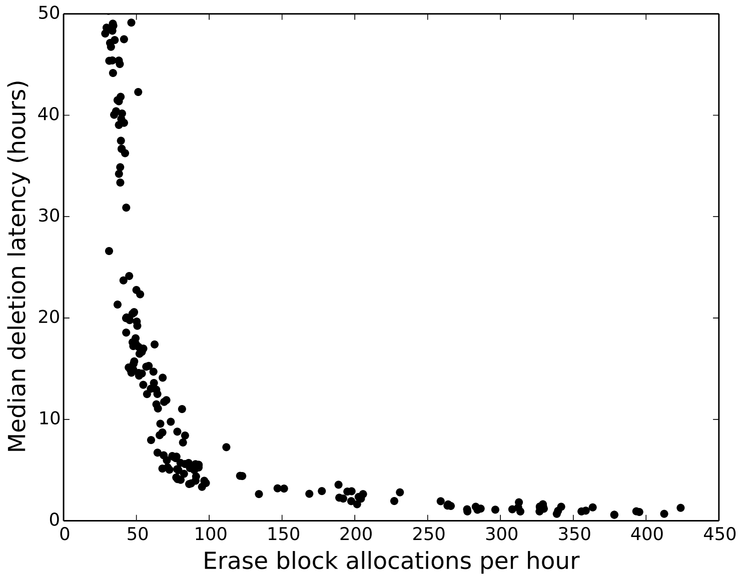


Figure 5.5: Scatter plot of deletion latency and erase block allocation for experiments on a 200 MiB storage medium with varied ballooning.

decreases. Since the data being stored comes from the same distribution, fuller erase blocks on identically-sized storage media imply that there are fewer erase blocks available to store data, so the expected erase block reallocation period decreases and therefore deleted data is removed from the system more frequently.

We observe an inverse relationship between the fill ratio and the erase block allocation rate for each partition type. Fewer available erase blocks mean more compaction and thus more frequent writes to the storage medium simply to copy data stored elsewhere. Figure 5.5 plots the relationship between the median deletion latency and the erase block allocation rate for simulations involving varying amounts of ballooning. The horizontal axis is the erase block allocation rate and the vertical axis is the median deletion time. A point on the plot represents an experiment with some amount of ballooning that resulted in the observed allocation rate and deletion latency.

The device's size is not a overriding factor in deletion latency—deletion latency can be reduced for any storage medium simply by applying the appropriate amount of ballooning to consume the excess capacity. Small amounts of ballooning on large storage media—which slightly increase the erase block allocation rate—can significantly drop the deletion latency. This is because the vast number of unused erase blocks are not allocated by greedy or cyclic allocation algorithms as the file system believes them to be full.

Hybrid Ballooning and Purging. The purge cost column of Table 5.2—where cost is measured as the number of erase blocks that must be erased to execute purging—was computed by executing purging after each experiment and measuring the number of erase block allocations that resulted. We see that when ballooning is not used, the purging cost is equal to the full size of the partition. For large partitions, this results in an unreasonable number of erase block allocations required for purging. We see that mild amounts of ballooning drastically reduce the cost of purging. In fact, for the two gigabyte YAFFS partition, a 50% increase in erase block allocations results in a ten-fold improvement in both deletion latency and purging cost.

Ballooning and Storage Medium Lifetime. The primary drawback of our solutions is the cost of increased erasures, both in terms of damage to the flash memory and power consumption. The additional wear is directly proportional to the increase in the erase block allocation rate, and inversely proportional to the lifespan. We compute an expected lifetime in years from the erase block allocation rate and present this in Table 5.2. We use a conservative (i.e., pessimistic) estimate of 10^4 erasures per erase block. Recall that a typical flash erase block can handle between 10^4 and 10^5 erasures [95], and some studies have indicated this is already orders of magnitude more conservative than reality [96].

Our results show that even at high erase block allocation rates, we still expect to see the storage medium live for upwards of a decade; this is well in excess of the replacement period of mobile phones that varies between two to eight years [97]. Users who require decades of longevity from their mobile phone can simply use mild ballooning. In particular, large capacity storage media combined with mild ballooning yield a system with reasonable purging performance and reduced flash memory lifetimes.

Power Consumption. To test if our solutions have acceptable power requirements, we analysed the power consumption of write operations. We measured the battery level of our Nexus One through the Android API, which gives its current charge as a percentage of its battery capacity. The experiment consisted of continuously writing data to the phone's flash memory in a background service while monitoring the battery level in the foreground. We measured how much data must be written to consume 10% of the total battery capacity. We ran the experiment four times and averaged the result. The resulting mean is within the range of 11.01 ± 0.22 GB with a confidence of 95%, corresponding to 90483 full erase blocks worth of data. Since this well exceeds the total of 1570 erase blocks on the device's data partition, we are certain that our experiment must have erased the erase blocks as well as written to them, thus measuring the power consumption of the electrically-intensive erasure operation.

Even using the most aggressive ballooning measurement for YAFFS, where nearly 250 erase blocks are allocated an hour, it would take 15 days for the ballooning application's writing behaviour to consume 10% of the battery. Furthermore, the built-in battery use information reported that the testing application was responsible for 3% of battery usage, while the Android system accounted for 10% and the display for 87%. We conclude that ballooning's power consumption is not a concern.

The power consumption required for purging is related to the size of the storage medium and the capacity of the battery—0.9% of the battery per gigabyte for the Nexus One. Other mobile phone batteries may of course yield varying results. Any mobile phone with a storage medium size exceeding a gigabyte therefore consumes significant time and energy to perform purging. Our hybrid solution, however, is perfectly suited for such storage media as it significantly drops the cost of purging.

5.7 Summary

In this chapter we considered deletion latency for log-structured file systems and showed that there is no guarantee of deletion on such file systems. We presented three useful user-level solutions for secure deletion on YAFFS file systems: purging, ballooning, and a hybrid of both. The hybrid provides secure data deletion against a computationally-unbounded unpredictable multiple-access coercive adversary, turning the storage medium into a SECDEL-CLOCK implementation. We have evaluated their effectiveness in terms of wear on the flash memory, as well as power consumption and time.

We restate that these solutions make strong assumptions on the implementation that stores the data; in particular, that by filling the capacity of the file system effects the secure deletion of all discarded data. Verifying this is simple for interfaces like MTD which provide raw access to the flash memory, however it is not as straightforwards when the memory is hidden behind an obfuscating controller.

We have also seen that user-level solutions are limited. The space of possible solutions is constrained to creating and deleting files. We showed that by filling certain log-structured file systems to capacity, we can securely deleted data. It requires that the file system reclaims all wasted storage resources before proclaiming the device is full.

In the next chapter, we consider what can be achieved without a user-level access restriction and develop an efficient and prompt secure deletion solution that can be integrated into any flash file system.

Chapter 6

Data Node Encrypted File System

6.1 Introduction

This chapter presents the Data Node Encrypted File System (DNEFS), a file system modification that augments a file system with efficient secure deletion. DNEFS is tailored to flash memory, which inhibits secure deletion by forcing all deletions to occur at a large granularity. Consequently, both flash file systems and flash translation layers implement a log-structured file system; one where new data is written at the log's end and logical addressing is used to locate file data. In log-structured file systems, data deletion happens only during compaction when the physical storage is needed for new data.

DNEFS encrypts each individual indivisible data node, i.e., the smallest unit of read/write for the file system.[1] Each data node is assigned a unique key and keys are colocated in a small set of erase blocks. This set is called the key storage area (KSA). While the main storage remains unchanged, the KSA must be capable of securely deleting data. The efficiency of the system comes from the size disparity between the key storage area and the main storage, as only the former needs to perform the expensive deletion operation.

[1]Note that we use the term data node for consistent terminology with our UBIFS-based implementation in the next chapter. A data node is the same as a data item: an indivisible piece of data, which forms the minimum unit of I/O for system.

6.2 System and Adversarial Model

DNEFS is designed to provide secure data deletion for flash memory. As with the solutions presented in the previous chapter, we assume that the user is continually storing and retrieving sensitive data on a flash-based storage medium such as a mobile phone.

The adversary can perform a coercive attack at multiple points in time to gain access to the user's device. The adversary is computationally bounded and therefore cannot decrypt encrypted data without the corresponding key.

The user's goal is to provide secure deletion for as much data as possible. In particular, it is to limit the information learned by the adversary through compromise to only the valid data currently stored on the device. DNEFS approaches this goal by providing secure data deletion with configurable existential and deletion latencies.

6.3 DNEFS's Design

In this section we describe DNEFS: a file system modification that provides efficient fine-grained secure deletion for flash memory. DNEFS uses encryption to provide secure deletion. It encrypts each individual data node (i.e., data item, or the unit of read/write for the file system) with a different key, and then manages the storage, use, and secure deletion of these keys in an efficient and transparent way for both users and applications. Data nodes are encrypted before being written to the storage medium and decrypted after being read; this is all done in-memory. The keys are stored on a reserved area of the file system called the key storage area (KSA). Figure 6.1 illustrates applying DNEFS to a file system and partitioning the storage medium.

DNEFS works independently of the notion of files; file count, file size, and file access patterns have no influence on the size of the key storage area. The encrypted data stored on the medium is no different than any reversible encoding applied by the storage medium (e.g., error-correcting codes) because all legitimate access to the data only observes the unencrypted form. This is not an encrypted file system, although in Section 6.4.5 we explain that it is easily extended to one. In our case, encryption is simply a coding technique that we apply immediately before storage to reduce the number of bits required to securely delete a data node from the data node size to the key size.

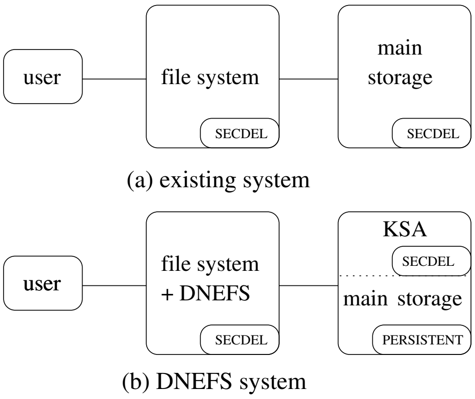


Figure 6.1: Adding DNEFS to an existing file system. (a) The existing system consists of a SECDEL storage medium accessed by the user through a file system; deletion may be an expensive operation. (b) A DNEFS system consists of a securely-deleting key storage area and a persistent main storage both compromising the storage medium. The efficiency of the system comes from applying the expensive secure-deletion operation only to the KSA.

6.3.1 Key Storage Area

We assume that the storage medium is divided into erase blocks which can be atomically erased, e.g., flash memory erase blocks. Our solution uses a small set of *erase blocks* to store all the data nodes' keys—this set is called the Key Storage Area (KSA). The erase blocks that are not part of the KSA belong to the main storage. The KSA is managed separately from the rest of the file system. It does not behave like a log-structured file system; instead secure deletion is explicitly provided by using batched compactions (see Section 4.3). Our solution therefore requires that the file system or flash controller that it modifies can logically reference the KSA's erase blocks and erase old KSA erase blocks promptly after writing a new version. Figure 6.1 emphasizes that the KSA, unlike the main storage, requires secure deletion.

Each data node's header stores the logical KSA position that contains its decryption key. The erase blocks in the KSA are periodically erased to securely delete any keys that decrypt discarded data. When the file system no longer needs a data node—i.e., the data node is removed or updated—the data node's corresponding key is discarded. This data-node-based approach is independent of files; keys are discarded whenever the data node they encrypt is discarded. A key remains in the discarded state until it is securely deleted from the KSA and its location is replaced with fresh, unused, random data; its state is then changed to unused.

When a new data node is written to the storage medium, an unused key is selected from the KSA and its position is stored in the data node's header. The key is used to encrypt the data node using a symmetric key block cipher in counter mode; we use a fixed initialization vector because keys are never used to encrypt multiple data nodes. DNEFS does cryptographic operations seamlessly, so applications are unaware that their data is being encrypted. Figure 6.2 illustrates DNEFS's (a) write, (b) read, and (c) discard algorithms.

6.3.2 Keystore

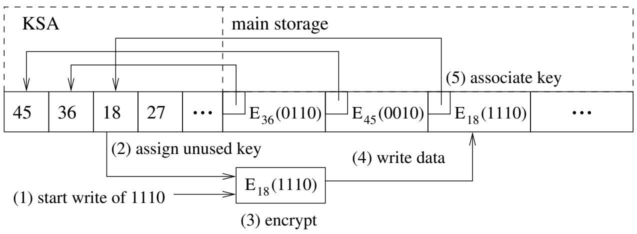
The KSA is used to implement a *keystore*, which is a special kind of storage medium whose purpose is to assign and securely delete cryptographic keys, which we call *key values* (KVs). These key values are assigned to encrypt new data and are deleted in lieu of the data they encrypt to provide secure deletion against our computationally-bounded adversary.

A *keystore* has a state and three functions: `assign`, `read`, and `discard`. The keystore's state can be examined by the adversary during a coercive attack. A keystore has a security parameter κ, which is used as the length of KVs. Additionally, each KV has an access token (AT) that uniquely identifies a KV from the time it is assigned until it is discarded. The `assign` function takes no parameters and returns an AT or $\perp$.[2] The `read` function takes an AT and returns a KV or $\perp$. The `discard` takes an AT and returns $\top$ for success or $\perp$ for failure.

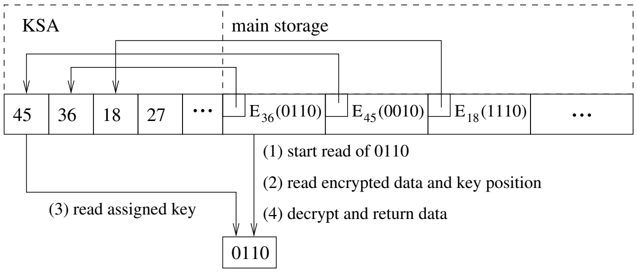
The following properties ensure that this system provides secure data deletion at a fine granularity:

- **P1** The KVs associated to the ATs returned by `assign` must be unpredictable.
- **P2** KVs returned by `read` must be the same for a particular AT from the time `assign` returns it until it is provided to `discard`; further, `read` must not return $\perp$ during this time.

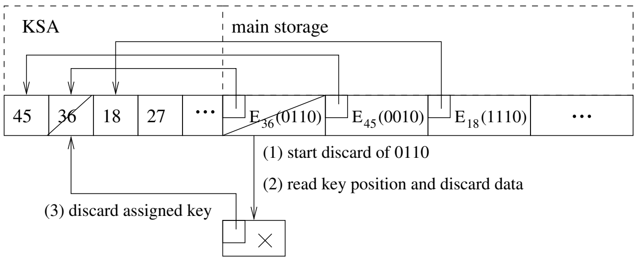
[2]In implementation the assign function returns both the AT and the KV in anticipation of an immediate read.



(a) DNEFS write operation



(b) DNEFS read operation



(c) DNEFS discard operation

Figure 6.2: DNEFS's read, write, and discard datapaths. Data blocks are represented with binary strings and encryption keys are represented with two digit numbers. (a) Writing a new data node 1110: 1110 is first encrypted with an unused key 18 and then written to an empty position in the main storage with a reference to the key's position stored alongside. (b) Reading a data node 0110: $E_{36}(0110)$ is first read from the main storage along with a reference to its key 18 in the KSA. The key is then read and used to decrypt the data. (c) Discarding a data node 0110: the key position associated to it is read, the data node and its corresponding key position are discarded.

- **P3** KVs returned by `read` must be unpredictable given the keystore's state at all times before the time `assign` returns its corresponding AT minus a bounded existential latency, and at all times after the time `discard` is called with its corresponding AT plus a bounded deletion latency.

With reference to Figure 6.2, DNEFS's use of the keystore consists of three operations: (a) assigning an unused key, (b) reading an assigned key, and (c) discarding an assigned key. We now describe an example implementation of a keystore, which we call a *clocked keystore*.

6.3.3 Clocked Keystore Implementation

A clocked keystore consists of a set of key positions (KPs), each one storing a KV and a key state. The three key states are **U** for *unused*, **A** for *assigned*, and **D** for *discarded*. The assign operation provides an *unused* (**U**) KP and changes its state to *assigned* (**A**). The read operation returns the KV for an *assigned* (**A**) KP. The discard operation takes an *assigned* KP and changes its state to *discarded* (**D**).

A periodic *clock* operation securely deletes any discarded keys and returns their state to unused. During this operation, KVs for both **U** and **D** KPs are replaced with fresh, unused random data taken from a cryptographically-suitable random source, e.g., `/dev/random`. Thus, unused KPs store a recently-generated random KV not previously assigned; assigned KPs are those that have been assigned but not discarded; discarded KPs are those that have been assigned and then discarded, though the KV remains until it is securely deleted.

The keystore design achieves properties **P1–3**. **P1** is holds because only **U** are assignable and they store unpredictable random KVs and assigning transitions it to **A**. **P2** holds because from the time a KP's state is **A** until it is **D**, the corresponding KV remains stored and is always returned. **P3** holds by replacing the KVs for both **U** and **D** KP during the clock function. The period between clock functions therefore forms an upper bound on the possible existential and deletion latencies.

Figure 6.3 shows the three states available for a KP and an example history of state transitions for a KP in a clocked keystore implementation. Observe that KVs are already stored on the storage medium before they are assigned and they remain on the storage medium for a limited amount of time after they are discarded. The periodic clock operation divides time into distinct deletion epochs, and the KV stored

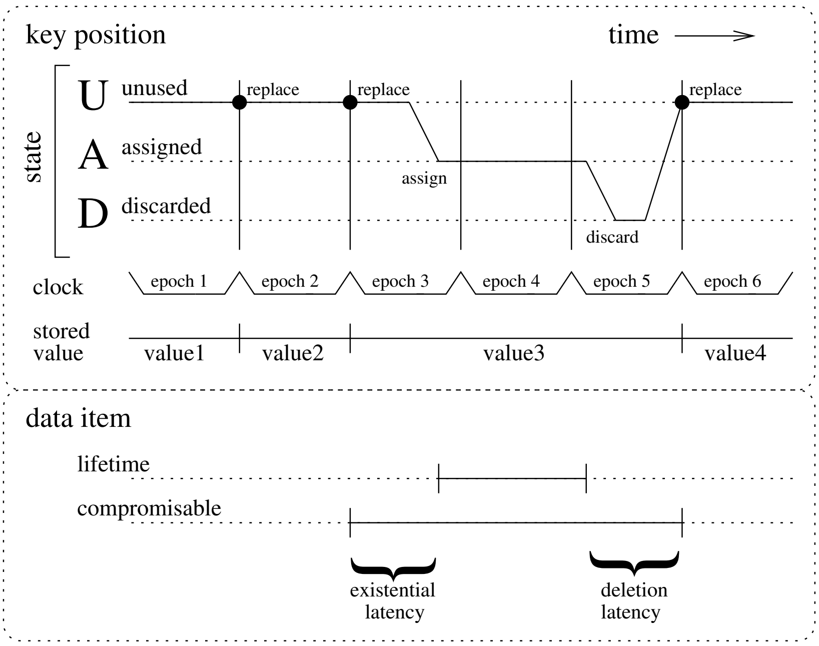


Figure 6.3: Example timeline for a KP's state and value in a clocked key-store as well as a data node that uses the KV. The three states are unused, assigned, and discarded. A solid line plots a function over time going among these states: starting unused, then being assigned in epoch two, discarded in epoch five, and finally returning to unused. The clock triggers periodically resulting in a vertical stroke in the state. Black dots and a *replace* label indicate the creation of a new KV. Over six deletion epochs, four different KVs occupy the KP. One KV, `value3`, is contained at the time the data node is created. It is assigned and discarded at the data node's creation and deletion times, respectively. The resulting existential latency and deletion latency are indicated in the increased time that the data node is compromisable.

in the KP changes only at the clock. An example data item has a lifetime from the third to fifth deletion epochs; its existential and deletion latencies expand its compromisable timespan in both directions to the nearest clock edge. Thus, a compromise at any timepoint is equivalent to a compromise at any other time point in the same epoch.

6.3.4 Clock Operation: KSA Update

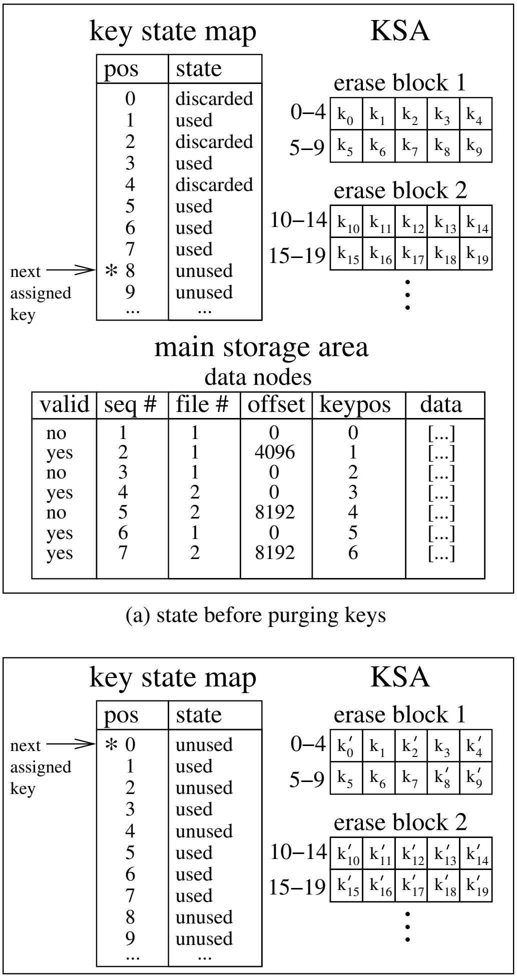
DNEFS clock operation, called KSA update, replaces all **U** and **D** keys with fresh values. It executes iteratively over each of the KSA's erase blocks as follows: a new version of the erase block is prepared where the **A** keys remain in the same position and all the **U** and **D** keys are replaced with fresh random data suitable for new keys. We keep assigned keys fixed because their corresponding data node has already written its logical position in the KSA for retrieval when reading. The new version of the erase block is then written to an arbitrary empty erase block on the storage medium. After completion, all erase blocks containing old versions of the logical KSA erase block are erased, thus securely deleting the unused and discarded keys along with the data nodes they encrypt.

This implementation requires the ability to securely delete an entire erase block, i.e., perform an erase block erasure. Therefore, for flash memory, DNEFS must be implemented either into the logic of a file system that provides access to the raw flash memory (e.g., UBIFS) or into the logic of the flash controller (e.g., FTL controller). Note that while both the KSA and the main storage are colocated on the same physical storage medium, it is only the KSA that has any secure deletion requirements; the main storage is assumed to be persistent. The efficiency of DNEFS comes from the fact that only the small number of KSA erase blocks must be erased to securely delete all data nodes that are discarded since the previous clock. This comes at the cost of assuming a computationally-bounded adversary—an information-theoretic adversary could decrypt the encrypted file data.

6.3.5 Key State Map.

The KPs' states are managed in memory with a *key state map*. Figure 6.4 shows an example key state map and a corresponding KSA before and after a KSA update: **U** and **D** positions are replaced with new KVs; **A** positions retain their KVs.

When the file system is mounted, the key state map must be correctly constructed. The actual procedure to do this depends on the file system in which it is integrated, but it must account for the possibility of a previous unsafe unmounting. We define a *correct* key state map as one that has (with cryptographically-high probability) the following three properties:


key state map
KSA
erase block 1
0–4 k_0 k_1 k_2 k_3 k_4
5–9 k_5 k_6 k_7 k_8 k_9
erase block 2
10–14 k_{10} k_{11} k_{12} k_{13} k_{14}
15–19 k_{15} k_{16} k_{17} k_{18} k_{19}
next assigned key
main storage area
data nodes

pos	state
0	discarded
1	used
2	discarded
3	used
4	discarded
5	used
6	used
7	used
* 8	unused
9	unused
...	...

valid	seq #	file #	offset	keypos	data
no	1	1	0	0	[...]
yes	2	1	4096	1	[...]
no	3	1	0	2	[...]
yes	4	2	0	3	[...]
no	5	2	8192	4	[...]
yes	6	1	0	5	[...]
yes	7	2	8192	6	[...]

(a) state before purging keys

key state map
KSA
erase block 1
0–4 k'_0 k_1 k'_2 k_3 k'_4
5–9 k_5 k_6 k_7 k'_8 k'_9
erase block 2
10–14 k'_{10} k'_{11} k'_{12} k'_{13} k'_{14}
15–19 k'_{15} k'_{16} k'_{17} k'_{18} k'_{19}
next assigned key

pos	state
* 0	unused
1	used
2	unused
3	used
4	unused
5	used
6	used
7	used
8	unused
9	unused
...	...

(b) state after purging keys

Figure 6.4: Example of a key state map, key storage area, and main storage area during a KSA update. (a) shows the state before and (b) shows the state after updating. Some keys are replaced with new values, corresponding to data nodes that were unused or discarded. The table of data nodes illustrate a log-structured file system, where newer versions of data nodes for the same file/offset invalidate older versions.

- **C1** Every unused key must not decrypt any data node—either valid or invalid.
- **C2** Every assigned key must have exactly one data node it can decrypt and this data node must be referred to by the file system's index.
- **C3** Every discarded key must not decrypt any data node that is referred to by the file system's index.

