User-Level Secure Deletion on Log-structured File Systems

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

Claudio Marforio
ETH Zurich, Switzerland
maclaudi@inf.ethz.ch

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

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

ABSTRACT

We address the problem of secure data deletion on log-structured file systems. We focus on the YAFFS file system, used on Android smartphones, and on the flash translation layer (FTL), used in SD cards and USB memory sticks. We show that neither of these systems provide temporal data deletion guarantees and that deleted data remains indefinitely on these systems if the storage medium is not used after the data is marked for deletion. Moreover, the time that data remains on log-structured file systems increases with the storage medium's size.

We propose two user-level solutions that achieve secure deletion: purging, which ensures that all data is deleted, and ballooning, which reduces the expected deletion latency. We show that these two solutions can be combined to guarantee 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. Our solutions empower users to ensure the secure deletion of their data without relying on the manufacturer to provide this functionality. We implement these solutions on Nexus One smartphones and show that they succeed in secure deletion. When used properly, our solutions neither prohibitively reduce the longevity of the flash memory nor noticeably reduce the device's battery lifetime.

Categories and Subject Descriptors

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASIACCS '12 Seoul, South Korea
Copyright 2012 ACM 978-1-4503-1303-2/12/05 ...$10.00.

General Terms

Secure deletion, privacy, flash memory

1. INTRODUCTION

Deleting a file from a storage medium serves two purposes: it reclaims storage and ensures that any sensitive information contained in the file becomes inaccessible. When done for the latter purpose, it is critical that the file is *securely* deleted, meaning that its content does not persist on the storage medium after deletion. Secure deletion enables users to protect the confidentiality of their data if their storage media are compromised, stolen, or confiscated under a subpoena. In the case of a subpoena, the user may even be forced to disclose all passwords or other credentials that enable access to the data stored on the storage medium; in such a scenario, to preserve the confidentiality of their data, users can only sanitize their storage medium before it is seized.

Secure deletion is almost always ignored in file system design [5, 6, 21, 27, 31], largely due to performance reasons. Typically, deletion is implemented as a rapid operation where a file is *unlinked*, meaning its metadata states that it is no longer present while the file's contents remain on the storage medium until overwritten by new data [9]. The time between deleting data and its actual removal from the storage medium is called the *deletion latency*. Therefore, deleted data remains accessible during the entire deletion latency.

Secure deletion is particularly important on modern smartphones, as they increasingly store personal data such as the owner's private communication, browsing history, and location history. Mobile phones further store business data, for which company policy or legislation may mandate the data's deletion after some time elapses or at some geographic locations. Prior to this work, the only effective user-level secure deletion solution available for mobile phones was the *factory reset*, which securely deletes all user data on the phone by returning the phone to its initial state. This is clearly inappropriate for users who wish to selectively delete data, such as some emails, but still retain their address books and installed applications. Other applications, such as FileShredderPro [20], claim to securely delete files by overwriting them with random data. However, due the nature of log-structured file systems, this solution is no more effective than deleting the file, since the new copy invalidates the old one but does not physically overwrite it [27].

In this work, we address secure deletion on modern smartphones, focusing on the flash file system YAFFS [6]. YAFFS is a log-structured file system developed specifically for flash memory storage. Android phones' internal memory uses YAFFS to store data such as browsing caches, maps caches, names of nearby wireless networks, GPS location data, SMS messages, electronic mails, and telephone call listings.

We also address secure deletion on flash memory when it is accessed through a flash translation layer (FTL). An FTL is an indirection layer between the flash memory and the file system that abstracts away all the nuances of flash memory, allowing traditional file systems such as FAT [14] to be used on the storage medium. Internally, FTLs implement a log-structured file system, mapping logical sectors to physical sectors [1]. FTLs are widely used on SD cards, USB sticks and solid state drives.

We analyze how deletion is performed in YAFFS and on an FTL and show that log-structured file systems in general provide no temporal guarantees on data deletion; the time deleted 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.

We propose user-level solutions for secure deletion in log-structured file systems: *purging*, which provides guaranteed time-bounded deletion of all data previously marked to be deleted, and *ballooning*, which continuously reduces the expected time that any piece of deleted data remains on the medium. We combine these two solutions into an effective hybrid solution.

We implement these solutions on an Android smartphone (Nexus One) 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, and also substantially reduces the deletion latency on large, sparsely occupied storage media.

The rest of this paper is organized as follows. In Section 2 we give background on flash memory and file systems. In Section 3 we examine the current state of secure deletion in log-structured file systems. In Section 4 we present our solutions, along with experimental results. In Section 5 we discuss related work and in Section 6 we draw conclusions.

2. SYSTEM MODEL AND BACKGROUND

System Model.

