Apps Can Quickly Destroy Your Mobile's Flash: Why They Don't, and How to Keep It That Way

Tao Zhang
The University of North Carolina at Chapel Hill
zhtao@cs.unc.edu

Aviad Zuck
Technion – Israel Institute of Technology
aviadzuc@cs.technion.ac.il

Donald E. Porter
The University of North Carolina at Chapel Hill
porter@cs.unc.edu

Dan Tsafrir
Technion – Israel Institute of Technology &
VMware Research
dan@cs.technion.ac.il

ABSTRACT

Although flash cells wear out, a typical SSD has enough cells and sufficiently sophisticated firmware that its lifetime generally exceeds the expected lifetime of its host system. Even under heavy use, SSDs last for years and can be replaced upon failure. On a smartphone, in contrast, the hardware is more limited and we show that, under heavy use, one can easily, and more quickly, wear out smartphone flash storage. Consequently, a simple, unprivileged, malicious application can render a smartphone unbootable ("bricked") in a few weeks with no warning signs to the user. This bleak result becomes more worrisome when considering the fact that smartphone users generally believe it is safe to try out new applications.

To combat this problem, we study the I/O behavior of a wide range of Android applications. We find that high-volume write bursts exist, yet none of the applications we checked sustains an *average* write rate that is high enough to damage the device (under reasonable usage assumptions backed by the literature). We therefore propose a rate-limiting algorithm for write activity that (1) prevents such attacks, (2) accommodates "normal" bursts, and (3) ensures that the smartphone drive lifetime is longer than a preconfigured lower bound (i.e., its warranty). In terms of user experience, our design only requires that, in the worst case of an app that issues continuous, unsustainable, and unusual writes, the user decides whether to shorten the phone's life or rate limit the problematic app.

CCS CONCEPTS

• **Information systems → Flash memory**; • **Security and privacy → Mobile platform security**.

ACM Reference Format:
Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. 2019. Apps Can Quickly Destroy Your Mobile's Flash: Why They Don't, and How to Keep It That Way. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3307334.3326108

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326108

1 INTRODUCTION

Smartphones typically include flash-based storage, because flash offers benefits such as fast random access, shock resistance, high density, and decreasing costs. A main drawback, however, is that flash cells can tolerate only a limited number of writes (i.e., program/erase cycles) before becoming unusable. Vendors therefore apply various methods to increase the lifetime of flash packages [19, 41, 44, 48, 55, 59, 69, 84, 92, 93, 147], primarily by (1) provisioning more physical than logical flash cells, and (2) using a sophisticated firmware layer to spread the wear across these cells.

One can determine how many physical flash cells are needed to support an expected device capacity in its logical block address (LBA) space, and a lifespan of, say, three years, by using a simple, back-of-the-envelope calculation: take the expected number of writes for the advertised LBA space over a three year period, and divide it by the number of per-cell program-erase cycles that individual cells can tolerate [36, 40, 42, 67, 97, 106, 120, 129, 131, 133]. Consequently, most high-end SSDs have multi-year warranties [76, 120, 124, 131, 133]. Field studies on the lifetime of consumer-grade SSDs demonstrate that even these relatively inferior devices can last for years under strenuous workloads before failing [36, 129]. Section 2 provides background on flash storage and discusses how the lifetime of SSDs is estimated and managed. Because the lifetimes of high-end SSDs are warrantied, we speculate there is a common, but inaccurate perception, both in academia [42, 67, 106] and among users [40, 97, 129], that flash endurance is effectively a non-issue for *any* flash device.

The first contribution of this paper is **an empirical evaluation of the lifespan of flash storage on a range of mobile phones**. Our results in Section 3 demonstrate that flash lifespan is not a solved problem in this context. In Section 4 we explain how to exploit this vulnerability using a simple, unprivileged app that can easily issue a lifetime's worth of writes in a few short weeks, rendering the phone unbootable, or "bricked". Moreover, we show how this malware can throttle its write activity so as to remain undetected by the user, until the phone stops working. We call this application the "wear-out app", or WAPP. We measure the lifespan and minimal time to wear-out several different mobile flash devices, at various price points, and the results are consistent: wearing out mobile flash is much easier than wearing out a regular SSD. This

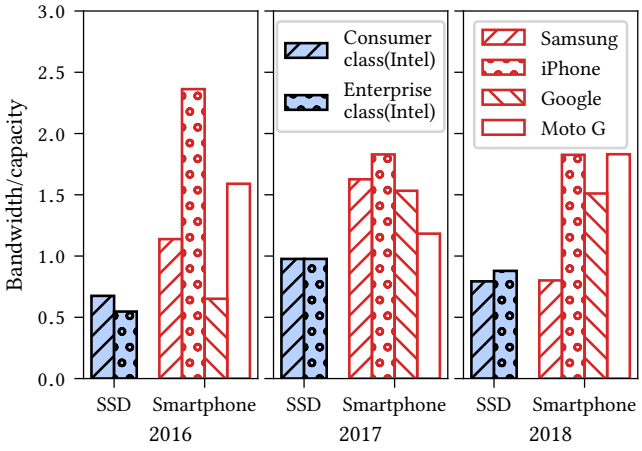


Figure 1: *The bandwidth-to-capacity ratio over time for smartphones and Intel's SSDs. The ratio of bandwidth-to-capacity for an SSD tends to balance over time, whereas phones can become dangerously skewed toward high bandwidth and low capacity.*

paper adds additional experiments and data to our preliminary results on wear-out in mobile devices [150].

The problem stems from an imbalance between the write bandwidth and the capacity of the drives, illustrated in Figure 1 for SSDs [137] and smartphones [13, 15, 16, 22–26, 54, 79]. Technological advancements, such as faster interfaces or improvements in flash technology, may temporarily change the balance. Over time, the bandwidth-to-capacity ratio evolves in a manner that tends to even out for SSDs. In contrast, the lower capacity of smartphones, coupled with more limited hardware and less sophisticated firmware, tend to create a dangerous imbalance: smartphone apps can easily issue a lifetime's worth of writes in a short time.

This problem is alarming because mobile ecosystems have created an arguably false sense of security—that trying out an app will not permanently damage your device. Mobile operating systems (OSes) have a tighter security model than a desktop OS [27, 28, 34], and app stores have some review process, even if it can be lax in practice [110, 134, 152]. As a result, users often give little-to-no consideration before downloading third-party applications [65, 74, 107], rooting their devices, and trusting applications. Yet neither the vetting process for these applications, nor the underlying OS storage abstractions are designed to manage permanently consumable resources [114] and prevent wear-out attacks. If a phone's flash is worn out, whether maliciously or not, it is not user-serviceable; in terms of repair cost and data integrity, *destroying the flash is tantamount to destroying the device.* To make things worse, unlike their full-fledged SSD counterparts, the specifications, performance characteristics, and lifetime estimates of mobile storage devices are not made public. Vendors make no claims about the longevity of these devices. Furthermore, the tight hardware budget of these devices limits the applicability of common lifespan extension techniques.

The second contribution of this paper is a **characterization of the write I/O behavior of a wide range of benign smartphone apps**, with a particular focus on write-intensive apps (Section 5). Our findings show that many applications issue minimal I/Os (well below 200 KiB/s) but some applications can indeed generate intense write bursts. In practice, these bursts are not sustained long enough to be problematic, except in the most extreme cases.

The third contribution is **an empirical explanation of why benign apps do not wear out smartphone flash in practice**. Although write-intensive apps can issue significant bursts of I/O activity, when averaged over a typical day of use, the average write bandwidth can be sustained for several years. As supported by recent studies [33, 49], we assume that a typical user (1) uses apps on their smartphone for two hours a day; and (2) this usage is spread among a range of apps from different categories.

Because I/O-intensive malicious apps can pose a threat to users, Section 6 contributes **a defense against dangerous I/O behaviors and the WAPP attack**, using unmodified, commodity hardware. Based on the characterization of expected app behavior, we design a policy such it is *never felt by users in the vast majority of cases*, unless: (i) they are under attack, or (ii) they exceed reasonable usage assumptions. Our solution begins by setting a target lifespan for the device, say 2–3 years. The OS tracks the remaining total writes and lifespan, and periodically allocates available writes to applications. Some writes are given to applications directly, and some are held in a slack pool. As the slack writes are consumed, heavy consumers are flagged and potentially rate-limited.

Our analysis shows that there are, admittedly less likely, scenarios where a user could shorten the lifespan of her phone through extremely heavy, intended use. For example, Final Fantasy is a write-heavy application, and one who plays for 16 hours per day every day could shorten the lifespan of the phone. Unfortunately, there is no general way to distinguish between this situation and an attack; in any case the result is the same: the user has installed an app that will shorten the lifespan of the phone. In terms of user experience, we believe this is the point at which the user should be warned that one or more apps will shorten the lifespan of the device, and she can decide to remove the app, rate-limit the app, or accept risks and continue. For reasonable use cases, our defense does not disrupt normal apps, and can resist the WAPP attack.

Although this paper focuses on smartphones, we believe this concern generalizes to any mobile or embedded device, from a smartwatch to internet-of-things gadgets to critical infrastructure, such as smart meters. The requirements are simply (1) less expensive flash devices and (2) the ability to execute user-level code, either via an app store, or perhaps loaded through another exploit. Although the main harm in the case of a smartphone user is the cost of a phone and its data, one could imagine more drastic consequences in rendering critical infrastructure or smart medical devices unexpectedly inoperable. Thus, it is likely that wear management will be a growing concern in new classes of devices that integrate small computers with other aspects of daily life.

2 BACKGROUND

NAND Flash. Flash packages are composed of large arrays of serially connected floating-gate cells [59]. Data is written by charging cells to a target voltage levels. The logical value of a cell is read by comparing its voltage to a voltage threshold. Data is read and written at page units, typically 4–16 KiB. Pages are further grouped into blocks, typically 256–4096 KiB in size.

Flash memories cannot update data in-place [61, 145, 154]. Updating a page requires that it first be erased, which reinitializes the voltage levels of the cells at the coarser granularity of a block. This process of writing (or programming) and erasing is called a Program/Erase cycle, or P/E cycle.

Because flash memories prohibit updates in-place, the SSD firmware, called a flash translation layer (FTL) must reclaim and move live data to different physical blocks, so that blocks can be erased and accept new data. Consequently, host-level writes may result in internal garbage collection overheads, where a larger amount of live data must be internally rewritten in order to erase a block for new writes, or, firmware-level write amplification [52, 69].

Flash blocks can endure a limited number of P/E cycles. Over time, additional electrons become trapped in the floating gate, causing uncorrectable bit errors. Some number of bit errors can be corrected transparently with parity checks, but a flash block can age to the point that it generates more bit errors than parity can correct. This problem is exacerbated as flash memories become denser. Earlier generations of flash chips, storing 1 bit per cell, endured up to 100K P/E cycles. Modern chips, storing 2–4 logical bits per cell, can only endure 1–3K P/E cycles [43, 62, 125].