Observe that an unused key that is marked as discarded still results in a correct key state map, as it affects neither the security of discarded data nor the availability of valid data.

We note that it is always possible to build a correct key state map. By design, file systems are capable of generating (some representation of) a file system index data structure that maps each valid data node to the location of its most-recently stored version. To build a correct key state map, we require that for each data node in the index, its corresponding KP's state is **A**. This approach, however, requires enumerating all data nodes. In the next chapter we show that our implementation of DNEFS for UBIFS leverages UBIFS's commit and replay mechanism to greatly improve the performance of rebuilding a correct key state map.

6.3.6 Summary

DNEFS provides guaranteed secure deletion against a computationally-bounded unpredictable multiple-access coercive attacker. When an encryption key is securely deleted, the data it encrypted is then inaccessible, even to the user. All discarded data nodes have their corresponding encryption keys securely deleted during the next KSA update. KSA updates occur as a periodic clock operation, so during operation the deletion latency *for all* data is bounded by the clock period. Neither the key nor the data node is available in any deletion epoch prior to the one in which it is written, so the existential latency *for all* data is also bounded by the clock period.

6.4 Extensions and Optimizations

In this section we present some extensions to DNEFS that may improve performance or security.

6.4.1 Granularity Tradeoff

DNEFS encrypts each data node with a separate key using a symmetric key block cipher in counter mode; this allows efficient secure deletion of data from long-lived files, e.g., databases. Other related work instead encrypts each file with a unique key, allowing secure deletion only at the granularity of an entire file [90]. This is well suited for media files, such as digital audio and photographs, which are usually created, read, and discarded in their entirety. Using a single key per file, however, means that modifications to files require re-encrypting its entire contents with a new key and securely deleting the old key. This cost grows with the size of the file, and then becomes more efficient to use naive compaction for files larger than an erase block.

We note that random read access can be made efficient by storing periodic IVs for long files.[3] Effectively, DNEFS does this by eschewing IVs altogether and using the storage to instead store encryption keys, which, unlike IVs, can also be used to secure deletion purposes as well as efficient random access.

Thus, a tradeoff exists between the storage costs of keys and the copying costs for modifications. At one extreme, DNEFS stores one key per data node and allows modifications with no additional cost. At the other extreme, one key per file (or storage medium) requires minimal storage but modifications are expensive. Between these extremes lies a range of possible encryption granularities, e.g., one key every eight data nodes.

Table 6.1 compares the encryption granularity trade off for a flash drive with 64 2-KiB pages per erase block. To compare DNEFS with schemes that encrypt each file separately, simply consider the data node size as equal to the IV granularity or the expected file size. The KSA size, measured in erase blocks per GiB of storage space, is the amount of storage required for IVs and keys, and is the worst case number of erase blocks that must be erased during each KSA update. The copy cost, also measured in erase blocks, is the amount of data that must be re-written to the flash storage medium due to a data node modification that affects only one page of flash memory. For example, with a data node size of 16 KiB and a page size of 2 KiB, the copy cost for a small change to the data node is 14 KiB. This is measured in erase blocks

[3]For completeness we mention (but do *not* advise) that a cipher in electronic codebook mode permits random access at the steep cost of the loss of semantic security.

Data node size (KiB)	Pages per data node	KSA size (EBs per GiB)	Copy cost (EBs)
2	1	64	0.0
4	2	32	0.016
8	4	16	0.047
16	8	8	0.11
32	16	4	0.23
64	32	2	0.48
128	64	1	0.98

Table 6.1: Data node granularity tradeoffs assuming 64 2-KiB pages per erase block.

because the additional writes, once filling an entire erase block, result in an additional erase block erasure that is otherwise unnecessary with a smaller data node size. Observe that in the final row, the data node size equals the erase block size and consequently any small change requires rewriting an erase block worth of data. In this case, however, the file system should instead store each data node on its own erase block and perform erase block erasure whenever a new version is written.

6.4.2 KSA Update Policies

While DNEFS uses batching to improve efficiency, there is no technical reason that prohibits immediate secure deletion. KSA update can be automatically triggered, for example, if data from a file marked with a *sensitive* attribute is discarded. KSA update is also triggered by an `ioctl`, which means that users or applications can force its operation, e.g., after clearing the web browsing cache. Note that batching of discards is required for DNEFS to provide any benefits over naive compaction of the erase block containing discarded data.

In addition to periodic updates, KSA updates can also be triggered once the number of discarded keys exceeds a threshold; this ensures that both the deletion latency and the amount of exposable data is limited. This effects a natural user interface, where discarding many files triggers secure deletion in the same way that a full garbage bin causes it to be emptied.

6.4.3 KSA Organization

The KSA can be divided into groups with different properties. This can be to provide extra feature or to improve efficiency. For example, KSA groups may vary in their clock frequency, so that sensitive data may be more quickly securely deleted. KSA groups may also vary in their encryption key sizes. We revisit this idea in Chapter 11.

Since the efficiency of DNEFS comes from batching, colocating the keys for data that is discarded simultaneously results in more efficient erase block erasures (i.e., more discarded keys per erased erase block). When the expiration time of data is not known in advance, a coarse division into short-term and long-term KSA groups can be approximated. When a data node is written to the file system it is encrypted with a short-term storage key. If the file system's free-space compaction results in that data node being moved, it can be re-encrypted with a new key from the long-term storage area. Thus, a form of generational garbage collection is used to as a heuristic to promote longer-lived data to the long-term group of the KSA [98].

6.4.4 Improving Reliability

As a technical note, flash erase blocks may become unreadable or unwritable. If a KSA erase block becomes unreadable, only a few KiB of keys are lost. Unfortunately, this corresponds to the loss of a much larger amount of data. Depending on the characteristics of the flash memory, it may be appropriate to replicate the KSA to prevent the loss of any data.

If a KSA erase block becomes a bad block while erasing it, it may be possible that its contents remain readable on the storage medium without the ability to remove them [92]. In this case, it is necessary to re-encrypt any data node whose encryption key remains available and to force the compaction of those erase blocks on which the data nodes reside. More generally, the implementation of the keystore as a set of KSA erase blocks does not guarantee robustness in data storage: one that always stores data correctly and always securely deletes data correctly. In Chapter 11 we show how to make a keystore that is robust against partial failures in confidentiality, integrity, and availability.

6.4.5 Encrypted File System

Our design can be trivially extended to offer a passphrase-protected encrypted file system: we simply encrypt the KSA whenever we write random data with a key derived from a password-based key derivation function, e.g., similar to LUKS [99].

Because each randomly-generated key in the KSA is unique (with high probability), we can encrypt the KSA using a block cipher in ECB mode to allow rapid decryption of randomly accessed offsets without storing additional initialization vectors [100]. Provided that the ciphertext block size is the same as the encryption key, no data is needlessly decrypted.

6.5 Summary

DNEFS is a generic file system extension designed for adding secure deletion to data and is particularly suited to flash memory. It provides secure deletion against a *computationally-bounded unpredictable multiple-access coercive adversary*, turning the storage medium into a SECDEL-CLOCK-EXIST implementation.

DNEFS works by encrypting each data node with a different key and storing the keys together on the flash storage medium. The erase blocks containing the keys are periodically updated to remove old keys, replacing them with fresh random data that can be used as keys for new data. DNEFS provides *fine-grained* deletion in that parts of files that are overwritten are also securely deleted.

In the next chapter, we describe UBIFSec, which is an implementation of DNEFS for the flash file system UBIFS. We further deploy UBIFSec on an Android mobile phone and test its performance in practice to verify that it is efficient.

Chapter 7

UBIFSec: Adding DNEFS to UBIFS

7.1 Introduction

The previous chapter presents DNEFS, a generic file system extension that provides efficient secure deletion. This chapter validates DNEFS by building and testing UBIFSec: the implementation of DNEFS for the flash file system UBIFS. We measure the increased flash memory wear caused by DNEFS as well as the battery consumption and conclude that UBIFSec has excellent performance and efficiently solves the problem of secure deletion for flash memory.

DNEFS is easily integrated into UBIFS with changes to about 100 lines of existing UBIFS source code and the inclusion of a new component, the KSA. We deploy UBIFSec on a Google Nexus One smartphone [78] running an Android OS. The system and applications (including video and audio playback) run normally on top of UBIFSec.

7.2 System and Adversarial Model

This chapter focuses on a concrete instantiation of the general solution DNEFS, whose design is described in Chapter 6. UBIFSec uses the same system and adversarial model as the one described in Section 9.2.

7.3 Background

Before detailing the integration of DNEFS with UBIFS, we first provide the necessary background information. We briefly recall the MTD layer, describe a logical interface for it called UBI, and then introduce the UBI-based flash file system UBIFS. Recall that Figure 2.1 shows the layers and interfaces involved in accessing flash memory.

7.3.1 MTD and UBI Layers

On Linux, flash memory is accessed through the Memory Technology Device (MTD) layer [80]. MTD has the following interface: read a page, write a page, erase an erase block, check if an erase block is bad, and mark an erase block as bad. Erase blocks are referenced sequentially, and pages are referenced by the erase block number and offset.

Unsorted Block Images (UBI) is an abstraction of MTD, where logical erase blocks are transparently mapped to physical erase blocks [42]. UBI's logical mapping implements wear-levelling and bad block detection, allowing UBI file systems to ignore these details. UBI also permits the atomic updating of a logical erase block—the new data is either entirely available or the old data remains.

UBI exposes the following interface: read and write to a logical erase block (LEB), erase an LEB, and atomically update the contents of an LEB. UBI LEBs neither become bad due to wear, nor should their erasure counts be levelled. Each UBI LEB has a unique number that orders the LEBs.

Underlying this interface is an injective partial mapping from LEBs to physical erase blocks (PEBs), where PEBs correspond to erase blocks at the MTD layer. The lower half of Figure 7.1 illustrates this relationship. Wear monitoring is handled by tracking the erasures at the PEB level, and a transparent remapping of LEBs occurs when necessary. Remapping also occurs when bad blocks are detected. Despite remapping, an LEB's number remains constant, regardless of its corresponding PEB.

Atomic updates of LEBs occur by invoking UBI's update function, passing as parameters the LEB number to update along with a buffer containing the desired contents. An unused and empty PEB is selected and the page-aligned data is then written to it. UBI then updates the LEB's mapping to the new PEB, and the previous PEB is queued for

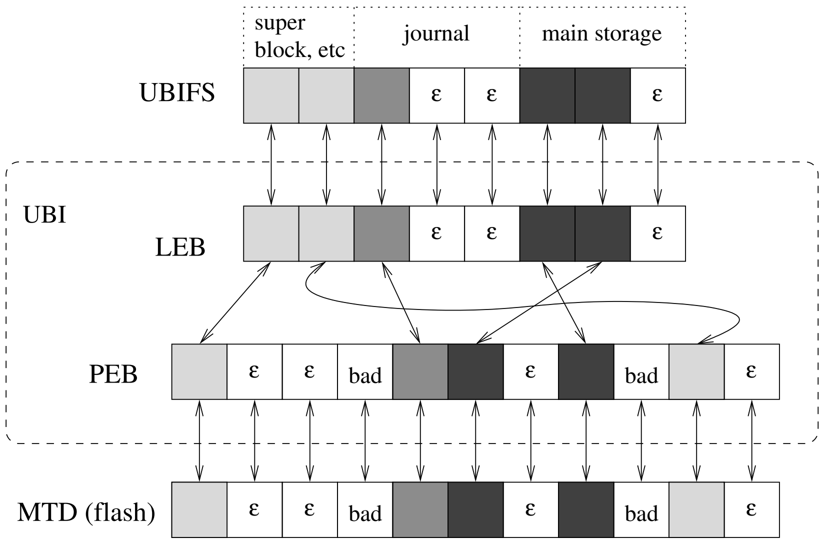


Figure 7.1: Erase block relationships among MTD, UBI, and UBIFS. Different shades label different areas of the file system: the super block, journal, main storage, etc. Empty LEBs are labelled by ε and are not mapped to a corresponding PEB by UBI. Similarly, bad PEBs are labelled and not mapped onto by UBI.

erasure. This erasure can be done either automatically in the background or immediately with a blocking system call. If the atomic update fails at any time—e.g., because of a power loss—then the mapping is unchanged and the old PEB is not erased.

7.3.2 UBIFS

The UBI file system, UBIFS [88], is designed specifically for UBI, and Figure 7.1 illustrates UBIFS's relationship to UBI and MTD. UBIFS divides file data into fixed-sized data nodes. Each data node has a header that stores the data's inode number and its file offset. This inverse index is used by UBIFS's compactor (called the garbage collector) to determine if the nodes on an erase block are valid or can be discarded.

UBIFS writes all new data in a journal similar to a log-structured file system; the journal consists of a set of LEBs. When the UBIFS journal is full, it is committed to the main storage area and emptied by logically moving the journal to an empty set of LEBs and growing the main storage area to encompass the old journal. An index is used to locate data nodes, and this index is also written to the storage medium. At its core, UBIFS is a log-structured file system; in-place updates are not performed. As such, UBIFS does not provide guaranteed secure data deletion.

UBIFS uses an index to determine which version of data is the most recent. This index is called the Tree Node Cache (TNC), and it is stored both in volatile memory and on the storage medium. The TNC is a B+ search tree [101] that has a small entry for every data node in the file system. When data is appended to the journal, UBIFS updates the TNC to reference its location. UBIFS implements truncations and deletions by appending special non-data nodes to the journal. When the TNC processes these nodes, it finds the range of TNC entries that correspond to the truncated or deleted data nodes and removes them from the tree.

UBIFS uses a commit-and-replay mechanism to ensure that the file system can be mounted after an unsafe unmounting without scanning the entire device. Commit periodically writes the current TNC to the storage medium, and starts a new empty journal. Replay loads the most recently-stored TNC into memory and chronologically processes the journal entries to update the stale TNC, thus returning the TNC to the state immediately before the previous unsafe unmounting.

UBIFS accesses flash memory through UBI's logical interface, which provides two features useful for our purposes. First, UBI allows updates to KSA erase blocks (called KSA LEBs in the context of UBIFSec) using its atomic update feature. After updating, all assigned KVs remain in the same *logical* position, so references to KSA positions remain valid after updating. Second, UBI handles wear-levelling for all the PEBs, including the KSA. This is useful because erase blocks assigned to the KSA see more frequent erasures; fixed physical assignment would therefore present wear-levelling concerns.

7.4 UBIFSec Design

UBIFSec is a version of UBIFS that is extended to use DNEFS to provide secure data deletion. UBIFS's data nodes have a size of 4096 bytes, and UBIFSec assigns each of them a distinct 128-bit KV used as an AES encryption key. AES keys are used in counter mode, which turns AES into a semantically-secure stream cipher [100]. Since each AES key is only ever used to encrypt a single block of data, we can safely omit the generation and storage of initialization vectors (IVs) and simply start the counter for each AES key at a static value. Our solution requires about 0.4% of the storage medium's capacity for the KSA, although there exists a tradeoff between the KSA's size and the data node's size (see Section 6.4.1).

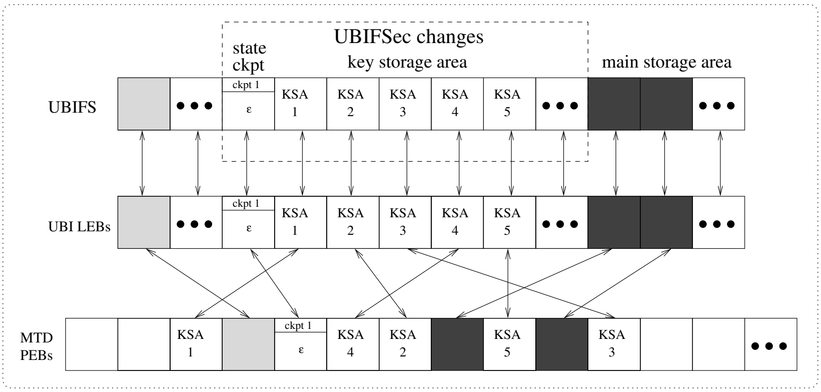
7.4.1 Key Storage Area

The KSA is composed of a set of LEBs that store random data used as encryption keys. When the file system is created, cryptographically-suitable random data is written from a hardware source of randomness to each of the KSA's LEBs and all the KVs are marked as unused. The KSA update writes new versions of the KSA LEBs using UBI's atomic update feature; immediately afterwards, `ubi_flush` is called to ensure that all PEBs containing old versions of the LEB are synchronously erased. All KVs they contain are therefore securely deleted. This flush feature ensures that all copies of LEBs made through internal wear-levelling are also securely deleted. Figure 7.2 shows the LEBs and PEBs during a KSA update.

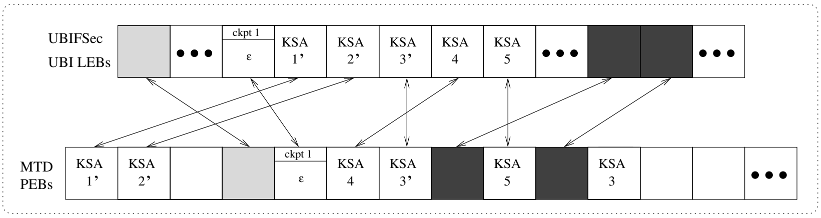
Only KSA erase blocks with discarded data are updated, though erase blocks that are not updated are not used to assign new KVs. To further reduce the number of KSA erase blocks that must be updated, we use KSA groups to concentrate KVs for long-term data. Our implementation uses two KSA groups: a short-term group and a long-term group. New data nodes initially get a short-term KV. If a data node is ever compacted by UBIFS, it is re-encrypted with a KV assigned from the long-term group and we say that the data node is *promoted* to the long-term group. The short-term KV is then discarded.

7.4.2 Key State Map

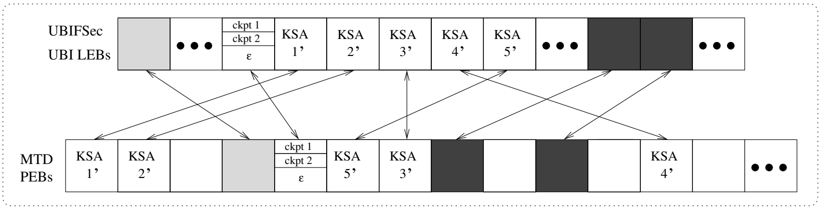
The key state map stores the key positions' states. The correctness of the key state map is critical in ensuring the soundness of secure deletion and data integrity. We now describe how the key state map is created and stored in UBIFSec. As an invariant, we require that UBIFSec's key state map is always correct (properties **C1**–**3** from Chapter 6) before and after executing a KSA update. This restriction—instead of



(a) before update



(b) during update



(c) after update

Figure 7.2: Erase block relationships among MTD, UBI, and UBIFSec, showing the new regions added by UBIFSec (cf. Figure 7.1). This figure has three components that illustrate the state (a) before, (b) during, and (c) after a KSA update. Observe in (b) that new versions of KSA blocks 1, 2, and 3 are written to new locations; the old version of block 3 remains. Observe in (c) that no old KSA erase block remains and a new key state checkpoint is written.

requiring correctness at all times after mounting—is to allow writing new data during KSA updates, and to account for the time between marking a key as assigned and writing the data it encrypts onto the storage medium.

The key state map is stored, used, and updated in volatile memory. Initially, the key state map of a freshly-formatted UBIFSec file system is correct as it consists of no data nodes, and every KV is fresh, random data that is marked as unused. While mounted, UBIFSec performs appropriate key management to ensure that the key state map is always correct when new data is written and discarded. We now show that we can always create a correct key state map when mounting an arbitrary UBIFSec file system.

The key state map is built from a periodic checkpoint combined with a logical replay of the file system's changes since the most recent checkpoint. We checkpoint the current key state map to the storage medium immediately after each KSA update. (This is even before logically replaying cached changes that occurred while updating.) After the KSA update, every key is either unused or assigned, and so a checkpoint of this map can be stored using one bit per key—less than 1% of the KSA's size—which is then compressed. A special LEB is used to store checkpoints, where each new checkpoint is appended; when the erase block is full then the next checkpoint is written at the beginning using an atomic update.

The checkpoint is correct when it is written to the storage medium, and therefore it is correct when it is loaded during mounting if no other changes occurred in the file system. If the file system changed after committing and before unmounting, then UBIFS's replay mechanism is used to generate the correct key state map: first, the checkpoint is loaded, then the replay entries are simulated. To simplify the logic for our integration, we perform KSA updates during regular UBIFS commits; the nodes that are then replayed for UBIFS are exactly the ones that must be replayed for DNEFS. If the stored checkpoint gets corrupted, then a full scan of the valid data nodes rebuilds the correct key state map. A consistency check for the file system also confirms the correctness of the key state map with a full scan.

As it is possible for the storage medium to fail during the commit operation (e.g., due to a loss of power), we now show that our invariant holds regardless of the condition of unmounting. Figure 7.3 shows a flow chart of the UBIFSec commit operation, annotated with the locations where it may fail. Each action in a rectangle is atomic: it either

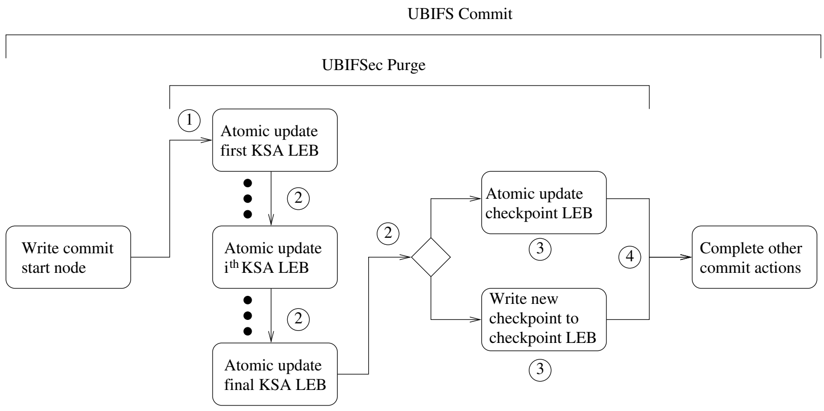

UBIFS Commit
UBIFSec Purge
1
Atomic update first KSA LEB
2
Write commit start node
Atomic update i^th KSA LEB
2
Atomic update final KSA LEB
2
Atomic update checkpoint LEB
3
Write new checkpoint to checkpoint LEB
3
4
Complete other commit actions

Figure 7.3: Flow chart of UBIFSec commit process labelled with four distinct potential failure locations. Each rounded rectangle contains an action in the process that either succeeds or fails atomically. Numbers indicate the unique failure points in our analysis.

succeeds or fails entirely. KSA update consists of atomically updating each LEB containing discarded KVs and afterwards writing a new checkpoint. UBI's atomic update feature ensures that any failure before completing the update is equivalent to failing immediately before beginning. Therefore, with reference to the numerical labels in Figure 7.3, the following is the complete list of distinct failure points: (1) before the first LEB update, (2) between some LEB updates, (3) after all the LEB updates but before or during the key state map checkpoint, (4) after the checkpoint but before finishing other UBIFS commit actions. We now discuss each of these failure points in detail.

First, failure can occur before updating the first LEB, which means the KSA is unchanged. When remounting the device, the loaded checkpoint is updated with the replay data, thereby constructing the exact key state map before updating—correct by assumption.

Second, failure can occur after updating one, several, or all of the KSA's LEBs. When remounting the device, the loaded checkpoint merged with the replay data reflects the state before the first update, so some updated LEBs contain new unused data while the key state map claims it is a deleted key. As these are cryptographically-suitable random values, with high probability they cannot successfully decrypt any existing valid data node.

prev stored ckpt	possible journal	current stored ckpt	state after double replay	correct key state map
U	∅	**U**	**U**	yes
U	**U→A**	**A**	**A**	yes
U	**U→A→D**	**U**	**D**	yes
A	∅	**A**	**A**	yes
A	**A→D**	**U**	**D**	yes

Table 7.1: Consequences of replaying false information during committing.

Third, failure can occur while writing to the checkpoint LEB. When the checkpoint is written using atomic updates, then failing during the operation is equivalent to failing before it begins. Incomplete checkpoints are detected and so the previous valid checkpoint is loaded instead. After replaying all the nodes, the key state map is equal to its state immediately before updating the KSA. This means that all discarded entries are actually unused entries, so the key state map invariants hold.

Fourth and finally, failure can occur after successfully updating the KSA and checkpointing the key state map, but before completing the regular UBIFS commit. In this case, the checkpointed key state map correctly reflects the contents of the KSA. When mounting, the replay mechanism incorrectly updates the key state map with the journal entries of the previous iteration. In other words, the journal's contents are doubly applied to the key state map. Table 7.1 shows the space of possibilities when replaying old changes on the post-updated checkpoint; it omits impossible checkpoint-journal combinations. For all possible double-replay scenarios, the generated key state map is always correct.

In summary, the correctness of the key state map before and after KSA updates is invariant, regardless of when or how the file system was unmounted. This ensures secure deletion's soundness as well as valid data's integrity on the storage medium.

7.4.3 Summary

UBIFSec instantiates DNEFS for UBIFS, and so it provides efficient fine-grained guaranteed secure deletion. UBIFSec is efficient in storage space: the overhead for keys is fixed and it needs less than one percent of the total storage medium's capacity. The periodic checkpointing of UBIFSec's key state map ensures that UBIFS's mounting time is not significantly affected by our approach.