We consider a scenario in which users have private sensitive data on mobile storage media that they wish to selectively securely delete. This includes cache files that should be *continually* deleted, such as location data encoded by GPS data or observed wireless networks. All valid data must remain available during the deletion.

We consider the *coercive attacker* as our adversary. This attacker can—at any moment—both obtain the user's storage medium and compel the user to reveal any secret keys and passphrases [26]. The attack's unpredictable nature prevents the user from performing any sanitization procedures before disclosure. This differentiates our secure deletion problem from secure data deletion in the context of repurposed hardware [9].

Under this attacker model, common solutions to preserving data confidentiality fail. Encrypting all the data on the storage medium does not work since the adversary is given all our encryption keys. The use of factory reset is impractical as the unpredictable compromise time requires deleting the entire phone's memory with such frequency that little useful data could reside on the storage medium.

We therefore need novel solutions to this problem. Solutions can exist at either of two system levels: user-level and kernel-level. User-level means that the solution must be executable as an application that does not require any elevated permissions or modifications to the operating system. This mode of access is greatly limited: an application's interaction with the file system consists solely of the creation and deletion of its own local files. It cannot change the file system's behaviour in any way to achieve secure deletion. However, as we show, it empowers users to integrate secure deletion solutions themselves without waiting until they are offered by storage medium manufacturers. This is important given the failure of manufacturers to provide sanitization or to correctly implement sanitization [30]. Kernel-level access is much less limited: it assumes that arbitrary changes can be made to the file system and a new kernel can be installed on the device—allowing for more efficient solutions, as kernel-level capabilities are a strict superset of user-level capabilities.

Flash Memory.

Flash memory is a non-volatile storage medium consisting of an array of electrical components that store information [1]. A *page* of flash memory is *programmed* to store data, which can thereafter be *read* until the page is *erased* [8]. Once programmed, the contents of flash memory should not be altered in place, but rather an erase procedure must be performed on erase blocks, which have a granularity larger than read/write chunks [8]. The support of multiple programmings between erasures varies between flash types, and its use is generally discouraged [23]. Flash erasure is costly: its increased voltage requirement eventually wears out the medium. Erase blocks can only handle 10^4 to 10^5 erasures [29] before wearing out and becoming unusable.

Log-structured File Systems.

A log-structured file system differs from a traditional block-based file system (such as FAT [14] or ext2 [5]) in that the entire file system is stored as a chronological record of changes from the initial empty state. As files are written, new fixed-size chunks are appended to the log indicating the resulting change; a page can store either a file's header or some data. The file system maintains in volatile memory the appropriate data structures to quickly find the newest version of each header and data page [8, 6].

Log-structured file systems complicate secure deletion because the traditional solution of overwriting a file with new content simply appends a second version of the file, while the first copy still remains in the log. Similarly, encrypting a file also appends a new encrypted version of that file, while the plaintext remains in the log.

Data is removed from a log-structured file system during *garbage collection* [8]. The garbage collector operates at the *erase block* level, which has a larger granularity than a page. While implementation details may vary, in principle

the garbage collector erases any 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.

Flash Translation Layer (FTL).

An FTL is a hardware or software device that interacts directly with the flash memory controller and exposes a block device interface [1]. Any block file system, such as FAT [14] or ext2 [5], can be used to store data on the flash memory, and the FTL opaquely handles bad block detection, wear-levelling, and garbage collection of erase blocks [1]. FTLs are widely used in SD cards, USB sticks, and SSD drives. FTLs are also used for the internal memory of iOS devices.

FTLs vary in implementation [7, 10], however they all have a simple design. They map logical block addresses to physical flash memory addresses [15] and effectively implement a log-structured file system over the flash erase blocks. Updates to logical sectors are written to unused locations, and the mapping is updated to reflect this change. The file system should tell the FTL when it discards a sector, meaning that the data in that sector has been deleted by the file system and the FTL no longer needs to preserve the data during erase block garbage collection.

The Linux *ftl* driver is an open-source implementation of an FTL based on Intel's specification [15]. This driver selects erase blocks for garbage collection using a greedy algorithm, where the erase block with the most wasted space is selected to be compacted. To add some wear-levelling, it periodically selects the erase block with the fewest past erasures instead.

YAFFS.

Yet Another Flash File System (YAFFS) is a log-structured file system designed specifically for flash memory [6]. It is notably used as the file system for the internal memory of Android mobile phones [11].

YAFFS allocates memory by selecting an unused erase block and allocating sequentially the numbered chunks in that erase block. YAFFS searches for empty erase blocks sequentially by the erase block number as defined by the physical layout of memory on the storage medium, wrapping cyclically when necessary. It begins its search from the last allocated erase block and returns the first empty erase block it finds. When the allocation of an erase block reduces the total number of empty erase blocks in the system below a minimum threshold, then YAFFS performs garbage collection to reclaim wasted space on partially-full erase blocks. Garbage collection copies the valid chunks from the partially-full erase block to the end of the log and then erases the erase block. 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.