A plethora of methods have been proposed to improve the lifetime of flash storage, primarily by evenly distributing wear over flash blocks [19, 39, 59] and using error-correcting codes [41, 55] to compensate for bit errors. Additional methods for extending SSD lifetime include high and low-endurance flash hybrids [46, 93], intra-SSD redundancy [86], data reduction [48, 140, 155], and reshaping [80, 84, 92]. Cai et al. [43] provide an extensive review of flash lifetime extension techniques.

SSD Lifetime Estimation in the Wild. Lifetime estimates of flash packages allow us to speculate on the lifetime of SSDs. Conservatively assuming that various optimizations in hardware, firmware, and software balance out ill-behaved user workloads, one can assume that the SSD can endure at least as many rewrites as its underlying storage media. For a typical consumer-grade SSD, this means 3K rewrites [125] of the drive's advertised capacity, or, three drive writes per day over three years. Several recent studies by datacenter operators corroborate this calculation and independently conclude that various SSDs last for years [103, 108, 125], despite relatively strenuous usage patterns. Other studies also concluded that SSDs can write petabytes of data before failing [36, 129]. Such findings lead many consumers to believe that SSDs last for extremely long periods of time [40, 42, 67, 97, 106, 129]. Vendor drive warranties reflect more concrete and conservative estimates of SSD usage and lifetime expectations; vendors also expect modern flash drives to last for years under typical usage patterns [76, 120, 124, 131, 133].

The combination of strong vendor warranties and the experience of commercial SSD products lasting for years under strenuous usage leads to a common perception that the endurance of flash-based drives is effectively a non-issue.

3 MEASURING WEAR-OUT

This section measures the performance characteristics of several mobile flash devices, under random and sequential write workloads. We demonstrate that these workloads can wear out the devices quickly, despite differences in the underlying hardware.

Device (Storage info.)	OS
eMMC 8GB (THGBMBG6D1KBAIL eMMC 5.0 [132])	Ubuntu 16.04 (kernel 3.14.79)
eMMC 16GB (iNAND 7030 eMMC 5.0 [123])	
μSD 16GB (Kingston SDC4/16GB [90])	
Samsung S9 64GB (Toshiba THGAF8G9T43BAIR UFS 2.1) [18]	Android 9
Samsung S6 32GB (Samsung KLUBG4G1BE-E0B1 UFS 2.0) [10]	Android 6.0.1
Moto E 8GB (Samsung QN1SMB eMMC 4.5) [8]	Android 5.1
BLU 512MB (K524G2GACH-B050 NAND Flash) [6]	Android 4.4
BLU 4GB (TYC0FH121638RA eMMC) [5]	

Table 1: *Evaluated mobile storage devices.*

3.1 Evaluation Setup

Our experiments use a range of prevailing mobile storage solutions on the market, including eMMC, UFS, and MicroSD (μSD) [37, 77, 78]. Table 1 lists the devices used in our experiments. The first class of measured devices includes two external eMMC chips and a conventional μSD card, also commonly used as external, additional storage in smartphones. All experiments on external eMMC chips were performed using the ODROID C2 platform [64].

We also measure smartphones. Most experiments use a mid-range Moto E 2nd Gen smartphone [8] and two high-end Samsung smartphones, S6 and S9 [10, 18], which have UFS [78] storage devices. We also examine two budget smartphones, referred to as "BLU 512MB"[6] and "BLU 4GB"[5], to see how cheaper hardware affects lifespan. Smartphone experiments are conducted using their stock Android systems and default Ext4 file system, except the Moto E 8GB, which uses F2FS[92], a flash-friendly file system. To study the effect of file systems, we ran the same experiments on two Moto E 8GB phones, one with F2FS and the other with Ext4. All experiments in this section use the `fio` benchmarking tool [7].

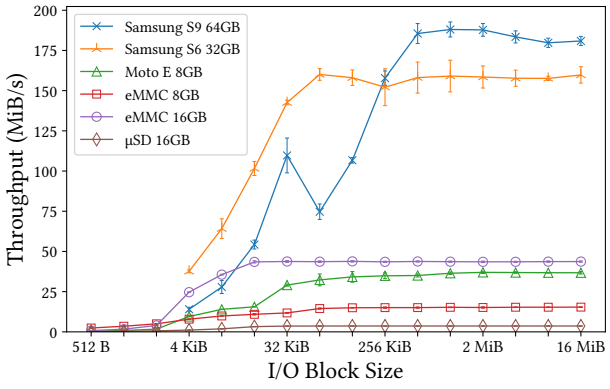
3.2 Performance Characteristics

We first explore the I/O characteristics of eMMC chips, to understand their behavior and find the most problematic I/O access patterns for these devices. Figure 2 shows the throughput of a sequential and random I/O microbenchmark on an empty device, with varying synchronous request sizes. For brevity, we omit read results, which were similar to the write results.

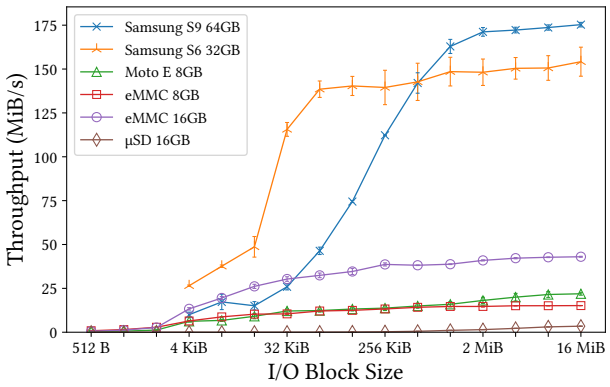
Our results demonstrate that *eMMC chips are faster than the μSD card in all I/O patterns*, contrary to a common conception that eMMC chips are essentially repackaged μSD cards [53, 88]. Furthermore, random and sequential write performance of eMMC chips are similar and generally scale linearly until the device is saturated. We conclude that *the I/O performance of modern eMMC devices hinges on request size*, as larger requests better utilize internal hardware parallelism [19, 82, 144] and reduce storage software overheads.

3.3 External eMMC Wear-out

Based on the micro-benchmark results, we hypothesize that eMMC chips can serve a large volume of intense write I/O activity within



(a) *Sequential Write*



(b) *Random Write*

Figure 2: *Mobile storage write throughput as a function of I/O write size (higher is better).*

a relatively short span of time. Such write activity can quickly consume the P/E cycle quota of the underlying flash cells.

To test this hypothesis, we use the eMMC lifetime estimation indicator [77]. This standard indicator is reported by the device firmware, as a number from 1 to 11; when the indicator has value n, it means the chip's consumed lifetime is between $(n-1)*10\%$ and $n*10\%$. The specifics of how the device's lifetime is estimated are proprietary and most likely vendor-specific. An indicator value of 11 does not necessarily mean that the chip will immediately stop functioning; rather, a value of 11 signifies that, according to the firmware's estimation, the chip has exceeded its maximum guaranteed lifetime, may introduce uncorrectable errors in stored data, and should be considered unreliable [77]. A chip at this state could stop functioning at any moment.

We repeatedly issued 4 KiB writes in randomly selected regions of four 100 MiB files on each external card, and measured the wear-out indicator. Because flash cannot update in place, writing to the same file and logical block on the device causes a new write to a new physical location in the storage device. Notably, this simple I/O pattern induces no write amplification, but simply forces repeated re-programming of its flash pages. As a result, increasing the size of the device's over-provisioned space cannot significantly reduce

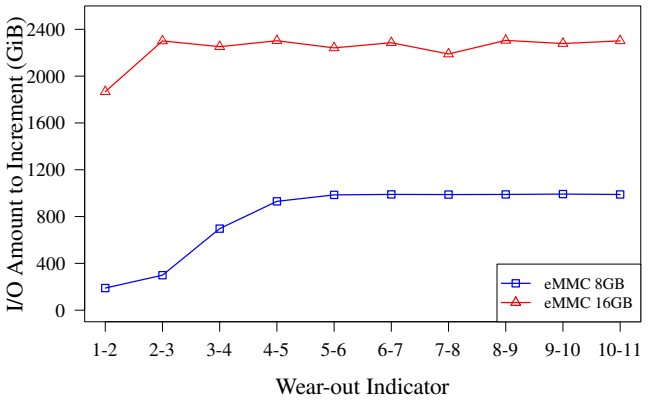


Figure 3: *Amount of I/O needed to increment the wear-out indicator on two external eMMC chips.*

the wear-out effect of this access pattern without exceeding the device's price point.

The results in Figure 3 show that the required I/O volume to increment the wear-out indicator is mostly constant throughout the lifetime of the devices. The first few increments tend to happen faster, which we hypothesize is attributable to initial software installation and vendor testing. In total, it takes a maximum of 992 GiB to increment the wear-out level by 10% in the 8GB eMMC chip. Interestingly, this result is roughly 3× lower than the "back-of-the-envelope" 3,000 complete rewrites one would expect to wear out the device [43, 62, 125]. Moreover, at a maximum throughput of 20 MiB/s, one could write this volume of data in 140 hours (6 days). For the 16GB eMMC chip, 23 TiB of writes are required to reach end-of-life after 164 hours (7 days) at 40 MiB/s.

3.4 Smartphone Wear-out

In prior experiments [150], we found that both small, random writes (as in the previous subsection), and large, sequential writes had the same impact on flash lifespan in terms of total bytes written to increment the wear indicator. However, we found that large, sequential writes could be issued to the device faster, and realize higher throughput on the underlying device. Thus, in this section, we issued continuous, large, synchronous, sequential write operations to wear out mobile flash devices. All smartphone wear-out experiments were run using a simple, malicious app (§4).

Our key result, illustrated in Figure 4, is that *the storage device in all phone models can be worn out in a matter of days to a few weeks.* Here, we show only the time to get to level 6 (except S9, which had only reached 5 at the camera-ready due date); we took every device all the way to 11, except the S6 and S9. Timing results vary, even for the same device model, most likely due to firmware-specific behaviors and optimizations, as well as the maximum throughput the malicious app can achieve (e.g. by using native library support which can better utilize the higher bandwidth provided by high-end storage chips, time spent in 3=>4 decreased significantly on the Samsung S6 phone we tested, as shown in Figure 4). Specifically, time consumption in 1=>2 on Samsung S9 bursts to ~98 hours due to a non-optimized WAPP in terms of I/O size, which is easily fixed

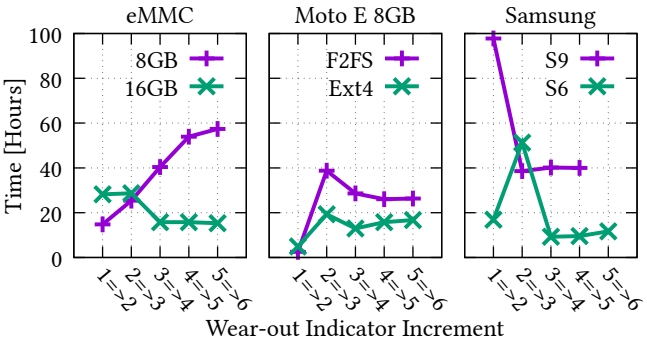


Figure 4: *Time to increment wear-out indicators on two smartphone models and two external eMMC chips.*

in following experiments. In the two budget smartphones "BLU 512MB" and "BLU 4GB" the flash storage chips did not provide reliable wear-out indications. However, both phones were bricked within two weeks. As for the two Moto E phones, both models were bricked two weeks after reaching the maximum wear indicator value. Notably, and consistent with Figure 1, the results are not sensitive to the capacity of the mobile storage device being used. This result is attributable to vendors increasing flash capacities by either using high-density, low-endurance memories (e.g., TLC flash [62]), or by scaling the number of flash units operated in parallel. When combined with higher-bandwidth interfaces, the result is a larger capacity device that is almost as susceptible to wear-out attacks as its smaller predecessors.