Our implementation of UBIFSec is available as a Linux kernel patch for version 3.2.1 [102]. Table 7.2 lists the small amount of changes to the original UBIFS source code required to integrate our solution. The keystore's implementation comprises most of the implementation effort.

7.5 Experimental Validation

We have patched an Android Nexus One smart phone's Linux kernel to include UBIFSec and modified the phone to use it as the primary data partition. In this section, we describe experiments with our implementation on both the Android mobile phone and on a simulator.

Our experiments measure our solution's cost: additional battery consumption, wear on the flash memory, and time required to perform file operations. The increase in flash memory wear is measured using a simulator, and the increase in time is measured on a Google Nexus One smartphone by instrumenting the source code of UBIFS and UBIFSec to measure the time it takes to perform basic file system operations. We further collected timing measurements from the same smartphone running YAFFS: the flash file system used on Android phones at the time that this research was undertook.

7.5.1 Android Implementation

To test the feasibility of our solution on mobile devices, we port UBIFSec to the Android OS. The Android OS is based on the Linux kernel and it is straightforward to add support for UBIFS. The UBIFS source code is already available so we apply our patch (backporting it for Linux kernel version 2.6.35.7) and configure the kernel compiler to include the UBI device and the UBIFS file system in compilation. We modify the Android boot image to create UBI devices from Android's data partition's MTD device and mount the data partition as file system type UBIFS. Because the default file system for this Android version is YAFFS, some of our experiments compare UBIFS not only to UBIFSec but also to YAFFS.

Mounting (25 lines of code)
mount the file system
- allocate and initialize the keystore
- deallocate keystore if an error occurs
- read the size of the KSA from the master node
unmount the file system
- deallocate the keystore
create default file system
- use storage medium's geometry to compute the required KSA size
- store this information in the master node
- call keystore's initialize KSA routine

Commit (3 lines of code)
commit the journal
- call the keystore's update operation

Input/Output (21 lines of code)
write data
- obtain an unused key position from the keystore
- store the key's position in the data node's header
- use the keystore and key position to look up the key
- provide the key to the compress function
recompute data after truncation
- obtain the original key, decrypt the data
- obtain a new key, encrypt the data with it after truncating
read data
- use the keystore and data node's key position to look up the key
- provide the key to the decompress function

Tree Node Cache (42 lines of code)
add/update the TNC
- provide a key position when adding data nodes
- store the key position inside TNC entries
- assign key position
- if updating, discard old key position as discarded
delete/truncate the TNC
- when removing a data node from the TNC, discard key position
commit the TNC
- read and write key position to stored tree nodes

Garbage Collection (13 lines of code)
promote key
- decide whether to promote data node
- re-encrypt promoted data node
- discard old key, assign new key

Table 7.2: Changes to UBIFS source code required to integrate UBIFSec.

Update Period	Erased PEBs per hour	Erasures per KSA update	KSA LEBs updated per hour	Discarded KVs per up'ed LEB	Wear ineq (%)	Life years
Std. UBIFS	21.3 ± 3.0	-	-	-	16.6 ± 0.5	841
60 minutes	26.4 ± 1.5	6.8 ± 0.5	6.8 ± 0.5	64.2 ± 9.6	17.9 ± 0.2	679
30 minutes	34.9 ± 3.8	5.1 ± 0.6	9.7 ± 2.0	50.3 ± 9.5	17.8 ± 0.3	512
15 minutes	40.1 ± 3.6	3.7 ± 0.4	14.9 ± 1.6	36.3 ± 8.2	19.0 ± 0.3	447
5 minutes	68.5 ± 4.4	2.6 ± 0.1	30.8 ± 0.7	22.1 ± 4.3	19.2 ± 0.5	262
1 minute	158.6 ± 11.5	1.0 ± 0.1	61.4 ± 4.6	14.1 ± 4.4	20.0 ± 0.2	113

Table 7.3: Wear analysis for our modified UBIFS file system. The expected lifetime is based on the Google Nexus One's flash specifications, which have 1571 erase blocks with a (conservative) lifetime estimate of 10^4 erasures.

7.5.2 Wear Analysis

We measure UBIFSec's wear on the flash memory in two ways: the number of erase cycles that occurs on the storage medium, and the distribution of erasures over the erase blocks. To reduce the wear, it is desirable to minimize the number of erasures that are performed, and to evenly spread the erasures over the storage medium's erase blocks.

We instrument both UBIFS and UBIFSec to measure PEB erasure frequency during use. We vary UBIFSec's KSA update period and compute the resulting erase block allocation rate. We do this with a low-level control (`ioctl`) that forces UBIFS to perform a commit. We also measure the expected number of deleted keys and updated KSA LEBs during KSA updates.

Using `nandsim` we simulate in memory a UBI storage medium using the geometry of Nexus One's flash memory [78]. We vary the period between UBIFSec's updates, i.e., the duration of a deletion epoch: one of 1, 5, 15, 30, and 60 minutes. We use the discrete event simulator based on the observed writing behaviour from the data earlier collected (see Section 5.4.1). Writing is performed until the file system begins compaction; henceforth we take measurements for a week of simulated time. We average the results from four attempts and computed 95% confidence intervals.

To determine whether our solution negatively impacts UBI's wear levelling, we perform the following experiment. Each time UBI unmaps an LEB from a PEB (thus resulting in an erasure) or atomically updates an LEB (also resulting in an erasure), we log the erased PEB's number. From this data, we compute the PEBs' erasure distribution.

To quantify wear-levelling, we use the Hoover economic wealth inequality indicator [103]—a metric that is independent of the storage medium size and erasure frequency. This metric comes from economics, where it quantifies the unfairness of wealth distributions. It is equal to the normalized sum of the difference of each measurement to the mean. For our purposes, it is the fraction of erasures that must be reassigned to other erase blocks to obtain completely even wear. Let the observations be $c_1, \ldots, c_n$, and $C = \sum_{i=1}^{n} c_i$, then the inequality measure is $\frac{1}{2} \sum_{i=1}^{n} |\frac{c_i}{C} - \frac{1}{n}|$.

Table 7.3 presents the results of our experiment. We see that the rate of block allocations increases as the KSA update period decreases, with 15 minutes providing a palatable tradeoff between the additional wear and timeliness of deletion. The KSA's update rate is computed as the product of the KSA update frequency and the average number of KSA LEBs that are updated each time. As such, it does not include the additional costs of executing UBIFS commits, which are captured by the disparity in the block allocations per hour. We see that when committing each minute, the additional overhead of committing compared to the updates of KSA blocks becomes significant.

As a remedy, we argue that while we integrated KSA update with UBIFS's commit to simplify the recovery logic, it is possible to separate these operations. Indeed, UBIFSec can add `KSA-update start` and `KSA-update finish` nodes as regular non-data journal entries. The replay mechanism is then extended to correctly update the key state map while processing these update nodes.

The expected number of KVs deleted per updated KSA LEB decreases sublinearly with the update period and linearly with the number of updated LEBs. This is because a smaller interval results in fewer expected deletions per interval and fewer deleted keys.

Finally, UBIFSec affects wear-levelling slightly. The unfairness increases with the update frequency, likely because the set of unallocated PEBs is smaller than the set of allocated PEBs; frequent updates cause unallocated PEBs to suffer more erasures. However, the effect is slight. It is certainly the case that the additional block erasures are, for the most part, evenly spread over the device.

7.5.3 Power Consumption

To measure battery consumption over time, we disable the operating system's suspension ability, thus allowing computations to occur continuously and indefinitely. This has the unfortunate consequence of maintaining power to the screen of the mobile phone. We first determine the power consumption of the device while remaining idle over the course of two hours starting with an 80% charged battery with a total capacity of 1366 mAh. The result was nearly constant at 121 mA. We subtract this value from all other power consumption measures.

To measure read throughput and battery use, we repeatedly read a large (85 MiB) file; we mount the drive as read-only and remount it after each read to ensure that all read caches are cleared. We read the file using `dd`, directing the output to `/dev/null` and record the observed throughput. We begin each experiment with an 80% charged battery and ran it for 10 minutes observing constant behaviour. We choose 80% charge to simulate steady state conditions—avoiding extremal charge states.

Table 7.4 presents the results for this experiment. For all file systems, the additional battery consumption was constant: 39 mA, about one-third of the idle cost. Depending on the file system, however, that amount of power achieved a varying throughput. We therefore include in our results a computation of the amount of data that can be read using 13.7 mAh—1% of the Nexus One's battery. The write throughput and battery consumption was measured by using `dd` to copy data from `/dev/zero` to a file on the flash file system. Compression is disabled for UBIFS for comparison with YAFFS. When the device is full, the throughput is recorded. We immediately start `dd` to write to the same file, which begins by overwriting it and thus measuring the battery consumption and reduction in throughput imposed by erase block erasures concomitant with writes.

7.5.4 Throughput Analysis

Table 7.4 shows read and write throughput achieved for different file systems. We observe that the use of UBIFSec reduces the throughput for both read and write operations when compared to UBIFS. Some decrease is expected, as the encryption keys must be read from flash while reading and writing. To determine whether there is any added latency due to the cryptographic operations, we performed these experiments with a modified UBIFSec that immediately returned zeroed memory when asked to read a key, but otherwise performed all cryptographic operations correctly. The resulting throughput for read and write was identical to UBIFS, suggesting that (for multiple reads) cryptographic

	YAFFS	UBIFS	UBIFSec
Read rate (MiB/s)	4.4	3.9	3.0
Power usage (mA)	39	39	39
GiB read per 1% battery	5.4	4.8	3.7
Write rate (MiB/s)	2.4	2.1	1.7
Power usage (mA)	30	46	41
GiB written per 1% battery	3.8	2.2	2.0

Table 7.4: I/O throughput and battery consumption for YAFFS, UBIFS, and UBIFSec.

operations are easily pipelined into the relatively slower flash memory read/write operations.

Some key caching optimizations can be added to UBIFSec to improve the throughput. Whenever a page of flash memory is read, the entire page can be cached at no additional read cost, allowing efficient sequential access to keys, e.g., for a large file. Long-term use of the file system may reduce its efficiency as fragmented gaps between unused and assigned keys result in sequential blocks of data not being assigned sequential keys in the KSA, causing frequent cache misses for sequential reads. Improved KSA organization can help retain this efficiency.

Write throughput, alternatively, is easily improved with caching. The sequence of keys for data written in the next deletion epoch is known at update time when all these keys are randomly generated and written to the KSA. By using a heuristic on the expected number of keys assigned during a deletion epoch, the keys for new data can be kept in memory as well as written to the KSA. Whenever a key is needed, it is taken and removed from this cache while there are still keys available.

Caching keys in memory exposes UBIFSec to attacks. We ensure that all memory buffers containing keys are overwritten when the key is no longer needed during normal cryptographic operations. Caches contain keys for a longer time but are cleared during KSA update to ensure deleted keys never outlive their deletion epoch. Sensitive data stored in volatile memory by applications may remain after the data's deletion; secure memory deallocation should be provided by the operating system to ensure its irrecoverability [104].

7.5.5 Timing Analysis.

We time the following file system functions: mounting/unmounting the file system and writing/reading a page. Additionally, we time the following functions specific to UBIFSec: allocation of the cryptographic context, reading the encryption key, performing an encryption/decryption, and updating a KSA LEB. We collect dozens of measurements for updating, mounting and unmounting, and hundreds of measurements for the other operations (i.e., reading and writing). We control for the delay caused by our instrumentation by repeating the experiments instead of executing nested measurements, i.e., we timed encryption and writing to a block in separate experiments.

We mounted a partition of the Android's flash memory first as a standard UBIFS file system and then as UBIFSec file system. We execute a sequence of file I/O operations on the file system. We collect the resulting times and present the 80th percentile measurements in Table 7.5. Because of UBIFS's implementation details, the timing results for reading data nodes contain also the time required to read relevant TNC pages (if they are not currently cached) from the storage medium, which is reflected in the increased delay. Because the data node size for YAFFS is half that of UBIFS, we also doubled the read/write measurements for YAFFS for a fair comparison. Finally, the mounting time for YAFFS is for mounting after a safe unmount—for an unsafe unmount (e.g., a crash), YAFFS requires a full device scan, which takes several orders of magnitude longer. This difference is because YAFFS checkpoints the file system's data structures when safely unmounting and simply reads them when mounting, continuing from whence it was.

The results show an increase in the time required for each of the operations. Mounting and unmounting the storage medium continue to take a fraction of a second. Reading and writing to a data node increases by a little more than a millisecond, an expected result that reflects the time it takes to read the encryption key from the storage medium and encrypt the data. We also test for noticeable delay by watching a movie in real time from a UBIFSec-formatted Android phone running the Android OS: the video was 512x288 Windows Media Video 9 DMO; the audio was 96.0 kilobit DivX audio v2. Both the video and audio play as expected on the phone; no observable latency, jitter, or stutter is observed during playback while background processes ran normally.

File system	80th percentile execution time (ms)		
operation	YAFFS	UBIFS	UBIFSec
mount	43	179	236
unmount	44	0.55	0.67
read data node	0.92	2.8	4.0
write data node	1.1	1.3	2.5
prepare cipher	-	-	0.05
read key	-	-	0.38
encrypt	-	-	0.91
decrypt	-	-	0.94
update one LEB	-	-	21.2

Table 7.5: Timing results for various file system functions on an android mobile phone.

Each atomic update of an erase block takes about 22 milliseconds. This means that if every KSA LEB is updated, the entire data partition of the Nexus One phone can be securely deleted in 200 milliseconds. The cost to securely delete data grows with its storage medium's size. The erasure cost for KSA updates can be reduced in a variety of ways: increasing the data node size to use fewer keys, increasing the KSA's update period, or improving the KSA's organization and key assignment strategy to minimize the number of KSA LEBs that contain deleted keys. The last technique can work alongside lazy updates of KSA LEBs that contain no deleted keys (i.e., only unused and assigned keys) so that they are only updated before being used to assign new keys to ensure freshness.

7.6 Conclusions

UBIFSec implements DNEFS for UBIFS and shows that DNEFS is a feasible solution for efficient secure deletion for flash memory operating within flash memory's specifications. It provides guaranteed periodic per-data-block secure deletion against a computationally-bounded unpredictable multiple-access coercive adversary. It turns the storage medium into a SECDEL-CLOCK-EXIST implementation where the clock frequency is a tradeoff between the deletion/existential latencies and the device wear.

Our experiments validate that UBIFSec has little cost. It requires a small evenly-levelled increase in flash memory wear and modest computational overhead. UBIFSec is seamlessly added to UBIFS. Cryptographic operations are inserted into UBIFS's existing read/write data path, and the key states are managed by UBIFS's existing index of data nodes.

Part III

Secure Deletion for Remote Storage

Chapter 8

Cloud Storage: Background and Related Work

8.1 Introduction

DNEFS uses encryption not as a secrecy technique but instead as a compression technique. This was not, however, the first time encryption was used to facilitate secure deletion. While DNEFS only treated the main storage as persistent for efficiency reasons, a variety of storage media are actually effectively indelible. Examples of such *persistent* media are write-once media, off-line tape archives, media that leave ample analog remnants [45], and storage media under adversarial control. We argue that media under adversarial control effectively models many scenarios involving remote storage systems such as cloud storage.

In the following chapters, we present a detailed examination of secure deletion solutions for persistent storage. The remainder of this chapter presents different persistent storage media and the object store abstraction, which is a common means by which they are accessed. We then present related work for secure deletion on persistent storage.

Chapters 9, 10, and 11 provide our contributions. Chapter 9 presents our results on generalizing the related work, formalizing the problem of key disclosure using graph theory, and then using the formalism to prove the security of a wide class of potential solutions. Chapter 10 describes a particular solution our of own design, which we then implement and analyse. Chapter 11 considers the problem of an unreliable securely-deleting storage media. We design a system robust against failures to delete data, failures to correctly store data, failures maintain the confidentiality of data, and failures to be available.

8.2 Persistent Storage

As described in Section 3.3, a persistent storage medium implementation is one that is indelible. All data stored on the medium remains stored; any previously stored data is given to the adversary at the time of compromise. Persistent storage models a variety of distinct situations, such as read-only memory, paranoia over analog remnants, and forward secrecy for network traffic (i.e., the secure deletion of previous communication sessions).

In some cases, like read-only memory, physical destruction is an option but does not provide effective secure deletion against our adversary who compromises at an unanticipated time.[1] In other cases, like remote storage and network traffic, physical destruction is not an option because the adversary may obtain the data immediately. As a result, we assume that there is no physical destruction option available for the persistent storage, and that all data written to a persistent storage medium is *immediately given to the adversary.* We do not consider this a compromise, that is, we still want to securely delete this data.

8.2.1 Securely-Deleting and Persistent Combination

While data cannot be deleted from a persistent medium, there exist a variety of mixed-media solutions where it is assumed that the user stores data using both a *persistent* storage medium and a *securely-deleting* storage medium [1, 2, 4, 9, 11, 105, 106]. In these situations, adversarial compromise only concerns the securely-deleting storage medium, not the persistent one.

[1]Recall from Section 4.2.4 that if many units of read-only memory are available and can be independently destroyed, then they can form the constituents of an archive whose erasure granularity is larger than the read and write granularity.

setting: tape vault
reason to use: cheap massive backups
persistent storage: magnetic tapes
securely-deleting storage: guarded machine at tape drive site
adversary: insider at vault or in transit

setting: remote storage
reason to use: convenience of ubiquitous access
persistent storage: networked file systems
securely-deleting storage: smart card, laptop, mobile phone
adversary: operator of remote storage server

setting: forward secrecy
reason to use: shared access to data, communication
persistent storage: network communication
securely-deleting storage: session keys, long-term signing key
adversary: network eavesdropper, key compromise

setting: analog remnants
reason to use: limited memory
persistent storage: digital storage
securely-deleting storage: human memory
adversary: one with unimaginable forensic capabilities

Table 8.1: Situations modelled by persistent storage media.

A non-trivializing assumption is that the user is, for some reason, compelled to use the persistent storage medium for storing data; perhaps because the securely-deleting medium is small in capacity, slow in performance, inconvenient to use, not able to share data, etc. The securely-deleting medium may be, for example, a trusted platform module or a portable smartcard that allows users to access remotely-stored data anywhere; in both cases we can expect it to only store a limited amount of data. Table 8.1 presents relevant secure deletion scenarios represented by this model and characterizes example types of media and the reasons why the persistent storage is used.

8.2.2 Cloud Storage

Our model assumes that the cloud storage is untrusted and thus is itself adversarial. While this characterization is not necessarily always true, there is a compelling reason to model it as such: once the data has left the users' control, they can no longer themselves ensure access control and secure deletion and must instead trust that it is performed correctly.

When data is stored remotely, it may be replicated many times, with backups and snapshots being stored in offline tape vaults. Resources may be shared or security vulnerabilities may enable unauthorized access. Centralized servers become a more valuable target for attacks. The data may be housed in data centres whose legal jurisdictions differ from those of the user. Legislative requirements may complicate remote storage of some kinds of data, such as the geolocation of banking information. Government adversaries may obtain user data through legal means, for instance, by obtaining surreptitious access to the storage medium through a legal subpoena to the storage provider.

The trustworthiness of an organization can also change over time: bankruptcy may legally require the liquidation of assets to satisfy creditors, or the organization may be purchased by a larger, less trustworthy one. Even private cloud infrastructures are vulnerable to insider attacks, poor configurations, mismanagement, human error, etc.

8.3 Related Work

In this section, we describe related work on secure deletion for persistent storage media that are augmented with a securely-deleting storage media. We note that some of these works did not explicitly consider this system model; in these cases, it is our opinion that their model fits into this framework.

Some related work uses hierarchical key wrapping structures to achieve secure deletion. For a visual reference, Figure 8.1 illustrates an example of the general shape of the key structures for relevant related work using directed graphs. In each subfigure, each node (black circle) corresponds to a key; a directed edge means that the source key is used to wrap the destination key. Encircled nodes have no incoming edges and correspond to master keys stored on the securely deleting

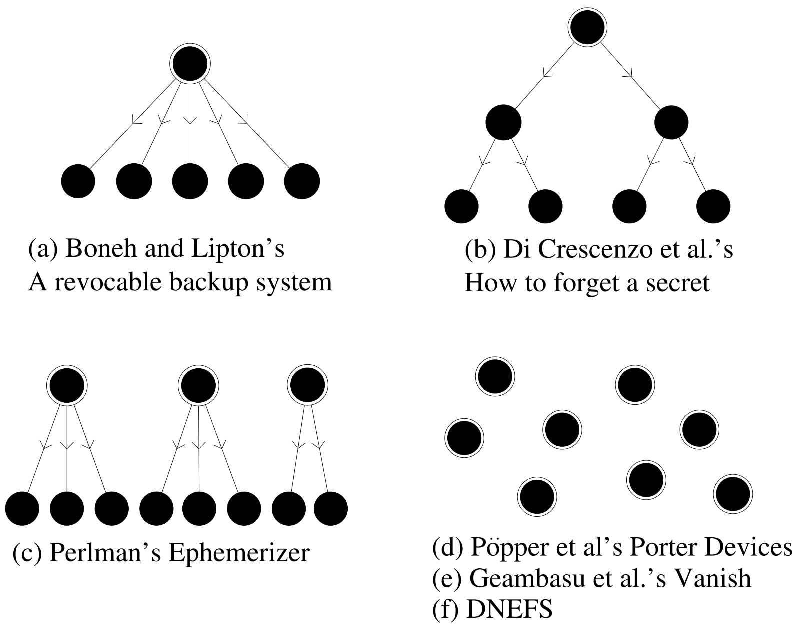

(a) Boneh and Lipton's
A revocable backup system
(b) Di Crescenzo et al.'s
How to forget a secret
(c) Perlman's Ephemerizer
(d) Pöpper et al's Porter Devices
(e) Geambasu et al.'s Vanish
(f) DNEFS

Figure 8.1: Key wrapping structures for secure deletion solutions for persistent storage. Each node is a key; directed edges mean that the source node wraps the destination; circled nodes are keys stored on the securely-deleting storage medium and leaf nodes are used to encrypt data items.

storage medium; nodes with no outgoing edges are data item keys used to encrypt individual data items.

Revocable Backup System. Boneh and Lipton (Figure 8.1 (a)) propose the first scheme that uses secure deletion of cryptographic keys to securely delete encrypted data under computational assumptions [4]. They created a revocable backup system for off-line (i.e., tape) archives consisting of three user-level applications. Backup files are made revocable *before* writing them to tape. Backups are revoked and then securely deleted without needing physical access to the tapes on which they are stored. Each backup is encrypted with a unique key; each key is then encrypted with a temporary master key. Their solution is clocked, so time is discretized into intervals and each interval is assigned a new master key that encrypts all the backup keys.

Backups are deleted from the archive simply by *not re-encrypting* the corresponding backup key with the new master key at the next clock edge. They extend their user interface to include master-key management with a *secure deletion* feature; in their work, they propose to write the new key on paper or on a floppy diskette and then physically destroy the previous one. The paper or floppy diskette therefore constitutes the securely-deleting medium while the tape archive constitutes the persistent storage.

How to Forget a Secret. Di Crescenzo et al. (Figure 8.1 (b)) first explicitly considered secure deletion on a storage medium consisting of both a large *persistent medium* and a small *securely-deleting medium* [105]. 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's keys. 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 the *rekeying* of all nodes on the path from the leaf to the root. Rekeying means that a new key is generated to encrypt the new key of the children, recursively until a new master key is generated and stored on the securely-deleting storage. In this scheme, the securely-deleting storage medium needs only to store a single key value.

Ephemerizer. Perlman's Ephemerizer (Figure 8.1 (c)) aims to securely delete communicated messages after an expiration time [1]. 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. Thus, the Ephemerizer deletes data at the granularity of an expiration time.

In this scheme, the Ephemerizer is the securely-deleting medium: it manages a number of a securely-deletable values that are used for many data items in a flat hierarchy, each corresponding to an expiration time. Unlike Boneh and Lipton, however, costly re-wrapping operations linear in the number of data items are avoided by requiring foreknowledge of the data item's expiration time. The message recipient acts as the persistent storage medium: instead of requiring that recipients perform suitable secure deletion promptly after the message expires, the Ephemerizer is introduced to ensure this is done correctly.

A notable aspect of the system is that the Ephemerizer's operator is not the user but a shared service that is not entrusted to know the contents of messages. Two systems are presented to achieve this: one involving triple encryption and one involving blinded decryption.

File System Design with Assured Delete. Perlman extends her previous work on the Ephemerizer to efficiently, reliably, and scalably integrate the service into a file system [8]. To improve reliability and availability, she uses multiple Ephemerizers. Secret sharing [107] is used to divide data encryption keys into n shares, any k-sized quorum of which can determine the key. Each share is made securely-deletable with a different Ephemerizer. Thus, only k such services need to be available for the key to be available, and only $n - k + 1$ such services need to securely delete their key for the key to be securely deleted.

She proposes three manners of storing data with secure deletability: (i) data is stored with its expiration time known in advance; (ii) individual files can be deleted on demand; (iii) classes of files can be created and deleted on demand. The first solution is equivalent to her original Ephemerizer with the addition of a quorum.