Garbage collection in YAFFS is either initiated by a thread that performs system maintenance, or takes place during write operations. Usually, a few chunks are copied at a time, whereby 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 garbage collection is performed. In this case, erase blocks with any deleted space are collected, and the procedure continues until the entire erase block can be reclaimed.

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

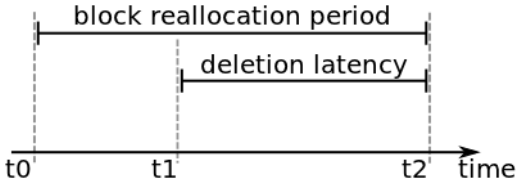


Figure 1: A lifetime of stored data. At time t_0 the erase block is allocated and data written onto it soon after. At time t_1 the data is deleted. At time t_2 the erase block is reallocated, thus removing the data from the medium. $t_2 - t_1$ is called the deletion latency, and $t_2 - t_0$ is called the erase block reallocation period.

3. DATA DELETION IN EXISTING LOG-STRUCTURED FILE SYSTEMS

In this section, we investigate data persistence on log-structured file systems by analyzing the internal memory of a Nexus One running Android/YAFFS and simulating larger storage media for both YAFFS and an FTL. We instrument the file system at the kernel level to log erase block allocation information. Afterwards, we examine the deletion latency for storage media of various types. Deletion latency, illustrated in Figure 1, is the time that deleted data remains accessible on the storage medium. In particular, we measure the average and worst-case data deletion latency for specific devices, application configurations and usage patterns.

Our results show that modern Android smartphones have large deletion latency, where deleted data can remain indefinitely on the storage media. This motivates our secure deletion solutions in the next section.

3.1 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 allocations 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 median time between erase blocks reallocations is 44.5 hours. When an erase block is reallocated, it means that data deleted prior to reallocation is gone, thus forming an upper bound for deletion latency.

3.2 Simulating Larger Storage Media

Log-structured file systems favour allocating empty erase blocks before compacting partially-empty erase blocks [10,

6]. 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 simulating the writing behaviour of an Android phone on simulated YAFFS and FTL 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 or a block file system with an FTL.

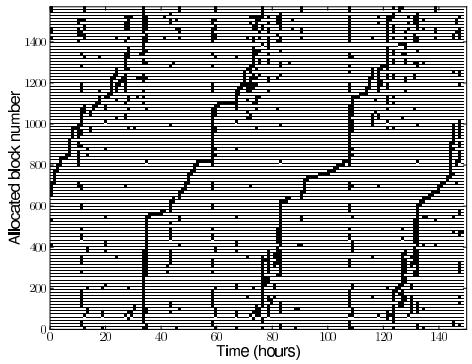
We used the collected statistics from our instrumented phone in Section 3.1 to determine the writing behaviour for our discrete event simulator. We logged every chunk that was written to the device for a week, and used this data to compute the period between successive creations of new files, and the type of file to create. A file type is defined by its lifetime, a distribution over the period between opening a file for write, a distribution over the number of chunks to write to a file each time it is opened, and a distribution over a file's chunks that indicates where the writes will occur.

Additionally, we implemented a secret writer that operated alongside the simulated writers. It periodically wrote a one-page secret message, waited until a new erase block was allocated, and then deleted the secret message. We used this to determine the deletion latency for data written at that particular moment in time.

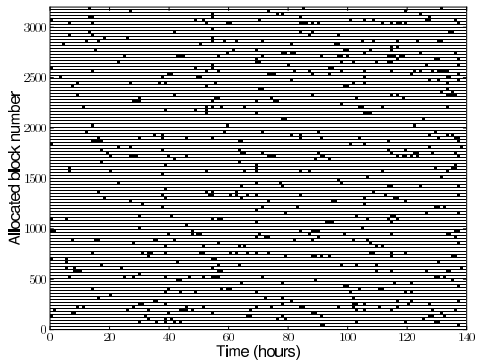
We performed experiments using the flash file system YAFFS and the block file system FAT when accessed through an FTL. YAFFS was mounted on a virtual flash storage medium created by the kernel module *nandsim*. For our tests with FTL, we used the *ftl* kernel module that implements a software FTL layer for flash hardware, which we then simulated in memory.

There were some differences between the YAFFS and FTL configurations. The internal memory of the Android phone has a page size of 2048 bytes, whereas the FTL driver has a hardcoded size of 512 bytes that is consistent with the page size of the file systems that are used on consumer devices such as SD cards. Therefore, each FTL experiment we performed has a four-fold larger YAFFS complement. In our experiments, we kept the same number of page writes the same between file systems, regardless of the difference in size. We also used an erase block size of 64 chunks for YAFFS (consistent with the Google Nexus One phone [11]) and 32 chunks for FTL (consistent with nandsim's flash devices with a 512-byte chunk size). This difference implies that the same number of writes to both YAFFS and FTL results in twice as many erase block allocations for the latter.