Consistent with Figure 3, the time to get to the first few wear indicators is typically less than the rest. In general, the measurements after level 5 are more consistent for smartphones as well as individual chips.

4 WEAR-OUT ATTACK

This section describes our threat model and a proof-of-concept wear-out attack that works on real-life mobile systems using only a simple, virtually permission-less app. We also discuss how our app can evade various detection mechanisms.

4.1 Threat Model

We consider a benign system with a file system on a flash-based mobile storage device (e.g., an eMMC chip), which provides users the ability to install executable applications. The mobile device is warrantied for L years (e.g., 2 years) and can sustain W terabytes of flash-level writes. The device supports a maximum write bandwidth of B_{max}, which, if applied constantly, will wear out the device before L. We assume the file system respects synchronous I/O, i.e., an `fsync()` immediately pushes dirty file data onto the underlying storage device. We also assume user apps may write at least 100 MiB to either private or public files.

We assume the adversary's code is installed and can run on the victim device. We envision a common case where the adversary's code hides its I/O-intensive nature from the end-user, such as by disguising itself as a benign app or running as a "trojan horse" in an advertising library in another useful app. We do not assume the app has any special privileges, nor do we require any other exploit of the system.

4.2 Implementation and Avoiding Detection

We implemented a simple attack application, called WAPP. WAPP continuously rewrites 100MiB files in the app's private storage area, which is allocated to the app by default. The app rewrites data with large, sequential I/Os, in order to realize the maximum throughput B_{max} supported by the device. WAPP is only 963 lines of code.

Next, we experimented with the difficulty of hiding a malicious, I/O intensive application on Android. We observe two likely indicators of a problematic app that would manifest before the device is bricked. First, Android monitors energy consumption, but only when on battery. Thus, we can evade detection via power monitoring by only running I/O intensive work when the phone is charging; the app can tell when the phone is charging. Also note that phones today spend a significant portion of time charging because most devices do not have a removable battery. Second, even with proper social engineering, continuously running WAPP in the foreground may alert users. We therefore run WAPP in the background. Most Android versions show apps currently running in background or as services (cf., `ps` on Unix). We observe that the refresh time for this monitor is around one second, and the app can detect when the screen is lit. Because most phones spend a significant fraction of the day charging with the screen disabled, WAPP can effectively evade monitoring by suspending its malicious I/O activity when the screen is turned on. In summary, most phones spend a significant fraction of the day charging with the screen disabled; even a stealthy version of this experiment could brick a phone within some reasonable factor of the time in these experiments.

We note that continuously running malicious applications may cause the phone to abnormally heat up, which may raise the suspicion of users, though such extent of heating may be attributed to heat generated by the charging process. We leave the exploration of this, and other possible detection methods for future work.

4.3 Permissions and Capabilities

One telltale sign of a malicious app is requesting more privileges than would seem needed for the advertised functionality [38, 136]. The only permission WAPP requires is the ability to read and write its own files. This permission is usually considered fundamental and harmless.

WAPP also requires several additional capabilities. In order to operate when the screen is off WAPP currently utilizes Android's WakeLock mechanism [30]. To avoid suspiciously high power consumption, WAPP detects the charging state of the devices and listens to relevant system broadcasts on charging states [31]. These capabilities are granted by default to all apps. That said, our use of WakeLock may be restricted by the most recent version of Android [4]. To realize the same behavior without the use of WakeLock, we can use periodic tasks, such as scheduled jobs or alarms, to repeatedly re-initiate malicious activities in the background [1, 35]. In conclusion, WAPP can mount an effective wear-out attack with essentially no permissions beyond those routinely granted to any app.

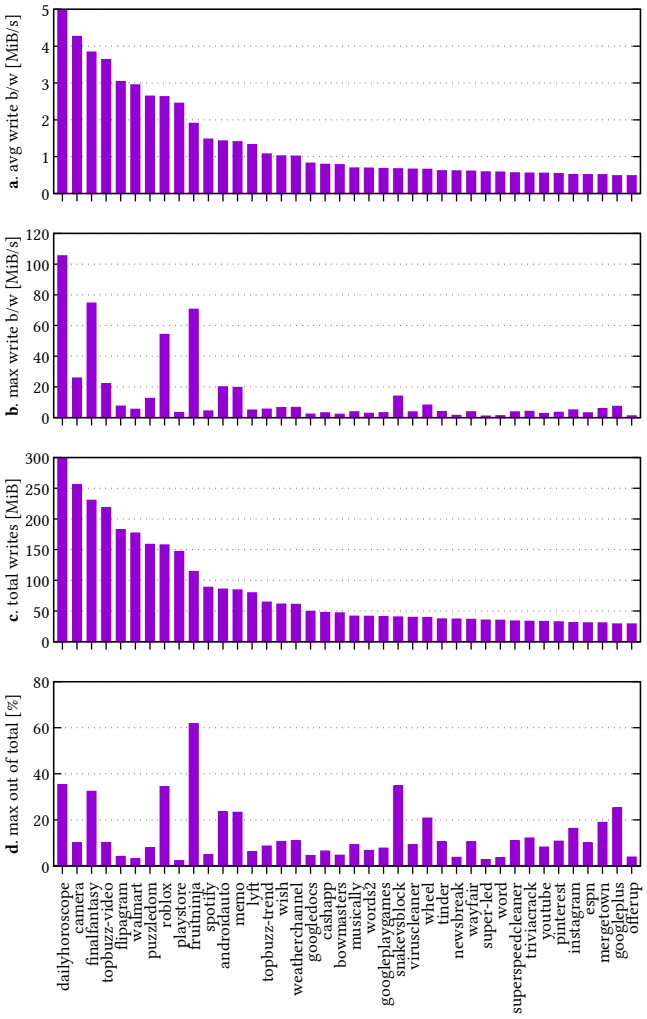


Figure 5: *I/O characterization of the top 40 write-intensive apps, during a one minute run.*

5 MOBILE APP I/O CHARACTERIZATION

A goal of this work is to defend against premature wear-out and wear-out attacks without harming normal app behavior or user experience. Thus, an important question to ask is, what level of write bandwidth is typically used by current mobile apps? With this information, we can assess the potential to destroy the device in practice, as well as whether one can differentiate benign and dangerous write behavior. This section presents a measurement study of disk write I/O rates for popular apps, especially during write-heavy activities.

5.1 Measurement Setup

We perform our measurements on the Samsung S6 model used in Section 3. The phone uses the default Android 6.0.1 operating system, and has 32 GB of internal storage and 3 GiB of DRAM. In our experiments we test two sets of usage scenarios. The first set is comprised of 27 preloaded apps (e.g., camera, voice recorder) and the top 150 free applications in the daily download chart of Google's Play store from App Annie [33]. From this group we excluded ten apps that would not install or run properly for technical issues, such as failing to access the network, or missing external dependencies, such as needing to connect to Google Home device manager. Eight more phone customization and keyboard apps were excluded because they are irrelevant to I/O activity. Three apps were removed from the store by the time we evaluated them, and two were unavailable for download because of geographical restrictions. A second set of usage scenarios is comprised of I/O-intensive workloads and apps including an FTP server, file copies, and device backup and restore.

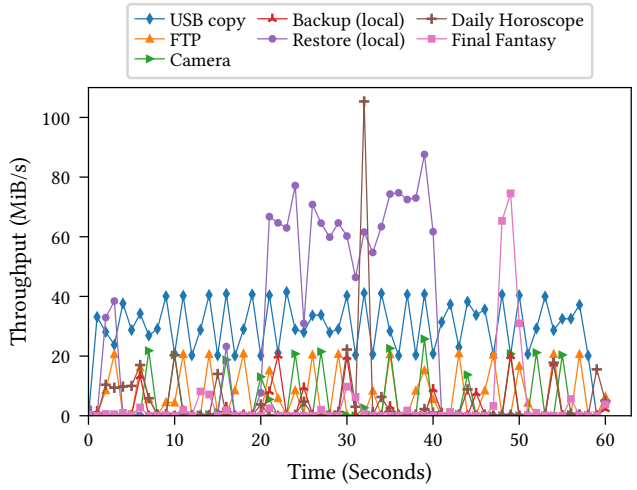
We used the following method to measure the I/O activity of each app. First, we installed the app from the Play store (when necessary), opened it, and performed any necessary setup and registration. We then closed the app and any other running apps. We started the application until it fully loaded and then cleared the file cache. At this point we start monitoring disk I/O activity using `/proc/diskstats`. For each app, we identify actions expected to create the highest write I/O volume, such as recording video at highest quality; we execute these operations for a period of 60 seconds. Otherwise, we manually operate the app as a normal user, such as playing a gaming app, or scrolling and posting on social media apps. We repeated this procedure three times, using data from the most write-intensive run. Finally, we uninstalled the app and removed it from the device.

Notably, our goal is to understand the limits of typical app write I/O behaviors during *normal* usage behaviors. Automated tools [2, 32] are often unsuitable for testing apps with non-deterministic behavior. Another issue is that gaming apps use custom libraries that are often not inter-operable with automated testing tools. Gaming apps constitute close to half of our test set. For these reasons, we measure the apps under manual operation, rather than in an automated framework.