The second solution takes Boneh and Lipton's approach [4] and uses Ephemerizers. Each file is encrypted with a unique file key and stored in memory in a file key table; she calls this table the *F-Table*. The encrypted F-Table is periodically backed up to persistent storage—encrypted with a master key that is made securely deletable by a quorum of Ephemerizers. Secure deletion occurs when the corresponding file key is not included in the backup. Perlman notes two risks in this approach. First, there is a high deletion latency depending on the costs of using the Ephemerizer. Second, there is a risk of data loss if the

F-Table is surreptitiously corrupted; this risk is because the file keys are continually re-encrypted with new short-lived keys. If the file key is corrupted and then backed up with a new key, the old (correct) version is irrecoverable.

The third solution aggregates files into *classes*, an idea she further develops in collaboration with Tang et al.'s Fade [2]. Instead of providing the lifetime of files, classes ensure that all the files they contain remain available until the Ephemerizer is instructed to delete the class. This addresses both her concerns with her second solution: deletion occurs promptly and class keys remain available from their creation until they are destroyed. If the class key is once backed up in its correct form, then it is available until a quorum of Ephemerizers discards their keys that permit deriving the corresponding shares.

Keeping Data Secret under Full Compromise using Porter Devices. Pöpper et al. [9] (Figure 8.1 (d)) formalize the problem of communicating secretly against an adversary that observes communications between parties at all times and can also perform a coercive attack to compromise both parties' storage media and secret keys or passphrases. They propose a protocol using a trusted porter device, such as a mobile phone, to store and later securely delete keys that encrypt time-limited data.

Messages are encrypted with a session key negotiated by both parties using Diffie-Hellman key negotiation. The storage and timely deletion of session keys is then managed by the porter device. The sender encrypts the message with the session key and then deletes the key. The encrypted message, along with its expiration time, is then sent to the recipient. The recipient retrieves the key from the porter device to decrypt and read the message. When the message's expiration time is reached, the porter device securely deletes the session key that decrypts the message. In this system, the porter device acts as the securely-deleting storage medium and the communicating parties normal storage is the persistent storage.

Fade. Tang et al.'s Fade [2] extends the third solution from Perlman's File System Design with Assured Delete [8] 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 corresponds to a specific *policy atom* that can expire or be revoked. These policy atoms can be combined using logical OR and AND operators, thus allowing more sophisticated policies to be expressed in a canonical form. The result is a collection of derivable policy keys that can be computed only if the logical expression is TRUE where truth is defined as the securely-deleting storage medium storing the corresponding key.

For instance, a policy may state 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. Logical AND is implemented using nested key wrapping, i.e., all keys must be available to derive the corresponding policy key. Logical OR is implemented by having the key corresponding to each OR operand independently wrap the resulting policy key. Similar to the Ephemerizer, Fade deletes data at the granularity of an entire policy.

Policy-Based Secure Deletion. Cachin et al.'s also design a policy-based secure deletion system with an expressive policy language [106], as well as cryptographic proofs for all constructions. Their system builds a directed policy graph that maps attributes to policies. Attributes' values are either true or false; boolean values feed forward through the graph. Each node is a threshold operator, e.g., if at least k-out-of-n parents are TRUE then it is TRUE. Logical OR and AND are special cases: $k = 1$ and $k = n$, respectively.

Each attribute is associated with a key, and these keys are stored on a securely-deleting storage medium used by the system. When an attribute is no longer true (e.g., a user no long has a role or the data lifetime has expired), then the key for that attribute is securely deleted. Each node is a threshold-cryptographic operator; if the policy language's expression is no longer true, then the corresponding policy key is no longer retrievable. As with Fade, this solution deletes data at the granularity of an entire policy.

Vanish. Geambasu et al.'s Vanish (Figure 8.1 (e)) is a system for securely-deletable communication over the Internet [11]. Messages are each encrypted with random unique keys and communicated between parties. Encryption keys are split into shares that are stored across the Internet in a distributed hash table (DHT). The security of their scheme relies on the nodes in the DHT together implementing a securely-deleting medium: due to the natural churn of data in DHTs, key shares have a limited lifetime before they become irrecoverable. Once there are fewer key shares available than the key-sharing threshold, the session key is irrecoverable and so is the message it encrypts.

Vanish imposes no access control on key shares; in fact, key shares are stored on a volunteer network in which anyone can participate and thus learn key material. The authors present an economic argument that adversaries are unable to view the entire network. However, later work [108] shows that low-cost Sybil attacks are indeed possible as the adversary need only to inspect one part of the network briefly, compile a full collection of key shares and then refuse to delete them.

DNEFS. For completeness, we observe that DNEFS (Figure 8.1 (f)) also provides secure deletion using a small securely-deleting key storage area and a large persistent main storage; they just happen to be the same physical medium (see Chapter 6).

8.4 Summary

Achieving secure deletion on persistent storage media requires encrypting the data and managing the keys on a securely-deleting storage medium. Figure 8.1 shows the different key encryption structures of some related work. Boneh and Lipton store a single master key and have linear updates, Di Crescenzo et al. store a single master key and have logarithmic updates [105], Perlman stores multiple master keys that expire at known times in lieu of updates [1], and Vanish [11], Pöpper et al. [9], and DNEFS (Chapter 6) store a linear number of keys with constant updates. Systems that store a linear number of keys are preferred if the securely-deleting storage medium can fit them all. Otherwise, the choice of data structure is a trade-off between the costs of reading data versus deleting data and depends on the intended workload.

Di Crescenzo et al.'s binary tree key structure has logarithmic read, write and delete operations, however it fixes the tree's size and shape before storing any data. Only the values associated with the nodes can change. The total amount of data that can be stored, therefore, is limited to what is initially fixed. Many useful data structures, however, are dynamic: they grow and shrink to accommodate data at the cost of more complicated update logic. In the next chapter, we prove that any tree-like data structure can be used to provide secure deletion and formalize the requirements on how updates are performed to achieve this. In Chapter 10, we implement a dynamic B-Tree-based securely-deleting data structure from the space of data structures we prove secure.

Another concern with all related work is that the securely-deleting storage medium is always assumed to have perfect storage properties: it never loses data, it always deletes data, it is always available. These assumptions are unrealistic in practice; it is particularly problematic when the securely-deleting storage medium stores a single master key required to access *all data stored.* In Chapter 11, we relax these assumptions and explore the problems that arise as a consequence.

Chapter 9

Secure Data Deletion from Persistent Media

9.1 Introduction

This chapter explores how to securely delete data that is stored on the combination of a persistent storage medium and a securely-deleting one, under the assumption that the data cannot be only stored on the securely-deleting storage. Instead, the persistent storage stores encrypted versions of all the user's data while the encryption keys required to access it are stored on the securely-deleting medium.

To support efficient random-access modifications to data, the data must be encrypted at the appropriate deletion granularity. Small deletion granularities may easily overwhelm the capacity of a limited securely-deleting medium. Key wrapping and key derivation are therefore used to build hierarchies of keys where a small number of master keys are used to derive many fine-grained data item keys.

In this chapter, we develop a new approach to reasoning about this problem by modelling adversarial knowledge as a directed graph of keys and verifying the conditions that result in secure data deletion. We define a generic shadowing graph mutation that models how the adversary's knowledge grows over time. We 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 a computationally-bounded unpredictable multiple-access coercive adversary who is additionally given live access to the persistent medium.

The generic shadowing mutation can express the update behaviour for a broad class of dynamic data structures: those whose underlying structure forms a directed tree (henceforth called an arborescence [109]). This includes self-balancing binary search trees and B-Trees [110], but also linked lists and extendible hash tables [111]. It also expresses the update behaviour of the related work presented in Figure 9.4. In the next chapter, we design a B-Tree-based securely-deleting data structure from the space of arborescent data structures, implement it, and analyse its performance.

9.2 System and Adversarial Model

Our system model is generally consistent with the main model developed in Chapter 3. The user is provided with two storage media: a fixed-sized securely-deleting medium and a dynamic-sized persistent storage medium. We assume that the securely-deleting medium automatically securely deletes any discarded data, i.e., it behaves like a SECDEL implementation. We also assume that the persistent medium behaves like a PERSISTENT implementation and therefore does not securely delete any data. The goal is to use both these media to provide secure deletion for as many data items as possible.

Our adversary is a computationally-bounded unpredictable multiple-access coercive adversary. The adversary also has live access to the persistent storage medium and so learns the data stored on it immediately; adversarial compromise refers only to obtaining access to the securely-deleting storage medium. As always, the adversary has full knowledge of the algorithms and implementation of the system of both the persistent and securely-deleting media.

For clarity in our presentation, we assume that all keys k have a name $\phi(k) \in \mathbb{Z}^+$, where ϕ 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.

9.3 Graph Theory Background

The work in this chapter relies heavily on graph theory. 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 [109].

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. We prohibit self-edges in G: $(u, v) \in V(G) \Rightarrow u \neq v$.

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_v^u to denote this path. The *ancestors* of a vertex v, called $\mathrm{anc}_G(v)$, is the largest subset of $V(G)$ such that v is reachable from each element. The *descendants* of a vertex u, called

$\mathrm{desc}_G(u)$, is the largest subset of $V(G)$ such that each element is reachable from u. If P_v^u 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, $V(G_v) = \mathrm{desc}_G(v)$ and $E(G_v) = E(G)|_{V(G_v)\times V(G_v)}$.

Arborescences and Mangroves. An *arborescence A diverging from a vertex* $r \in V(A)$ 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 [109]. 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 [109]). 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_v^r (Theorem VI.8 [109]). 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 [109]): (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 9.1 shows an example mangrove as well as an arborescent subdigraph induced by a vertex.

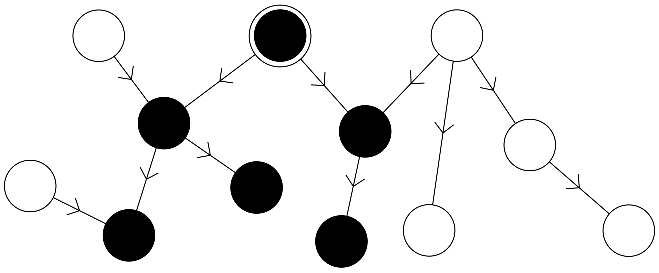


Figure 9.1: An example mangrove. Shaded vertices belong to the arborescent subdigraph induced by the circled vertex.

9.4 Graph-Theoretic Model of Key Disclosure

We now characterize secure deletion in the context of key wrapping and persistent storage. We use this to prove the security of a broad class of mutable data structures when used to retrieve and securely delete data stored on persistent storage. 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 valid instantiations of the mutation's parameters preserve a graph property that simplifies secure deletion.

9.4.1 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 symmetric-key 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 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. In other words, if one knows the key associated with the origin of a path in this graph, one can compute the key associated with the terminus. 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 $\phi : K \to \mathbb{Z}^+$, 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 $G^{adv} = G$.

9.4.2 Secure Deletion

Secure data deletion against an adversary with live access to the persistent storage medium requires that the data's encryption key is securely deleted as well as any values that may derive its value. This means that any ancestor of the data's corresponding vertex in the adversary's key disclosure graph must be securely deleted. 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 ϕ, 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 $R_{live} \subseteq R$ 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_{live} and for all $r \in R \setminus R_{live}$, 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_{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.

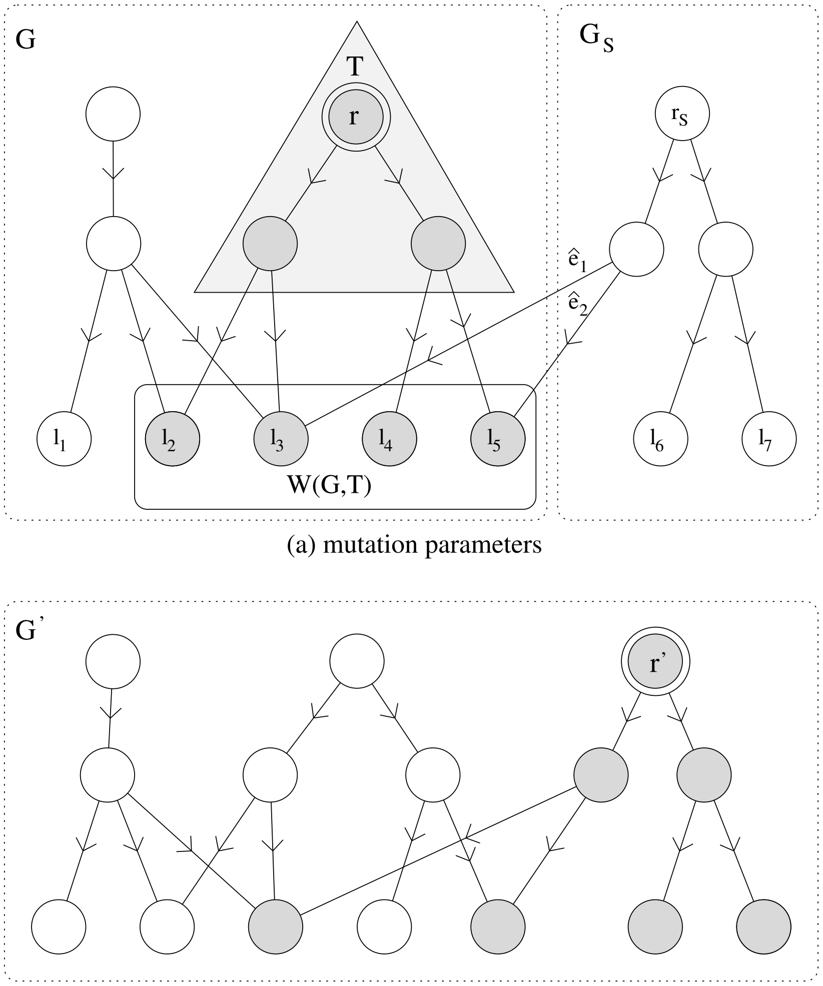
We have shown that to securely delete the data corresponding to a vertex v, we must securely delete data corresponding to all ancestors of v that are not already securely deleted. This is burdensome if it requires a full graph traversal, because the adversary's key disclosure graph perpetually grows. 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., the graph is a mangrove). In the next section, we define a family of graph mutations that preserves this invariant.

9.5 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 [112]. 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 9.2. 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* tree path 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 9.2 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



(a) mutation parameters

(b) post–mutated graph

Figure 9.2: An example shadowing mutation. (a) The parameters of a shadowing graph mutation. (b) The resulting graph. The pre-mutated 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.

user generating new random keys and sending wrapped keys to the adversary, respectively. 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 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.

Mangroves ensure that during the entire course of operations, no additional paths to compute keys were ever unexpectedly generated. Therefore, the client-side cost of managing the key disclosure graph is significantly reduced; secure deletion is achieved by shadowing along the unique path from the vertex that should be deleted to the root vertex, and securely deleting the key corresponding to the previous root.

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 $r_S \in V(G_S)$ 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) $\forall \{(i,j),(i',j')\} \subseteq \hat{E}\ .\ i \neq i' \Rightarrow j \neq j'$ (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 μ that takes as input a graph mutation $(G, r, G_S, T, \hat{E})$ and outputs the mutated graph G', defined by $V(G') = V(G) \cup V(G_S)$ and $E(G') = E(G) \cup E(G_S) \cup \hat{E}$. 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 $\hat{e} \in \hat{E}$.

9.5.1 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^i_v and P^j_v. Since $i, j \in V(G_r)$, there exist distinct paths P^r_i and P^r_j. Consequently, $P^r_i P^i_v$ and $P^r_j P^j_v$ 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) $P^r_v = P^r_i P^i_j P^j_v$ and $P^r_j = P^r_i P^i_j$. However, by definition of $W(G, T)$, all edges except the final one in P^r_i and P^r_j are in $E(T)$. If P^i_j is non-degenerate, then $P^r_i P^i_j \neq P^r_j$ as P^r_i has an edge outside of T followed by more than one edge. Therefore, P^i_j is degenerate and $i = j$, as needed. □

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

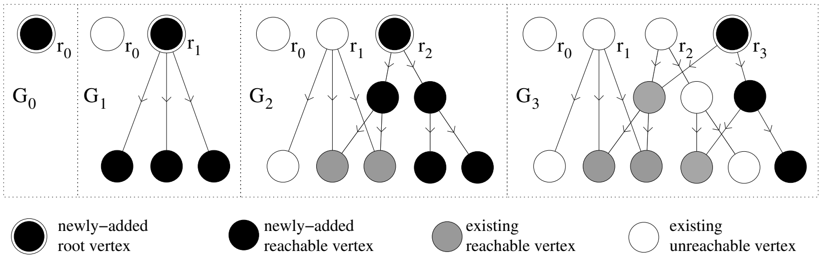

r0
G0
r0
r1
G1
r0
r1
r2
G2
r0
r1
r2
r3
G3
newly–added root vertex
newly–added reachable vertex
existing reachable vertex
existing unreachable vertex

Figure 9.3: 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.

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 $G' = \mu(G, r, G_S, T, \hat{E})$, 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, 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_v^r and Q_v^r 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_v^r and Q_v^r are elements of $V(G)$. Also, by construction, $E(G')|_{V(G)\times V(G)} = E(G)$, and thus all edges of P_v^r and Q_v^r are elements of $E(G)$. Therefore, P_v^r and Q_v^r 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_v^r and Q_v^r 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: $P_v^r = P_u^r, (u,w), P_v^w$ and $Q_x^r, (x,y), Q_v^y$, where (u,w) and (x,y) are elements of $\hat{E}$. We know that $P_v^r \neq Q_v^r$, 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 $P_u^r \neq Q_x^r$ or $P_v^w \neq Q_v^w$. 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_v^w and Q_v^y 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_v^r and Q_v^r cannot exist. Therefore, for all $r \in V(G')$, G'_r is an arborescence and so G' is a mangrove. □

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

A shadowing graph mutation chain describes a sequence of mutations applied on a key disclosure graph. Figure 9.3 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. *Let $\mathcal{M} = (M_0, \ldots, M_p)$ be a shadowing graph mutation chain. 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 $P_v^{r_{j+1}}$ in G_{j+1}. Since $r_{j+1} \in V(G_{S,j})$ and $v \notin V(G_{S,j})$, such a path has the form $P_{\hat{e}}^{r_{j+1}}(\hat{e}, \hat{e}')P_v^{\hat{e}'}$ where $(\hat{e}, \hat{e}') \in \hat{E}_j$ and $P_v^{\hat{e}'}$ is a path in G_j. Then $\hat{e}' \in W(G_j, T_j)$ and so $P_{\hat{e}'}^{r_j}$ is a path in G_j, implying that $P_{\hat{e}'}^{r_j} P_v^{\hat{e}'}$ is a path from r_j to v in G_j, which leads to 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 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.

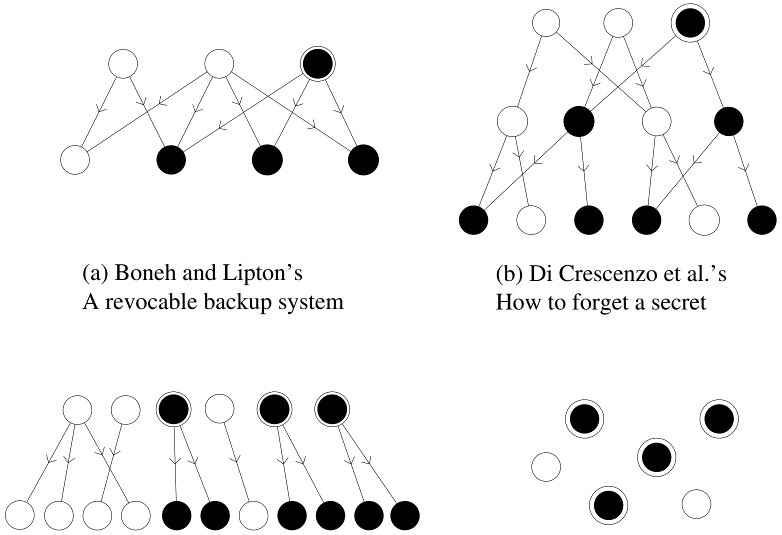
Proof. The proof is by establishing the premises required for Definition 2. First, $R = \{k_0, \ldots, k_p\}$ and $R_{\text{live}} = \{k_i, \ldots, k_{i+l}\}$ which means that $R_{\text{dead}} = \{k_0, \ldots, k_{i-1}\} \cup \{k_{i+l+1}, \ldots, k_p\}$. Because no compromise occurs from time t_i until 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. $\square$

Theorem 2 shows that mangrove-shaped key disclosure graphs that evolve through a sequence of shadowing graph mutations provide simple criteria for secure data deletion. Interestingly, as we show next, some related work also induce mangrove-shaped key disclosure graphs.

9.5.3 Mangrove Key Disclosure Graphs in Related Work

Figure 8.1 in Section 8.3 illustrated the key wrapping structures for some related work. Figure 9.4 extends this earlier figure to show example key disclosure graphs that can be generated after a couple update operations for each of the related work. Observe that they all have mangrove-shaped key disclosure graphs. As a result, the security of these systems follow from the results in this section. Admittedly, the machinery of our proofs is beyond what is necessary if our only goal was to prove the security of a clocked keystore such as DNEFS. Nevertheless, we observe that its security conveniently follows as a corollary from this work due to the fact that its key disclosure graph is a mangrove.


(a) Boneh and Lipton's
A revocable backup system

(b) Di Crescenzo et al.'s
How to forget a secret

(c) Perlman's Ephemerizer

(d) Pöpper et al's Porter Devices
(e) Geambasu et al.'s Vanish
(f) DNEFS

Figure 9.4: Mangrove-shaped key disclosure graphs for related work. Circled nodes represent keys currently stored on the securely-deleting storage; black nodes are currently derivable keys; white nodes are securely deleted keys.

9.6 Summary

We developed a general approach for the design and analysis of secure deletion solutions from persistent media. We defined a key disclosure graph that models the growth of adversarial knowledge as wrapped keys are written to the persistent storage. We defined the conditions by which data is securely deleted against this adversary. We showed that if the key disclosure graph has the shape of a mangrove then ensuring secure deletion is more easily achieved: no additional data must be stored about the adversary's knowledge other than the data structure used to manage currently valid data.

To ensure that the key disclosure graph retains its mangrove property, we defined a shadowing graph mutation, which is sufficiently generic to express the update behaviour of any arborescent data structure. We proved that applying any shadowing graph mutation to any mangrove always results in a mangrove, and that chains of these mutations can be constructed to reflect arbitrary sequences of the data structure storing and deleting data items. This provides secure data deletion against a computationally-bounded unpredictable multiple-access coercive adversary, turning the storage medium into either a SECDEL or SECDEL-CLOCK implementation, depending on whether shadowing mutations are performed immediately or batched, respectively.

Chapter 10

B-Tree-Based Secure Deletion

10.1 Introduction

This chapter builds on the previous chapter of secure deletion for persistent storage by using a small securely-deleting storage medium, which explored the general space of possibilities. We design and implement a concrete securely-deleting data structure from this space. Our motivation is to provide a *dynamic* data structure, whose capacity can grow and shrink as necessary based on the current requirements.

The B-Tree implements a securely-deleting key-value map that maps data handles to data items; new pairs can be inserted, existing pairs can be removed, and any stored data item can be updated. Our B-Tree collects multiple updates and performs them in batch. It therefore consists of two parts, a skeleton tree managed locally to the user and the full tree stored on the persistent storage medium. Periodically, all changes local to the skeleton tree are collected into a single shadowing graph mutation and applied to the full tree. After each update, a new root key value is stored in the securely-deleting storage medium, which divides time into deletion epochs; the solution then behaves like a securely-deleting clocked implementation with a corresponding deletion latency (Figure 3.2 in Chapter 3). An optional crash-safety mechanism we propose further adds existential latency (Figure 3.3 in Chapter 3).

We implement our B-Tree-based instance 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 analysing 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` [113]. We show that the caching strategy approximates the theoretical optimal (i.e., Bélády's "clairvoyant" strategy [114]) 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.

10.2 System and Adversarial Model

This chapter focuses on the design and implementation of a solution whose general space and security is described in Chapter 9. The system and adversarial model we use is identical to the one described in Section 9.2. The update behaviour of our B-Tree design is expressible as a shadowing graph mutation. By applying the results of Chapter 9, our B-Tree comes from the general space of possible solutions that we proved secure.

10.3 Background

A B-Tree is a self-balancing search tree [110] 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 [110]. (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.

10.3.1 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.

10.3.2 B-Tree Balance Operations

A B-Tree of order N is balanced when (i) the number of children of each non-root node is inclusively between $\lceil \frac{N}{2} \rceil$ and N and (ii) the number of children in the 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 and can propagate 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.

10.4 Securely-Deleting B-Tree Design

We use a B-Tree to organize, access, and securely delete data. We assume that only a constant number of B-Tree nodes can be stored on the securely-deleting storage medium. 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 [115]. An authentic parent node guarantees the authenticity of its children. The root hash is stored with the key.

We use a form of *shadowed updates* [112] when updating B-Tree nodes. A shadowed update means that when a new version of a node is written, it is written to a new location. A node that references it (i.e., its parent) must also be updated to store the new location, propagating shadowing to the root. We use a variation on shadowed updates: instead of updating the location of the data, we instead update the key that decrypts it. Consequently, any change to a leaf node results in new versions of all ancestor nodes up to the root. This is analogous to normal shadowed updated if one imagines the encryption key as a pointer to the data's location.