In our experiments, we ignored the small unaligned writes to the virtual page map and to fixed sectors in the FAT file system corresponding to updates to the file allocation table. The Linux *ftl* driver is a simple implementation based on Intel's specification [15], while FTLs used in consumer devices have had well-funded development, and can be optimized to deal effectively with these frequent writes [32]. The exact implementation can also vary among devices, as the design space and optimization possibilities of FTLs is a research area of its own [10, 7]. We therefore collected data on the writes we are certain must occur, and recall that for each write, a (possibly amortized) number of additional writes must also occur.



(a) YAFFS' erase block allocation over time on an Android phone.



(b) FTL's erase block allocation over time in simulation.

Figure 2: Plot of erase block allocation over time for YAFFS and an FTL. The time between two points on the same horizontal line is the erase block reallocation period.

Deletion Latency.

We plotted the flash storage media's erase block allocation over time to gain intuition on its behaviour. Figure 2 (a) shows the results for our YAFFS storage medium and (b) for our FTL storage medium. The X-axis corresponds to time, and the Y-axis represents the space of sequentially-numbered erase blocks. A black square on the graph means that an erase block was allocated at that time. For clarity, Figure 2 (a) shows the allocations for every 15th erase block and (b) for every 30th.

We present the results of our experiment in Table 1, which gives the median and 95th percentile deletion times in hours for secrets written onto the storage medium during simulation. The results are provided for YAFFS and FTL partitions of different sizes. The maximum measurement is undefined because these systems provide no deletion guarantee.

We observe the effect of cyclic erase block allocation in YAFFS, as there is both a linear growth in deletion latency as the size of the partition increases, and a high percentile

Partition size / type	Deletion latency (hours) 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
50 MB FTL	53.5 ± 4.2	91.6 ± 10.3
250 MB FTL	131.6 ± 15.4	335.5 ± 40.5
500 MB FTL	233.0 ± 17.9	714.4 ± 27.4

Table 1: Deletion latency in hours for different configuration parameters.

observation close to the median. FTL's greedy allocation is evident in the unpredictability of sequential allocations and the substantially larger gap between the median and 95th percentile deletion latencies. For instance, a YAFFS implementation on a 2 GB partition (e.g, the data partition on the Samsung Galaxy S [28]) 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.

4. 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 and are designed for the scenario where a security-conscious mobile phone user wants to install a secure deletion application from an application marketplace, but is unwilling (or insufficiently skilled) to install a new phone operating system or is prohibited from doing so by the mobile phone's manufacturer. User-level solutions allow the users themselves to achieve secure deletion without voiding warranties or relying on the hardware manufacturer to provide secure deletion.

User-level solutions have limited access to the flash storage medium. Such solutions 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 garbage collection 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 garbage collection, 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 periodically purges to guarantee deletion.

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

4.1 Purging

Purging completely fills the file system's empty space with junk files, thereby ensuring that all unneeded data is securely deleted. 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, whenever 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 before crossing a border.

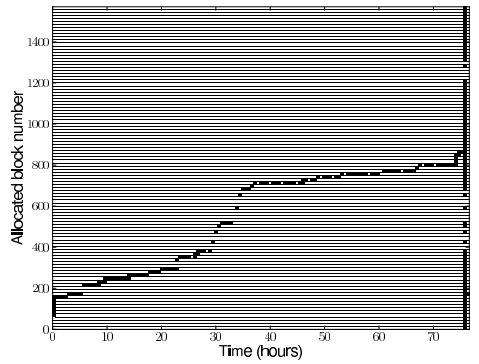
Completely filling the storage medium is possible provided the user is not subjected to disk-quota limitations. It typically requires garbage collecting (i.e., erasing) most erase blocks on the storage medium. This is because deleted chunks can occur in any erase block that sees active use, resulting in small data gaps throughout the file system.

The fact that the storage medium must be completely filled follows from a worst case analysis of a log-structured file system's erase block allocation strategy. 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 secret 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 data chunks are no longer available on the storage medium. This condition holds for YAFFS and the Linux FTL implementation, however other file systems and FTL hardware implementations may differ.