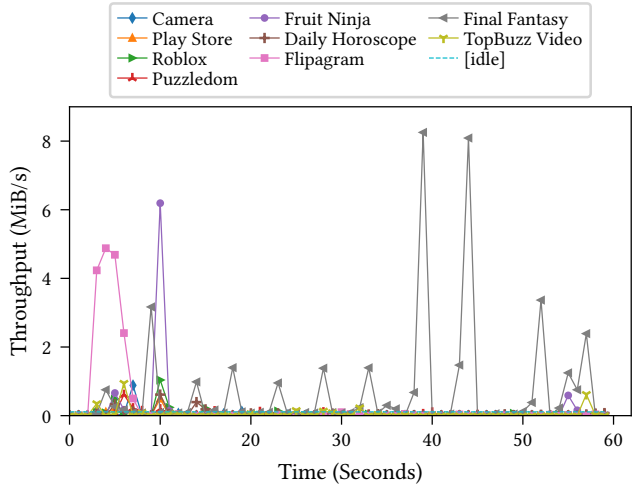
5.2 Popular Apps Characterization

Figures 5a and 5b show the average and maximum write I/O throughput for the first group of applications (preloaded and popular free apps). For brevity, we only illustrate results for the 40 apps that displayed the highest average I/O throughput. Surprisingly, the most write-I/O-heavy app was "Daily Horoscope", probably due to poorly written code in earlier versions (experiments with newer versions of Daily Horoscope observed significantly lower write I/O volumes). The preloaded Camera application also issued relatively high disk write volumes on average. This result is expected, since we operated the camera by repeatedly recording videos with the highest configurable picture quality. Notably, *even the most write-heavy apps still utilize on average less than 5% of the device's maximum write throughput* (160 MiB/s sequential).

Various workloads can cause apps to display bursty I/O behavior. Figure 5c shows the total I/O volume issued by each app over the measurement period. Figure 5d reports how much the burst of maximum throughput contributed to the write volume issued by an app—for each app, effectively selecting the writes for the highest bandwidth second in Figure 5b, and dividing this by the total I/Os in Figure 5c. The results show that many apps issue high write



(a) *Write I/O throughput over 60 seconds for three write-heavy usage scenarios and the three most write-heavy popular apps.*



(b) *Background write I/O throughput over 60 seconds for 10 write-intensive apps.*

Figure 6: *App I/O characterization results*

volumes in short, bursty periods, presumably due to persisting of cached files and discarding [66] temporary files[1].

5.3 Write-heavy Applications

Apps excluded from this study may still issue exceptionally high write volumes to the device. Therefore, we further tested known apps that are likely to issue such high write volumes during normal operation. We measure an FTP server app serving a file put request over LAN, copying large files from a PC connected via the USB cable, and backup/restore operations using Titanium, a popular backup app. For comparison we also include results from the three most write-intensive applications tested in the first app set (§5.2).

[1] Android, as well as Linux, counts discard operations as writes to the device. Firmware discard handling is implementation-specific and proprietary, and it is unknown if and when discards translate to flash writes. Therefore, our results are an upper bound on each app's related wear.

App	Avg. throughput (MiB/s)	Daily usage (Hour)
USB copy	29.74	1.18
FTP	6.39	5.50
Camera	4.26	8.24
Backup (local)	2.30	15.25
Restore (local)	23.29	1.51
Daily Horoscope	4.98	7.05
Final Fantasy	3.84	9.15

Table 2: *Average I/O throughput for apps in Figure 6a, and average daily usage required to shorten the device's lifetime.*

Figure 6a shows the measured write I/O throughput (excluding discard operations) for each workload over a 60 second run, where $B = 1.46$ MiB/s is the maximum average bandwidth level that does not shorten device lifetime. For each app, Table 2 details the average throughput and estimated length of daily usage that would shorten the device's lifetime. The results demonstrate that *most write-intensive workloads' average daily write throughput is not high enough to shorten the device's expected lifetime.*

5.4 Background I/O

Thus far, we have characterized write I/O activity for apps when actively operating them in the foreground. Previous studies found that the average mobile device is active for up to two hours daily and that most app sessions last no more than a few minutes [33, 45, 58]. Therefore, we investigate app's background I/O activity.

We define background operation as app activity occurring when the screen is off, without any user interaction. We first establish a baseline by measuring background I/O activity when no app is active, and only preloaded apps are installed during a 5-minute idle session. In this state, the average I/O throughput is 0.11 MiB/s.

Next, we measure app background I/O activity when the screen is closed and the device is left idle for one minute. We measure when the device is charging, in order to avoid entering power-saving modes that significantly reduce app activity [56]. Figure 6b illustrates our results. With the exception of Final Fantasy, which appears to perform periodic checkpoints to disk, *most apps cause little to no background I/O activity.*

5.5 Discussion

From the results thus far, one may be concerned that some apps can shorten the device's lifetime even under normal behavior. The S6 model used in our experiments has 32 GB of capacity and a maximum two year warranty [122]. According to our measurements, the minimum volume of writes required to consume 10% of the S6's maximum wear is 8.8 TiB. Conservatively assuming that the device is able to sustain a maximum of 88 TiB writes over the course of its lifetime, then the device is capable of serving writes from all apps at an average write throughput of $B = 1.46$ MiB/s. In our experiments only 10 apps demonstrated such behavior over time when operated in the foreground (and none in the background). However, usage scenarios that require nonstop operation of such write-intensive apps appear unlikely.

Modeling wear. To further illustrate the small amount of wear due to normal app behaviors, we model the amount of wear each app will cause to the mobile storage device under typical usage.

First, we parameterize the time users spend using mobile apps, estimated by recent studies at an average $App_{time} = 2.3$ hours daily [45, 49, 57]. Unfortunately, we could not find publicly available data on the average time consumed by typical users for each specific app in our test set. Instead, we approximate the time spent on each app using T_{cat}, the fraction of time spent on each app's category according to a recent study [49], by conservatively assuming it is the only app of its category being used (e.g., the only game being played).

We calculate each app's wear by taking into account its average I/O throughput IO_{thr}, as observed in our experiments. We conservatively assume that the app is used every day during the device's warrantied lifetime L. Finally, we divide the result by W, the amount of I/O that the device can serve before it reaches the vendor's estimated wear limit. In summary, app-related wear can be modeled as:

$$Wear = \frac{App_{time} \times T_{cat} \times IO_{thr} \times L}{W}$$

For example, we can calculate the wear induced by the "Final Fantasy" app ($IO_{thr} = 3.83$ MiB/s) in our S6 device ($W = 88$ TiB, $L = 2$ years). We assume apps are being used for $App_{time} = 2.3$ hours a day, $T_{cat} = 10\%$ of which is used for games [49]. Even under our conservative model this write-intensive gaming app would consume only 2.5% of the device's lifetime over a two-year period. In summary, we conclude that *under reasonable usage assumptions, the vast majority of apps do not display I/O behavior which causes significant wear on the device.*

6 MANAGING WEAR

Existing systems do not protect against apps issuing destructive volumes of writes. This section describes and evaluates a policy that protects against the unlikely event of writes that might harm the device. We reiterate that increasing device capacities or overprovisioning more space does not resolve the fundamental problem that an app can deplete the device's flash P/E cycles before the expected lifespan of the overall device. Instead, mobile system designers should follow policies that given a total budget of P/E cycles for the device's lifespan ensure that these cycles are not being consumed too quickly. We present one possible solution in this design space that meets this goal.

6.1 Wear Management Policy

We model our device as having a total number of block writes (W) that it accepts before being at risk of failure, and a target remaining lifespan L (e.g., for a new device with a warranty of 2 years, $L = 2$). As W is not typically provided by vendors, it can be be empirically estimated or given a conservative lower bound. Over time L decreases. W decreases as writes are issued to the device. Table 3 enumerates the important parameters used in our policy, and expected values for the S6 phone.

Our algorithm dynamically tracks $B = W/L$, or the average write bandwidth the device can sustain without violating the goal of operating for L years (§5.5). The high-order goal is that apps should not be able to consume more than B average bandwidth over the device's lifespan. Intuitively, we start with a naïve rate-limiting scheme: all apps may write freely until the apps' cumulative bandwidth (total writes divided by the time since installation) reaches our target B, at which point all apps are throttled to B. The next few paragraphs refine this simple policy to accommodate typical usage patterns without harming device safety.

Accommodating Bursts with Daily Quotas. The first drawback of the naïve policy is that it may not handle short "bursts" of heavy write I/Os that will average out over, say, one day. For instance, playing Final Fantasy consumes 3.83 MiB/s of bandwidth on our S6 device ($B = 1.46$ MiB/s). If the user only plays for two hours a day, the average over the period of a day will still be a safe 0.32 MiB/s. Our goal is to accommodate a good user experience, potentially borrowing against future periods of idleness, without leaving the device open to exploit. We refine the policy with two notions: **daily quotas** and **slack**.

We first refine the policy by dividing B for the remaining life into base bandwidth ($\hat{B}$) and slack bytes (S). We set S at 50% of W by default, which reduced B to $\hat{B} = 0.73$ MiB/s. Once the apps have collectively reached bandwidth $\hat{B}$ they may still write an additional S bytes to disk before I/O is throttled.

Second, we apportion slack into equal, per-day quotas S_{day}. For a device with $L = 2$ years, $S_{day} \approx 61.72$ GiB in case of our S6 device. This parameter should easily suffice for the typical case where a user operates apps for an average two hours a day and no malicious apps are installed. Underutilized portions of S_{day} are accumulated and re-distributed to the S_{day} values of the remaining days.

Without per-day slack, a single app could drain all of the slack for the lifespan of the device, leading to a potential denial-of-service attack for any app (or combination of apps) whose throughput is larger than $\hat{B}$. This is true for malicious apps, but also for any app violating our usage assumptions. For example, a user playing Final Fantasy for more than 9 hours a day on the S6 device would consume S before L, the two-year warranty period.

To exemplify how these thresholds work in more intensive usage scenarios, consider two edge cases for high-throughput apps. First, consider a camera app which takes an exuberant 3K photos a day. Assuming each picture occupies ~5 MiB of storage, the camera app would still yield an average 0.17 MiB/s throughput, for an overall consumption of 14.65 GiB a day—well below $\hat{B}$ when averaged over the course of a day. If the user were to take all 3K photos in a rapid-fire burst of, say, 120 seconds, the instantaneous throughput would be 125 MiB/s, which could violate our daily threshold; however, this would fit comfortably within daily slack. As a second case, consider an intensely avid gamer playing Final Fantasy for four hours daily; the gamer consumes 53.86 GiB of daily writes. Both I/O-intensive apps would operate in these cases without fully utilizing the daily slack and would neither disrupt user experience nor risk damage to the device.

In contrast, a malicious app operating at the maximum throughput of 160 MiB/s will drain S_{day} within 15 minutes. At this point, bandwidth that all apps can issue will be throttled to $\hat{B}$, until S_{day} is replenished.