10.4.1 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 static IV. 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.

10.4.2 Data Integrity

To ensure that the persistent storage has correctly returned all data, it is necessary to verify the integrity of the data received. Merkle Hash Trees [115] provide such a construction for a binary tree data structures: each node in the tree contains a hash of the concatenated values of its two children, and an authentic copy of the hash of the root is sufficient to verify the integrity of all nodes in the tree. Mykletun et al. propose extending this to provide integrity for the many children of a B-Tree [116]. The use of a cryptographic hash function ensures that the received data is protected against both accidental and deliberate modification.

We use a variant of this approach in our solution: each node is hashed but the hashes of the children are independently stored in their parent (alongside the child's decryption key). Therefore, the parent stores for each child a key, a hash, a search index, and a storage location. The leaves of our B-Tree then store the hashes of the actual data blocks they index. Hashes for data blocks are computed when they are written, and all other B-Tree nodes have their hashes recomputed before writing them to persistent storage. This allows efficient updates to B-Trees with large numbers of children because only the children that actually changed need to have their hashes recomputed, not all children of a node. Moreover, it is these modified nodes that are available in the user's skeleton tree and must be also re-encrypted during committing.

10.4.3 Versioning

Some cloud storage research focuses on ensuring that the most recent version of data is always returned to protect against remote storage media that may return previous versions of valid data instead of the most recent. For instance, tahoefs [117] queries a variety of servers and each returns a data block and a version; tahoefs assumes that the highest received version number is the correct one.

A nice property of our solution is that whenever a data block is updated, the old version becomes irrecoverable even to the user. This means that our solution achieves authenticated versions as a side effect. If a data block can be correctly decrypted, it is therefore the newest version.

10.4.4 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 rekey. Finally, it allows the user to control how often the securely-deleting storage medium is updated (i.e., the clock period and relevant latencies). This is useful if using the medium's deletion operation 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 10.1 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 confirmed by using its hash value stored at the parent and 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) children, 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 10.6 we present our experimental results with eviction strategies.

All B-Tree modifications—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

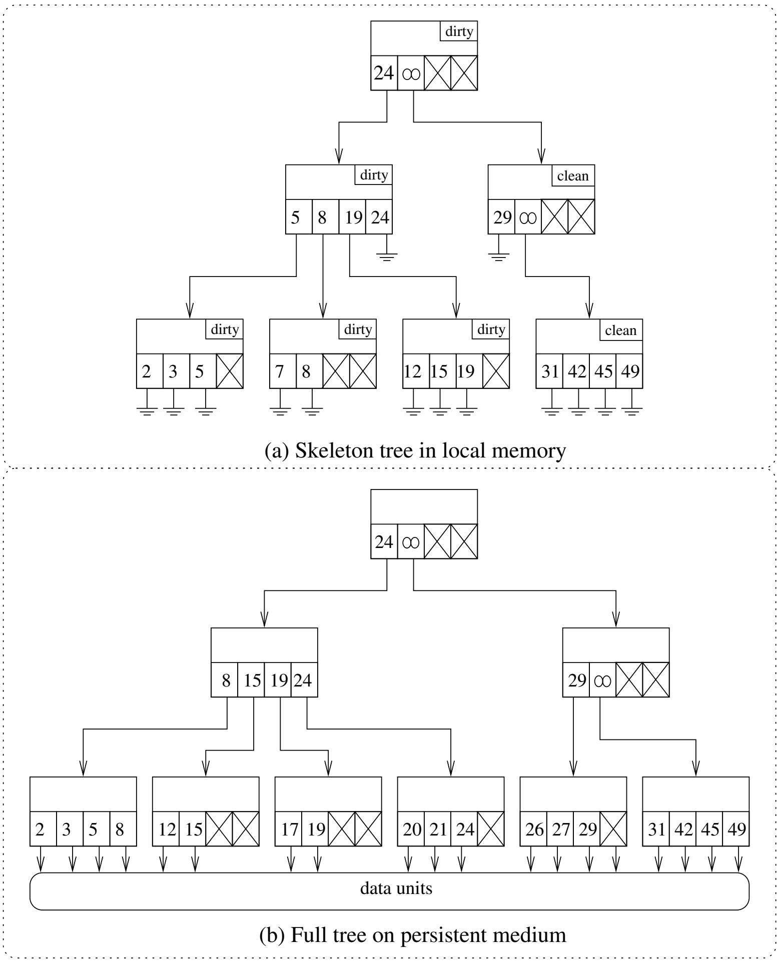


Figure 10.1: Example of a B-Tree stored on the persistent medium along with an in-memory skeleton tree. (a) shows the skeleton tree of B-Tree nodes, where node 42 was read and local changes were made: the node 7 was added and the node 17 was deleted, causing a split operation and a fuse operation respectively. (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.

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.

10.4.5 Commitment

The B-Tree commit operation is periodically performed and is the clock of our system. Time is thus divided into deletion epochs characterized by the particular master encryption key stored in the securely-deleting storage medium. Commit 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. This means that deleted data items have a deletion latency bounded by the clock period, i.e., a SECDEL-CLOCK implementation.

The commit operation handles two kinds of dirty nodes: *deleted* ones that have been deleted from the B-Tree through 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., its children's keys, hashes, and search values) is then encrypted with its newly-generated key and stored on persistent storage.

10.4.6 Crash Safety

A critical feature in the design of storage systems is crash safety, which aims to minimize the data loss due to unexpected events such as a system crash. Unsaved data may reside in memory buffers waiting to be committed; such data is lost in the event of power loss. Thus, the commit period is a trade-off between data loss risk and increased overhead. The overhead is induced by the cost of rekeying and storing dirty tree nodes. Therefore, we use a journalling mechanism that allows the recovery of uncommitted data. This removes the risk of data loss from the trade-off.

When data is written, its index reference and encryption key are written to a journal on the local storage medium. This permits the encrypted data stored on remote storage to be decrypted without updating the encrypted parents. Similarly, when data is deleted from the remote storage, a record of this deletion is made in the journal. The journal is securely deleted after flushing the tree and replayed whenever the server application is started. This results in the correct reconstruction of the B-Tree's internal state at the time of system crash.

If the securely-deleting storage medium is too resource constrained to maintain an adequate journal, then the user can safely store all changes by wrapping the including the data item's key directly wrapped by the current root key. This permits secure deletion because the user is assured that the old root key is destroyed at the next commit operation—the wrapped key is useful only in the event of power loss before that commit operation occurs. The consequence of this is that it introduces an existential latency for the data: the compromise of the root key before the data is written as well as continuous access to the persistent storage provides access to the data. In this case, the existential latency is bounded by the clock period and the B-Tree has the behaviour of a SECDEL-CLOCK-EXIST implementation.

Observe that while this direct wrapping is not a shadowing mutation, it is easy to show that applying it to a mangrove still preserves the key disclosure graph's mangroveness: only one vertex and one edge are added such that the new vertex has outdegree zero (preventing cycles) and indegree one; the indegree of all other vertices is unchanged.

10.5 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 described in this chapter 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.

10.5.1 Data Storage

Our solution assumes the user's storage system divides the data into data items 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 data items, and keys (handles) reference this data. Data items can be entire files, components of files, etc. This allows our solution to implement a simple object storage device (OSD) [71]. If each data item is uniquely indexed by the B-Tree, then modifying the data item 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.

10.5.2 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.

By default, the `nbd-server` program uses a local file to implement the block device. This is similar to the loopback device (e.g., `/dev/loop0`), except that the `nbd-server` can run on a separate machine than the file that corresponds to the block device. The `nbd-client` program tells the running kernel how to connect to a `nbd-server` program and which `nbd` device number it should connect to it. After successfully executing `nbd-client`, the corresponding block device forwards all requests to the desired `nbd-server` program. This permits our solution to be easily integrated into any Linux system without modifying the operating system. By default, however, running `nbd-client` and mounting the device driver `nbd` as a file system requires root privileges. Our solution replaces only the user-level `nbd-server` tool with our B-Tree implementation.

10.5.3 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).

10.5.4 Caches

Our solution caches data in multiple locations. Two important caches are the skeleton B-Tree and a working space for the `nbd` device. The first ensures that the B-Tree's dirty nodes do not need to be flushed—and the root key changed—whenever data is stored or removed. The second ensures that if the `nbd` device sequentially issues many small read or write requests on the same stored block, then the block is only retrieved once.

In Section 10.6 we test a variety of cache sizes and eviction strategies to quantify the success of caching. In our final design, we use a simple least-recently-used eviction strategy for our B-Tree cache. Our approach is modified from simple cache eviction because we require a skeleton tree and a full commit of all dirty nodes during the clock operation. We therefore only perform a full flush of all dirty cached B-Tree nodes whenever there is insufficient cache space to accommodate the requirements of the worst-case B-Tree operation. The clean nodes can then be heuristically evicted as needed when they are leaves in the skeleton tree.

10.6 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.

10.6.1 Workloads

We test our implementation's performance on a variety of different file system workloads. We used the `filebench` utility [113] 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 prewritten 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 prewritten 25 GiB file. This tests the performance for 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.

		Cache size: 10 items			Cache size: 50 items		
Workload	block size	LRU	LFU	Bélády	LRU	LFU	Bélády
seq	1 KiB	97.5	28.2	97.7	98.8	38.7	98.8
	16 KiB	99.7	47.7	99.8	99.9	99.0	99.9
rand (1 KiB)	1 KiB	11.8	15.3	19.8	26.4	26.3	33.5
	16 KiB	49.3	40.1	58.2	63.4	67.2	70.3
rand (1 MiB)	1 KiB	97.5	25.4	97.7	97.9	38.0	98.0
	16 KiB	98.6	62.6	98.7	98.9	98.3	99.1
svn	1 KiB	96.0	47.2	96.1	97.1	75.9	97.4
	16 KiB	97.8	81.2	97.8	98.6	96.2	99.0

Table 10.1: Caching hit ratio (%) for B-Tree nodes

10.6.2 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 [114], least recently used (LRU), and least frequently used (LFU). Bélády's strategy is included as an objective reference point when comparing caching strategies. We only maintain cache usage statistics for items currently in the cache.

The results of our experiment are shown in Table 10.1. 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.

10.6.3 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 and workload; it is independent of using a 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.

		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 overhead (%)	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	20 671.4
	block-size overhead (%)	497.9	2293.2	9473	38 191.8
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 overhead (%)	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 overhead (%)	0	63.4	247.9	750.2

Table 10.2: B-Tree secure deletion overhead

- 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.

Table 10.2 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 benefit the most from 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. As expected, the block size overhead resulting from fetching unnecessary data shows a large amount of waste.

10.7 Conclusions

We designed, implemented, and analysed a securely-deleting B-Tree, whose design is taken from the space of securely-deleting data structures that we prove secure in Chapter 9. It uses a local skeleton tree to cache all modifications and performs a period commit operation to synchronize them and securely delete discarded data. 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.

In our a B-Tree, a single master key is stored on the securely-deleting storage medium and is required to access all stored data. In the next chapter, we consider the problem of an unreliable securely-deleting storage medium, i.e., one that may fail to be available, to correctly store data, or correctly delete data. We design a robust storage medium to account for these risks.

Chapter 11

Robust Key Management for Secure Data Deletion

11.1 Introduction

As we have seen throughout the previous chapters, data encryption is a useful tool for reducing the problem of secure data deletion to the problem of deleting the corresponding encryption keys. In general, these keys are smaller and so more easily managed and controlled than the data itself. In related work [1,4,9,105], as well as our own work from Chapters 6, 7, 9, and 10, the securely-deleting storage medium is assumed to have perfect storage characteristics: it never loses data, never exposes data except through compromise, it always correctly deletes data, and it is always available. These are strong and unrealistic assumptions to place on a storage medium. Moreover, the risk of data loss is amplified by the ratio between the size of the key and the data it encrypts. In particular, Chapter 10's B-Tree-based design used only a single encryption key to encrypt all the data stored on the persistent storage. The loss of these 16 bytes is devastating to such a system.

In this chapter, we remove the strong and unrealistic assumptions on the securely-deleting medium's reliability, integrity, and confidentiality. We allow the securely-deleting medium to be unavailable or to partially fail—either by discarding valid data or by failing to securely delete discarded data. In this chapter we explore the effect that this has on the secure deletability and availability guarantees on the securely-deleting storage medium.

We use a keystore system like DNEFS (Chapter 6) to provide secure deletion. We improve the robust storage and secure deletion of data by distributing the securely-deleting medium over multiple nodes. This is particularly challenging since storing additional copies of data decreases the chance of its loss but increases the chance that one copy's deletion fails. At the minimum, we ensure that the failure or compromise of any single node in the system has no effect on the secure deletability or availability of data. We present a two-dimensional secret-sharing and erasure code that balances these properties. In one dimension, key values are replicated across a subset nodes; in the other dimension, multiple key values are used to create encryption keys. We ensure that multiple key values can be selected such that no single node in the network store any two of them.

We implement both our distributed keystore system and a FUSE-based [118] file system that uses the keystore to store securely-deletable data with a variable encryption granularity. We test our design and measure our system's performance with respect to various parameters including the number of nodes and replication. We find that the latency of key operations is very small and that the service rate is high; the latency remains small even as the service rate reaches its capacity. Individual nodes can service 14 thousand requests per second with a latency (including local network delay) of approximately 725 microseconds. Moreover, the communication among keystore nodes is small, requiring, for example, 4 KiB/s to generate and synchronize millions of keys every ten minutes.

11.2 System and Adversarial Model

Figure 11.1 illustrates our system model. Each arrow is a mutually-authentic, forward-secure communication channel, that is, a secure channel offering secrecy and authentication that also achieves perfect-forward secrecy. Multiple clients store and retrieve data using two shared storage media: a distributed system of keystore nodes that together form a clocked keystore and a PERSISTENT storage medium (henceforth called the *content store*). This section overviews each en-

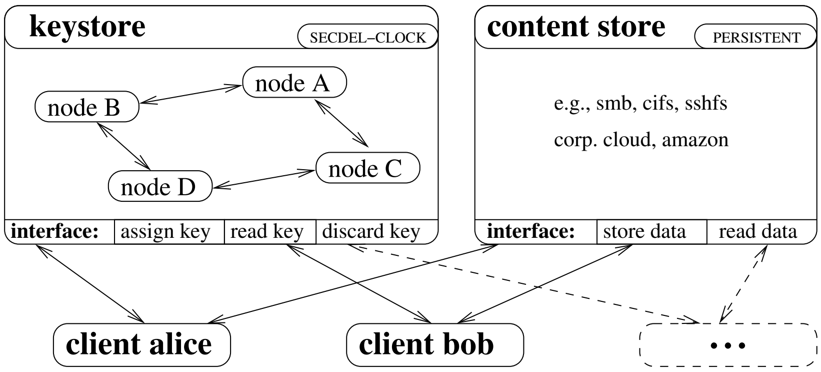


Figure 11.1: Diagram of the keystore system. Each arrow is a mutually-authentic, forward-secure communications channel. Multiple clients interact both with a content store and a distributed keystore. The content store has a PERSISTENT implementation. The keystore consists of a set of keystore nodes which mutually communicate and together implement a CLOCKED KEYSTORE.

tity and presents our adversarial model. Further details on components are presented in the subsequent sections: Section 11.3 describes the keystore; Section 11.4 describes how keystore nodes communicate; Section 11.6 describes how the client uses these components to store data in a securely-deletable way.

In this chapter, the clocked keystore is not a single storage medium but is now a distributed collection of component keystore nodes, each with access to a securely-deleting storage medium. We also assume that there are multiple users of the system and that access to the storage media is shared, though user data is not necessarily shared. Finally, we extend the abilities of the adversary. This remainder of this section describes these model extensions in more detail.

Keystore. The *keystore* is a set of *keystore nodes*, which store, retrieve, and securely delete randomly-generated binary strings (i.e., key values). We assume that keystore nodes can authentically and forward-securely communicate among themselves. Authenticity means that communicating nodes know their partner's identity and agree on data sent. Forward security means that the compromise of a node's long-term key does not reveal the content of previous communications secured using that key. For example, TLS [119] with client authentication and Diffie-Hellman key negotiation [120] satisfies these properties, provided that each node can verify the other nodes' certificates. We further assume that all nodes can store data such that it is securely deletable, i.e., each node has access to a securely-deleting storage medium. We assume that this securely-deleting storage medium may occasionally fail to correctly store data, delete data, or keep data confidential.

Content Store. The *content store* is the persistent storage medium (e.g., a virtual network file system), which stores encrypted data. The content store also performs appropriate access control on the data: authenticating clients and checking permissions to access stored items. We make no assumptions on the content store's ability to perform secure deletion. Moreover, the way that it handles data's erasure coding, striping, migrating, defragmenting, backing up, etc., is unaffected by our solution.

Client. The *client* stores and retrieves data.[1] There can be multiple clients using the system and clients must not be able to access other clients' data. We assume that clients can establish an authentic and forward-secure connection to the *keystore nodes* and the *content store* [119]. This can be scalably achieved in various ways. Our implementation uses a public-key infrastructure with a certificate authority [120, 121]; clients then verify the certificates of the entities to ensure that the public keys are valid before negotiating a TLS-secured communication channel.

Adversarial Model. We have the same concept of data items, data lifetimes, and secure deletion as the other work in this thesis. We assume the existence of a computationally-bounded unpredictable multiple-access coercive adversary who can gain access to the keystore nodes and clients' secrets that may be needed to access their data.

Our adversarial model augments the adversary with the following *non-coercive attacks*:

- **A1** The content store is perpetually compromised. Data stored on it is immediately given to the adversary. This models settings where the content store is implemented using an untrusted

[1]We use the new term client as a means of emphasizing that there are now multiple entities sharing a common securely-deleting keystore.

or potentially-compromised third-party storage system, such as Amazon Cloud [122], Dropbox [123], or Google Drive [124].

- **A2** There are multiple legitimate clients who have access to some, but not all, data. The adversary may be a legitimate client (or a coalition thereof) and thus use client privileges to access other clients' data.
- **A3** Some keystore nodes may fail. In particular, they may fail to delete data, fail to correctly store data, fail to maintain the confidentiality of data, fail in their availability, e.g., by being unable to respond to requests to delete data, and fail by generating predictable random numbers.
- **A4** There is a computationally-bounded active network adversary for all communication—both client to storage medium and storage medium to storage medium. We assume that a long-term denial-of-service attack on all keystore nodes is outside the adversary's abilities.

11.3 Distributed Keystore

In Chapter 6 we introduced the notion of a clocked keystore, which provides individually-assignable securely-deletable *key values* (KVs). KVs are used to generate encryption keys. Each key has a corresponding *access token* (AT). Each data item is encrypted with its own unique KV and secure deletion is achieved against a computationally-bounded adversary by securely deleting the KV that corresponds to the data item.

The following properties ensure that this system provides secure data deletion at a fine granularity:

- **P1** The KVs associated to the ATs returned by `assign` must be unpredictable.
- **P2** KVs returned by `read` must be the same for a particular AT from the time `assign` returns it until it is provided to `discard`; further, `read` must not return $\perp$ during this time.
- **P3** KVs returned by `read` must be unpredictable given the keystore's state at all times before the time `assign` returns its corresponding AT minus a bounded existential latency, and at all

times after the time `discard` is called with its corresponding AT plus a bounded deletion latency.

In this section, we design a *distributed clocked keystore* that is robust against partial failures in the storage medium. It is composed of a set of keystore nodes. Each node maintains a local state, which includes the state and value for a set of key positions (KPs). We describe the keystore nodes' client interface, and how they together effect a clocked keystore. We then present the distributed synchronization method, and measures to detect and correct for Byzantine failures.

In our setting, multiple clients share access to the keystore. It is therefore necessary to provide access control on KVs. We use unpredictable access tokens to accomplish that. As such, we further require an additional keystore property:

- **P4** ATs returned by `assign` must be unpredictable.

This provides the condition that knowledge of an AT allows knowledge of the corresponding KV.

11.3.1 Distributed Clocked Keystore

We now explain how to distribute our clocked keystore implementation over multiple keystore nodes and achieve correctness and robustness. Each keystore node maintains a local state, which includes the state and value for a set of positions. Table 11.1 lists the node-local state used by our algorithms. In it, Π represents the space of possible KPs, κ is a security parameter, and δ is the desired bound for existential and deletion latencies. Recall from Chapter 6 that the possible states for a KP are unused (**U**), assigned (**A**), and discarded (**D**).

As before, each node offers the client a keystore interface: assign a KV, read the KV associated with a KP, and discard the KV associated with a KP. Nodes are, however, now responsible for only a subset of KPs; the set of nodes responsible for a KP is called the KP's *replication set*. Reading a KP for which a node is not responsible returns $\perp$. Algorithms 1, 2, and 3 provide the client-side assign, read, and discard algorithms respectively. The DOM keyword takes the domain of a mapping. The **with** keyword indicates arbitrary selection.

```
$\Pi$                                              // set of KPs
$\kappa$                                           // security parameter
$\delta$                                           // bound for latencies
St {
  me: $\mathbb{Z}^n$                               // keystore node number
  PK: $\mathbb{Z}^n \to$ pubkey                    // public keys for peers
  assigner: $\Pi \nrightarrow \mathbb{Z}^n$        // maps position to unique assigner
  replicators: $\Pi \nrightarrow 2^{\mathbb{Z}^n}$ // maps position to set of replicators
  state: $\Pi \nrightarrow \{\mathbf{U}, \mathbf{A}, \mathbf{D}\}$   // maps position to state
  value: $\Pi \nrightarrow \{0,1\}^\kappa$         // maps position to KV
  update_number: $\Pi \nrightarrow \mathbb{Z}^+$   // maps position to update count
  update_commit: $\Pi \nrightarrow \{0,1\}^\kappa$ // maps position to current commit
  last_update: $\Pi \nrightarrow \mathbb{Z}^+$     // timestamp of last update
  updating: $\Pi \nrightarrow \{\mathbf{true}, \mathbf{false}\}$   // is position being updated

  // check that the position stores a fresh value
  is_recent(t) $\triangleq (\mathbf{now} - t < \delta)$
  // the set of positions that are assignable for this node
  assignable $\triangleq \{i \in \Pi : \text{state}(\pi) = \mathbf{U} \wedge \text{assigner}(\pi) = \text{me}$
                 $\wedge\ \text{is\_recent}(\text{last\_update}(\pi)) \wedge \neg\text{updating}(\pi)\}$
}
```

Table 11.1: Keystore node local state.

Algorithm 1: assign

```
local state: St – keystore node state (Table 11.1)
output     : AT or $\perp$ – access token or fail
begin
    if $St.assignable = \emptyset$ then
        return $\perp$;
    with $\pi \in St.assignable$ do
        St.state($\pi$) $\leftarrow$ $\mathbf{A}$;
        return $\pi \| \texttt{hash}(\text{St.value}(\pi))$;
```

Algorithm 2: read

```
local state: St – keystore node state (Table 11.1)
input      : $\pi \| t$ – access token (position and hash)
output     : KV or $\perp$ – key value for the position or fail
begin
    if $\pi \notin \textsc{dom}(St.value)$ then
        return $\perp$;
    if $\mathtt{hash}(St.value(\pi)) \neq t$ then
        return $\perp$;
    return $St.value(\pi)$;
```

Algorithm 3: discard

```
local state: St – keystore node state (Table 11.1)
input      : $\pi \| t$ – access token (position and hash)
output     : $\top$ or $\perp$ – success or fail
begin
    if $\pi \notin \textsc{dom}(St.value)$ then
        return $\perp$;
    if $\mathtt{hash}(St.value(\pi)) \neq t$ then
        return $\perp$;
    St.state($\pi$) $\leftarrow$ D;
    return $\top$;
```

11.3.2 Distributed Keystore Correctness

In order for our distributed system to implement a keystore, it must guarantee properties **P1–3** as well as **P4** for our shared setting. It does this by implementing a distributed version of the clocked keystore, which has these properties.

An immediate problem with assignment of KPs in a distributed system is ensuring that two nodes do not assign the same KP. We remedy this by associating each KP with a unique *assigner*. Only the assigner may assign the value and all other nodes that store the value are *replicators*; a KP's *replication set* therefore consists of a single assigner and some number of replicators. The set of assignable positions among keystore nodes are pairwise disjoint. This restriction does not limit our solution because the client has no preference over assignable positions: the client asks a node for an unused KV and accepts it. Nodes create their own assignable KPs as necessary and cooperate to replicate them, ensuring that all nodes can satisfy a client's assign request.

To see why **P1** holds, first observe that it holds for each node individually since **U** KPs store unpredictable KVs and are changed to **A** when assigned; the KV is replaced with a new value when the corresponding KP returns to **U**. Second, each keystore node is given a unique set of assignable KPs, so given all assignable KVs from all other nodes, the assignable KVs from the remaining node are still unpredictable. **P2** holds because nodes in the replication set can read the KV. **P3** holds because a distributed synchronization protocol (described in Section 11.4) is periodically executed among nodes that store the KP, which implements the clock function of the keystore. **P4** holds because ATs include the cryptographic hash of the corresponding KV—itself unpredictable through **P1**.

Note that the keystore properties are only provided assuming that no node fails. In the next section, we describe in detail the synchronization protocol that runs among the nodes in the replication set and then consider the Byzantine failures that can occur and design our system to be robust against them.