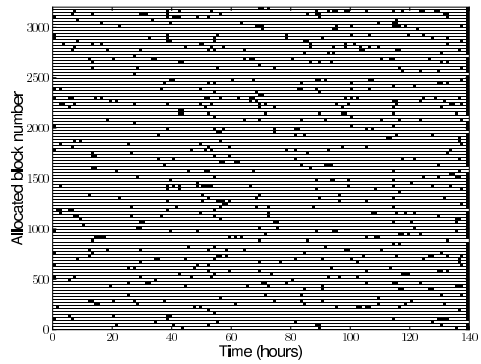
A natural concern for purging's correctness is its behaviour on multithreaded systems. However, using the previous reasoning, purging needs to keep writing to the storage medium until it reports that it completely full. This ensures that any data that has been deleted prior to purging is irrecoverable 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 will also be told 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 secret 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.

Figure 3 (a) shows the resulting erase block allocations reported by an instrumented version of YAFFS executing purging, and (b) is the same for the FTL implementation. The X-axis corresponds to time in hours, and the Y-axis shows the numbered erase blocks. A small square in the graph indicates when each erase block was allocated. As with Figure 2, only a subset of erase block allocations are plotted for clarity. At the right side of both graphs, we see the near immediate allocation of every erase block on

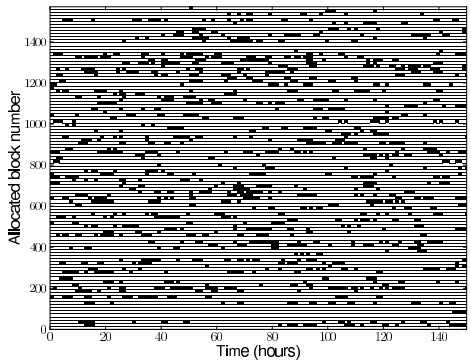


(a) Executing purging on YAFFS.

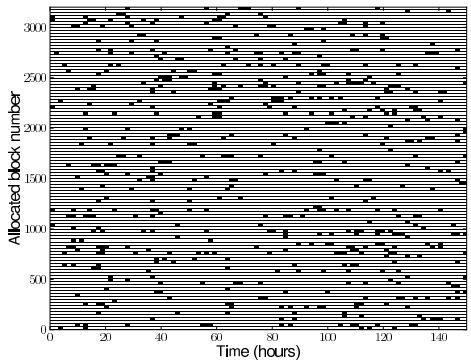


(b) Executing purging on an FTL.

Figure 3: Plot of erase block allocation over time for (a) YAFFS and (b) FTL. 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.



(a) Executing ballooning on YAFFS.



(b) Executing ballooning on an FTL.

Figure 4: Plot of erase block allocation over time for (a) YAFFS and (b) FTL when running ballooning.

the medium. This is the consequence of filling the storage medium to capacity; a log-structured file system must garbage collect most of its erase blocks to reclaim every available page.

4.2 Ballooning

In contrast to purging, which guarantees time-bounded secure data deletion, we now present ballooning, which achieves probabilistic continuous secure deletion. Ballooning artificially constrains the the file system's or FTL's available free space. This results in more frequent garbage collection 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 space of the storage medium, which reduces the total number of erase blocks available for allocation. This reduces the expected erase block reallocation period, which is an upper bound on the deletion latency for data contained in that erase block (cf. Figure 1).

In Section 4.4, we will explore how varying free space thresholds—the aggressiveness of ballooning—affect deletion latency and other measurements. However, we first visualize our hypothesis that ballooning reduces the erase block reallocation period. Figures 4 (a) and (b) show the erase block allocations that result from executing ballooning on YAFFS and an FTL respectively. We see a stark difference when compared with Figure 2. As the number of allocatable erase blocks decreases, YAFFS' sequential allocation becomes much more erratic, and the erase block reallocation period decreases. Similarly, the randomness of FTL's greedy strategy focuses more on specific rows of erase block allocation activity. The rows of Figure 4 that contain no allocation activity likely correspond to erase blocks that have now been assigned to junk files. Both show a decrease in the erase block allocation period, which therefore reduces the expected deletion latency.

4.3 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 data is deleted—or indeed even if. By combining both these approaches, we create a hybrid scheme that has neither disadvantage. We use purging to *guarantee* the secure deletion of data, and we use ballooning to ensure that a large storage medium's empty space must not be refilled during every purging operation.

Reducing the number of erase blocks that must be filled during purging mitigates three concerns about purging: its wear on the storage medium, its power consumption, and its execution time. Large capacity storage media are particular

suitable to this solution, as 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 effect with experimental results for various storage medium sizes and ballooning aggressiveness settings.

4.4 Experimental Evaluation

We developed an application that implements our hybrid solution. It periodically examines the file system to determine the amount of free space, and appropriately creates and deletes junk files to maintain the free space within the upper and lower thresholds. The oldest junk file is always deleted before more recent ones to load-balance the erasure wear on the flash memory. 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 will run 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 secrets remained on the storage medium to determine ballooning's effectiveness. We measured the ratio of deleted chunks 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 chunks 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 garbage collector. Since we are using fixed write distributions, the expected rate of writes from the simulator is identical between experiments. Therefore, the observed disparity in erase block allocation rates reflects exactly the additional writes resulting from the increased garbage collections caused by our application to achieve secure deletion.