Var.	Meaning (default value for Samsung S6 32GB)
W	Estimated total lifetime device write (88 TiB)
L	Warrantied device lifetime (2 years)
S	$= W/2$, slack capacity for burst I/O (44 TiB)
$\hat{B}$	$= (W - S)/L$, base bandwidth (0.73 MiB/s)
S_{fg}	Per day slack capacity for foreground apps (61.72 GiB)
S_{bg}	Per hour slack capacity for background apps (2.57 GiB)

Table 3: *Parameters (Var.) used in wear management policy.*

User Discretion and Malicious Apps. Typical users only operate apps for two hours a day [33, 49], making it *highly unlikely for benign apps to wear out the device in the common case.* In the absence of a malicious app, our rate limiting mechanism is not activated, even in extreme cases where benign, I/O-intensive apps are used over much longer periods. However, *benign apps may still wear-out the device if they issue a sufficiently heavy volume of write I/Os.*

Indiscriminately throttling I/O can result in a false positive identification of a wear-out attack. Moreover, we suspect that any heuristic that tries to automatically differentiate benign dangerous use from malicious dangerous use may simply lead malicious apps to operate just below the detection threshold.

Thus, the decision whether to allow dangerous I/O behaviors is better left to the user. This approach has precedents, such as services on Samsung devices [3] that alert users for computationally intensive apps that may drain the battery faster than expected, as well as slow down the device.

To assist the user in making this decision, our system tracks each app's I/O activity. Whenever an app utilizes a pre-determined watermark threshold of $W_{mark} = 0.5 \times S_{day}$ (30.86 GiB in our S6 device) our system warns the user that some apps are wearing out the storage device. The system then lists the daily top I/O consuming apps. The user can then limit I/O-consuming apps to a safe rate (e.g., $\hat{B}/Num_{Apps}$), or let them continue to exhaust the slack with user's consent. When an app is allowed to run without any rate limiting, it still consumes slack for other apps, in the interest of preserving overall lifespan.

Finally, we proportionally add unused $\hat{B}$ portions to S_{day} and W_{mark}, giving I/O-intensive apps more slack leeway after long periods of inactivity. Malicious apps operating at full speed will still be detected, while studiously remaining above the watermark still caps a malicious app at 50% of the adjusted S_{day}.

Differentiating foreground and background apps. Our measurements in §5.4 indicate that when benign apps are running in the background, the apps are less I/O-intensive than when running in the foreground. Meanwhile, our WAPP in §4 heavily relies on background I/O to stealthily carry out the attack. So we can improve the policy by detecting apps' foreground/background status, and treating each app differently depending on its status.

Our basic approach is to monitor slack usage at different granularities for foreground and background apps; slack usage for background apps is tracked at an hourly granularity, rather than daily. This approach bounds the amount of slack a malicious background app can consume before it is rate limited, on the assumption that a benign app should not issue this many writes in background mode.

More formally, foreground apps may use the daily slack (S_{day}); background apps may only use hourly slack ($S_{bg} = S_{fg}/24$), which is roughly 2.57 GiB on our running example S6 device. Similarly, the threshold to warn the user about excessive slack consumption is prorated differently for foreground and background apps. Each app's foreground/background status is available via Android activity stack, and, based on this status, our modified Android policy applies either foreground or background threshold values for that app. Thus, a malicious app in the background will hit the lower background threshold sooner, reducing potential harm to normal apps. At the end of each hour, any leftover hourly slack in S_{bg} is reclaimed and re-appropriated according to the current value of S_{fg}. Algorithm 1 details our monitoring policy.

Multiple Malicious Apps. Our analysis to date has assumed that one malicious app would issue an adversarial amount of I/O. The user is warned when one app consumes more than a configurable watermark of slack usage ($W^{fg}_{mark} = 0.5 \times S$, or half of the slack by default). One could imagine trying to evade this warning, or confuse the user, by spreading the malicious load across multiple coordinating apps.

At a minimum, our design warns the user whenever 75% of S_{fg} is consumed, so that she is aware that the device's lifetime is being consumed in an alarming rate, and can look at list of processes, ordered by I/O activity. One can also set W lower to increase sensitivity, perhaps at a cost of more queries to the user.

6.2 Implementation

We implement a wear management defense on the Samsung S6 with Android 6.0.1. This defense includes an extension to Linux kernel version 3.10.101, which tracks application I/O behavior, as well as a policy executor that can dynamically apply a configurable write-limiting policy. We track per-process writes and export this data via `/proc`. We use the Android notion of an App ID to correlate aggregate writes for an app across multiple process instantiations. Specifically, for each app, the kernel exports two values to `/proc`: (1) *DataWritten*, the total I/O written by the app, and (2) $Bandwidth_{app}$, the app's last observed per-second I/O bandwidth usage. Notably, a similar I/O monitoring mechanism tracking total individual app I/O was recently added to the Android kernel [29]. The policy executor tracks *DataWritten* and $Bandwidth_{app}$ values for all apps, and configures a rate-limiting policy, defaulting to what is described in §6.1. When necessary, rate limiting can be applied to all processes related to an app. In newer Linux kernels (4.5 or newer), `cgroups` v2 [83] includes native support for I/O rate limiting, obviating the need for our kernel changes; however, the phones we experimented with did not support such new kernels in Android.

6.3 Defense Evaluation

We evaluate the effectiveness of our mechanism in the presence of malicious apps, as well as its performance impact. We ran our experiments on a Samsung S6 phone, together with Android 6.0.1 on top of our Linux kernel variant based on version 3.10.101. The phone has 32 GB of UFS storage, running the Ext4 filesystem [24]. The relevant parameters used for this device are $L = 2$ years and $W = 88$ TiB, based on measurements in §3.

Algorithm 1: Wear management policy

Monitor($Apps, L, W$);
Input: Set of installed apps, Lifetime guarantee, Max. write volume of flash storage device
$S \leftarrow W \times 0.5$;
foreach *day in L* **do**
  $S_{fg} \leftarrow \frac{S}{\text{Days left in L}}$;
  $S \leftarrow S - S_{fg}$;
  $W^{fg}_{mark} \leftarrow S_{fb} \times 0.5$;
  $L \leftarrow L - 1$ day;
  $\hat{B} \leftarrow \frac{W-S}{L}$;
  reset all apps daily I/O counters;
  foreach *hour in day* **do**
    $S_{bg} \leftarrow \frac{S_{fb}}{\text{Hours left in day}}$;
    $S_{fg} \leftarrow S_{fg} - S_{bg}$;
    $W^{bg}_{mark} \leftarrow S_{bg} \times 0.5$;
    foreach *second t in hour* **do**
      $R_t \leftarrow$ total I/O rate in t;
      $R^{bg}_t \leftarrow$ total background I/O rate in t;
      $R^{fg}_t \leftarrow$ total foreground I/O rate in t;
      $W \leftarrow W - R_t$;
      foreach x *in Apps* **do**
        if $R_t > \hat{B}$ **then**
          Update $x.state$ to fg or bg;
          $x.slk^{x.state} \leftarrow x.slk^{x.state} + (R_t - \hat{B}) \times \frac{x.rate}{R_t}$;
          if $x.slk^{x.state} > W^{x.state}_{mark}$ **then**
            alert user to rate limit x;
      end
      $S_{fg} \leftarrow S_{fg} - R^{fg}_t + \hat{B}$;
      $S_{bg} \leftarrow S_{bg} - R^{bg}_t + \hat{B}$;
      if $R_t < \hat{B}$ **then**
        $W^{fg}_{mark} \leftarrow W^{fg}_{mark} + (\hat{B} - R_t) \times 0.5$;
      if $\frac{\text{total daily consumption}}{\text{initial } S_{fg}} > 0.75$ **then**
        alert user on lifetime consumption rate;
    end
    $S_{fg} \leftarrow S_{fg} + S_{bg}$;
  end
  $S \leftarrow S + S_{fg}$;
end

Our setup emulates common usage scenarios, where the phone is either (a) running one I/O-intensive app in the foreground, such as Camera or Final Fantasy, or (b) idle with some background activities. The phone is loaded with popular apps of different categories, including Spotify, Facebook, Gmail, and Chrome. In most cases, the apps' background activities are negligible, except that some apps might handle asynchronous events; for example, instant messaging apps may become active upon incoming messages. Here, we pick three scenarios to demonstrate the effectiveness of our wear management policy while inducing negligible interference on benign apps.

The first scenario uses the Camera app to shoot a video clip. The Camera app in our Samsung S6 will automatically stop the video capture after 5 minutes. An I/O activity trace of this scenario over 6 minutes is illustrated in Figure 7a. The average write throughput is ~7 MiB/s; if the user continues shooting video with the Camera app, it will take over 1.2 hours to reach W^{fg}_{mark}, at which point user will be alerted and the Camera app be rate limited or permission to exceed the limit is explicitly granted by the user. The I/O trace also shows a few spikes, notably from system processes, GMS (Google Mobile Service, detailed in next paragraph) and the Chrome browser. These are commensurate to normal background activities of benign apps, as measured in §5.

The second scenario involves Google Hangouts running in the background, but with one incoming message every 5 seconds. The write throughput is shown in Figure 7b. One interesting observation is that this activity involves not only the Hangouts app, but also the GMS service. GMS is a set of preset apps and essential service frameworks that can be used by apps[12]. FCM (Firebase Cloud Messaging, formerly Google Cloud Messaging), one of the services provided by GMS, is widely used by instant messaging and email clients, including Google Hangouts, for message/notification delivery [14, 17]. So the Hangouts app together with GMS processes stand out hand in hand in this 6 minutes I/O trace. However, their combined I/O throughput only adds up to ~300 KiB/s, which is just below the W^{bg}_{mark} if it keeps receiving messages continuously for the whole hour.

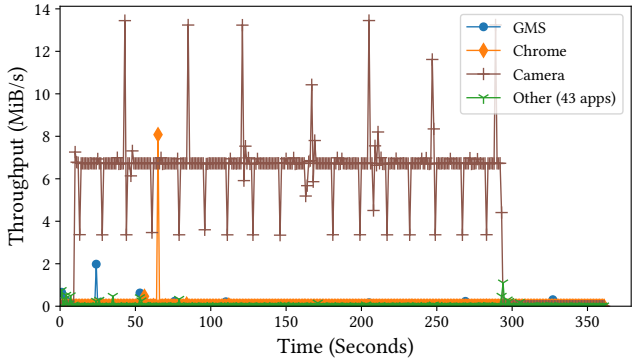
The third scenario adds the malicious WAPP, which spikes out to around 70 MiB/s (Figure 7c). Our wear management policy reacts within 30 seconds and throttles the malicious app to a safe rate.

Finally, we evaluated the performance overhead of our monitor on multiple write-intensive micro-benchmarks using Androbench [89], including all major I/O access patterns (random/sequential, read/write) as well as repeated accesses to an SQLite database. We compared the results of the micro-benchmarks on an Android kernel 3.10.101 with and without our write tracking changes. Our results show that any differences in performance do not display specific trends and are within experimental noise. We omit detailed results in the interest of brevity.