11.4 Synchronization

Keystore nodes in the distributed keystore use a *synchronization* protocol to ensure that KPs have consistent states and values. Synchronization results in the distributed keystore having the behaviour of a simple keystore and therefore achieving properties **P1–4**. Synchronization is an assigner-led two-phase protocol, similar to the two-phase commit atomic commitment protocol [125]. The first *pull* phase (cf. voting) collects all state changes from replicators. The second *push* phase (cf. commit) provides replicators with the pulled information, allowing them to all independently consent on the new state and value.

Algorithm 4 presents the synchronization procedure as run by the assigner. The **assert** keyword represents a check or condition that must hold; if any asserted statements fail—e.g., because a signature is forged—then the update is aborted. We note that while the algorithms presented here describe the synchronization of a single key position, our implementation improves the communication complexity by synchronizing many key positions simultaneously. We restate that communication between nodes occurs over mutually-authentic forward-secure TLS-encrypted communication channels.

In the *pull* phase the assigner requests the current state from each replicator as well as a random contribution used to generate the new KV. Algorithm 5 presents the replicator-executed pull algorithm. The assigner commits to its random contribution at this time, which the replicators store to check during the push phase. Each replicator returns its identity, the synchronization round number, the KP's ID and state, a random contribution, and the assigner's commitment, as well as a public-key signature of this data. These pulled values are collected for all nodes in the replication set and distributed to each (including the assigner) during the subsequent *push* phase.

Algorithm 6 presents the push algorithm executed by all nodes in the replication set. It takes as arguments the KP to push, the assigner's previously committed random value, and the set of pull messages aggregated by the assigner. It checks that only nodes in the replication set appear in the set of messages and that each node appears exactly once, that each message concerns the correct position and synchronization round number, that the assigner's random contribution agrees with its previous commitment, that all nodes were given the same commitment, and that each push message is correctly signed by the providing node. If so, then the vote-state algorithm is used to update the state and determine if the value must be replaced, using the same procedure as the assigner. The push algorithm returns the hash of the update number, position number, and the value. The assigner checks for each replicator that their hash matches its own version.

Algorithm 4: synchronize

```
local state: St – keystore node state (Table 11.1)
           : peer – map from peer number to actual peer
input      : $\pi$ – key position to synchronize
begin
    assert St.assigner($\pi$) = St.me;
    atomic
        if St.updating($\pi$) = true then
            return
        St.updating($\pi$) ← true;
    /* setup                                              */
    St.update_number($\pi$) += 1;
    $i \leftarrow$ St.update_number($\pi$);
    $k_{\mathrm{me}} \leftarrow$ random-key();
    $c \leftarrow$ hash($k_{\mathrm{me}}$);
    $s \leftarrow$ St.state($\pi$);
    St.update_commit($\pi$) ← $c$;
    /* pull phase                                         */
    $R \leftarrow [(\pi, \mathrm{me}, i, s, k, c, \mathtt{sign}(\pi \| \mathrm{me} \| i \| s \| k \| c))]$;
    for $r \in$ St.replicators($\pi$) do
        $R$.append(peer($r$).pull($i, \pi, c$));
    /* push phase                                         */
    $h \leftarrow$ peer(St.me).push($\pi, R$);
    for $r \in$ St.replicators($\pi$) do
        $h' \leftarrow$ peer($r$).push($\pi, R$);
        assert $h' = h$;
    St.updating($\pi$) ← false;
```

Algorithm 5: pull

```
local state: St – keystore node state (Table 11.1)
input      : $i$ – update number
           : $\pi$ – key position to pull
           : $c$ – commitment for push phase
output     : $\pi$ – key position being pulled
           : $r$ – my keystore number
           : $i$ – output number
           : $s$ – position state
           : $k$ – random contribution
           : $\sigma$ – signature of $(\pi \| r \| i \| s \| k)$
begin
    $r \leftarrow$ St.me;
    assert St.update_number$(\pi) + 1 = i$;
    St.update_number$(\pi) \leftarrow i$;
    St.update_commit$(\pi) \leftarrow c$;
    $s \leftarrow$ St.state$(\pi)$;
    $k \leftarrow$ random-key();
    return $(\pi, r, i, s, k, c, \mathit{sign}(\pi \| r \| i \| s \| k \| c))$;
```

Algorithm 6: push

```
local state: St – keystore node state (Table 11.1)
input       : $\pi$ – key position to push
            : $R$ – list of pull messages
output      : $h$ – hash of new key value
begin
    $P \leftarrow$ St.replicators($\pi$) $\cup$ St.assigner($\pi$);
    $k \leftarrow \mathbf{0}$;
    $S \leftarrow [\,]$;
    for $(\pi', r', i', s', k', c', \sigma') \in R$ do
        assert $r' \in P$;
        $P \leftarrow P \setminus \{r'\}$;
        assert $c'$ = St.update_commit($\pi$);
        if $r' = \mathit{St.assigner}(\pi)$ then
            assert St.update_commit($\pi$) = $H(k')$;
        assert $\pi' = \pi$;
        assert St.update_number($\pi$) = $i'$;
        assert verify(St.PK($r'$), $\pi' \| r' \| i' \| s' \| k' \| c'$, $\sigma'$);
        $k \leftarrow k \oplus k'$;
        $S$.append($s'$);
    assert $P = \emptyset$;
    ($s_{\text{new}}$, replace) $\leftarrow$ vote-state($S$);
    if *replace* = ***true*** then
        St.value($\pi$) $\leftarrow k$;
        St.last_update($\pi$) $\leftarrow$ now;
    $h \leftarrow$ hash($i \| \pi \|$St.value($\pi$));
    return *h;*
```

```
Algorithm 7: vote-state
local state: $q_a$ – assign quorum
           : $q_d$ – discard quorum
input      : $\{s_1, \ldots, s_n\}$ – replication set's states
           : $s_{\text{me}}$ – my state
output     : $s_{\text{new}}$ – new state of position
           : replace – whether the key value must be replaced
begin
    $n_a \leftarrow |\{i \in [1,n] : s_i = \mathbf{A}\}|$;
    $n_d \leftarrow |\{i \in [1,n] : s_i = \mathbf{D}\}|$;
    if $n_d \geq q_d$ then
        return $(\boldsymbol{D}, \mathbf{true})$;
    if $n_a \geq q_a$ then
        return $(s_{me} = \boldsymbol{D}\ ?\ \boldsymbol{D} : \boldsymbol{A}, \mathbf{false})$;
    return $(\mathbf{D}, \mathbf{true})$;
```

before			after			new
N1	**N2**	**N3**	**N1**	**N2**	**N3**	val
U	**U**	**U**	**D**	**D**	**D**	true
U	**U**	**D**	**D**	**D**	**D**	true
U	**U**	**A**	**A**	**A**	**A**	false
U	**A**	**A**	**A**	**A**	**A**	false
A	**A**	**A**	**A**	**A**	**A**	false
U	**A**	**D**	**A**	**A**	**D**	false
A	**A**	**D**	**A**	**A**	**D**	false
U	**D**	**D**	**D**	**D**	**D**	true
A	**D**	**D**	**D**	**D**	**D**	true
D	**D**	**D**	**D**	**D**	**D**	true

Table 11.2: Keystore nodes before and after synchronizing state with three nodes, an assign quorum of one and a discard quorum of two. The *new val* column indicates whether the key value is replaced during update. The table is complete up to permutations.

The use of multiple random contributions from the nodes protects against a case where an assigner has a broken random number generator that produces predictable numbers. While making the assigners commit to their random key values is not strictly necessary in our security model, it ensures that the assigner—or indeed any single entity—cannot dictate what are the ultimate values for new keys. In Section 11.6.3 we discuss a scenario where this is useful.

Algorithm 7 presents the vote-state algorithm used to determine the new key state and value. It takes a set of states from among the replication set and its own local state. It returns the new state and whether it should be replaced with a new value. Further details are presented in the next section on Byzantine robustness. In principle, the algorithm works by requiring a quorum of nodes to agree on assigns and discards. Each node votes on whether a position is **D**; if the vote passes—that is, the *discard quorum* is met or exceeded—then the position is discarded and replaced. If not, then a vote occurs on whether the position is **A**; if the vote passes—that is, the *assign quorum* is met or exceeded—then the position is **A** and the value is retained. Otherwise, then position is **D** and replaced; this ensures the secure deletion of unused key positions to bound the existential latency.

Table 11.2 illustrates the results of vote state for a system of three keystore nodes, an assign quorum of one, and a discard quorum of two. The table gives all possible input states (unique up to permutations) and provides the state after synchronization as well as whether each node gives a new value to the position.

Figure 11.2 shows an example state timeline for a KP stored by an assigner and two replicators. The nodes in the replication set periodically synchronize. The thick lines for the assigner and replicators represent a concurrent example history for the KP's state. The final sequence consists of the KP being assigned and deleted by the assigner before synchronizing.

11.5 Byzantine Robustness

Our distributed keystore provides properties **P1**–**4** under perfect conditions. In this section we make it robust against Byzantine failures. A Byzantine failure in a distributed system is a failure where a node behaves in an arbitrary way [126]. Table 11.3 lists the Byzantine failures during the synchronization protocol. The first part of the table

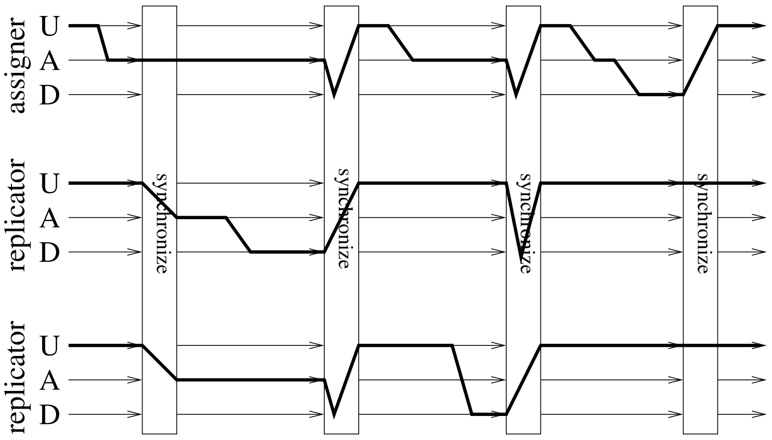


Figure 11.2: Example position state timeline for a three-node replication set. The position is assigned then discarded, thrice. This example uses an assign and discard quorum of one.

lists all possible false state reports from peers by enumerating them. For example, F1 is when a KP's state is **U** but a node reports that it is **A**. The effect column shows the effect of this failure, and the remedy column shows how it is avoided. For clarity, we call failures where a node invents a state change that did not occur *fraudulent failures* and failures where a node does not report a state change that did occur *negligent failures*. The second part of the table lists non-state-related Byzantine failures found by inspecting the algorithms.

A node may also have a Byzantine failure when serving a client's requests, however most of these are equivalent to a particular failure during synchronization. For example, during assign, a node may return incorrect data to the client—this is equivalent to F9: the node gives a false random seed to peers during synchronization and thereby updates the KV to a different value. Table 11.4 presents the different Byzantine failures that can occur for client operations by inspecting Algorithms 1–3. It also shows the synchronization failure to which they are equivalent, and the remedy that makes the keystore robust against it.

Peer Eviction. F7 is the failure where nodes delay or disrupt synchronization, such as by being offline. An assigner may delay initiating synchronization or completing the push phase; replicators may delay

	state change			
ID	**true**	**claim**	**effect**	**remedy**
F1	**U**	**A**	needless key storage	load balance assigner pages
F2	**U**	**D**	no negative effect	discard quorum
F3	**A**	**U**	lost assigned key	over-report assigns, check value
F4	**A**	**D**	lost assigned key	discard quorum
F5	**D**	**U**	no negative effect	discard quorum
F6	**D**	**A**	keep discarded key	over-report deletions

ID	**Byzantine failure**	**effect**	**remedy**
F7	delay synchronization	increase latencies	peer eviction
F8	send bad signatures	increase latencies	peer eviction
F9	false random seed	assigned keys differ	over-report assigns
F10	send false pull msg	fake consensus	peer signature
F11	replay pull msg	fake consensus	sign over update number & position
F12	(i) missing pull msg (ii) duplicated pull msg	fake consensus	check pull messages against replication set

Table 11.3: Byzantine failures in keystore node synchronization.

responding to pull requests. Both delays can effect a longer deletion latency, however existential latency is unaffected because the definition of an assignable KP excludes ones that were not recently synchronized.

Delayed synchronization is remedied with replication set eviction. After a configurable amount of time passes without synchronization, unresponsive nodes are evicted from the replication set. The assigner then replicates the KP with new replicators at the next opportunity. Discarded KPs known locally to the evicted replicator are lost, but this is equivalent to F6. The offline node may still retain deleted values, however, but we discuss how to resolve this later. Unresponsive assign-

operation	**byzantine failure**	**equiv.**	**remedy**
read	deny service	-	multiple nodes
	return garbage	F9	client check
assign	deny service	-	multiple nodes
	unfresh value	-	discussed in Section 11.6.3
	multiple assign	-	discussed in Section 11.6.3
	do not update state	F3	over-report assigns
	return garbage	F9	over-report assigns
discard	deny service	-	multiple nodes
	do not update state	F6	over-report discards
	allow fake token	F4	deletion quorum

Table 11.4: Byzantine failures in keystore node client interface.

ers can also be evicted. This results in assignments being unreported (equivalent to F3) and discards being unreported (equivalent to F6).

Over-Reporting Thresholds. Clients over report their assigned and discarded KVs to multiple nodes in the replication set to protect against negligent failures. Configurable *over-reporting thresholds* r_a and r_d are the number of additional nodes to which each assigned or discarded KP is reported, respectively. When over reporting assigns, the access token further serves as the proof that the assigned key value is in fact the same one stored by the peer, thus simultaneously protecting against providing false data (F9).

Assign and Discard Quorums. We are robust against fraudulent failures by setting a configurable *assign quorum* q_a and *discard quorum* q_d. The assign quorum is the number of nodes that must agree that a KP was assigned to change its state during synchronization; the discard quorum is the number of nodes that must agree that a KP was discarded to securely delete it during synchronization.

Observe that Algorithm 7 checks the assign and discard counts against the relevant quorums during synchronization. We emphasize that $q_a > 1$ implies that assigned KVs can be lost during synchronization and so we use $q_a = 1$ as a safe default. If fraudulent assigns are a concern, however, then it is necessary that the client only uses the KV after confirming the assignment through over reporting. Discard quorums are important to protect against data loss due to a fraudulent failure. Assign quorums, however, only protect against wasted storage resources for KVs that were not actually assigned.

Summary. Quorums and over reporting place the onus on the client to report assigns and discards to sufficiently many nodes to meet the threshold; clients report assigns to peers to ensure that the value is correct and report discards to ensure that the state change occurs.

An assign and discard quorum of q_a and q_d respectively means that the system can handle at most $q_a - 1$ fraudulent assigns and $q_d - 1$ fraudulent discards. An over-reporting threshold of r_a (for assigns) and r_d (for discards) means that the system can handle at most r_a and r_d negligence failures for assigns and discards respectively. Clients are required to communicate with $q_a + r_a$ nodes per assign and $q_d + r_d$ nodes per discard.

11.6 Keystore Secure Deletion

In the last section we saw how to implement a distributed securely-deleting keystore. We now present how to use such a keystore to achieve secure deletion. We discuss policies on key values, encryption key construction, and present an example encoding scheme that balances secure deletability and availability of data. Finally, we analyze the system's security.

11.6.1 Key Pools and Encryption Keys

Clients use the keystore to obtain securely-deletable KVs that are used to build encryption keys. Different properties may be associated with different KVs. For example, KVs can have different deletion latencies, different expected reliabilities, and be stored and securely deleted with different procedures. We encapsulate all policy-based aspects by which KVs may be discriminated by associating each one with a *key pool*. When a client requests a key, it does so by specifying the corresponding key pool from which the key should be taken.

We say a keystore node *serves* a key pool if it stores KVs for that pool. A node can serve multiple pools, but to serve a pool it must fulfill the pool's policy requirements. This therefore requires oversight in the initial deployment of keystore hardware, e.g., where they are located and how they store and securely delete keys. Keystore node certificates also bind their service to key pools. Communication between two nodes is only necessary if the nodes serve the same pool. Figure 11.3 illustrates a system of six nodes serving 24 KVs from four pools.

An important notion for our system is that of *complementary key pools*: two key pools, L and L', are *complementary* if no keystore node serves both L and L'. This provides clients with the technical means of selecting two key values with the knowledge that no correct keystore node stores both these items. Building an encryption key from the logical XOR of these values therefore ensures that the compromise or failure of any single machine does not affect the secure-deletability of the encryption key.

A *key recipe* or *recipe* is a sequence of ATs. On the content store, the key recipe is stored alongside the data encrypted with the resulting key. Clients are free to select how their encryption keys are composed, which they do by selecting the number of KPs and the pools from which to assign for each position. A sequence of pools that scaffolds the contents of a key recipe is called a *key class* or *class*.

The three client-side keystore operations—assign, read, and discard—extend naturally to classes and recipes. Clients assign a recipe from a class, read a key from a recipe, and discard a recipe. Figures 11.4 (a), (b), and (c) show the data path for the client write, read, and discard operations respectively. When writing data, two KVs are assigned from keystore nodes, and the data is encrypted with the logical XOR of these values. These values' ATs are then encrypted with a password-derived key [99] and stored alongside the encrypted data. The password's strength is important for ensuring that the key recipes cannot be read while valid, however, the password's strength is irrelevant once the KVs are securely deleted.

When reading data, the key recipe provides the ATs required to obtain the KVs used to build the encryption key. When discarding data, only the recipe is read and it is used to determine which keys to

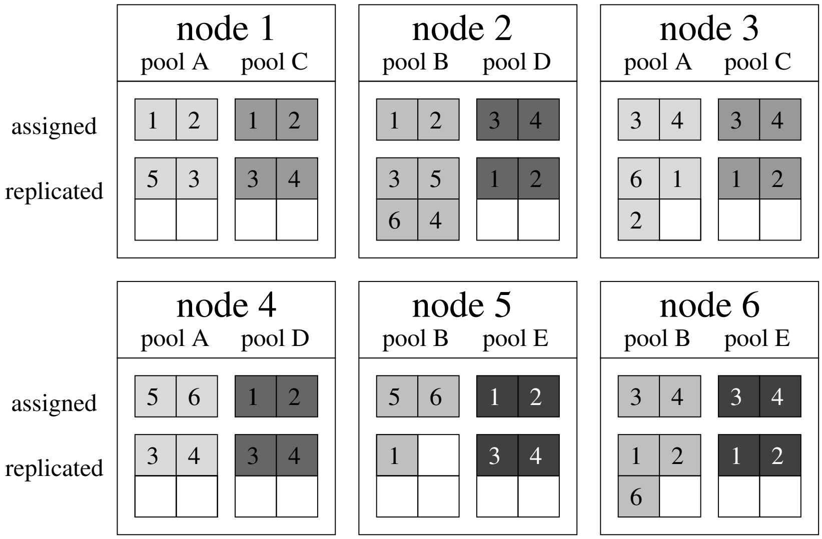
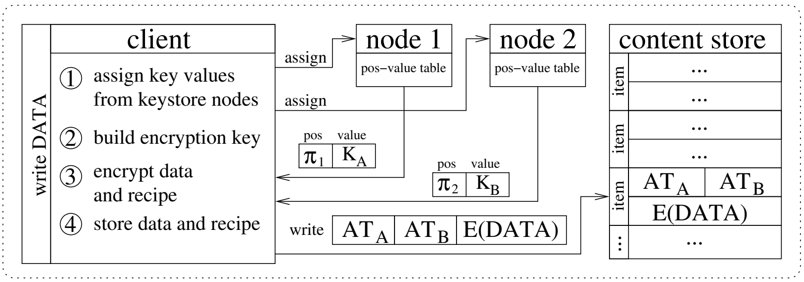
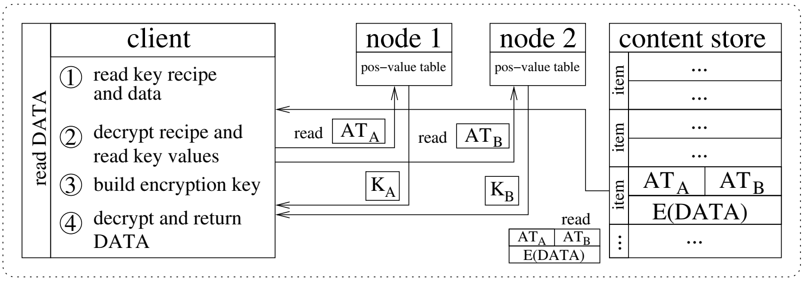


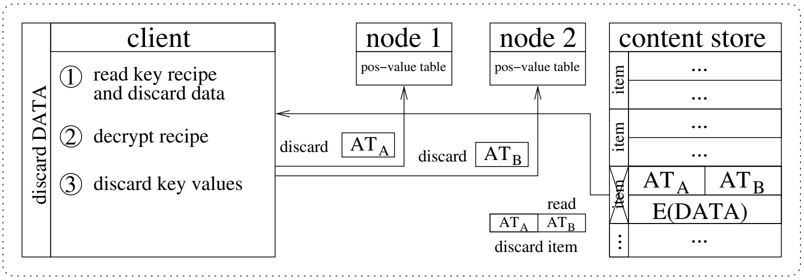
Figure 11.3: Example pool to node assignment. Different shades indicate different pools, and the number corresponds to the pool-local key values. Each pool requires a minimum of one replication. No node serves both A and B; no node serves any two of C, D, and E. All nodes have two assignable values per pool and a number of replicated values.



(a) write



(b) read



(c) discard

Figure 11.4: Datapath for client operations using a distributed keystore. This example assumes an assign and discard quorum of 1 and an over-reporting threshold of 0. (a) The write operation. (b) The read operation. (c) The discard operation. We use the following abbreviations: $\text{AT}_A = E_{K_{\text{PK}}}(\pi_1 \| H(K_A))$ (access token for pool A), $\text{AT}_B = E_{K_{\text{PK}}}(\pi_2 \| H(K_B))$ (access token for pool B), and $\text{E(DATA)} = E_{K_A \oplus K_B}(\text{DATA})$ (encrypted data).

discard. Figure 11.4 is simplified in that the client does not over report assigns and deletions and only communicates with the KV's assigner.

Many different encoding can be used to create cryptographic keys out of the KVs. Encodings that require algebraic structure, such as polynomial-interpolation encoding [107], cannot be implemented using only client assigns: either all nodes involved must know the resulting encryption key to correctly generate shares or the client must generate additional shares and store them after the assignment with further nodes. The first case is undesirable because we require that the client is the sole entity aware of actual encryption keys. The second case is acceptable provided that the client can wait until the shares are generated and stored reliably before using the encryption key. In the next section we describe a simple XOR-based (n-out-of-n) encoding scheme that does not require client-side share generation.

11.6.2 XOR-Based Encoding

The XOR-based encoding for encryption key generation consists of a client obtaining a set of KVs and deriving an encryption key from their logical XOR. We can imagine this encoding as consisting of two parameters: the number of operands XORed together and the number of times each operand is replicated. Both these parameters affect the availability and secure deletability of data. Figure 11.5 (a) illustrates these parameters as a two-dimensional matrix: each row corresponds to an operand and each column corresponds to a replication of that operand on a different keystore node. The key class represented by the figure has four complementary key pools, each of which is replicated three times. Note that replication across columns is only one possible scheme; in general any erasure coding can be used to replicate KVs.

For the key to be available, at least one replicator in each KV's replication set must correctly store and return the KV (Figure 11.5 (b): one cell per row). For the encryption key to be securely deleted, at least one entire replication set must securely delete their KV (Figure 11.5 (c): one full row). As the size of the key class increases, secure deletability increases while availability decreases. Similarly, increasing the size of the replication set increases availability and decreases secure deletability. The client selects the key class—both the number of entries and the pools for each entry—permitting the client to select a desirable tradeoff.

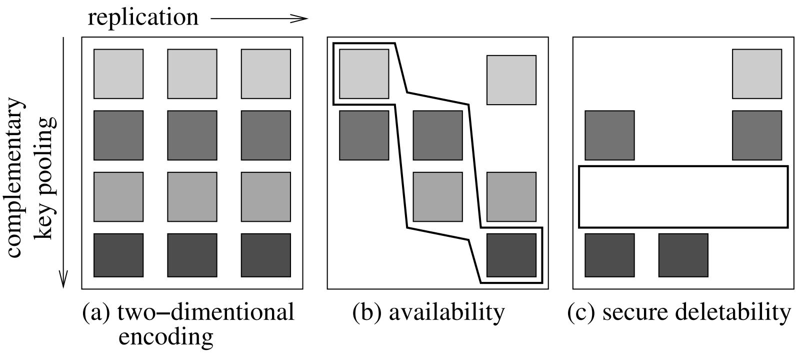


Figure 11.5: Two-dimensional value encoding. Each row represents a different KV and each square in a row represents a copy of that KV. (a) Perfect replication for each KV. (b) Some KVs are missing but a KV from each row is still available and therefore so is the resulting encryption key. (c) After secure deletion, some KVs remain but an entire row of one is gone and therefore the encryption key is securely deleted.