To quantify the benefit of our application—that is, how promptly the secure deletion of sensitive data occurs—we measure the expected time sensitive data remains on the storage medium. We calculate this measurement using our secret writer that periodically writes one page secrets onto the medium and deletes them. We then compute how long the written secrets remain on the storage medium.

Experimental Results.

Tables 2 and 3 present the results for YAFFS and the Linux FTL implementation on simulated storage media of different sizes while using different ballooning thresholds. The partition size is the full storage capacity of the medium, and the type is either the file system YAFFS or an FTL which stores a FAT file system. The fill ratio is the average proportion of valid 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 erasure cycle lifetime of 10^4 erasures, 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 some measurements.

Ballooning, Deletion Latency, and Block Allocation Rate.

As discussed in Section 3.2, without ballooning both the fill ratios and the deletion latency are highly dependent on the size of the storage medium. However, as ballooning increases the fill ratio, the deletion latency similarly 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 garbage collection and thus more frequent writes to the storage medium simply to copy data stored elsewhere.

We see from the deletion latency that device size is not a deciding 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 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

Partition size / type	Free blocks	Fill ratio	Block allocs per hour	Lifetime (years)	Purge cost (blocks)	Deletion latency (hours) median	95th %ile
	603.8	20%	32.7 ± 2.3	54	1556.8	41.5 ± 2.6	46.2 ± 0.5
200 MB	91.8	63%	53.4 ± 4.7	33	705.2	10.8 ± 1.7	14.6 ± 1.3
YAFFS	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
	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
1 GB	88.2	64%	59.8 ± 8.4	29	765.0	10.4 ± 0.5	16.1 ± 2.0
YAFFS	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
	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
2 GB	254.5	48%	41.1 ± 3.7	43	1237.5	28.7 ± 1.5	34.8 ± 6.1
YAFFS	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 2: Block allocations, storage medium lifetimes, and deletion times for the YAFFS file system.

Partition size / type	Free blocks	Fill ratio	Block allocs per hour	Lifetime (years)	Purge cost (blocks)	Deletion latency (hours) median	95th %ile
	1850.7	14%	67.2 ± 7.5	26	4463	53.5 ± 4.2	91.6 ± 10.3
50 MB	42.8	50%	90.2 ± 10.2	19	3491	7.0 ± 2.0	71.7 ± 36.7
FTL	1.0	75%	148.9 ± 33.1	12	1388	2.2 ± 0.5	39.7 ± 9.6
	1.0	80%	183.9 ± 11.1	9	1247	1.9 ± 0.5	22.7 ± 5.5
	1.0	85%	230.7 ± 21.6	7	501	1.3 ± 0.7	18.9 ± 1.6
	13906.1	14%	65.6 ± 4.4	27	21121	131.6 ± 15.4	335.5 ± 40.5
250 MB	14483.9	13%	67.6 ± 5.4	26	20993	62.3 ± 7.5	104.1 ± 28.6
FTL	662.2	69%	261.7 ± 17.1	6	11554	6.5 ± 2.2	46.6 ± 0.7
	511.9	82%	379.4 ± 10.1	4	3234	3.5 ± 1.5	15.4 ± 1.0
	29795.9	13%	62.6 ± 0.6	28	35665	233.0 ± 17.9	714.4 ± 27.4
500 MB	30619.2	15%	65.0 ± 5.7	27	35367	65.4 ± 18.6	118.7 ± 16.8
FTL	4882.9	71%	204.1 ± 6.8	8	20649	54.6 ± 2.3	96.3 ± 0.3
	2207.4	75%	329.9 ± 10.6	5	16896	12.4 ± 3.0	57.2 ± 13.1

Table 3: Block allocations, storage medium lifetimes, and deletion times for the Linux FTL implementation.

of erase block allocations required for purging. However, we see that even 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 tenfold improvement in both deletion latency and purging cost.

Ballooning and Storage Medium Lifetime.

The primary drawback of our solutions is the additional wear that increased erasures put on the mobile phone, both in terms of damage to the flash memory and power consumption. It would be a concern for adoption if our solution significantly reduces the phone's lifetime or battery life.

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 2. We use a conservative estimate of 10^4 erasures per erase block. (Recall that a typical flash erase block can handle between 10^4 and 10^5 erasures [29], and some studies have indicated this is already orders of magnitude more conservative than reality [4].) Our results show that even at high erase block allocation rates, we still expect to see the storage medium life for upwards of a decade; this is well in excess of the average replacement period of mobile phones. Users who require decades of longevity from their mobile phone can simply use very mild ballooning. In particular, large capacity storage media combined with mild ballooning yield a system with reasonable purging performance and flash memory lifetimes that likely exceed that of the owner.