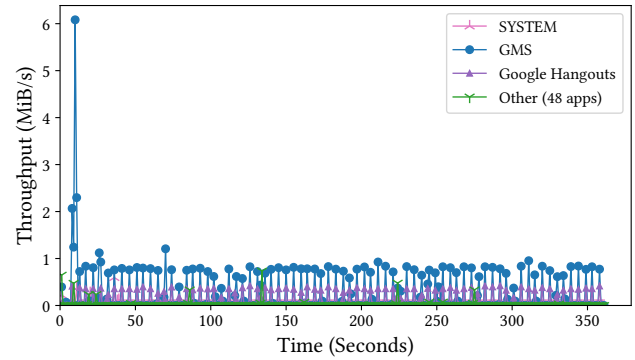
6.4 Write Amplification

Some I/O patterns, most notably random writes, tend to cause write amplification (WA) at the firmware level (§2). Thus, a malicious app could potentially circumvent our baseline rate limiting policy by issuing write patterns that maximize WA with significantly lower host-level write bandwidth.

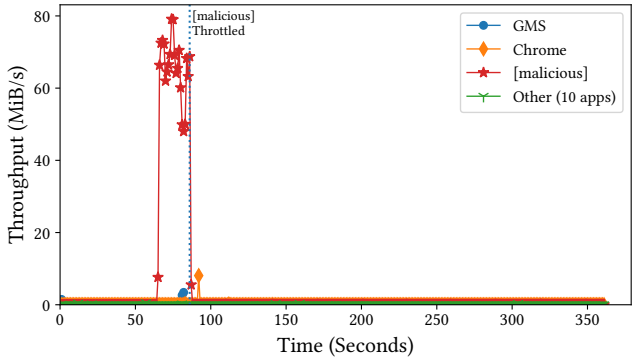
To demonstrate the destructive potential of this enhanced attack, we modified our WAPP application to first fill all of the free space left on the device with a large file. The malicious app then continues to rewrite this file using small 4 KiB random writes. Our measurements show that the modified WAPP only required 18 hours and 112 GiB worth of host-level writes to increment the lifetime indicator, i.e., $WAF = 80$. The resulting I/O rate of only 1.7 MiB/s (1% of the device's maximum throughput) is similar to that of many benign



(a) *Write throughput of video recording in the foreground with the Camera app, over a 5 minute period.*



(b) *Write throughput of Hangouts, running in background mode, with an incoming message every 5 seconds.*



(c) *Write throughput over time with the malicious app running in background. The throughput drops when the malicious app is rate limited.*

Figure 7: *I/O trace under wear management policy.*

apps and only slightly higher than the device's $B = 1.46$ MiB/s. As a comparison, to increment the indicator by 1, the original WAPP requires as much as 2000 GiB of host-level writes, but less than 10 hours at a much higher I/O rate of ~70 MiB/s. Similar tests on the external eMMC cards yielded a smaller WAF of 7–8x, most likely attributable to having an additional high-endurance flash landing zone. To avoid detecting the malicious app by its high space utilization, the malicious app could temporarily delete its files when the display is on. Notably, the effect of write amplification hinges on the amount of free space in the device, which effectively serves as additional over-provisioned space. Consequently, per-app storage space quotas may be used to impede this kind of attack.

Ideally, our model could account for write amplification by tracking physical writes, rather than logical writes. Unfortunately, most mobile devices do not directly report physical writes. Without reliable, fine-grain wear indicators or knowledge of the inner-workings of a device's specific firmware, one must infer write amplification. Specifically, workloads with a high degree of locality are less likely to induce write amplification. Furthermore, WA is strongly and inversely correlated to the amount of free, or over-provisioned, space in the system [52]. By taking into account the amount of free space and estimating the locality of each request using a technique such as in LAST [95], which classifies the locality of I/O requests according to their size, we can estimate the WAF and lower the expected number of logical writes (W) over the device's lifespan.

To conclude, an attacker may further deploy the wear-out attack using harmful, flash-specific, and maliciously shaped I/O workloads. We leave the exploration of a more refined, write-efficient wear-out attack and defense as future work.

7 FUTURE OF MOBILE STORAGE

Although this paper focuses on smart phones, the problems described in this paper are applicable to any device that has high bandwidth and low capacity storage with relatively low endurance. This includes wearables [119, 128, 135, 139] and IoT devices [75], which present attackers with a plethora of small-capacity non-volatile storage devices. There is a region where this trade-off is less of a problem. High-capacity SSD devices will require long periods of time to wear out, even under consistently strenuous write workloads. However, phone designers need to be cognizant of the potential danger of this trade-off, as mobile storage devices are likely to continue combining high bandwidth [21, 78] with smaller storage capacities when compared to their full-fledged SSD counterparts. It is easy to envision a future where mobile storage devices remain in a bad point in the bandwidth/endurance trade-off space.

We also note that users find increasingly creative ways to use smartphones for other purposes, such as Wi-Fi hotspots, servers, or security cameras [81, 118, 148]. This trend is likely to continue as users accumulate older phones with low resale value. Some of these usages could be considered normal use cases, such as parents occasionally using smartphones as baby monitors for short-periods, or repurposing an older phone for this purpose. We leave the exploration of such use cases as future work.

Our proposed attack has implications for future storage technologies as well—the key issues are the same as 2D, or planar flash—including overall write endurance per cell, the amount of over-provisioning, and the time it would take to write a lifetime's worth of data on the device. 3D flash [104, 121] stacks flash cell layers on top of each other to form denser flash devices; current 3D flash improves endurance and performance over planar flash. However, future 3D flash generations are predicted to suffer from similar

performance and reliability issues as those of their planar predecessors [43], as newer generations likely adopt known techniques that trade endurance for higher density and capacity.

Another important trend is the introduction of rival non-volatile memory technologies (NVM) [143] such as Phase-Change Memory (PCM) [85, 98, 105, 113, 138]. PCM promises significant improvements over flash in both bandwidth and endurance [91, 153]. Higher-bandwidth devices can be worn out faster by a malicious app. On the other hand, higher endurance of the underlying storage media means that a successful wear-out attack must issue larger volumes of writes. Thus, it is too early to say specifically how vulnerable PCM and other upcoming NVM technologies will be to this attack.

8 RELATED WORK

Flash/Android storage analysis. A long series of studies have examined whether durability and reliability of flash-based storage is a problem, as each generation of device is a moving target. Some studies raise concerns around the trend to sacrifice durability for density and lower cost [43, 62, 67]. Some have argued that the endurance of flash storage is sufficient for common-case, realistic workloads, especially with the help of wear-leveling techniques [42, 106, 125]. Meza et al. also demonstrate that flash durability is affected by factors like internal buffering, wear leveling, and controllers [103].

A large body of work is dedicated to modeling and improving flash device response times by improving garbage collection [146], device-level parallelism [47, 60, 70, 71], and using flash-aware schedulers [87, 112, 127]. However, these works mostly focus on improving request response times rather than estimating the wear effect of individual requests and applications.

Several studies [50, 63, 109] recently explored the I/O activity of Android applications. However, their measurements focused on the effect of I/O on app responsiveness and user experience.

Flash and Mobile I/O attacks. This work continues the theme of Prabhakaran et al. [114], who previously identified the importance of adjusting the storage stack for systems with "depletable storage", such as NAND flash. They proposed that flash P/E cycles be tracked, attributed and appropriated to specific applications by assigning "write credits", but did not implement, nor evaluate, mechanisms to limit the consumption of flash P/E cycles. The authors of GANGRENE [130] demonstrated the potential for a WAPP-like attack on small USB sticks in a desktop environment. We expand these attacks in a mobile environment, investigate how benign apps can also wear out the device and explore ways to defend against this phenomenon. Our prior work establishes wear-out as a problem with preliminary data [150]; this paper expands this motivating data on the problem, and proposes a solution, based on a thorough characterization of normal application behavior.

Exhausting depletable resources. The closest works to ours explore attacks on the lifetime of PCM-based RAM alternatives. Like flash, PCM cells can also endure only a limited number of writes, typically 10^7. Therefore, wear-leveling is also employed in PCM management [115, 151, 153]. Many works explored PCM wear out attacks and mitigation [72, 101, 116, 117, 126, 141]. However, these attacks often exploit specific properties of PCM, or rely on out-of-band knowledge of the wear-leveling implementation. Furthermore, PCM supports in-place writes and access units much smaller than flash (e.g., 64B). The ensuing simpler nature of PCM management schemes can make them more vulnerable to manipulations that repeatedly write data to the same physical location.

The closest work to this paper, in terms of in managing flash lifetime, is by Lee et al. [94], who explore techniques to limit writes in order to ensure that enterprise SSDs meet their expected lifetime. However, their work focuses on enterprise workloads, does not handle malicious workloads, uses system-level write throughput throttling, and utilizes flash-level behaviors which may not be applicable in newer memories [100].

The closest work in OS support for managing a depletable resource is managing power for mobile devices and applications [9, 9, 11, 11, 20, 51, 56, 68, 96, 99, 102, 111, 142, 149]. Unfortunately, approaches to managing power cannot be applied directly to flash since, unlike power, flash lifetime is a non-renewable resource, whose consumption is only reported by mobile devices at coarse granularity and typically obfuscated by firmware-specific proprietary details. DefDroid [73] protects against overuse of battery, as well as storage, but is disabled when "the device is being charged and connected to WiFi".

9 CONCLUSIONS

Mobile OSes assume, often incorrectly, that it is safe to allow apps to issue unconstrained writes to the device. Our results show that the ratio of bandwidth to capacity in the underlying device is often dangerously skewed, especially in the presence of less sophisticated firmware and simpler supporting hardware. We demonstrate how this can be corrected at the system software level with a carefully designed rate-limiting algorithm and configuration, without disrupting normal behavior.

This paper focuses on smartphones, but we believe the same issues apply to any small, flash-based devices on which third-party software can be loaded, potentially including critical infrastructure or internet-connected medical devices. The continuing proliferation of IoT and embedded devices to everyday life presents new opportunities to attack high-bandwidth, flash-based, storage devices with relatively small capacity that can be maliciously worn-out.

ACKNOWLEDGMENTS

We thank our shepherd, Mary Baker, and the anonymous reviewers for their insightful comments on earlier drafts of the work. This research was supported in part by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, grant # 2017702; the United States National Science Foundation (NSF) grant # CNS-1816263; VMware; and the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber directorate.

REFERENCES

[1] 2011. Android Alarm Manager. Android.com, https://developer.android.com/reference/android/app/AlarmManager.
[2] 2015. Android UI Automator. Android.com, https://developer.android.com/training/testing/ui-automator.
[3] 2016. Samsung Smart Manager. Samsung.com, https://www.samsung.com/global/galaxy/what-is/smart-manager/.

[4] 2017. Background Execution Limits. Android.com, https://developer.android.com/about/versions/oreo/background.

[5] 2017. BLU Advance 4.0L. Amazon.com, https://www.amazon.com/BLU-Advance-4-0L-Unlocked-Smartphone/dp/B00YCTIL88.

[6] 2017. BLU Dash D171a. Amazon.com, https://www.amazon.com/BLU-Dash-D171a-Unlocked-White/dp/B00F9AK2D6.