Given observed or predicted failure rates and assuming independent failures among keystore nodes, computing the effective availability and secure-deletability is straightforward. Let A be the mean time between failures (MTBF) for availability, that is, the expected time between node failures that results in the loss of stored data [127]. Let S be the MTBF for secure deletability, that is, the expected time before a node cannot securely delete data. This can occur because the drive is lost, becomes read-only, breaks down and is sent for repair, develops bad blocks that cannot be deleted, etc. Note that A is often provided by the hardware's data sheets, although research has shown that the estimations can be significantly inaccurate [128, 129].

An XOR-based encoding with x operands and c copies of each operand yields an effective secure-deletion MTBF of $\frac{S^x}{c^x}$ and an availability MTBF of $\frac{A^c}{x}$. A client with target MTBFs for availability and secure-deletability can select the smallest satisfactory c and x.

11.6.3 Security Analysis

We now analyse our system's security. We begin by showing how it provides secure deletion against our adversary by inspection of the adversary's capabilities. We then present other security considerations.

In this section, we use the term *data lifetime* to refer to the data's lifetime expanded in both directions by the upper bound on existential and deletion latency.

Secure data deletion requires that an adversary that performs coercive attacks at all times outside the data's lifetime and non-coercive attacks during its lifetime is unable to recover the data. By encrypting stored data, this corresponds to the adversary being unable to recover the KVs that comprise the data item's encryption key.

The keystore properties ensure that the adversary's coercive attacks do not reveal non-valid data. **P4** ensures that coercive attacks outside the lifetime of data do not recover it, while **P1** ensures that the KVs are uniquely assigned and unguessable. Now we show how we provide security despite the adversary's four non-coercive attacks, which can be performed at any time. Section 11.2 in this Chapter describes the these attacks in detail.

- **A1** (accesses content store). This provides no information because all data written to the content store is encrypted. ATs can only be decrypted by performing a coercive attack, but non-valid data is securely deleted by the keystore.

- **A2** (acts as a keystore client). The adversary cannot guess access tokens and so is unable to read or discard KVs for which it was not assigned. The adversary is able to repeatedly call `assign` to obtain KVs. We discuss this denial of service later in this section.

- **A3** (operates a keystore node). The adversary can obtain a fraction of the KVs in the system. We use complementary key pooling to ensure that no single node is capable of deriving the key using local information. Certificates that bind keystore nodes to the pools they serve also prevent nodes from joining replicating sets for KVs from complementary pools and from performing Sybil attacks.

- **A4** (mounts network attacks). All communication is done over TLS-secured forward-secure channels and so eavesdropping or modification is not possible. An adversary may mount a denial-of-service attack, the implications and mitigations of which we discuss later in this section.

Denial of Service Attacks. Denial-of-service (DoS) attacks are an open problem in computer security that we do not solve. Our adversarial model assumes that a long-term DoS of all keystore nodes is not possible. There are precautions, however, that can be taken to mitigate the damage done by DoS attacks.

The first DoS attack occurs when the adversary uses **A2** to assign all possible KVs and thus consume all keystore nodes' free storage capacity. If clients are authenticated and known, then simple accounting can be used to tally the difference between assigned and discarded keys; either charging the client for the storage or placing limits on the number of assignable positions. When keystore nodes operate as a public service then a micropayment mechanism may be used when assigning KVs.

The second DoS attack occurs when the adversary, using **A4**, interferes with network communications. While this inconveniences users' storing and retrieving of data, it also has implications for secure deletion. In particular, it increases the deletion latency beyond the key pool's promised upper bound in two ways: (i) by preventing the replication set from synchronization; (ii) by preventing the client from being able to contact keystore nodes to discard KPs. Note that neither effects existential latency: the assignable set of KPs excludes ones that are not recently synchronized. To mitigate (i), keystore nodes that observe a long delay in synchronization may securely delete all discarded KVs without updating the rest of the page. If the client informed the entire replication set before the DoS began then this achieves the secure deletion of this data. Mitigating (ii) is more troublesome; the distribution of keystore nodes should make it difficult for the adversary to inhibit all communication.

Session Keys and Entropy Pools. Bounded existential and deletion latencies require that compromising the keystore nodes' states does not reveal KVs outside their valid lifetime. When refining the design to an implementation, it is important to securely delete any data that can violate this property. Perfect forward secrecy is often implemented with periodic key renegotiation; random numbers are typically generated by drawing from a hardware entropy pool. An adversary with knowledge of the entropy pool may be able to predict session keys; an adversary with knowledge of a session key can determine the cryptographic random number generator used to generate new KVs. To ensure that

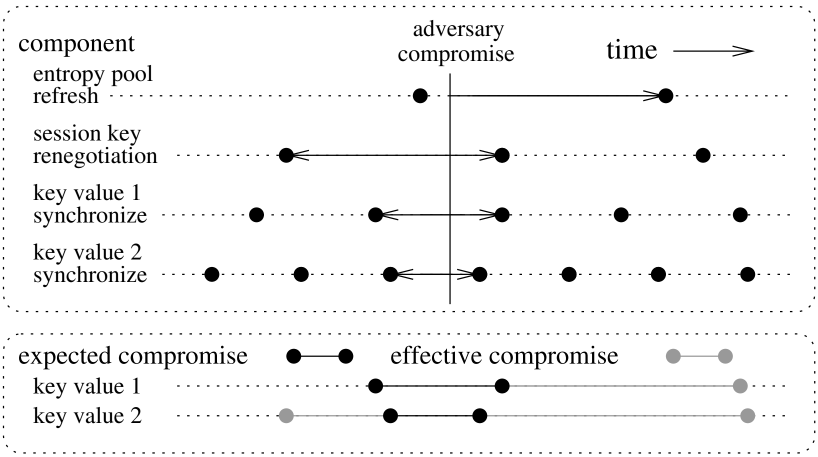


Figure 11.6: Effect of entropy pool and session key disclosure on the compromisability of KVs. The expected compromise is indicated in black while the effective compromise due to session keys and entropy pools is indicated in grey.

latency bounds are meaningful, session keys and the entropy pool's contents must be periodically refreshed and securely deleted after use.

Figure 11.6 shows how the adversary's knowledge of the entropy pool and session keys effects the compromise of KVs. While our design expects that the adversary's single attack only obtains KVs valid at that time, knowledge of the entropy pool allows future session keys to be predicted and therefore effect a compromise of future KVs. Knowledge of the current session key allows the adversary to decrypt previously-collected network traffic and determine earlier KVs. We must renegotiate session keys and clear the entropy pool with the same timeliness as the shortest key pool synchronization.

Key Value Freshness. The client can confirm KV freshness when over reporting an assign because the KV is fresh if any replicator believes that the position is unused. This is the case unless the assigner led a synchronization before the client is able to report. Further protection can be added by having each node maintain a set of recently-assigned positions.

Multiple Assigns. We assume that assigners do not intentionally multiply assign positions, and unintentional failures due to lost state are avoided by first resynchronizing before assigning. We nevertheless consider how to avoid malicious assigners.

KV freshness ensures that a malicious assigner cannot doubly assign KVs that were assigned in previous epochs. Assigners can, however, assign an unused position to multiple clients before synchronization. Client IDs for assigned positions can be compared during synchronization, however this increases the communication cost between nodes.

Covert Disclosure. A possible attack is a keystore node disclosing future KVs to an adversary well in advance of their use, thus avoiding suspicious behaviour at the time the KV is assigned. Observe that this is equivalent to effecting a large existential latency for data.

A node can increase existential latency in two ways: (i) by fraudulently assigning all stored KVs among the nodes in advance, reporting the KVs to the adversary, and then only later actually assigning them to clients; (ii) by generating new KVs in a predictable way, e.g., with predictable random seeds. In the first case, value freshness defends against this attack: clients do not use KVs whose freshness check fails. In the second case, assigner commitments and replicator contributions ensure that no entity in the system controls the ultimate random values used to generate KVs. Of course, this still does not prevent a hostile node from simply allowing an active adversary to read any KV without the access token.

11.7 Implementation Details

We have implemented our design as an open-source project in C++11. The project has two main components: the server-side keystore node and the client-side keystore file system (KSFS). We now discuss their implementations and the reasoning behind our design decisions.

Key Pages. KVs are aggregated into variable-sized key pages. Pages are the atomic unit of synchronization, replication, assigner ownership, key pool membership, and indexing. Thus, a position replicated by a set of nodes implies that its containing page is replicated by the same set of nodes. Among other things, pages allow the storage of KVs to scale: they reduce the memory footprint required for organizing the KVs. Key pools therefore consist of a set of key pages.

Key Page State and Synchronization. Each page stores two bits for its position's state. The first bit (the state bit) for each position is **true** if the position's state is **A** and **false** if it is not **A** (i.e., either **U** or **D**). The second bit (the delta bit) is **true** if and only if the state bit is different from the last synchronized consensus.

Synchronization is performed for an entire page. Instead of providing the full state, nodes provide their LZO-compressed [130] delta bit vector. The KPs for KVs that are both assigned and discarded during the previous synchronization are specifically listed after the delta vector for efficiency. The assigner also provides the state vector as it was after the previous synchronization, which the replicators check against their own. Each node in the replication set then updates the KP's state according to the `vote-state` algorithm. Assigned positions whose local discard was rejected by consensus retain the KV but keep the state discarded and the delta bit set.

The page's **U** and **D** KVs are replaced with new values taken from a cryptographic random number generator [131]. This generator is seeded by the bitwise XOR of all the random value contributions from each node in the replication set. Note that this refinement replaces the i.i.d. distribution with a pseudo-random distribution seeded by randomness.

Key Page Cache. The key page cache manages access to pages, keeping some keys in memory and loading the others from long-term storage when needed. Any use of a page requires that it is available in the page cache. Each pool has its own page cache. The policy defines the minimum size of the memory used to cache pages, thereby improving the expected performance.

Depending on how pages are stored, loading them from long-term storage may require complicated operations, including determining the page's key and decrypting its contents. All page metadata, including each KP's state, is loaded alongside. The caching strategy keeps assignable pages ready in the cache and evicts the replicated pages based on a least-recently-used metric, which has low overhead and high performance. Pages with local modifications are first written to long-term storage before eviction.

Key Page Locking. For thread-safety, key pages are locked when performing mutable operations, e.g., assigning and deleting keys. Once a page is locked, the time required to handle an assign or delete request is extremely brief. Originally, when assigning a key, we polled for an unlocked page if one was available instead of waiting on a locked page; testing, however, revealed that at high load it is more efficient (and at low load it made no difference) simply to pick a random assigner page and wait for the lock.

While KV operations (i.e., assign, read, and discard) take constant time, a few key page operations take time proportional to the page size. These are marshalling a page for peer replication, preparing the pull message, and updating the page with the new version. The synchronization protocol does not affect performance for KV operations: from the time a page's pull message is generated until the time the page is finished synchronizing, the page is replaced with an update wrapper. This update wrapper caches all discard requests itself, rejects any assign requests (nodes maintain multiple assigner pages per pool to account for locked pages), and handles read requests without locking the page. After updating the key page, the update wrapper's modifications (i.e., deletions) are replayed and the wrapper removed. Therefore, the only operation that may affect the latency of KV operations is marshaling a key page for replication. Since this operation primarily happens before a page is ready for use and only occasionally when a replicator leaves the network, we conclude that page lock time is not a concern.

Key Builder. The *key builder* is a client-side service that processes key classes and key recipes, performs the appropriate calls to keystore nodes, and returns the actual encryption keys. The key builder's functions are the following: `read_key`, which turns a key recipe into an encryption key by reading the referenced KVs and assembling the key; `assign_key`, which takes a key class and returns both an encryption key and its corresponding key recipe by getting new KVs for each key pool in the class; `delete_key`, which takes a key recipe and discards each constituent key position. Each of these functions executes in parallel with respect to keystore operations: reading a key recipe of three KVs, for example, spawns three simultaneous read requests. When KVs are replicated over multiple keystore nodes, the key builder selects among them until the request is satisfied.

Keystore File System. The keystore file system (KSFS) is a file system that provides secure deletion for its data using the solution proposed in this work. KSFS is a FUSE-based file system that stacks on top of an existing file system, e.g., a file system that proxies access to the content store.

KSFS encrypts each data block with its own unique key. The recipe is stored along with the encrypted block in the file or as a separate related file. Moreover, each file is optionally given its own master key. This key is assigned from the nodes in the same manner as other keys, though it may use a different key class. The master key is used to decrypt the block keys used by the file, therefore, the block key retrieved through the recipe is interpreted as an encrypted key. When the file master key is securely deleted, then all the content in the file is also securely deleted even if some per-block keys remain. This provides another defense for compromised keys as well as greater efficiency in securely deleting an entire file.

Only a few features were added to the FUSE file system to implement our design. The new features are the following: initialize a key builder service, store and retrieve key recipes alongside encrypted data, and perform the relevant cryptographic operations in the I/O datapath. The KSFS is implemented by making the following changes to the FUSE file system:

- `create`: obtain a file master key and store its file recipe at the beginning of the file; allocate auxiliary state.
- `open`: read the file master key and build its key, allocate auxiliary state for file.
- `truncate`: delete the recipes for all truncated blocks; re-encrypt the file's tail if the truncation is unaligned with the file system block size.
- `unlink`: delete all recipes for the file including the master key.
- `flush`: flush file auxiliary state if necessary.
- `read`: read a part of the file as well as the key recipes for that part; build the corresponding keys and decrypt the data.
- `write`: obtain new keys from the key builder, encrypt the data and write the encrypted data and key recipes to the file.

- `statfs/getattr`: correct file sizes to account for recipes.

In reality, the read and write functions are more complicated. Since the file system's blocksize is the granularity of encryption, writing data that replaces an existing part of a file—but not a complete block—requires reading and decrypting that part of the file, making the change, re-encrypting the block with a new key, and updating the full block and key recipe. Consequently, the encryption blocksize affects system performance because all operations must occur with that block size as the minimum unit of I/O.

To improve performance for repeated small reads or writes within a block, a one-item cache is used to store the current working data block. When a read or write requires a particular block, the cache is first checked for its presence. If absent, the current block is written to the content store (if modified in cache) and replaced with the desired block. All reads and writes are performed only on the cached entry.

Keystore Communication. We use RPC calls for all keystore communication. Each keystore node is assigned an IP address and port to run an RPC server. Our implementation uses `xmlrpc` [132] for the RPC server and `stunnel` [133] to expose it as an `https` [134] service using OpenSSL [135]. Consequently, the IP address of the RPC server is only accessible locally and used for forwarding traffic from the publicly-available TLS-secured RPC server. Additionally, the IP, port, and public key of some initial *bootstrapper* nodes' RPC servers are given to each new peer. On initialization, the node queries one of the bootstrapper nodes to request known peers. The peers that are provided are further queried until the relevant network view is known. When querying for peers, the pools served by the node are also provided. The node ignores any peers for which it does not share a pool. The KSFS also uses RPC to communicate with the nodes to request keys, etc. The KSFS is also bootstrapped in the same way. Since the KSFS defines the classes it uses, it can also determine the set of pools of interest.

11.8 Experimental Validation

We validate our design by experimenting with our implementation. We focus on the inter-keystore-node communication cost during synchronization and the keystore's latency and throughput for satisfying clients' requests.

Inter-node Communication. We measure the amount of communication in our system by summing the size of outgoing inter-node RPC calls. Communication mostly occurs when nodes synchronize pages. Our goal is to verify that our design decisions to compress synchronization messages and to seed random number generators for creating key values result in low communication overhead.

We measure the average inter-node communication in a distributed system of 6 or 12 computers running keystore nodes. We use a baseline configuration that consists of 6 nodes each assigning from 1000 16-KiB pages replicated among 3 nodes and synchronized every 10 minutes. The other configurations are generated by taking the baseline and changing one parameter. We run the system for 90 minutes and measure the average inter-node communication over this time with and without data compression. Note that compression reflects the best case as no state changes occur; without compression reflects the worst case where so many state changes occur that compression does not help. We further compute the number of unique key positions provided by this configuration. Table 11.5 presents our results. Note that the communication is measured as the *per node outgoing* communication: the total system communication is scaled by the number of nodes.

Table 11.5 fails to refute our hypothesis on the utility of aggregating KVs into key pages: given a fixed client load, pages can be synchronized independent of page size. Note that while our tests had all pages synchronized, in practice keystore nodes only need to synchronize pages with local changes along with sufficient assigner pages to handle the expected demand in the next interval; unchanged pages can be ignored, however a mechanism must exist for replicators to signal to the assigner to initiate synchronization if a local change (i.e., deletion) occurs.

The size of the replication set affects the communication complexity because messages are passed among more peers. Communication scales quadratically to the replication set size.

Configuration	Key Values (millions)	Communication (B/s) Compressed	Uncompressed
baseline	6.1	4380 ± 24	7285 ± 3
2000 pages	12.3	8597 ± 24	14281 ± 11
5 minute sync.	6.1	8704 ± 22	14460 ± 7
64 KiB pages	24.6	4221 ± 14	13089 ± 6
1 replicator	6.1	1856 ± 23	3053 ± 26
5 replicators	6.1	8738 ± 22	14513 ± 3
12 nodes	12.3	4638 ± 23	7724 ± 23

Table 11.5: Average inter-node communication with 95% confidence. Each row has different parameter configurations. The `baseline` configuration has the following values: 6 nodes each assigning from 1000 16-KiB pages replicated by 2 nodes and synchronized every 10 minutes. The remaining measurements vary from the baseline by the identified parameter.

op.	Latency (μs) internal	external	Key building latency (μs) 2 parts	4 parts	6 parts
assign	24 ± 1.0	726 ± 7	1034 ± 21	1115 ± 36	1414 ± 39
read	16 ± 0.7	725 ± 8	1102 ± 27	1181 ± 10	1416 ± 30
delete	28 ± 1.0	719 ± 9	1070 ± 34	1198 ± 11	1423 ± 35

Table 11.6: Keystore operation 95th percentile latency with 95% confidence. Internal latency is the operation's processing time within the node. External latency is the client's observed processing time, including local network latency. The key building latency is the time required for the `client-assign`, `client-read`, and `client-delete` algorithms to execute, for different recipe sizes.

Increasing the number of nodes does not affect communication complexity. The additional overhead in discovering peers and negotiating secure channels is therefore significantly less than the cost of performing synchronization. The communication complexity scales with the number of pages being synchronized and inversely with the period between synchronizations.

Latency and Throughput. Table 11.6 presents latency for basic key operations. Internal latency is the time a node requires to process a request after parsing the RPC message; external latency is the peer's observed latency for request processing at the moment the request is issued, including HTTPS tunneling and local network latency. Key building latency is the time taken to assign, read, or delete an entire key; we measure this using a recipe size of two, four, and six. Measurements are computed by averaging six 95th-percentile observations; the true mean is within the interval 19 times of out 20.

Latency measurements are stable across the parameters and workloads we tested: all values from 4 KiB to 256 KiB exhibit the same latencies. The internal latency of read operations is smaller because both assign and delete events are logged for crash recovery and the journal synchronized before returning to the client. If the synchronization is disabled then the three operations have approximately equal latency.

The RPC library, the HTTPS server, and the network add significant overhead to the latency of requests, from a couple dozen microseconds to a millisecond. The use of HTTP-based transport is not optimal for binary data because it must be encoded into base64. Replacing the RPC library should therefore yield better performance. The ping between machines is 300 microseconds and the loopback device (for communications between `stunnel` and the HTTP server) has a delay of 10 microseconds.

We see that the latency for recipes operations scales very well with the recipe size, even at the 95th percentile. Provided that the recipe components are on physically-separate machines, the per-position operations can be dispatched in parallel and the overhead involved in assembling the result and waiting for the slowest response is negligible.

To measure the throughput of the keystore, we created a client that issues random requests with high frequency. In order to actually consume the entire capacity of the keystore, however, we used a network of two nodes serving a single pool and used the other ten computers to issue requests. With this setup, we achieved a throughput of 28 ± 0.1 kilo-operations per seconds, with each node handling approximately 14 kilo-operations per second. Despite the high load, the key operations' internal and external latencies during throughput measurements still fell within the mean's confidence interval. Latency at high load is therefore competitive with low load.

11.9 Conclusions

We developed a robust key storage solution for secure data deletion using both a persistent storage medium and a securely-deleting storage medium. The proposed secure deletion system can handle the partial failure of the securely-deleting storage, either by failing to delete data or failing to store data. Our design distributes the securely-deleting storage medium over many nodes and allows the client to select from key values with different storage and deletion policies. We consider an encoding scheme that balances both data secure deletability and data availability. We implement our design and analyze its performance, observing that it has a high service rate and can synchronize many securely-deletable key values with very low communication complexity. Further research on its performance with a deployed system is needed to validate its throughput and usability.

Part IV

Conclusions and Future Work

Chapter 12

Conclusion and Future Work

With this final chapter we conclude our thesis. First, we summarize our contributions and relate them with related work. Second, we outline unanswered questions and directions for future work. We finish by drawing our conclusions.

12.1 Summary of Contributions

In Chapter 2, we presented related work on secure deletion and compared their environmental assumptions and behavioural properties in Table 2.3. Related work for flash and cloud storage were presented in subsequent chapters as well as our own contributions: user-level deletion, DNEFS/UBIFSec, securely-deleting B-Trees, and distributed keystores. Our work was generally designed to defeat a strong adversary: the computationally-bounded unpredictable multiple-access coercive adversary. Our solutions are further designed to provide efficient secure deletion at a fine (per-data-item) granularity.

Table 12.1 relates all this work. Building on Table 2.3, it includes flash and cloud related work as well as our own.

Solution Name	**Target Adversary**	**Integration**	**Granularity**	**Scope**
overwrite [48, 49, 61]	unbounded coercive	user-level[a]	per-file	targeted
fill [50, 51, 70]	unbounded coercive	user-level	per-data-item	untargeted
NIST clear [10]	internal repurposing	varies	per-medium	untargeted
NIST purge [10]	external repurposing	varies	per-medium	untargeted
NIST destroy [10]	advanced forensic	physical	per-medium	untargeted
ATA secure erase [41]	external repurposing	controller	per-medium	untargeted
renaming [59]	unbounded coercive	kernel[a]	per-data-item	targeted
ext2 sec del [14]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3 basic [59]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3 comprehensive [59]	unbounded coercive	kernel[a]	per-data-item	targeted
purgefs [62]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3cow sec del [64]	bounded coercive	kernel[a]	per-data-item	untargeted
compaction	unbounded coercive	kernel	per-data-item	untargeted
batched compaction	unbounded coercive	kernel	per-data-item	untargeted
per-file encryption [90]	bounded coercive	kernel	per-file	targeted
scrubbing [39]	unbounded coercive	kernel[a]	per-data-item	untargeted
flash SAFE [6]	external repurposing	controller	per-medium	untargeted
purging Chap. 5	unbounded coercive	user-level	per-data-item	untargeted
ballooning Chap. 5	unbounded coercive	user-level	per-data-item	untargeted
hybrid Chap. 5	unbounded coercive	user-level	per-data-item	untargeted
DNEFS Chap. 6	bounded coercive	kernel	per-data-item	untargeted
revocable backup [4]	bounded coercive	composed[b]	per-file[c]	targeted
forget secret [105]	bounded coercive	composed[b]	per-data-item	targeted
ephemerizer [1]	bounded coercive	composed[b]	per-data-item[de]	targeted
ephem. time based [8]	bounded coercive	kernel	per-data-item[e]	targeted
ephem. on demand [8]	bounded coercive	kernel	per-file	targeted
ephem. classes [8]	bounded coercive	kernel	per-data-item[f]	targeted
porter devices [9]	bounded coercive	composed[b]	per-data-item[de]	targeted
vanish [11]	bounded coercive	composed[b]	per-data-item[d]	targeted
fade [2]	bounded coercive	composed[b]	per-data-item[f]	targeted
policy-based [106]	bounded coercive	composed[b]	per-data-item[f]	targeted
B-Tree Chap. 10	bounded coercive	composed[b]	per-data-item	targeted
keystore Chap. 11	bounded coercive	composed[b]	per-data-item	targeted

[a] Assumes interface performs in-place updates.
[b] Non-standard interface, assumes two storage media: one securely-deleting and one persistent.
[c] Works at the granularity of a backup, which is composed of arbitrary files.
[d] Data items are messages communicated between two peers.
[e] Data items' lifetimes are known in advance.
[f] Data items are grouped into classes and deleted simultaneously.