Power Consumption.

To test if our solutions have acceptable power requirements, we analyzed 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 still takes 15 days for the ballooning application 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—0.9% of the battery per gigabyte. Other mobile phone batteries may of course yield varying results. Clearly, any mobile phone with a storage medium size in excess of a gigabyte will consume an unreasonable amount of time and energy to perform purging. The hybrid solution with mild ballooning is suited for such storage media as it significantly drops the cost of purging.

5. RELATED WORK

Sanitizing data by overwriting is the most intuitive approach to secure deletion solution given its analogue in the analog world. To sanitize a digital file, one opens the file and overwrites its contents with new data. This is the solution used by secure deletion tools such as *shred* [25] and *srm* [16]. These operate at the user-level; they can be executed by a normal system user to securely delete the user's own files. They requires that a file's data is stored only at one location; when a file is updated, the new data must replace the old one. This solution is not compatible with flash memory because in-place updates cannot be performed.

Kernel-level secure deletion solutions have been proposed for some widely-used block-structured file systems [2, 17, 18]. These solutions typically modify the kernel and enforce that when any data chunk is marked for deletion, it is then overwritten with arbitrary data. This is useful to securely delete truncated parts of files and to ensure secure deletion for file systems that do not perform immediate updates, i.e., file systems that employ journals. Our solutions shares this property: any chunk of data that is deleted will be securely deleted. However, these overwriting solutions require in-place updates of data, where old (sensitive) data is directly overwritten with new (insensitive) data; in-place updates are not supported by flash memory.

User-level encryption tools, such as GPG [12], can be used to prevent the data's disclosure under device compromise [12]. Care must still be taken to *securely* delete the original copy of the data after encrypting it and whenever it is decrypted for use. A more sophisticated implementation of this solution is to use a cryptographic file system that supports full-drive encryption [3, 13]. This solution is both more effective and usable, but it still has limitations: users can be coerced to disclose their credentials, or may be legally obliged to delete data. In our attacker model, we assume that the attacker can access our secret keys, so this solution is not appropriate. Moreover, a user-level application cannot encrypt data from other applications on the system. We observe, however, that it would be very useful if mobile phones encrypted all short-term cache data and kept all encryption keys only in volatile memory. This way, securely deleting this data is trivial and a power loss would still not lose important data.

Wei et al. [30] considered secure deletion on flash storage in the context of solid state drives, which access the memory through an FTL. They executed traditional block-based in-place overwrites to perform secure deletion and determined that it did not sanitize data on flash storage; this was their expected hypothesis as flash memory cannot perform in-place updates. They also showed that some built-in sanitization methods do not function correctly. They proposed adding secure deletion by having hardware manufacturers perform zero overwriting whenever an erase block is discarded, where zero-overwriting is a technique that removes the information by programming a flash page more than once between erasures. However, programming flash memory multiple times between erasures is discouraged for all flash memory types and outright forbidden for some [23]. It operates outside flash memory's specification [24], where officially it results in undefined behaviour [24]. Flash manufacturers prohibit this due to *program disturb* [22]: bit errors that can be caused in spatially proximate pages while programming flash memory. The second version of YAFFS removed its earlier use of multiple programming to set a deleted flag on data for exactly this reason [6]. Wei et al. propose the use of *scrub budgets* for each flash memory type, which corresponds to the number of times multiple programmings can be performed for that specific memory type without resulting in a high risk of bit errors. When the scrub budget is exceeded, secure deletion occurs the by copying the valid data elsewhere and erasing the erase block. They investigate the scrub budgets for some flash memory types and observe that the scrub budgets for modern, large, dense flash memories are too small for their solution to provide great utility.

Lee et al. [19] present a secure deletion solution for YAFFS using encryption. They propose encrypting all files and including the corresponding encryption key in every file header written to the file system. Secure deletion is thus achieved whenever all the headers for a file are deleted. They propose changing the deletion code to force deletion of erase blocks containing file keys, and ensuring that the file system always colocates the same key on the same erase block. Their approach suffers from a number of limitations. Foremost is that their solution (assuming they use cipher block chaining mode) is not easily integrated into existing file systems. They required the entire file is available in memory to encrypt before it is written to the storage medium and make no mention of initialization vectors (IV). They do not discuss how to allow efficient random access to files. If instead, they further modify every data block to include a unique IV, then deleted data can be recovered as long as the file remains, which does not provide security for truncations and overwrites of files. This is particularly important for applications that store all user data in a single long-lived database. They do not discuss wear-levelling for the reserved erase blocks storing frequently-erased file headers; our solution only adds regular file data, relying on the file system's implementation of wear-levelling.