[7] 2017. fio benchmarking tool. https://github.com/axboe/fio.

[8] 2017. Motorola Moto E LTE. Amazon.com, https://www.amazon.com/Motorola-Moto-LTE-Contract-Cellular/dp/B00XQVDW6Y.

[9] 2017. Power Profiles for Android. Android.com, https://source.android.com/devices/tech/power/.

[10] 2017. Samsung Galaxy S6. Amazon.com, https://www.amazon.com/Samsung-Galaxy-S6-Factory-Unlocked/dp/B0143NXM68.

[11] 2017. Sensors power consumption. Android.com, https://source.android.com/devices/sensors/.

[12] 2018. Android – Google Mobile Services. Android.com, https://www.android.com/gms/.

[13] 2018. DiskBench on the App Store. App Store – Apple, https://itunes.apple.com/us/app/diskbench/id1166519285.

[14] 2018. Engage your users across Android, iOS and Chrome. Google.com, https://developers.google.com/cloud-messaging/.

[15] 2018. Google Pixel 3 Smartphone Review. NotebookCheck.net, https://www.notebookcheck.net/Google-Pixel-3-Smartphone-Review.366326.0.html.

[16] 2018. Motorola Moto G6 Smartphone Review. NotebookCheck.net, https://www.notebookcheck.net/Motorola-Moto-G6-Smartphone-Review.303202.0.html.

[17] 2019. Firebase Cloud Messaging. Google.com, https://firebase.google.com/docs/cloud-messaging/.

[18] 2019. Samsung Galaxy S9. Amazon.com, https://www.amazon.com/Samsung-SM-G960F-5-8-inches-Factory-Unlocked/dp/B079SQ5VHX.

[19] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (ATC)*.

[20] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab. Hamid, Feng Xia, and Muhammad Shiraz. 2015. A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications* 58, Supplement C (2015), 42 – 59.

[21] anand-nvme2015 2015. iPhone 6s and iPhone 6s Plus Preliminary Results. Anandtech, https://www.anandtech.com/show/9662/iphone-6s-and-iphone-6s-plus-preliminary-results.

[22] Anandtech.com. Apr. 2013. Samsung Galaxy S4 Review -Part 1. https://www.anandtech.com/show/6914/samsung-galaxy-s-4-review/6.

[23] Anandtech.com. Apr. 2014. Samsung Galaxy S5 Review. https://www.anandtech.com/show/7903/samsung-galaxy-s-5-review/7.

[24] Anandtech.com. Apr. 2015. The Samsung Galaxy S6 and S6 Edge Review. https://www.anandtech.com/show/9146/the-samsung-galaxy-s6-and-s6-edge-review/7.

[25] Anandtech.com. Jul. 2017. Samsung Galaxy S8 Showdown: Exynos 8895 vs. Snapdragon 835, Performance & Battery Life Tested. https://www.anandtech.com/show/11540/samsung-galaxy-s8-exynos-versus-snapdragon/3.

[26] Anandtech.com. Mar. 2016. The Samsung Galaxy S7 & S7 Edge Review, Part 1. https://www.anandtech.com/show/10120/the-samsung-galaxy-s7-review/3.

[27] Android.com. 2016. Application security. https://source.android.com/security/overview/app-security.html#how-users-understand-third-party-applications.

[28] Android.com. 2017. Android Practices for Security and Privacy. https://developer.android.com/training/articles/security-tips.html#StoringData.

[29] Android.com. 2017. Flash Wear Management in Android Automotive. https://source.android.com/devices/tech/perf/flash-wear.

[30] Android.com. Accessed Apr. 2018. Keeping the Device Awake. https://developer.android.com/training/scheduling/wakelock.html.

[31] Android.com. Accessed Apr. 2018. Monitoring the Battery Level and Charging State. https://developer.android.com/training/monitoring-device-state/battery-monitoring.html.

[32] Android.com. Accessed Apr. 2018. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[33] annie-usage Accessed Apr. 2018. Spotlight on Consumer App Usage. App Annie, https://www.appannie.com/en/insights/market-data/global-consumer-app-usage-data/.

[34] Apple.com. 2017. iOS App Sandbox Design Guide. https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html.

[35] ARS-ANDR8 2017. Android 8.0 Oreo, thoroughly reviewed. Arstechnica, https://arstechnica.com/gadgets/2017/09/android-8-0-oreo-thoroughly-reviewed/4.

[36] ArsTechnica. 2014. Consumer-grade SSDs actually last a hell of a long time. https://arstechnica.com/gadgets/2014/06/consumer-grade-ssds-actually-last-a-hell-of-a-long-time/.

[37] SD Association. 2000. SD Standard Overview. https://www.sdcard.org/developers/overview/index.html.

[38] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[39] Avraham Ben-Aroya and Sivan Toledo. 2006. Competitive Analysis of Flash-memory Algorithms. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14 (ESA)*. 100–111.

[40] Betanews. 2014. Modern SSDs can last a lifetime. https://betanews.com/2014/12/05/modern-ssds-can-last-a-lifetime/.

[41] Richard E. Blahut. 2003. *Algebraic Codes for Data Transmission:*. Cambridge University Press. https://doi.org/10.1017/CBO9780511800467

[42] Simona Boboila and Peter Desnoyers. 2010. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*.

[43] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. 2017. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. IEEE* 105, 9 (Sept 2017), 1666–1704.

[44] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. 2013. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *IEEE 31st International Conference on Computer Design (ICCD)*. 123–130. https://doi.org/10.1109/ICCD.2013.6657034

[45] Hong Cao and Miao Lin. 2017. Mining smartphone data for app usage prediction and recommendations: A survey. *Pervasive and Mobile Computing* 37 (2017), 1 – 22. https://doi.org/10.1016/j.pmcj.2017.01.007

[46] Li-Pin Chang. 2008. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *Asia and South Pacific Design Automation Conference*. 428–433. https://doi.org/10.1109/ASPDAC.2008.4483988

[47] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *Transaction on Storage* 12, 3, Article 13 (May 2016), 39 pages.

[48] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*.

[49] comscore 2017. The 2017 U.S. Mobile App Report. Comcast, https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report.

[50] J. Courville and F. Chen. 2016. Understanding storage I/O behaviors of mobile applications. In *32nd Symposium on Mass Storage Systems and Technologies (MSST)*.

[51] T. A. Dao, I. Singh, H. V. Madhyastha, S. V. Krishnamurthy, G. Cao, and P. Mohapatra. 2017. TIDE: A User-Centric Tool for Identifying Energy Hungry Applications on Smartphones. *IEEE/ACM Transactions on Networking* 25, 3 (June 2017), 1459–1474.

[52] Peter Desnoyers. 2012. Analytic Modeling of SSD Write Performance. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*.

[53] Peter Desnoyers. 2013. What Systems Researchers Need to Know about NAND Flash. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/desnoyers

[54] Devicespecifications.com. Mar. 2018. Samsung Galaxy S9 review. https://www.devicespecifications.com/en/editor-review/b9e76e/8.

[55] G. Dong, N. Xie, and T. Zhang. 2011. On the Use of Soft-Decision Error-Correction Codes in NAND Flash Memory. *IEEE Transactions on Circuits and Systems I: Regular Papers* 58, 2 (Feb 2011), 429–439. https://doi.org/10.1109/TCSI.2010.2071990

[56] Doze Accessed Apr. 2018. Optimizing for Doze and App Standby. Android.com, https://developer.android.com/training/monitoring-device-state/doze-standby.html.

[57] emarket 2017. eMarketer Unveils New Estimates for Mobile App. eMarketer, https://www.emarketer.com/Article/eMarketer-Unveils-New-Estimates-Mobile-App-Usage/1015611.

[58] Denzil Ferreira, Jorge Goncalves, Vassilis Kostakos, Louise Barkhuus, and Anind K. Dey. 2014. Contextual Experience Sampling of Mobile Application Micro-usage. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services (MobileHCI)*.

[59] Eran Gal and Sivan Toledo. 2005. Algorithms and Data Structures for Flash Memories. *Computing Surveys* 37, 2 (June 2005), 138–163. https://doi.org/10.1145/1089733.1089735

[60] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. M. Sha. 2014. Exploiting parallelism in I/Os scheduling for access conflict minimization in flash-based solid state drives. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*.

[61] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[62] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Conference on File and*

Storage Technologies (FAST). USENIX Association. http://dl.acm.org/citation.cfm?id=2208461.2208463

[63] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. 2018. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC)*.

[64] Hardkernel. 2017. ODROID | Hardkernel. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G145457216438.

[65] Mark A. Harris, Steven Furnell, and Karen Patten. 2014. Comparing the Mobile Device Security Behavior of College Students and Information Technology Professionals. *Journal of Information Privacy and Security* 10, 4 (2014), 186–202. https://doi.org/10.1080/15536548.2014.974429 arXiv:http://dx.doi.org/10.1080/15536548.2014.974429

[66] Christoph Hellwig. 2017. Improving Block Discard Support throughout the Linux Storage Stack. Vault Linux Storage and Filesystems Conference.

[67] Damien Hogan, Tom Arbuckle, and Conor Ryan. 2013. Estimating MLC NAND Flash Endurance: A Genetic Programming Based Symbolic Regression Application. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 1285–1292. https://doi.org/10.1145/2463372.2463537

[68] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. 2015. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* 48, 3, Article 39 (Dec. 2015), 40 pages.

[69] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of The Israeli Experimental Systems Conference (SYSTOR)*. ACM, Article 10, 9 pages. https://doi.org/10.1145/1534530.1534544

[70] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. 2013. Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance. *IEEE Trans. Comput.* 62, 6 (June 2013), 1141–1155.

[71] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the International Conference on Supercomputing (ICS)*.

[72] F. Huang, D. Feng, W. Xia, W. Zhou, Y. Zhang, M. Fu, C. Jiang, and Y. Zhou. 2016. Security RBSG: Protecting Phase Change Memory with Security-Level Adjustable Dynamic Mapping. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[73] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[74] James Imgraben, Alewyn Engelbrecht, and Kim-Kwang Raymond Choo. 2014. Always connected, but are smart mobile users getting more security savvy? A survey of smart mobile device users. *Behaviour & Information Technology* 33, 12 (2014), 1347–1360. https://doi.org/10.1080/0144929X.2014.934286 arXiv:http://dx.doi.org/10.1080/0144929X.2014.934286

[75] Intel. 2013. A guide to the internet of things. http://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html.

[76] Intel. 2017. Limited Warranties for Intel® Solid State Drives. Intel, http://www.intel.com/content/www/us/en/support/solid-state-drives/000005861.html.

[77] JEDEC. 2015. JEDEC e.MMC standard v5.1. https://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc.

[78] JEDEC. 2016. JEDEC Universal Flash Storage (UFS) standard v2.1. http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[79] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. 2013. AndroStep: Android Storage Performance Analysis Tool.. In *Software Engineering (Workshops)*, Vol. 13. 327–340.