Solution Name	Lifetime	Latency	Efficiency
overwrite [48,49,61]	unchanged	immediate	number of overwrites
fill [50,51,70]	unchanged	immediate	inverse to medium size
NIST clear [10]	varies	immediate	varies with medium type
NIST purge [10]	varies	immediate	less efficient than clearing
NIST destroy [10]	destroyed	immediate	varies with medium type
ATA secure erase [41]	unchanged	immediate	inverse to medium size
renaming [59]	unchanged	immediate	truncations copy the file
ext2 sec del [14]	unchanged	immediate	batches to minimize seek
ext3 basic [59]	unchanged	immediate	batches to minimize seek
ext3 comprehensive [59]	unchanged	immediate	slower then ext3 basic
purgefs [62]	unchanged	immediate	number of overwrites
ext3cow sec del [64]	unchanged	immediate	deletes multiple versions
compaction	some wear	immediate	inefficient, lots of copying
batched compaction	some wear	periodic	no worse than compaction
per-file encryption [90]	some wear	immediate	one erasure at file deletion
scrubbing [39]	unchanged	immediate	varies with memory type
flash SAFE [6]	some wear	immediate	inverse to medium size
purging Chap. 5	some wear	immediate	depends on medium size
ballooning Chap. 5	variable wear	no guarantee	tradeoff with deletion
hybrid Chap. 5	variable wear	periodic	periodic duration trade-off
DNEFS Chap. 6	little wear	periodic	periodic duration trade-off
revocable backup [4]	varies[a]	periodic	re-encrypt all backup keys
forget secret [105]	varies[a]	immediate	re-encrypt tree path
ephemerizer [1]	varies[a]	periodic[b]	one key per expiration time
ephem. time based [8]	varies[a]	periodic[b]	one key per expiration time
ephem. file on demand [8]	varies[a]	periodic	re-encrypt all file keys
ephem. custom classes [8]	varies[a]	immediate[c]	must delete all data in a class
porter devices [9]	varies[a]	immediate	uses public key crypto
vanish [11]	varies[a]	no guarantee	requires DHT access
fade [2]	varies[a]	periodic[c]	only delete policy atoms
policy-based [106]	varies[a]	periodic[c]	only delete policy atoms
B-Tree Chap. 10	varies[a]	periodic	re-encrypt tree path
keystore Chap. 11	varies[a]	periodic	depends on key encoding

[a] Wear depends on how the securely-deleting storage medium is implemented.
[b] Deletion time is selected in advance from a finite set of possibilities.
[c] Deletion is based on logical expressions from a finite number of terms.

Table 12.1: Full spectrum of secure deletion solutions. This table expands on Table 2.3 to include related work for cloud storage, remote storage as well as the solutions proposed in this thesis.

12.2 Future Work

12.2.1 New Types of Storage Media

Storage technology advances the state of the art in many ways: capacity, reliability, performance, and price. Secure deletion, however, is not a design requirement and creative approaches to achieve it are usually needed after new hardware is introduced.

Though new storage media may fall into a category for which solutions are known—e.g., by permitting in-place updates, by having an write/erase granularity asymmetry, or by being persistent but augmented with a securely-deleting storage medium. For example, two kinds of technology currently on the horizon are shingled magnetic recording (SMR) and heat-assisted magnetic recording (HAMR).

SMR technology increases capacity by 25% by overlapping tracks (imagine roof shingles) and writes to one track can affect parallel tracks. Managing access is required either by an obfuscating hardware controller or by special drivers in the operating system. The unique geometry of this device may warrant new approaches for efficient secure deletion.

HAMR technology uses a tiny laser to heat the area of the magnetic storage being written to so that a very weak magnetic field is capable of being a magnetically-coercive force. HAMR storage promises to increase the storage density 100 fold, though consumer-level devices are nowhere in sight. Perhaps the high density and weak magnetic field may introduce analog remnants of deleted data.

12.2.2 Benchmarks for Different Storage

In our experiments for mobile storage, we collected usage data of a mobile file system from a limited sample size. This was acceptable to analyse DNEFS since it is generally independent of the file system usage: the KSA size depends only on the storage medium size and the worst case erasure count depends on the KSA size and the clock frequency. Similarly, we used the replay of our research group's revision control history to simulate a shared storage medium being used to perform periodic commits of local data.

It would be useful, however, to have more datasets describing the write and discard behaviour representative of a variety of types of mobile-phone users and cloud-storage users. Having real-world open data available to the scientific community that models cloud storage use cases, however, would provide more confidence in the performance measurements. Critically, such benchmarks must include discarding data: it is insufficient to record when a storage location is overwritten because the data item's corresponding discard time may have occurred long earlier.

12.2.3 Secure Deletion Data Structure Selection

Our analysis of secure deletion for persistent storage proved the security for all arborescent structure. Our B-Tree-based design was motivated by the utility of having a dynamically-sized data structures and a large blocksize when accessing data remotely; it is inspired by the ubiquitous use of B-Trees in the niche of database and file system storage. However, there is no benchmarked comparison to show that our design is better than a simple static tree or other possible arborescent structures.

It would be useful to determine what are the workloads and situations that are the most amendable to our design, and how other candidates compare. In additions to comparing different workloads, it would be interesting it compare the cost of adding a securely-deleting B-Tree layer on top of a file system versus integrating it into an existing (B-Tree) access structure.

12.2.4 Formalization

A notable aspect missing from this work is the formalization of secure data deletion and a mathematical model of storage media. The storage medium models and definitions of recoverable versus irrecoverable data is used as intuitive concepts and not formalized using a mathematical description of a storage medium and a definition of what can be recovered. Cryptography was assumed to be perfect: a computationally-bounded adversary cannot decrypt AES-encrypted messages without the key.

It would be useful to develop formal models of the concepts we present in this work and thereby formally prove the security of our designs. One may consider a storage medium's contents through history as a transition system trace. Secure deletion might mean that the set of possible walks—with the states during the data's lifetime redacted—is indistinguishable from the set of possible walks for a different value stored instead. Another possibility is a game-based definition where the adversary wins if it correctly chooses which among two storage media actually contained some data item. A proof of correctness for a secure deletion solution may involve proving its behaviour is indistinguishable from one of our idealized SECDEL models. In the process of formalizing and determining the best way of modelling storage media for secure deletion, useful new concepts or perspectives may be discovered.

12.2.5 DNEFS for FTLs

DNEFS efficiently solves the problem of secure data deletion for flash memory when accessed with a flash file system. However, flash file systems are less common than flash translation layers (FTL). This is because an FTL allows the storage medium to contain a traditional block-based file system such as FAT; these file systems are more widely-supported among operating systems. It would therefore be useful to integrate DNEFS into a security-enhanced FTL so that secure deletion can be provided for these users as well.

While FTLs vary in implementation, many of which are not publicly available, in principle DNEFS can be integrated with FTLs in the following way. All file-system data is encrypted before being written to flash, and decrypted whenever it is read. A key storage area is reserved on the flash memory to store keys, and key positions are assigned to data. The FTL's in-memory logical remapping of sectors to flash addresses stores alongside a reference to a key position. The FTL mechanism that rebuilds its logical sector to address mapping must also rebuild the corresponding key positions. A key position consists of a KSA erase block number and the offset inside the erase block. KSA erase blocks can be logically referenced for wear levelling by storing metadata in the final page of each KSA erase block. This page is written immediately after successfully writing the KSA block and stores the following information: the logical KSA number so that key position references remain valid after updating, and an deletion epoch number so that the most recent version of the KSA block is known.

Generating a correct key state map when mounting is tied to the internal logic of the FTL. Assuming that the map of logical to physical addresses along with the key positions is correctly created, then it is trivial to iterate over the entries to mark the corresponding keys as assigned. The unmarked positions are then updated to contain new data. The FTL must also generate cryptographically-secure random data or be able to receive it from the host. Finally, the file system mounted on the FTL must issue TRIM commands [84] when a sector is deleted, as only the file system has the semantic context to know when a sector is deleted.

12.3 Concluding Remarks

This thesis provided a thorough examination of the problem of secure data deletion. We focused on two main environments—mobile storage and remote storage—both of which lack secure data deletion despite storing sensitive data. Our examination of related work not only provides a survey of the literature but further builds a framework necessary for the comparison of secure deletion solutions and determine what are the salient features of solutions. We then presented a system and adversarial model in which we design and analyze our contributions.

For flash memory, we showed that it is possible for a user to delete data without any modifications to their operating system. When able to modify the operating system, however, DNEFS can be integrated into flash file systems to create a file system with comprehensive and efficient secure data deletion. Our design and implementation of UBIFSec validated DNEFS by showing that it is indeed efficient and unobtrusive to use.

For remote storage, we design a key disclosure graph: a tool for modelling and reasoning about the adversary's growth of knowledge when storing wrapped encryption keys on a persistent storage medium. We defined useful conditions on mutations for this graph that correspond to ways of updating data structures to easily effect secure deletion. We further designed and built a B-Tree-based solution to analyze the performance in practice. Our final contribution is the analysis of an unreliable securely-deleting storage medium. We allow it to fail in storing and deleting data, and make it robust against these failures by distributing it among multiples nodes. We examined the secure-deletion complications that arise from this and implement our designed distributed system for analysis.

Throughout, we provided new ways for thinking about the problem of secure deletion: asymmetries in erase/write granularities, mangrove-shaped key disclosure graphs, keystores with pre-written keys, existential latency, and multiple-access adversaries. We hope that these concepts are found useful in the design of future systems.

Bibliography

[1] R. Perlman, "The Ephemerizer: Making Data Disappear," Sun Microsystems, Tech. Rep., 2005.

[2] Y. Tang, P. Lee, J. Lui, and R. Perlman, "Secure Overlay Cloud Storage with Access Control and Assured Deletion," *Dependable and Secure Computing*, pp. 903–916, 2012.

[3] A. Rahumed, H. C. H. Chen, Y. Tang, P. P. C. Lee, and J. C. S. Lui, "A Secure Cloud Backup System with Assured Deletion and Version Control," in *ICPP Workshops*, 2011, pp. 160–167.

[4] D. Boneh and R. J. Lipton, "A Revocable Backup System," in *USENIX Security Symposium*, 1996, pp. 91–96.

[5] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computing Surveys*, vol. 43, no. 1, pp. 1–37, 2010.

[6] S. Swanson and M. Wei, "SAFE: Fast, Verifiable Sanitization for SSDs," UCSD, Tech. Rep., October 2010.

[7] N. Provos, "Encrypting virtual memory," in *USENIX Security Symposium*, 2000, pp. 35–44.

[8] R. Perlman, "File System Design with Assured Delete," in *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[9] C. Pöpper, D. Basin, S. Capkun, and C. Cremers, "Keeping Data Secret under Full Compromise using Porter Devices," in *Computer Security Applications Conference*, 2010, pp. 241–250.

[10] R. Kissel, M. Scholl, S. Skolochenko, and X. Li, "Guidelines for Media Sanitization," September 2006, National Institute of Standards and Technology.

[11] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, "Vanish: increasing data privacy with self-destructing data," in *USENIX Security Symposium*, 2009, pp. 299–316.

[12] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. Levy, P. Gardner, and V. Moscaritolo, "New directions for self-destructing data systems," University of Washington, Tech. Rep., 2010.

[13] S. Diesburg, C. Meyers, M. Stanovich, M. Mitchell, J. Marshall, J. Gould, A.-I. A. Wang, and G. Kuenning, "TrueErase: Per-File Secure Deletion for the Storage Data Path," *ACSAC*, 2012.

[14] S. Bauer and N. B. Priyantha, "Secure Data Deletion for Linux File Systems," *Usenix Security Symposium*, pp. 153–164, 2001.

[15] Privacy Commissioner of Canada, "Personal Information Protection and Electronic Documents Act," 2011. [Online]. Available: http://laws-lois.justice.gc.ca/eng/acts/p-8.6/

[16] United States Department of Health and Human Services, "HIPAA Security Guidance," 2006.

[17] Electronic Privacy Information Center, "Investigations of Google Street View," 2012. [Online]. Available: https://epic.org/privacy/streetview/

[18] Mozilla Foundation, "Firefox." [Online]. Available: https://www.mozilla.org/

[19] Google, Inc., "Google chrome." [Online]. Available: http://www.google.com/chrome/

[20] Apple, Inc., "Safari." [Online]. Available: http://www.apple.com/safari/

[21] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a Type III Anonymous Remailer Protocol," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003, pp. 2–15.

[22] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[23] European Commission, "Progress on EU Data Protection Reform Now Irreversible Following European Parliament Vote," March 2014. [Online]. Available: http://europa.eu/rapid/press-release_MEMO-14-186_en.htm

[24] California State Legislature, "Senate Bill No. 568 CHAPTER 336," 2013.

[25] J. M. Rosenbaum, "In defense of the DELETE key," 2000.

[26] S. Garfinkel and A. Shelat, "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security & Privacy*, pp. 17–27, January 2003.

[27] H. Ritzdorf, N. Karapanos, and S. Capkun, "Assisted Deletion of Related Content," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. ACM, 2014.

[28] G. G. Richard III and V. Roussev, "Scalpel: A frugal, high performance file carver." in *DFRWS*, 2005.

[29] A. Pal, K. Shanmugasundaram, and N. Memon, "Automated reassembly of fragmented images," in *Proceedings of the 2003 International Conference on Multimedia and Expo - Volume 2*, ser. ICME '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 625–628.

[30] Simson L. Garfinkel, "Carving contiguous and fragmented files with fast object validation," *Digital Investigation*, vol. 4, Supplement, pp. 2–12, 2007.

[31] A. Pal and N. Memon, "The evolution of file carving," *Signal Processing Magazine, IEEE*, vol. 26, no. 2, pp. 59–71, 2009.

[32] A. D. McDonald and M. G. Kuhn, "Stegfs: A steganographic file system for linux." in *Information Hiding*, ser. Lecture Notes in Computer Science. Springer, 1999, pp. 462–477.

[33] R. Anderson, R. Needham, and A. Shamir, "The steganographic file system," 1998, pp. 43–60.

[34] S. Bajaj and R. Sion, “Hifs: history independence for file systems,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 1285–1296.

[35] S. Mitra and M. Winslett, “Secure deletion from inverted indexes on compliance storage,” in *Proceedings of the Second ACM Workshop on Storage Security and Survivability*, ser. StorageSS ’06. ACM, 2006, pp. 67–72.

[36] S. L. Garfinkel, “Leaking sensitive information in complex document files-and how to prevent it,” *IEEE Security & Privacy*, pp. 20–27, 2014.

[37] D. Perito and G. Tsudik, “Secure code update for embedded devices via proofs of secure erasure.” *IACR Cryptology ePrint Archive*, 2010.

[38] A. Young and M. Yung, “Cryptovirology: Extortion-based security threats and countermeasures,” *IEEE Symposium on Security and Privacy*, pp. 129–140, 1996.

[39] M. Wei, L. M. Grupp, F. M. Spada, and S. Swanson, “Reliably Erasing Data from Flash-Based Solid State Drives,” in *USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2011, pp. 105–117.

[40] P. T. McLean, “AT Attachment with Packet Interface Extension (ATA/ATAPI-4),” 1998. [Online]. Available: http://www.t10.org/t13/project/d1153r18-ATA-ATAPI-4.pdf

[41] G. Hughes, T. Coughlin, and D. Commins, “Disposal of disk and tape data by secure sanitization,” *Security Privacy, IEEE*, vol. 7, no. 4, pp. 29–34, 2009.

[42] T. Gleixner, F. Haverkamp, and A. Bityutskiy, “UBI - Unsorted Block Images,” 2006. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf

[43] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, “The GNU C Library Reference Manual,” 2012. [Online]. Available: http://www.gnu.org/software/libc/manual/pdf/libc.pdf

[44] Intel Corporation, "Intel Solid-State Drive Optimizer," 2009. [Online]. Available: http://download.intel.com/design/flash/nand/mainstream/Intel_SSD_Optimizer_White_Paper.pdf

[45] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *USENIX Security Symposium*, 1996, pp. 77–89.

[46] C. Wright, D. Kleiman, and R. S. S. Sundhar, "Overwriting Hard Drive Data: The Great Wiping Controversy," *Information Systems Security*, pp. 243–257, 2008.

[47] Y. Gardi, "mmc: card: IOCTL support for Sanitize feature of eMMC v4.5," 2011.

[48] D. Jagdmann, "srm - Linux man page."

[49] B. Durak, "wipe(1) - Linux man page."

[50] Apple, Inc., "Mac OS X: About Disk Utility's erase free space feature," 2012. [Online]. Available: https://support.apple.com/kb/HT3680

[51] J. Garlick, "scrub(1) - Linux man page."

[52] S. L. Garfinkel and D. J. Malan, "One big file is not enough: A critical evaluation of the dominant free-space sanitization technique." in *Privacy Enhancing Technologies, 6th International Workshop, PET 2006, Cambridge, UK, June 28-30, 2006, Revised Selected Papers*, 2006, pp. 135–151.

[53] Oracle Corporation, "About MySQL," 2012. [Online]. Available: http://www.mysql.com/about/

[54] Hipp, Wyrick & Company, Inc., "About SQLite," 2012. [Online]. Available: http://www.sqlite.org/about.html

[55] P. Stahlberg, G. Miklau, and B. N. Levine, "Threats to privacy in the forensic analysis of database systems," in *ACM SIGMOD conference on Management of data*, 2007, pp. 91–102.

[56] SQLite, "Pragma statements." [Online]. Available: http://www.sqlite.org/pragma.html#pragma_secure_delete

[57] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *Dutch International Symposium on Linux*, 1995. [Online]. Available: http://web.mit.edu/tytso/www/linux/ext2intro.html

[58] S. C. Tweedie, "Journaling the Linux ext2fs Filesystem," *Linux-Expo'98*, 1998.

[59] N. Joukov, H. Papaxenopoulos, and E. Zadok, "Secure Deletion Myths, Issues, and Solutions," *ACM Workshop on Storage Security and Survivability*, pp. 61–66, 2006.

[60] E. Zadok and J. Nieh, "FiST: A Language for Stackable File Systems," in *USENIX Technical Conference*, 2000, pp. 55–70.

[61] C. Plumb, "shred(1) - Linux man page."

[62] N. Joukov and E. Zadokstony, "Adding Secure Deletion to Your Favorite File System," *Third International IEEE Security In Storage Workshop*, pp. 63–70, 2005.

[63] Z. Peterson and R. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance," *Trans. Storage*, vol. 1, no. 2, pp. 190–212, 2005.

[64] Z. Peterson, R. Burns, and J. Herring, "Secure Deletion for a Versioning File System," *USENIX Conference on File and Storage Technologies*, 2005.

[65] R. L. Rivest, "All-Or-Nothing Encryption and The Package Transform," in *Fast Software Encryption Conference*, 1997, pp. 210–218.

[66] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, pp. 91–98, May 2009.

[67] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 142–154, 2011.

[68] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: a usability evaluation of PGP 5.0," in *USENIX Security Symposium*, 1999, pp. 169–184.

[69] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *Conference on Email and Anti-Spam*, 2004.

[70] V. Hauser, "sfill(1) - Linux man page."

[71] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-Based Storage," *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.

[72] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran, "The ANSI T10 object-based storage standard and current implementations," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 401–411, July 2008.

[73] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. Springer-Verlag, 2002, pp. 53–65.

[74] Jedec Solid State Technology Association, "Embedded MultiMedia Card (eMMC) Electrical Standard (5.01)," July 2014.

[75] M. Alfeld, W. De Nolf, S. Cagno, K. Appel, D. P. Siddons, A. Kuczewski, K. Janssens, J. Dik, K. Trentelman, M. Walton, and A. Sartorius, "Revealing hidden paint layers in oil paintings by means of scanning macro-XRF: a mock-up study based on Rembrandt's "An old man in military costume"," *Journal of Analytical Atomic Spectrometry*, vol. 28, pp. 40–51, 2013.

[76] A. Ban, "Flash file system," US Patent, no. 5404485, 1995.

[77] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, vol. 37, pp. 138–163, 2005.

[78] Google, Inc., "Google Nexus Phone." [Online]. Available: http://www.google.com/nexus/

[79] Micron Technology, Inc., "Technical Note: Design and Use Considerations for NAND Flash Memory," 2006. [Online]. Available: http://download.micron.com/pdf/technotes/nand/tn2917.pdf

[80] "Memory Technology Devices (MTD): Subsystem for Linux," 2008. [Online]. Available: http://www.linux-mtd.infradead.org/

[81] W. F. Heybruck, "An Introduction to FAT 16/FAT 32 File Systems," 2007.

[82] Charles Manning, "How YAFFS Works," 2010. [Online]. Available: http://www.yaffs.net/documents/how-yaffs-works

[83] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1992.

[84] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," 1998.

[85] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of Flash Translation Layer," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 332–343, 2009.

[86] G. Goodson and R. Iyer, "Design Tradeoffs in a Flash Translation Layer," *Workshop on the Use of Emerging Storage and Memory Technologies*, 2010.

[87] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa Linux Symposium*, 2001. [Online]. Available: http://sources.redhat.com/jffs2/jffs2.pdf

[88] A. Hunter, "A Brief Introduction to the Design of UBIFS," 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf

[89] J. Kim, "f2fs: introduce flash-friendly file system," 2012. [Online]. Available: https://lkml.org/lkml/2012/10/5/205

[90] J. Lee, S. Yi, J. Heo, and H. Park, "An Efficient Secure Deletion Scheme for Flash File Systems," *Journal of Information Science and Engineering*, pp. 27–38, 2010.

[91] Open NAND Flash Interface, "Open NAND Flash Interface Specification, version 3.0," 2011. [Online]. Available: http://onfi.org/specifications/

[92] Micron, Inc., "Design and Use Considerations for NAND Flash Memory Introduction," 2006. [Online]. Available: http://download.micron.com/pdf/technotes/nand/tn2917.pdf

[93] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: anomalies, observations, and applications," in *IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009, pp. 24–33.

[94] Samsung, Inc., "Samsung Galaxy S Phone." [Online]. Available: http://www.samsung.com/global/microsite/galaxys/index_2.html

[95] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and Repairing Write Performance on Flash Devices," in *Workshop on Data Management on New Hardware*, 2009, pp. 9–14.

[96] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," in *USENIX conference on File and storage technologies*, 2010.

[97] R. Entner, "International Comparisons: The Handset Replacement Cycle," Recon Analytics, Tech. Rep., 2011.

[98] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software Practice and Experience*, vol. 19, pp. 171–183, 1989.

[99] C. Fruhwirth, "New methods in hard disk encryption," 2005.

[100] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography, Chapter 7: Block Ciphers*, 1997.

[101] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* MIT Press, 1998.

[102] J. Reardon, "UBIFSec Implementation," 2011. [Online]. Available: http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/research/secure_deletion/ubifsec.patch

[103] Hoover, E.M., “The Measurement of Industrial Localization,” *The Review of Economics and Statistics*, pp. 162–171, 1936.

[104] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, “Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation,” in *USENIX Security Symposium*, 2005, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251420

[105] G. D. Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson, “How to Forget a Secret,” in *STACS*, ser. Lecture Notes in Computer Science. Springer, 1999, pp. 500–509.

[106] C. Cachin, K. Haralambiev, H.-C. Hsiao, and A. Sorniotti, “Policy-based secure deletion,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 259–270.

[107] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[108] S. Wolchok, O. S. Hoffman, N. Henninger, E. W. Felten, J. A. Haldermann, C. J. Rossback, B. Waters, and E. Witchel, “Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs,” in *Proc. 17th Network and Distributed System Security Symposium*, Feb. 2010.

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

[110] D. Comer, “The ubiquitous B-tree,” *ACM Computing Surveys*, vol. 11, pp. 121–137, 1979.

[111] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing—a fast access method for dynamic files,” *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, 1979.

[112] O. Rodeh, “B-trees, shadowing, and clones,” *Trans. Storage*, vol. 3, no. 4, pp. 2:1–2:27, 2008.

[113] McDougall, R. and Mauro, J., “FileBench,” 2005. [Online]. Available: www.solarisinternals.com/si/tools/filebench/

[114] L. A. Bélády, "A study of replacement algorithms for virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[115] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89. Springer-Verlag New York, Inc., 1989, pp. 218–238.

[116] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and Integrity in Outsourced Databases," *Trans. Storage*, pp. 107–138, 2006.

[117] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: The Least-Authority Filesystem," in *In Proceedings of the 4th ACM international workshop on Storage security and survivability*, 2008, pp. 21–26.

[118] FUSE Developers, "The Filesystem in Userspace Website," 2014. [Online]. Available: http://fuse.sourceforge.net

[119] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Internet Requests for Comments, RFC 5246, May 2000. [Online]. Available: https://tools.ietf.org/html/rfc5246

[120] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, pp. 644–654, Sep. 2006.

[121] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and T. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile ," Internet Requests for Comments, RFC 6818, May 2008. [Online]. Available: https://tools.ietf.org/html/rfc6818

[122] Amazon Inc., "Amazon Webservices," 2014. [Online]. Available: https://aws.amazon.com

[123] Dropbox, Inc., "The Dropbox Website," 2014. [Online]. Available: https://www.dropbox.com

[124] Google, Inc., "The Google Drive Website," 2014. [Online]. Available: https://drive.google.com

[125] N. A. Lynch, *Distributed Algorithms.* Morgan Kaufmann, 1996.

[126] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[127] G. Cole, "Tp-338.1: Estimating drive reliability in desktop computers and consumer electronics systems," 2000. [Online]. Available: kc.ors-pc.com/bbs/img/8.pdf

[128] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA: USENIX Association, 2007.

[129] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA: USENIX Association, 2007.

[130] Markus F.X.J. Oberhumer, "The LZO Website," 2014. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[131] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography, Chapter 5: Pseudorandom Bits and Sequences*, 1997.

[132] xmlrpc-c Developers, "The XML-RPC for C and C++ Website," 2014. [Online]. Available: http://xmlrpc-c.sourceforge.net

[133] stunnel Developers, "The stunnel Website," 2014. [Online]. Available: http://www.stunnel.org

[134] E. Rescorla, "HTTP Over TLS," Internet Requests for Comments, RFC 2818, May 2000. [Online]. Available: https://tools.ietf.org/html/rfc2818

[135] openssl Developers, "The OpenSSL Website," 2014. [Online]. Available: www.openssl.org