It is difficult to compare Lee et al.'s solution with ours in experiments because their solution was not implemented. They simulated their algorithm by assuming files were modified at most twice; our examination of Android phone data found that a third of all chunks were file headers, suggesting much more frequent modifications. They intend to erase an erase block whenever a file is removed; however our data indicates that Android phones delete nearly 10000 tiny cache files a day—securely deleting each results in frequent erase block erasures that could be easily batched.

6. CONCLUSIONS AND FUTURE WORK

We have considered deletion latency for log-structured file systems and showed that there is no guarantee of deletion on such file systems. We presented three user-level solutions for

secure deletion on YAFFS file systems: purging, ballooning, and a hybrid of both.

Our simulations of FTL using the provided kernel driver are limited. It would be useful to perform our experiments using the exact implementations of various popular FTL drivers in practice, or to access the raw flash memory behind an FTL in a SD card to know exactly what the erase block allocations per hour, deletion latency, and purge costs are as a result of using our solutions. Differences in implementation may even render our approach unsuccessful: Wei et al. [30] report the accessibility of deleted data after filling the entire contents of some particular solid state drives. As a particular case study, it would be useful to determine the applicability and optimal ballooning parameters for the FTL used on iOS devices (iPhones, iPads, etc.).

Our discrete event simulator uses live data from lengthy usage of an Android mobile phone, building a model of each application's writing patterns. It would be useful to extend this study by collecting large samples of many different users, thus getting a richer understanding of mobile phone writing behaviours under a variety of use conditions. The model could then be reduced to a small number of variables and distributions that control the frequency of the following events: a new page is written, an old page is overwritten and an old page is deleted. These distributions could be composed by appropriately-weighting subdistributions to model the writing behaviour of different user types.

In conclusion, we have presented useful user-level solutions for secure deletion of data from flash memory, and have evaluated their effectiveness in terms of wear on the flash memory, as well as power consumption and time. We have implemented our solutions and made an application available on the Android marketplace.

7. ACKNOWLEDGMENTS

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

8. REFERENCES

[1] A. Ban. Flash file system. US Patent, no. 5404485, 1995.

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

[3] M. Blaze. A Cryptographic File System for Unix. *ACM CCS*, pages 9–16, 1993.

[4] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[5] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. *Proceedings of the First Dutch International Symposium on Linux*, 1995.

[6] Charles Manning. How YAFFS Works. 2010.

[7] 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*, 55(5-6):332–343, 2009.

[8] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37:138–163, 2005.

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

[10] G. Goodson and R. Iyer. Design Tradeoffs in a Flash Translation Layer. *Proceedings of Workshop on the Use of Emerging Storage and Memory Technologies*, January 2010.

[11] Google, Inc. Google Nexus Phone.

[12] G. P. Guard. gpg(1) - Linux man page.

[13] M. A. Halcrow. ecryptfs: An enterprise-class cryptographic filesystem for linux. 2005.

[14] W. F. Heybruck. An Introduction to FAT 16/FAT 32 File Systems. 2007.

[15] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. 1998.

[16] D. Jagdmann. srm - Secure File Deletion for POSIX Systems.

[17] N. Joukov, H. Papaxenopoulos, and E. Zadok. Secure Deletion Myths, Issues, and Solutions. *ACM Workshop on Storage Security and Survivability*, pages 61–66, 2006.

[18] N. Joukov and E. Zadokstony. Adding Secure Deletion to Your Favorite File System. *Third International IEEE Security In Storage Workshop*, pages 63–70, 2005.

[19] J. Lee, S. Yi, J. Heo, and H. Park. An Efficient Secure Deletion Scheme for Flash File Systems. *Journal of Information Science and Engineering*, pages 27–38, 2010.

[20] L. Marttala. File Shredder Pro. 2010.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New ext4 Filesystem: Current Status and Future Plans. 2007.

[22] Micron, Inc. Design and Use Considerations for NAND Flash Memory Introduction. 2006.

[23] Micron Technology, Inc. Technical Note: Design and Use Considerations for NAND Flash Memory. 2006.

[24] Open NAND Flash Interface. Open NAND Flash Interface Specification, version 3.0. 2011.

[25] C. Plumb. shred(1) - Linux man page.

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

[27] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[28] Samsung, Inc. Samsung Galaxy S Phone.

[29] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and Repairing Write Performance on Flash Devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 9–14, New York, NY, USA, 2009. ACM.

[30] M. Wei, L. M. Grupp, F. M. Spada, and S. Swanson. Reliably Erasing Data from Flash-Based Solid State Drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies*, pages 105–117, Berkeley, CA, USA, 2011.

[31] D. Woodhouse. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium*, 2001.

[32] H. Yun. Flash memory management method and flash memory system. US Patent, no. 6938116, 2007.