[80] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Presented as part of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 309–320. https://www.usenix.org/conference/atc13/technical-sessions/presentation/jeong

[81] D. A. Johnson and M. M. Trivedi. 2011. Driving style recognition using a smartphone as a sensor platform. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. 1609–1615. https://doi.org/10.1109/ITSC.2011.6083078

[82] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[83] Kernel.org. 2015. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[84] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. *ACM Transactions on Storage* 8, 4 (November 2012), 1–25.

[85] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. 2014. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. *Transactions on Storage* 10, 4, Article 15 (Oct. 2014), 21 pages. https://doi.org/10.1145/2668128

[86] J. Kim, E. Lee, J. Choi, D. Lee, and S. H. Noh. 2016. Chip-Level RAID with Flexible Stripe Size and Parity Placement for Enhanced SSD Reliability. *IEEE Trans. Comput.* 65, 4 (April 2016), 1116–1130.

[87] J. Kim, E. Lee, and S. H. Noh. 2016. I/O Scheduling Schemes for Better I/O Proportionality on Flash-Based SSDs. In *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[88] J. M. Kim and J. S. Kim. 2012. Advil: A Pain Reliever for the Storage Performance of Mobile Devices. In *IEEE 15th International Conference on Computational Science and Engineering (CSE)*. 429–436. https://doi.org/10.1109/ICCSE.2012.66

[89] Je-Min Kim and Jin-Soo Kim. 2012. *AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices*. Springer Berlin Heidelberg, Berlin, Heidelberg, 667–674. https://doi.org/10.1007/978-3-642-27552-4_89

[90] Kingston. 2017. Class 4 microSDHC Card - 4GB–32GB | Kingston. https://www.kingston.com/us/flash/microsd_cards/sdc4.

[91] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. *SIGARCH Computer Architecture News* 37, 3 (June 2009), 2–13. https://doi.org/10.1145/1555815.1555758

[92] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 273–286. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[93] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. 2009. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *Proceedings of the Conference on USENIX Annual Technical Conference (USENIX)*. 1. http://dl.acm.org/citation.cfm?id=1855807.1855816

[94] Sungjin Lee, Taejin Kim, Kyungho Kim, and Jihong Kim. 2012. Lifetime Management of Flash-Based SSDs Using Recovery-Aware Dynamic Throttling. In *In Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST)*.

[95] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *SIGOPS Operating Systems Review* 42, 6 (Oct. 2008), 36–42.

[96] Matthew Lentz, James Litton, and Bobby Bhattacharjee. 2015. Drowsy Power Management. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*.

[97] LifeHacker. 2015. How Long Will My Hard Drives Really Last? LifeHacker, http://lifehacker.com/how-long-will-my-hard-drives-really-last-1700405627.

[98] D. Loke, T. H. Lee, W. J. Wang, L. P. Shi, R. Zhao, Y. C. Yeo, T. C. Chong, and S. R. Elliott. 2012. Breaking the Speed Limits of Phase-Change Memory. *Science* 336, 6088 (2012), 1566–1569. https://doi.org/10.1126/science.1221561 arXiv:http://www.sciencemag.org/content/336/6088/1566.full.pdf

[99] Hong Lu, Jun Yang, Zhigang Liu, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. 2010. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[100] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. 2018. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[101] Haiyu Mao, Xian Zhang, Guangyu Sun, and Jiwu Shu. 2017. Protect Non-volatile Memory from Wear-out Attack Based on Timing Difference of Row Buffer Hit/Miss. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1627–1630. http://dl.acm.org/citation.cfm?id=3130379.3130758

[102] Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. 2015. Selectively Taming Background Android Apps to Improve Battery Lifetime. In *USENIX Annual Technical Conference (USENIX ATC)*.

[103] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A Large-Scale Study of Flash Memory Failures in the Field. *SIGMETRICS Performance Evaluation Review* 43, 1 (June 2015), 177–190.

[104] Micron. 2015. 3D NAND. http://www.micron.com/products/nand-flash/3d-nand.

[105] Micron. 2015. 3D XPoint Technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[106] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R. Stan. 2010. How I Learned to Stop Worrying and Love Flash Endurance. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. USENIX Association. http://dl.acm.org/citation.cfm?id=1863122.1863125

[107] Alexios Mylonas, Anastasia Kastania, and Dimitris Gritzalis. 2013. Delegate the smartphone user? Security awareness in smartphone platforms. *Computers & Security* 34 (2013), 47 – 66. https://doi.org/10.1016/j.cose.2012.11.004

[108] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD Failures in Datacenters: What, When and Why? *SIGMETRICS Performance Evaluation* 44, 1 (June 2016).

[109] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. 2015. Reducing Smartphone Application Delay Through Read/Write Isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[110] Jon Oberheide and Charlie Miller. 2012. Dissecting the android bouncer. *SummerCon* (2012).

[111] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[112] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*.

[113] PCWorld. 2016. Intel experiments with 3D Xpoint as it ships out SSDs to testers. http://www.pcworld.com/article/3098769/storage/intel-experiments-with-3d-xpoint-as-it-ships-out-ssds-to-testers.html.

[114] Vijayan Prabhakaran, Mahesh Balakrishnan, John D. Davis, and Ted Wobber. 2010. Depletable Storage Systems. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*.

[115] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. 2009. Enhancing lifetime and security of PCM-based Main Memory with Start-Gap Wear Leveling. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[116] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini. 2011. Practical and secure PCM systems by online detection of malicious write streams. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*.

[117] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini. 2011. Practical and secure PCM systems by online detection of malicious write streams. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*.

[118] S. K. Ray, R. Sinha, and S. K. Ray. 2015. A smartphone-based post-disaster management mechanism using WiFi tethering. In *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*. 966–971. https://doi.org/10.1109/ICIEA.2015.7334248

[119] Samsung. 2014. Wearable Mobile Devices. http://www.samsung.com/uk/consumer/mobile-devices/wearables/.

[120] Samsung. 2017. SAMSUNG SSD Limited Warranty For All Samsung SSDs. Samsung, http://www.samsung.com/semiconductor/minisite/ssd/downloads/warranty/SAMSUNG_SSD_Limited_Warranty_English_US.pdf.

[121] Samsung. 2017. V-NAND Technology. Samsung, http://www.samsung.com/semiconductor/products/flash-storage/v-nand/.

[122] Samsung.com. 2018. UK Mobile Device warranty. http://www.samsung.com/uk/support/warranty/.

[123] Sandisk. 2017. Sandisk iNand Embedded Flash Drives. https://www.sandisk.com/oem-design/mobile/inand.

[124] SanDisk. 2017. SanDisk Product Warranty. SanDisk, https://www.sandisk.com/about/legal/warranty/warranty-table.

[125] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 67–80. http://usenix.org/conference/fast16/technical-sessions/presentation/schroeder

[126] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. 2010. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-change Memory with Dynamically Randomized Address Mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*.

[127] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC)*.

[128] Microsoft Store. 2017. Wearable technology. https://www.microsoftstore.com/store/msusa/en_US/cat/Wearable-technology/categoryID.67937000.

[129] The TechReport. 2015. The SSD Endurance Experiment: They're all dead. TechReport, http://techreport.com/review/27909/the-ssd-endurance-experiment-theyre-all-dead.

[130] Robert Templeman and Apu Kapadia. 2012. GANGRENE: Exploring the Mortality of Flash Memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec)*.

[131] Toshiba. 2016. Product manual, SG5 client SSD series. Toshiba, https://toshiba.semicon-storage.com/content/dam/toshiba-ss/asia-pacific/docs/product/storage/product-manual/SSD-SG5-Series-Brochure-Revision1.0.pdf.

[132] Toshiba. 2017. e·MMC™| TOSHIBA Storage & Electronic Devices Solutions Company. https://toshiba.semicon-storage.com/us/product/memory/nand-flash/mlc-nand/emmc.html.

[133] Toshiba. 2017. Storage Products Warranty. Toshiba, http://toshiba.semicon-storage.com/us/product/storage-products.html.

[134] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security)*. USENIX, 559–572. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_tielei

[135] Information Week. 2016. 10 Medical-Device Wearables To Improve Patients' Lives. http://www.informationweek.com/healthcare/mobile-and-wireless/10-medical-device-wearables-to-improve-patients-lives/d/d-id/1323544.

[136] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2420950.2420956

[137] Wikipedia. Accessed Apr. 2018. List of Intel SSDs. https://en.wikipedia.org/wiki/List_of_Intel_SSDs.

[138] H.-S.P. Wong, S. Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, B. Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (Dec 2010), 2201–2227. https://doi.org/10.1109/JPROC.2010.2070050

[139] Computer World. 2016. Why 'invisible smart glasses' are the perfect wearable. http://www.computerworld.com/article/3138534/wearables/why-invisible-smart-glasses-are-the-perfect-wearable.html.

[140] Guanying Wu and Xubin He. 2012. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. ACM, 253–266. https://doi.org/10.1145/2168836.2168862

[141] G. Wu, H. Zhang, Y. Dong, and J. Hu. 2012. CAR: Securing PCM Main Memory System with Cache Address Remapping. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*.

[142] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. 2015. Automated OS-level Device Runtime Power Management. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[143] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. 2011. Emerging Non-volatile Memories: Opportunities and Challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/2039370.2039420

[144] S. y. Park, E. Seo, J. Y. Shin, S. Maeng, and J. Lee. 2010. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Computer Architecture Letters* 9, 1 (Jan 2010), 9–12. https://doi.org/10.1109/L-CA.2010.3

[145] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. 2015. Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, USA, 257–271. http://dl.acm.org/citation.cfm?id=2750482.2750502

[146] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail Flash: Near-perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)*.

[147] Chengen Yang, Hsing-Min Chen, Trevor Mudge, and Chaitali Chakrabarti. 2014. Improving the Reliability of MLC NAND Flash Memories Through Adaptive Data Refresh and Error Control Coding. *Journal of Signal Processing Systems* 76, 3 (2014), 225–234. https://doi.org/10.1007/s11265-014-0880-5

[148] W. Yi, W. Jia, and J. Saniie. 2012. Mobile sensor data collector using Android smartphone. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 956–959. https://doi.org/10.1109/MWSCAS.2012.6292180

[149] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[150] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. 2017. Flash drive lifespan *is* a problem. In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*.

[151] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[152] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*. IEEE, 95–109.

[153] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. 2013. Phase-change Memory: An Architectural Perspective. *ACM Comput. Surv.* 45, 3, Article 29 (July 2013), 33 pages. https://doi.org/10.1145/2480741.2480746

[154] Aviad Zuck, Yue Li, Jehoshua Bruck, Donald E. Porter, and Dan Tsafrir. 2018. Stash in a Flash. In *16th USENIX Conference on File and Storage Technologies (FAST)*.

[155] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How?. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*.